

AutoCorres2

Matthew Brecknell David Greenaway Johannes Hölzl
Fabian Immler Gerwin Klein Rafal Kolanski
Japheth Lim Michael Norrish Norbert Schirmer
Salomon Sickert-Zehnter Thomas Sewell Harvey Tuch
Simon Wimmer

May 14, 2024

Abstract

AutoCorres2 is a tool to facilitate the verification of C programs within Isabelle [5]. It is a fork of [AutoCorres](#).

Contents

1	Introduction	11
I	Library	12
2	Misc. Definitions and Lemmas	13
2.1	Take, drop, zip, <i>list-all</i> etc rules	58
3	ML Antiquotations	68
3.1	Building terms: <i>mk-term</i>	68
3.2	Term pattern: <i>term-pat</i>	69
3.2.1	Example	79
3.3	ML Antiquotation to Match and Instantiate (recombine) cterms and terms	81
3.4	Record Antiquotation	90
3.4.1	Motivation	90
3.4.2	Example	91
3.4.3	Implementation	91
3.4.4	Examples	94
3.5	Various Antiquotations	96
3.5.1	Antiquotations for terms with schematic variables	96
3.5.2	Antiquotation for types with schematic variables	96
4	Proof Tools	98
4.1	Tools for handling Tuples	98
4.1.1	Antiquotations for terms with schematic variables	98
4.2	Tools for intro-rule based term synthesis <i>synthesize-rules</i>	129
4.2.1	Commands	129
4.2.2	ML Antiquotations	130
4.2.3	Attributes	130
5	Rule by Method	132
5.1	Tagging	145
5.1.1	Basic Definitions and Theorems	145
5.1.2	Conversions	147

5.1.3	Globbering	151
5.1.4	Reordering Subgoals	152
5.1.5	<i>subgoalT</i> and <i>subgoalsT</i> , and <i>prefersT</i>	153
6	Verification Condition Generator <i>runs-to-vcg</i>	159
6.1	Marked Assumptions	159
6.2	<i>THEN-ALL-NEW-FORWARD</i>	160
6.3	Basic VCG	161
6.4	Setup for Tagging	165
6.5	<i>runs-to-vcg</i>	166
6.6	Check	166
7	Proof Methods	167
7.1	Debugging methods	167
7.2	Simple Combinators	167
7.3	Advanced combinators	172
7.3.1	Protecting goal elements (assumptions or conclusion) from methods	172
7.3.2	Safe subgoal folding (avoids expanding meta-conjuncts)	172
7.4	Utility methods	174
7.4.1	Finding a goal based on successful application of a method	174
7.4.2	Remove redundant subgoals	175
7.5	Attribute methods (for use with <i>rule-by-method</i> attributes)	177
7.6	Shortcuts for <i>prove-prop.</i>	177
8	Option Monad (State Reader)	179
8.1	"While" loops over option monad.	185
8.2	Lift <i>option-while</i> to the (<i>'a, 's</i>) <i>lookup</i> monad	187
9	Mutual CCPO Recursion <i>fixed-point</i>	197
9.1	Relate orders between locale and type classes	197
9.2	Prove admissibility of <i>corresXF</i>	199
II	C-Parser	202
10	Theory Variants for Target Architectures via <i>L4V-ARCH</i>	203
11	Unified Memory Model (UMM)	206
11.1	More Word Lemmas	206
11.2	Distinct Proposition	224
11.3	Type setup	234
11.3.1	Pointers	234
11.3.2	Raw heap	236

11.4	Intervals	236
11.5	<i>dt-tuple</i> : a reimplementa- tion of 3 item tuples	236
11.6	Properties of pointers	237
11.7	Properties of the raw heap	238
11.8	Properties of intervals	238
11.9	C types setup	242
11.10	<i>super-update-bs</i>	277
11.11	Rest	278
11.12	Words and Pointers	346
11.13	Finite Cartesian Products	362
11.14	Auxiliary Theorems	369
11.15	Legacy definition <i>fg-cons</i>	371
11.16	Lense definition using an update function <i>lense</i>	371
11.17	Update for functions	373
11.18	Scenes	373
11.19	More properties of maps plus map disjunction.	509
11.19.1	Properties of maps not related to restriction	510
11.19.2	Properties of map restriction	511
11.20	Definitions	514
11.21	Properties of ($'-$)	514
11.22	Properties of map disjunction	514
11.22.1	Map associativity-commutativity based on map disjunction	515
11.22.2	Basic properties	516
11.22.3	Map disjunction and addition	516
11.22.4	Map disjunction and updates	518
11.22.5	Map disjunction and (\subseteq_m)	519
11.22.6	Map disjunction and restriction	520
11.23	<i>sep-point-tac</i>	642
11.24	<i>sep-exists-tac</i>	643
11.25	<i>sep-select-tac</i>	644
11.26	Instantiation, inferring type instantiation from term instantiation.	687
11.27	(Partial) Pointer Lenses	716
11.27.1	<i>pointer-lense</i>	716
11.27.2	<i>partial-pointer-lense</i>	716
11.28	<i>heap-upd</i> and <i>heap-upd-list</i>	890
11.29	<i>heap-upd</i>	890
11.30	<i>merge-ti</i>	893
11.30.1	<i>merge-ti-list</i>	898
12	More Building Blocks for our C- Language Model	902
12.1	addr bounds	902
12.2	More Heap Typing	903

12.2.1	Heap type tag and valid simple footprint	905
12.3	Pointers to local (stack) variables	910
12.4	Misc derived language elements	974
13	Packed Types (no implicit padding)	993
13.1	Underlying definitions for the class axioms	993
13.2	Lemmas about <i>td-fafu-idem</i>	994
13.2.1	<i>td-fa-hi</i>	998
13.3	Simp rules for deriving packed props from the type combinators	1000
13.3.1	<i>td-fafu-idem</i>	1000
13.3.2	<i>td-fa-hi</i>	1003
13.4	The type class and simp sets	1005
13.5	Instances	1006
13.6	Theorems about packed types	1007
13.6.1	<i>td-fa-hi</i>	1007
13.6.2	<i>td-fafu-idem</i>	1007
13.6.3	Proof automation for packed types	1014
13.7	Prettier Printing for Programs	1016
13.8	Modelling Local Variables	1028
14	Setup Lex / Yacc and Translation from C to Simpl	1045
15	Misc. Lemmas	1064
15.1	Abbreviations and helpers	1064
15.2	Basic operations	1064
15.2.1	<i>clift</i>	1064
15.2.2	<i>h-val</i>	1065
15.2.3	<i>h-t-valid</i>	1065
15.3	<i>field-lvalue</i>	1066
15.3.1	<i>heap-update</i>	1066
15.3.2	<i>c-guard</i>	1067
15.3.3	<i>clift</i>	1067
15.3.4	<i>cparent</i>	1068
15.3.5	<i>h-val</i>	1070
15.3.6	<i>h-t-valid</i>	1071
15.3.7	Type Combinators and Padding	1074
15.3.8	The orphanage: miscellaneous lemmas pulled up to (roughly) where they belong.	1075
III	AutoCorres	1100
16	Spec-Monad	1101
16.1	<i>rel-map</i> and <i>rel-project</i>	1101

16.2	Misc Theorems	.1102
16.3	Galois Connections	.1103
16.4	<i>post-state</i> type	.1105
16.4.1	Order Properties	.1107
16.4.2	<i>holds-post-state</i>	.1108
16.4.3	<i>holds-post-state-partial</i>	.1110
16.4.4	<i>sim-post-state</i>	.1111
16.4.5	<i>rel-post-state</i>	.1113
16.4.6	<i>lift-post-state</i>	.1113
16.4.7	<i>map-post-state</i>	.1114
16.4.8	<i>vmap-post-state</i>	.1115
16.4.9	<i>pure-post-state</i>	.1116
16.4.10	<i>bind-post-state</i>	.1117
16.5	<i>exception-or-result</i> type	.1119
16.6	<i>spec-monad</i> type	.1125
16.6.1	<i>rel-spec-monad</i>	.1137
16.7	VCG basic setup	.1138
16.7.1	<i>res-monad</i> and <i>exn-monad</i> Types	.1139
16.8	<i>res-monad</i> and <i>exn-monad</i> functions	.1148
16.9	Monad operations	.1148
16.9.1	\top	.1148
16.9.2	\perp	.1149
16.9.3	<i>fail</i>	.1149
16.9.4	<i>yield</i>	.1149
16.9.5	<i>throw-exception-or-result</i>	.1150
16.9.6	<i>throw</i>	.1150
16.9.7	<i>get-state</i>	.1150
16.9.8	<i>set-state</i>	.1151
16.9.9	<i>select</i>	.1152
16.9.10	<i>unknown</i>	.1152
16.9.11	<i>lift-state</i>	.1153
16.9.12	<i>constexec-concrete</i>	.1153
16.9.13	<i>constexec-abstract</i>	.1154
16.9.14	<i>bind-exception-or-result</i>	.1154
16.9.15	<i>bind-handle</i>	.1156
16.9.16	(\gg)	.1159
16.9.17	<i>assert</i>	.1169
16.9.18	<i>assume</i>	.1170
16.9.19	<i>assume-outcome</i>	.1170
16.9.20	<i>assume-result-and-state</i>	.1171
16.9.21	<i>gets</i>	.1172
16.9.22	<i>assert-result-and-state</i>	.1173
16.9.23	<i>assuming</i>	.1174
16.9.24	<i>guard</i>	.1174

16.9.25	<i>assert-opt</i>	.1176
16.9.26	<i>gets-the</i>	.1176
16.9.27	<i>modify</i>	.1177
16.9.28	<i>condition</i>	.1178
16.9.29	<i>when</i>	.1182
16.9.30	<i>While</i>	.1182
16.9.31	<i>map-value</i>	.1199
16.9.32	<i>liftE</i>	.1201
16.9.33	<i>try</i>	.1204
16.9.34	<i>finally</i>	.1205
16.9.35	<i>(<catch>)</i>	.1206
16.9.36	<i>check</i>	.1208
16.9.37	<i>ignoreE</i>	.1209
16.9.38	<i>on-exit'</i>	.1212
16.9.39	<i>run-bind</i>	.1214
16.9.40	Iteration of monadic actions	.1215
16.9.41	<i>forLoop</i>	.1216
16.9.42	<i>whileLoop-unroll-reachable</i>	.1217
16.9.43	<i>on-exit</i>	.1218
16.10	Setup for Tagging	.1221
16.11	<i>succeeds</i> and <i>reaches</i>	.1222
16.11.1	Relational rewriting for Monads	.1224
16.11.2	\top	.1227
16.11.3	\perp	.1227
16.11.4	<i>fail</i>	.1227
16.11.5	<i>yield</i>	.1227
16.11.6	<i>return</i>	.1228
16.11.7	<i>skip</i>	.1228
16.11.8	<i>throw-exception-or-result</i>	.1229
16.11.9	<i>throw</i>	.1229
16.11.10	<i>get-state</i>	.1229
16.11.11	<i>set-state</i>	.1229
16.11.12	<i>select</i>	.1230
16.11.13	<i>unknown</i>	.1230
16.11.14	<i>lift-state</i>	.1230
16.11.15	<i>constexec-concrete</i>	.1230
16.11.16	<i>constexec-abstract</i>	.1231
16.11.17	<i>bind-handle</i>	.1231
16.11.18	(\gg)	.1233
16.11.19	<i>assert</i>	.1234
16.11.20	<i>assume</i>	.1235
16.11.21	<i>assume-outcome</i>	.1235
16.11.22	<i>assume-result-and-state</i>	.1235
16.11.23	<i>gets</i>	.1236

16.11.24	<i>assert-result-and-state</i>	.1236
16.11.25	<i>assuming</i>	.1236
16.11.26	<i>guard</i>	.1237
16.11.27	<i>assert-opt</i>	.1237
16.11.28	<i>gets-the</i>	.1237
16.11.29	<i>modify</i>	.1237
16.11.30	<i>condition</i>	.1238
16.11.31	<i>when</i>	.1238
16.11.32	<i>While</i>	.1239
16.11.33	<i>map-value</i>	.1239
16.11.34	<i>liftE</i>	.1239
16.11.35	<i>try</i>	.1240
16.11.36	<i>finally</i>	.1240
16.11.37	<i>(<catch>)</i>	.1240
16.11.38	<i>check</i>	.1241
16.11.39	<i>ignoreE</i>	.1241
16.11.40	<i>bind-exception-or-result</i>	.1242
16.11.41	<i>bind-finally</i>	.1243
16.11.42	<i>run-bind</i>	.1243
17	Basic Stuff	1252
18	L1 phase	1269
18.1	Peep-hole L1 optimisations	.1298
18.2	Hoare-Triples for L1 (internal use)	.1316
19	L2 phase: local variable abstraction with lambdas	1324
19.1	Some Relators	.1340
19.2	Nested Exceptions	.1381
19.3	Transformations from single level exceptions to nested exceptions	.1381
19.4	Transformations for procedure local exceptions	.1393
19.4.1	Transformations for <i>try</i> and <i>finally</i>	.1393
19.5	Transformations on global exceptions	.1393
19.5.1	Removing unused tuple components	.1393
19.5.2	Setup basic rules	.1393
19.6	Peep-hole optimisations applied to L2	.1411
20	IO phase: In/Out Parameters	1427
20.1	Heap Typing for Split Heap	.1427
20.2	Valid root footprint	.1427
20.3	More Stack Typing	.1506
20.4	In Out Parameter Refinement	.1524

21 HL phase: Heap Lifting / Split Heap	1633
21.1 Open Types1674
21.1.1 Syntax <i>PTR-VALID</i> ('a)1708
21.1.2 Syntax <i>IS-VALID</i> ('a)1711
21.2 Refinement Lemmas1714
22 WA phase: Word Abstraction	1821
22.1 Basic Definitions1821
22.2 Abstracting values and expressions1822
22.3 Refinement Lemmas1833
23 TS phase: Type Strengthening (find suitable target monad)	1858
23.1 Synthesize Rules Setup1862
23.2 Pure Monad1863
23.3 Reader Monad (Gets)1865
23.4 Option (Reader) Monad1867
23.5 Nondet Monad1872
23.5.1 Elimination of <i>L2-try</i> in the Error Monad1883
23.6 Error Monad (exit)1890
24 Polishing the Final Outcome	1898
24.1 Support to normalise guards and array index expressions1916
24.2 Monad simplification with custom congruence rules1921
IV Documentation	1970
25 Quickstart	1971
25.1 Introduction1971
25.2 A First Proof with AutoCorres1971
25.2.1 Two simple functions: <code>min</code> and <code>max</code>1972
25.2.2 Invoking the C-parser1972
25.2.3 Invoking AutoCorres1974
25.2.4 Verifying <code>min</code>1975
25.2.5 Verifying <code>max</code>1975
25.3 More Complex Functions with AutoCorres1976
25.3.1 A simple loop: <code>mult_by_add</code>1976
25.3.2 <code>swap</code>1979
25.4 Command Options and Invocation1982
25.4.1 Session Structure1982
25.4.2 C-Parser1982
25.4.3 AutoCorres1983
26 Overview of AutoCorres	1986
26.1 Building Blocks1986

26.2	C Parser	.1987
26.3	AutoCorres	.1988
26.4	AutoCorres Flow	.1991
26.4.1	General Remarks	.1991
26.4.2	Links to more documentation / examples	.1992
26.5	Overview on the Locales	.1992
26.6	Example Program	.1992
26.6.1	Incremental Build	.1993
26.6.2	All the rest	.1994
26.7	Simplification strategy and dealing with tuples in L2-optimization phases	.1997
26.8	Simplification of conditions (guards, loops, conditionals)	.2002
26.9	Tricks to enforce first-order unification	.2005
26.10	Exception Rewriting	.2007
26.10.1	Preliminary examples illustrating the usage of <i>rel-spec-mona</i>	.2007
26.10.2	From <i>L2-catch</i> and flat exceptions to <i>L2-try</i> and nested exceptions	.2010
26.10.3	Flatten the error type of calls, aka get rid of constructor <i>Nonlocal</i>	.2015
26.11	Tuple optimization by analysing variable use and removing unused variables.	.2016
26.12	Locales, Local-Theories, Named-Targets, Morphisms and Declarations...	.2022
26.13	Morphisms and Declarations	.2026
26.13.1	Excuse on Proof Context vs. Local Theory.	.2030
26.13.2	Attributes vs. Local Theory Declarations	.2031
26.14	ML Antiquotations	.2033
26.15	Markup and Reports	.2035
26.16	Term Synthesize via Intro Rules	.2037
26.17	Pointers to Local Variables	.2042
26.17.1	Design choices	.2046
26.17.2	Open Ends / TODOs	.2052
26.18	In-Out Parameters, Abstracting Pointers to Values	.2053
26.18.1	Overview	.2053
26.18.2	Building Blocks	.2053
26.18.3	Options	.2058
26.18.4	Examples	.2058
26.18.5	Handling <code>exit</code> in function calls	.2065
26.18.6	Global Heap Pointers	.2066
26.18.7	Disclaimer / Caution	.2067
26.18.8	Implementation Aspects	.2067
26.18.9	pointer-parameters as data	.2068
26.18.10	Future work / Open Ends	.2068
26.19	Function Pointers	.2070

26.19.1	Global locales2072
26.19.2	Function / Recursive-clique specific locales2075
26.20	Unions2089
26.20.1	Union support in C-Parser and Autocorres2089
26.21	Pointers into Structures in Split Heap2093
26.21.1	Overview2093
26.21.2	Example Program and some Intuition2094
26.21.3	Background2096
26.21.4	User Level2101
26.21.5	Simulation Proof2106
26.21.6	Examples for normalisation of array indexes2155
26.21.7	Essence of Heap Lifting2156
27	C-Translation Infrastructure	2159
27.1	Local Variables2159
27.1.1	Basic ML primitives2160
27.1.2	Syntax2161
27.1.3	Simplifier setup2161
27.2	Infrastructure for states2164
27.3	Cached simproc examples2165
28	Translation of the StrictC Dialect (Outdated)	2169
28.1	Introduction2169
28.1.1	StrictC Subset Summary2170
28.2	Abstract Syntax2170
28.2.1	Regions2172
28.3	The Symbol Table2174
28.3.1	Functional Record Updates in SML2175
28.4	Creation of the Hoare Environment State2176
28.4.1	Representing Values in Memory2177
28.4.2	Pointers2178
28.4.3	Arrays2178
28.4.4	C struct Types2179
28.5	Translating Expressions2180
28.5.1	Undefined Behaviour2182
28.6	Concrete Syntax: Parsing and Lexing2183
28.6.1	Lexing and typedef Names2184
28.6.2	GCC <code>__attribute__</code> Declarations2185

Chapter 1

Introduction

AutoCorres2 is a tool to facilitate the verification of C programs within Isabelle [5]. It is a fork of AutoCorres: <https://trustworthy.systems/projects/OLD/autocorres/>. Here some quick links into the document:

- Quickstart guide for users ([chapter 25](#)): `doc/quickstart/Chapter1_MinMax.thy`
- Background information, internals and some history of AutoCorres ([chapter 26](#)): `doc/AutoCorresInfrastructure.thy`
- C-Parser
 - Some internals ([chapter 27](#)): `c-parser/CTranslationInfrastructure.thy`
 - Original documentation (outdated) ([chapter 28](#)): The supported subset of C is extended. Moreover, the C-parser is integrated into Isabelle/ML and no standalone C-parser is supplied. The description of the design principles is still valid: `c-parser/doc/ctranslation_body.tex`

Part I
Library

Chapter 2

Misc. Definitions and Lemmas

theory *More-Lib*

imports

Introduction-AutoCorres2

HOL-Library.Prefix-Order

Word-Lib.Word-Lib-Sumo

HOL-Eisbach.Eisbach-Tools

begin

abbreviation (*input*)

split $:: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$

where

split == *case-prod*

lemma *hd-map-simp*:

$b \neq [] \implies \text{hd } (\text{map } a \ b) = a \ (\text{hd } b)$

by (*rule hd-map*)

lemma *tl-map-simp*:

$\text{tl } (\text{map } a \ b) = \text{map } a \ (\text{tl } b)$

by (*induct b,auto*)

lemma *Collect-eq*:

$\{x. P \ x\} = \{x. Q \ x\} \longleftrightarrow (\forall x. P \ x = Q \ x)$

by (*rule iffI*) *auto*

lemma *iff-impI*: $[[P \implies Q = R]] \implies (P \longrightarrow Q) = (P \longrightarrow R)$ **by** *blast*

definition

$fun\text{-}app :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixr** \$ 10) **where**
 $f \$ x \equiv f x$

declare $fun\text{-}app\text{-}def$ [*iff*]

lemma $fun\text{-}app\text{-}cong$ [*fundef-cong*]:

$\llbracket f x = f' x' \rrbracket \Longrightarrow (f \$ x) = (f' \$ x')$
by *simp*

lemma $fun\text{-}app\text{-}apply\text{-}cong$ [*fundef-cong*]:

$f x y = f' x' y' \Longrightarrow (f \$ x) y = (f' \$ x') y'$
by *simp*

lemma $if\text{-}apply\text{-}cong$ [*fundef-cong*]:

$\llbracket P = P'; x = x'; P' \Longrightarrow f x' = f' x'; \neg P' \Longrightarrow g x' = g' x' \rrbracket$
 $\Longrightarrow (if P then f else g) x = (if P' then f' else g') x'$
by *simp*

lemma $case\text{-}prod\text{-}apply\text{-}cong$ [*fundef-cong*]:

$\llbracket f (fst p) (snd p) s = f' (fst p') (snd p') s' \rrbracket \Longrightarrow case\text{-}prod f p s = case\text{-}prod f' p' s'$
by (*simp add: split-def*)

lemma $prod\text{-}injects$:

$(x,y) = p \Longrightarrow x = fst p \wedge y = snd p$
 $p = (x,y) \Longrightarrow x = fst p \wedge y = snd p$
by *auto*

definition

$pred\text{-}imp :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$

where

$pred\text{-}imp P Q \equiv \forall x. P x \longrightarrow Q x$

lemma $pred\text{-}impI$: $(\bigwedge x. P x \Longrightarrow Q x) \Longrightarrow pred\text{-}imp P Q$

by (*simp add: pred-imp-def*)

definition

$pred\text{-}conj :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$ (**infixl** and 35)

where

$pred\text{-}conj P Q \equiv \lambda x. P x \wedge Q x$

definition

$pred\text{-}disj :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$ (**infixl** or 30)

where

$pred\text{-}disj P Q \equiv \lambda x. P x \vee Q x$

definition

$$\text{pred-neg} :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool}) (\text{not} - [40] 40)$$
where

$$\text{pred-neg } P \equiv \lambda x. \neg P x$$
lemma *pred-neg-simp*[simp]:
$$(\text{not } P) s \longleftrightarrow \neg (P s)$$

$$\text{by } (\text{simp add: pred-neg-def})$$
definition $K \equiv \lambda x y. x$ **definition**

$$\text{zipWith} :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow 'c \text{ list} \textbf{ where}$$

$$\text{zipWith } f \text{ xs } \text{ ys} \equiv \text{map } (\text{case-prod } f) (\text{zip } \text{xs } \text{ys})$$
primrec

$$\text{delete} :: 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$$
where

$$\text{delete } y [] = []$$

$$| \text{delete } y (x \# \text{xs}) = (\text{if } y=x \text{ then } \text{xs} \text{ else } x \# \text{delete } y \text{xs})$$
definition

$$\text{swp } f \equiv \lambda x y. f y x$$
lemma *swp-apply*[simp]: $\text{swp } f y x = f x y$

$$\text{by } (\text{simp add: swp-def})$$
primrec (*nonexhaustive*)
$$\text{theRight} :: 'a + 'b \Rightarrow 'b \textbf{ where}$$

$$\text{theRight } (\text{Inr } x) = x$$
primrec (*nonexhaustive*)
$$\text{theLeft} :: 'a + 'b \Rightarrow 'a \textbf{ where}$$

$$\text{theLeft } (\text{Inl } x) = x$$
definition

$$\text{isLeft } x \equiv (\exists y. x = \text{Inl } y)$$
definition

$$\text{isRight } x \equiv (\exists y. x = \text{Inr } y)$$
definition

$$\text{const } x \equiv \lambda y. x$$
primrec

$$\text{opt-rel} :: ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \text{ option} \Rightarrow 'b \text{ option} \Rightarrow \text{bool}$$
where

$$\text{opt-rel } f \text{ None } y = (y = \text{None})$$

$$| \text{opt-rel } f (\text{Some } x) y = (\exists y'. y = \text{Some } y' \wedge f x y')$$

lemma *opt-rel-None-rhs*[simp]:
 $opt-rel\ f\ x\ None = (x = None)$
by (*cases x, simp-all*)

lemma *opt-rel-Some-rhs*[simp]:
 $opt-rel\ f\ x\ (Some\ y) = (\exists x'. x = Some\ x' \wedge f\ x'\ y)$
by (*cases x, simp-all*)

lemma *tranclD2*:
 $(x, y) \in R^+ \implies \exists z. (x, z) \in R^* \wedge (z, y) \in R$
by (*erule tranclE*) *auto*

lemma *linorder-min-same1* [simp]:
 $(min\ y\ x = y) = (y \leq (x::'a::linorder))$
by (*auto simp: min-def linorder-not-less*)

lemma *linorder-min-same2* [simp]:
 $(min\ x\ y = y) = (y \leq (x::'a::linorder))$
by (*auto simp: min-def linorder-not-le*)

A combinator for pairing up well-formed relations. The divisor function splits the population in halves, with the True half greater than the False half, and the supplied relations control the order within the halves.

definition

$wf-sum :: ('a \Rightarrow bool) \Rightarrow ('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$

where

$wf-sum\ divisor\ r\ r' \equiv$
 $(\{(x, y). \neg divisor\ x \wedge \neg divisor\ y\} \cap r')$
 $\cup \{(x, y). \neg divisor\ x \wedge divisor\ y\}$
 $\cup (\{(x, y). divisor\ x \wedge divisor\ y\} \cap r)$

lemma *wf-sum-wf*:

$\llbracket wf\ r; wf\ r' \rrbracket \implies wf\ (wf-sum\ divisor\ r\ r')$

apply (*simp add: wf-sum-def*)

apply (*rule wf-Un*)**+**

apply (*erule wf-Int2*)

apply (*rule wf-subset*)

[**where** $r=measure\ (\lambda x. If\ (divisor\ x)\ 1\ 0)$]

apply *simp*

apply *clarsimp*

apply *blast*

apply (*erule wf-Int2*)

apply *blast*

done

abbreviation(*input*)

$option-map == map-option$

lemmas *option-map-def = map-option-case*

lemma *False-implies-equals* [*simp*]:
 $((False \implies P) \implies PROP Q) \equiv PROP Q$
apply (*rule equal-intr-rule*)
apply (*erule meta-mp*)
apply *simp*
apply *simp*
done

lemma *split-paired-Ball*:
 $(\forall x \in A. P x) = (\forall x y. (x,y) \in A \longrightarrow P (x,y))$
by *auto*

lemma *split-paired-Bex*:
 $(\exists x \in A. P x) = (\exists x y. (x,y) \in A \wedge P (x,y))$
by *auto*

lemma *bexI-minus*:
 $\llbracket P x; x \in A; x \notin B \rrbracket \implies \exists x \in A - B. P x$
unfolding *Bex-def* **by** *blast*

lemma *delete-remove1*:
 $delete\ x\ xs = remove1\ x\ xs$
by (*induct xs, auto*)

lemma *ignore-if*:
 $(y\ and\ z)\ s \implies (if\ x\ then\ y\ else\ z)\ s$
by (*clarsimp simp: pred-conj-def*)

lemma *zipWith-Nil2* :
 $zipWith\ f\ xs\ [] = []$
unfolding *zipWith-def* **by** *simp*

lemma *isRight-right-map*:
 $isRight\ (case-sum\ Inl\ (Inr\ o\ f)\ v) = isRight\ v$
by (*simp add: isRight-def split: sum.split*)

lemma *zipWith-nth*:
 $\llbracket n < \min\ (length\ xs)\ (length\ ys) \rrbracket \implies zipWith\ f\ xs\ ys\ !\ n = f\ (xs\ !\ n)\ (ys\ !\ n)$
unfolding *zipWith-def* **by** *simp*

lemma *length-zipWith* [*simp*]:
 $length\ (zipWith\ f\ xs\ ys) = \min\ (length\ xs)\ (length\ ys)$
unfolding *zipWith-def* **by** *simp*

lemma *first-in-uptoD*:

$a \leq b \implies (a::'a::order) \in \{a..b\}$
by *simp*

lemma *construct-singleton*:
 $\llbracket S \neq \{\}; \forall s \in S. \forall s'. s \neq s' \longrightarrow s' \notin S \rrbracket \implies \exists x. S = \{x\}$
by *blast*

lemmas *insort-com = insort-left-comm*

lemma *bleeding-obvious*:
 $(P \implies True) \equiv (Trueprop True)$
by *auto*

lemma *Some-helper*:
 $x = Some\ y \implies x \neq None$
by *simp*

lemma *in-empty-interE*:
 $\llbracket A \cap B = \{\}; x \in A; x \in B \rrbracket \implies False$
by *blast*

lemma *None-upd-eq*:
 $g\ x = None \implies g(x := None) = g$
by (*rule ext*) *simp*

lemma *exx [iff]*: $\exists x. x$ **by** *blast*
lemma *ExNot [iff]*: $Ex\ Not$ **by** *blast*

lemma *cases-simp2 [simp]*:
 $((\neg P \longrightarrow Q) \wedge (P \longrightarrow Q)) = Q$
by *blast*

lemma *a-imp-b-imp-b*:
 $((a \longrightarrow b) \longrightarrow b) = (a \vee b)$
by *blast*

lemma *length-neq*:
 $length\ as \neq length\ bs \implies as \neq bs$ **by** *auto*

lemma *take-neq-length*:
 $\llbracket x \neq y; x \leq length\ as; y \leq length\ bs \rrbracket \implies take\ x\ as \neq take\ y\ bs$
by (*rule length-neq, simp*)

lemma *eq-concat-lenD*:
 $xs = ys @ zs \implies length\ xs = length\ ys + length\ zs$
by *simp*

lemma *map-upt-reindex'*: $map\ f\ [a ..< b] = map\ (\lambda n. f\ (n + a - x))\ [x ..< x + b - a]$

by (rule nth-equalityI; clarsimp simp: add.commute)

lemma map-upt-reindex: $\text{map } f [a ..< b] = \text{map } (\lambda n. f (n + a)) [0 ..< b - a]$
 by (subst map-upt-reindex' [where x=0]) clarsimp

lemma notemptyI:
 $x \in S \implies S \neq \{\}$
 by clarsimp

lemma setcomp-Max-has-prop:
 assumes a: $P x$
 shows $P (\text{Max } \{(x::'a)::\{\text{finite}, \text{linorder}\}\}. P x)$
proof –
 from a have $\text{Max } \{x. P x\} \in \{x. P x\}$
 by – (rule Max-in, auto intro: notemptyI)
 thus ?thesis by auto
qed

lemma cons-set-intro:
 $\text{lst} = x \# \text{xs} \implies x \in \text{set lst}$
 by fastforce

lemma list-all2-conj-nth:
 assumes lall: list-all2 P as cs
 and rl: $\bigwedge n. \llbracket P (as ! n) (cs ! n); n < \text{length as} \rrbracket \implies Q (as ! n) (cs ! n)$
 shows list-all2 $(\lambda a b. P a b \wedge Q a b)$ as cs
proof (rule list-all2-all-nthI)
 from lall show $\text{length as} = \text{length cs} ..$
next
 fix n
 assume $n < \text{length as}$

 show $P (as ! n) (cs ! n) \wedge Q (as ! n) (cs ! n)$
proof
 from lall show $P (as ! n) (cs ! n)$ by (rule list-all2-nthD) fact
 thus $Q (as ! n) (cs ! n)$ by (rule rl) fact
qed
qed

lemma list-all2-conj:
 assumes lall1: list-all2 P as cs
 and lall2: list-all2 Q as cs
 shows list-all2 $(\lambda a b. P a b \wedge Q a b)$ as cs
proof (rule list-all2-all-nthI)
 from lall1 show $\text{length as} = \text{length cs} ..$
next
 fix n
 assume $n < \text{length as}$

```

show  $P (as ! n) (cs ! n) \wedge Q (as ! n) (cs ! n)$ 
proof
  from lall1 show  $P (as ! n) (cs ! n)$  by (rule list-all2-nthD) fact
  from lall2 show  $Q (as ! n) (cs ! n)$  by (rule list-all2-nthD) fact
qed

```

```

lemma all-set-into-list-all2:
  assumes lall:  $\forall x \in set\ ls. P\ x$ 
  and     length  $ls = length\ ls'$ 
  shows  list-all2  $(\lambda a\ b. P\ a)$  ls ls'
proof (rule list-all2-all-nthI)
  fix n
  assume  $n < length\ ls$ 
  from lall show  $P (ls ! n)$ 
  by (rule bspec [OF - nth-mem]) fact
qed fact

```

```

lemma GREATEST-lessE:
  fixes  $x :: 'a :: order$ 
  assumes gts:  $(GREATEST\ x. P\ x) < X$ 
  and     px:  $P\ x$ 
  and     gtst:  $\exists max. P\ max \wedge (\forall z. P\ z \longrightarrow (z \leq max))$ 
  shows   $x < X$ 
proof -
  from gtst obtain max where pm:  $P\ max$  and g':  $\bigwedge z. P\ z \implies z \leq max$ 
  by auto

```

```

  hence  $(GREATEST\ x. P\ x) = max$ 
  by (auto intro: Greatest-equality)

```

```

  moreover have  $x \leq max$  using px by (rule g')

```

```

  ultimately show ?thesis using gts by simp
qed

```

```

lemma set-has-max:
  fixes  $ls :: ('a :: linorder)\ list$ 
  assumes ls:  $ls \neq []$ 
  shows   $\exists max \in set\ ls. \forall z \in set\ ls. z \leq max$ 
  using ls
proof (induct ls)
  case Nil thus ?case by simp
next
  case (Cons l ls)

  show ?case
  proof (cases  $ls = []$ )
    case True

```

```

thus ?thesis by simp
next
  case False
  then obtain max where mv: max ∈ set ls and mm: ∀ z ∈ set ls. z ≤ max
using Cons.hyps
  by auto
  show ?thesis
  proof (cases max ≤ l)
    case True
    have l ∈ set (l # ls) by simp
    thus ?thesis
    proof
      from mm show ∀ z ∈ set (l # ls). z ≤ l using True by auto
    qed
  next
  case False
  from mv have max ∈ set (l # ls) by simp
  thus ?thesis
  proof
    from mm show ∀ z ∈ set (l # ls). z ≤ max using False by auto
  qed
qed
qed
qed

```

lemma True-notin-set-replicate-conv:
 $True \notin \text{set } ls = (ls = \text{replicate } (\text{length } ls) \text{ False})$
by (induct ls) simp+

lemma Collect-singleton-eqI:
 $(\bigwedge x. P x = (x = v)) \implies \{x. P x\} = \{v\}$
by auto

lemma exEI:
 $[\exists y. P y; \bigwedge x. P x \implies Q x] \implies \exists z. Q z$
by (rule ex-forward)

lemma allEI:
assumes $\forall x. P x$
assumes $\bigwedge x. P x \implies Q x$
shows $\forall x. Q x$
using assms **by** (rule all-forward)

General lemmas that should be in the library

lemma dom-ran:
 $x \in \text{dom } f \implies \text{the } (f x) \in \text{ran } f$
by (simp add: domD ranI)

lemma orthD1:

$\llbracket S \cap S' = \{\}; x \in S \rrbracket \Longrightarrow x \notin S'$ **by** *auto*

lemma *orthD2*:

$\llbracket S \cap S' = \{\}; x \in S \rrbracket \Longrightarrow x \notin S$ **by** *auto*

lemma *distinct-element*:

$\llbracket b \cap d = \{\}; a \in b; c \in d \rrbracket \Longrightarrow a \neq c$
by *auto*

lemma *ball-reorder*:

$(\forall x \in A. \forall y \in B. P x y) = (\forall y \in B. \forall x \in A. P x y)$
by *auto*

lemma *hd-map*: $ls \neq [] \Longrightarrow \text{hd} (\text{map } f \text{ } ls) = f (\text{hd } ls)$

by (*cases* *ls*) *auto*

lemma *tl-map*: $\text{tl} (\text{map } f \text{ } ls) = \text{map } f (\text{tl } ls)$

by (*cases* *ls*) *auto*

lemma *not-NilE*:

$\llbracket xs \neq []; \bigwedge x xs'. xs = x \# xs' \Longrightarrow R \rrbracket \Longrightarrow R$
by (*cases* *xs*) *auto*

lemma *length-SucE*:

$\llbracket \text{length } xs = \text{Suc } n; \bigwedge x xs'. xs = x \# xs' \Longrightarrow R \rrbracket \Longrightarrow R$
by (*cases* *xs*) *auto*

lemma *map-upt-unfold*:

assumes *ab*: $a < b$

shows $\text{map } f [a ..< b] = f a \# \text{map } f [\text{Suc } a ..< b]$

using *assms upt-conv-Cons* **by** *auto*

lemma *tl-nat-list-simp*:

$\text{tl} [a ..< b] = [a + 1 ..< b]$

by (*induct* *b*, *auto*)

lemma *image-Collect2*:

$\text{case-prod } f \text{ } \{x. P (\text{fst } x) (\text{snd } x)\} = \{f x y \mid x y. P x y\}$

by (*subst image-Collect*) *simp*

lemma *image-id'*:

$\text{id } \text{' } Y = Y$

by *clarsimp*

lemma *image-invert*:

assumes *r*: $f \circ g = \text{id}$

and *g*: $B = g \text{ ' } A$

shows $A = f \text{ ' } B$

by (*simp add: g image-comp r*)

lemma *Collect-image-fun-cong*:

assumes $rl: \bigwedge a. P a \implies f a = g a$
shows $\{f x \mid x. P x\} = \{g x \mid x. P x\}$
using rl **by** *force*

lemma *inj-on-take*:

shows *inj-on* (*take* n) $\{x. \text{drop } n \ x = k\}$
proof (*rule inj-onI*)

fix $x \ y$
assume $xv: x \in \{x. \text{drop } n \ x = k\}$
and $yv: y \in \{x. \text{drop } n \ x = k\}$
and $tk: \text{take } n \ x = \text{take } n \ y$

from xv **have** $\text{take } n \ x \ @ \ k = x$
using *append-take-drop-id mem-Collect-eq* **by** *auto*
moreover from $yv \ tk$
have $\text{take } n \ x \ @ \ k = y$
using *append-take-drop-id mem-Collect-eq* **by** *auto*
ultimately show $x = y$ **by** *simp*

qed

lemma *foldr-upd-dom*:

$\text{dom } (\text{foldr } (\lambda p \ ps. \ ps \ (p \mapsto f \ p)) \ as \ g) = \text{dom } g \cup \text{set } as$
proof (*induct as*)

case *Nil* **thus** *?case* **by** *simp*

next

case (*Cons* $a \ as$)
show *?case*
proof (*cases* $a \in \text{set } as \vee a \in \text{dom } g$)
case *True*
hence $ain: a \in \text{dom } g \cup \text{set } as$ **by** *auto*
hence $\text{dom } g \cup \text{set } (a \ \# \ as) = \text{dom } g \cup \text{set } as$ **by** *auto*
thus *?thesis* **using** *Cons* **by** *fastforce*

next

case *False*
hence $a \notin (\text{dom } g \cup \text{set } as)$ **by** *simp*
hence $\text{dom } g \cup \text{set } (a \ \# \ as) = \text{insert } a \ (\text{dom } g \cup \text{set } as)$ **by** *simp*
thus *?thesis* **using** *Cons* **by** *fastforce*

qed

qed

lemma *foldr-upd-app*:

assumes $xin: x \in \text{set } as$
shows $(\text{foldr } (\lambda p \ ps. \ ps \ (p \mapsto f \ p)) \ as \ g) \ x = \text{Some } (f \ x)$
(is $(?f \ as \ g) \ x = \text{Some } (f \ x)$
using xin

proof (*induct as arbitrary: x*)
case *Nil* **thus** *?case* **by** *simp*

next
case (*Cons a as*)
from *Cons.prem*s **show** ?*case* **by** (*subst foldr.simp*s) (*auto intro: Cons.hyps*)
qed

lemma *foldr-upd-app-other*:
assumes *xin*: $x \notin \text{set } as$
shows (*foldr* ($\lambda p ps. ps (p \mapsto f p)$) *as g*) $x = g x$
(is ($?f as g$) $x = g x$)
using *xin*
proof (*induct as arbitrary: x*)
case Nil thus ?*case* **by** *simp*
next
case (*Cons a as*)
from *Cons.prem*s **show** ?*case*
by (*subst foldr.simp*s) (*auto intro: Cons.hyps*)
qed

lemma *foldr-upd-app-if*:
foldr ($\lambda p ps. ps (p \mapsto f p)$) *as g* = ($\lambda x. \text{if } x \in \text{set } as \text{ then } \text{Some } (f x) \text{ else } g x$)
by (*auto simp: foldr-upd-app foldr-upd-app-other*)

lemma *foldl-fun-upd-value*:
 $\bigwedge Y. \text{foldl } (\lambda f p. f(p := X p)) Y e p = (\text{if } p \in \text{set } e \text{ then } X p \text{ else } Y p)$
by (*induct e*) *simp-all*

lemma *foldr-fun-upd-value*:
 $\bigwedge Y. \text{foldr } (\lambda p f. f(p := X p)) e Y p = (\text{if } p \in \text{set } e \text{ then } X p \text{ else } Y p)$
by (*induct e*) *simp-all*

lemma *foldl-fun-upd-eq-foldr*:
 $!!m. \text{foldl } (\lambda f p. f(p := g p)) m xs = \text{foldr } (\lambda p f. f(p := g p)) xs m$
by (*rule ext*) (*simp add: foldl-fun-upd-value foldr-fun-upd-value*)

lemma *Cons-eq-neq*:
 $\llbracket y = x; x \# xs \neq y \# ys \rrbracket \implies xs \neq ys$
by *simp*

lemma *map-upt-append*:
assumes *lt*: $x \leq y$
and *lt2*: $a \leq x$
shows $\text{map } f [a ..< y] = \text{map } f [a ..< x] @ \text{map } f [x ..< y]$
proof (*subst map-append [symmetric]*, *rule arg-cong [where f = map f]*)
from *lt* **obtain** *k* **where** *ky*: $x + k = y$
by (*auto simp: le-iff-add*)

thus $[a ..< y] = [a ..< x] @ [x ..< y]$
using *lt2*
by (*auto intro: upt-add-eq-append*)

qed

lemma *Min-image-distrib*:

assumes *minf*: $\bigwedge x y. \llbracket x \in A; y \in A \rrbracket \implies \min (f x) (f y) = f (\min x y)$

and *fa*: *finite A*

and *ane*: $A \neq \{\}$

shows $\text{Min } (f ` A) = f (\text{Min } A)$

proof –

have *rl*: $\bigwedge F. \llbracket F \subseteq A; F \neq \{\} \rrbracket \implies \text{Min } (f ` F) = f (\text{Min } F)$

proof –

fix *F*

assume *fa*: $F \subseteq A$ **and** *fne*: $F \neq \{\}$

have *finite F* **by** (*rule finite-subset*) *fact+*

thus *?thesis F*

unfolding *min-def* **using** *fa fne fa*

proof (*induct rule: finite-subset-induct*)

case *empty*

thus *?case* **by** *simp*

next

case (*insert x F*)

thus *?case*

by (*cases F = \{\}*) (*auto dest: Min-in intro: minf*)

qed

qed

show *?thesis* **by** (*rule rl [OF order-refl]*) *fact+*

qed

lemma *min-of-mono'*:

assumes $(f a \leq f c) = (a \leq c)$

shows $\min (f a) (f c) = f (\min a c)$

unfolding *min-def*

by (*subst if-distrib [where f = f, symmetric]*, *rule arg-cong [where f = f]*, *rule if-cong [OF - refl refl]*) *fact+*

lemma *nat-diff-less*:

fixes *x :: nat*

shows $\llbracket x < y + z; z \leq x \rrbracket \implies x - z < y$

using *less-diff-conv2* **by** *blast*

lemma *take-map-Not*:

$(\text{take } n (\text{map } \text{Not } xs) = \text{take } n xs) = (n = 0 \vee xs = [])$

by (*cases n*; *simp*) (*cases xs*; *simp*)

lemma *union-trans*:

assumes *SR*: $\bigwedge x y z. \llbracket (x,y) \in S; (y,z) \in R \rrbracket \implies (x,z) \in S^*$

shows $(R \cup S)^* = R^* \cup R^* \circ S^*$

```

apply (rule set-eqI)
apply clarsimp
apply (rule iffI)
apply (erule rtrancl-induct; simp)
apply (erule disjE)
apply (erule disjE)
apply (drule (1) rtrancl-into-rtrancl)
apply blast
apply clarsimp
apply (drule rtranclD [where R=S])
apply (erule disjE)
apply simp
apply (erule conjE)
apply (drule tranclD2)
apply (elim exE conjE)
apply (drule (1) SR)
apply (drule (1) rtrancl-trans)
apply blast
apply (rule disjI2)
apply (erule disjE)
apply (blast intro: in-rtrancl-UnI)
apply clarsimp
apply (drule (1) rtrancl-into-rtrancl)
apply (erule (1) relcompI)
apply (erule disjE)
apply (blast intro: in-rtrancl-UnI)
apply clarsimp
apply (blast intro: in-rtrancl-UnI rtrancl-trans)
done

```

lemma *trancl-trancl*:

```

 $(R^+)^+ = R^+$ 
by auto

```

Some rules for showing that the reflexive transitive closure of a relation/predicate doesn't add much if it was already transitively closed.

lemma *rtrancl-eq-reflc-trans*:

```

assumes trans:  $\text{trans } X$ 
shows  $\text{rtrancl } X = X \cup \text{Id}$ 
by (simp only: rtrancl-trancl-reflc trancl-id[OF trans])

```

lemma *rtrancl-id*:

```

assumes refl:  $\text{Id} \subseteq X$ 
assumes trans:  $\text{trans } X$ 
shows  $\text{rtrancl } X = X$ 
using refl rtrancl-eq-reflc-trans[OF trans]
by blast

```

lemma *rtranclp-eq-reflcp-transp*:

assumes *trans*: *transp X*
shows $rtranclp\ X = (\lambda x\ y.\ X\ x\ y \vee x = y)$
by (*simp add*: *Enum.rtranclp-rtrancl-eq fun-eq-iff*
rtrancl-eq-reflc-trans trans[unfolded transp-trans])

lemma *rtranclp-id*:
shows $reflp\ X \implies transp\ X \implies rtranclp\ X = X$
apply (*simp add*: *rtranclp-eq-reflcp-transp*)
apply (*auto simp*: *fun-eq-iff elim*: *reflpD*)
done

lemmas *rtranclp-id2 = rtranclp-id[unfolded reflp-def transp-relcompp le-fun-def]*

lemma *if-1-0-0*:
 $((if\ P\ then\ 1\ else\ 0) = (0 :: ('a :: zero-neq-one))) = (\neg\ P)$
by (*simp split*: *if-split*)

lemma *neq-Nil-lengthI*:
 $Suc\ 0 \leq length\ xs \implies xs \neq []$
by (*cases xs, auto*)

lemmas *ex-with-length = Ex-list-of-length*

lemma *in-singleton*:
 $S = \{x\} \implies x \in S$
by *simp*

lemma *singleton-set*:
 $x \in set\ [a] \implies x = a$
by *auto*

lemma *take-drop-eqI*:
assumes *t*: $take\ n\ xs = take\ n\ ys$
assumes *d*: $drop\ n\ xs = drop\ n\ ys$
shows $xs = ys$
proof –
have $xs = take\ n\ xs @ drop\ n\ xs$ **by** *simp*
with *t d*
have $xs = take\ n\ ys @ drop\ n\ ys$ **by** *simp*
moreover
have $ys = take\ n\ ys @ drop\ n\ ys$ **by** *simp*
ultimately
show *?thesis* **by** *simp*
qed

lemma *append-len2*:
 $zs = xs @ ys \implies length\ xs = length\ zs - length\ ys$
by *auto*

lemma *if-flip*:

$(\text{if } \neg P \text{ then } T \text{ else } F) = (\text{if } P \text{ then } F \text{ else } T)$

by *simp*

lemma *not-in-domIff*: $f x = \text{None} = (x \notin \text{dom } f)$

by *blast*

lemma *not-in-domD*:

$x \notin \text{dom } f \implies f x = \text{None}$

by (*simp add: not-in-domIff*)

definition

$\text{graph-of } f \equiv \{(x,y). f x = \text{Some } y\}$

lemma *graph-of-None-update*:

$\text{graph-of } (f (p := \text{None})) = \text{graph-of } f - \{p\} \times \text{UNIV}$

by (*auto simp: graph-of-def split: if-split-asm*)

lemma *graph-of-Some-update*:

$\text{graph-of } (f (p \mapsto v)) = (\text{graph-of } f - \{p\} \times \text{UNIV}) \cup \{(p,v)\}$

by (*auto simp: graph-of-def split: if-split-asm*)

lemma *graph-of-restrict-map*:

$\text{graph-of } (m \upharpoonright S) \subseteq \text{graph-of } m$

by (*simp add: graph-of-def restrict-map-def subset-iff*)

lemma *graph-ofD*:

$(x,y) \in \text{graph-of } f \implies f x = \text{Some } y$

by (*simp add: graph-of-def*)

lemma *graph-ofI*:

$m x = \text{Some } y \implies (x, y) \in \text{graph-of } m$

by (*simp add: graph-of-def*)

lemma *graph-of-empty* :

$\text{graph-of } \text{Map.empty} = \{\}$

by (*simp add: graph-of-def*)

lemma *graph-of-in-ranD*: $\forall y \in \text{ran } f. P y \implies (x,y) \in \text{graph-of } f \implies P y$

by (*auto simp: graph-of-def ran-def*)

lemma *graph-of-SomeD*:

$\llbracket \text{graph-of } f \subseteq \text{graph-of } g; f x = \text{Some } y \rrbracket \implies g x = \text{Some } y$

unfolding *graph-of-def*

by *auto*

lemma *graph-of-comp*:

$\llbracket g x = y; f y = \text{Some } z \rrbracket \implies (x,z) \in \text{graph-of } (f \circ g)$

by (*auto simp: graph-of-def*)

lemma *in-set-zip-refl* :

$(x,y) \in \text{set } (\text{zip } xs \ xs) = (y = x \wedge x \in \text{set } xs)$
by (*induct xs*) *auto*

lemma *map-conv-upd*:

$m \ v = \text{None} \implies m \ o \ (f \ (x := v)) = (m \ o \ f) \ (x := \text{None})$
by (*rule ext*) (*clarsimp simp: o-def*)

lemma *sum-all-ex* [*simp*]:

$(\forall a. x \neq \text{Inl } a) = (\exists a. x = \text{Inr } a)$
 $(\forall a. x \neq \text{Inr } a) = (\exists a. x = \text{Inl } a)$
by (*metis Inr-not-Inl sum.exhaust*)⁺

lemma *split-distrib*: $\text{case-prod } (\lambda a \ b. T \ (f \ a \ b)) = (\lambda x. T \ (\text{case-prod } (\lambda a \ b. f \ a \ b) \ x))$

by (*clarsimp simp: split-def*)

lemma *case-sum-triv* [*simp*]:

$(\text{case } x \ \text{of } \text{Inl } x \Rightarrow \text{Inl } x \mid \text{Inr } x \Rightarrow \text{Inr } x) = x$
by (*clarsimp split: sum.splits*)

lemma *set-eq-UNIV*: $(\{a. P \ a\} = \text{UNIV}) = (\forall a. P \ a)$

by *force*

lemma *allE2*:

$\llbracket \forall x \ y. P \ x \ y; P \ x \ y \implies R \rrbracket \implies R$
by *blast*

lemma *allE3*: $\llbracket \forall x \ y \ z. P \ x \ y \ z; P \ x \ y \ z \implies R \rrbracket \implies R$

by *auto*

lemma *my-BallE*: $\llbracket \forall x \in A. P \ x; y \in A; P \ y \implies Q \rrbracket \implies Q$

by (*simp add: Ball-def*)

lemma *unit-Inl-or-Inr* [*simp*]:

$(a \neq \text{Inl } ()) = (a = \text{Inr } ())$
 $(a \neq \text{Inr } ()) = (a = \text{Inl } ())$
by (*cases a;clarsimp*)⁺

lemma *disjE-L*: $\llbracket a \vee b; a \implies R; \llbracket \neg a; b \rrbracket \implies R \rrbracket \implies R$

by *blast*

lemma *disjE-R*: $\llbracket a \vee b; \llbracket \neg b; a \rrbracket \implies R; \llbracket b \rrbracket \implies R \rrbracket \implies R$

by *blast*

lemma *int-max-thms*:

$(a :: \text{int}) \leq \text{max } a \ b$
 $(b :: \text{int}) \leq \text{max } a \ b$

by (*auto simp: max-def*)

lemma *sgn-negation* [*simp*]:
 $\text{sgn } -(x::\text{int}) = - \text{sgn } x$
by (*clarsimp simp: sgn-if*)

lemma *sgn-sgn-nonneg* [*simp*]:
 $\text{sgn } (a :: \text{int}) * \text{sgn } a \neq -1$
by (*clarsimp simp: sgn-if*)

lemma *inj-inj-on*:
 $\text{inj } f \implies \text{inj-on } f A$
by (*metis injD inj-onI*)

lemma *ex-eqI*:
 $\llbracket \bigwedge x. f x = g x \rrbracket \implies (\exists x. f x) = (\exists x. g x)$
by *simp*

lemma *pre-post-ex*:
 $\llbracket \exists x. P x; \bigwedge x. P x \implies Q x \rrbracket \implies \exists x. Q x$
by *auto*

lemma *ex-conj-increase*:
 $((\exists x. P x) \wedge Q) = (\exists x. P x \wedge Q)$
 $(R \wedge (\exists x. S x)) = (\exists x. R \wedge S x)$
by *simp+*

lemma *all-conj-increase*:
 $((\forall x. P x) \wedge Q) = (\forall x. P x \wedge Q)$
 $(R \wedge (\forall x. S x)) = (\forall x. R \wedge S x)$
by *simp+*

lemma *Ball-conj-increase*:
 $xs \neq \{\} \implies ((\forall x \in xs. P x) \wedge Q) = (\forall x \in xs. P x \wedge Q)$
 $xs \neq \{\} \implies (R \wedge (\forall x \in xs. S x)) = (\forall x \in xs. R \wedge S x)$
by *auto*

lemma *disjoint-subset*:
assumes $A' \subseteq A$ **and** $A \cap B = \{\}$
shows $A' \cap B = \{\}$
using *assms* **by** *auto*

lemma *disjoint-subset2*:
assumes $B' \subseteq B$ **and** $A \cap B = \{\}$
shows $A \cap B' = \{\}$
using *assms* **by** *auto*

lemma *UN-nth-mem*:

$i < \text{length } xs \implies f (xs ! i) \subseteq (\bigcup x \in \text{set } xs. f x)$
by (*metis UN-upper nth-mem*)

lemma *Union-equal*:

$f ' A = f ' B \implies (\bigcup x \in A. f x) = (\bigcup x \in B. f x)$
by *blast*

lemma *UN-Diff-disjoint*:

$i < \text{length } xs \implies (A - (\bigcup x \in \text{set } xs. f x)) \cap f (xs ! i) = \{\}$
by (*metis Diff-disjoint Int-commute UN-nth-mem disjoint-subset*)

lemma *image-list-update*:

$f a = f (xs ! i)$
 $\implies f ' \text{set } (xs [i := a]) = f ' \text{set } xs$
by (*metis list-update-id map-update set-map*)

lemma *Union-list-update-id*:

$f a = f (xs ! i) \implies (\bigcup x \in \text{set } (xs [i := a]). f x) = (\bigcup x \in \text{set } xs. f x)$
by (*rule Union-equal*) (*erule image-list-update*)

lemma *Union-list-update-id'*:

$\llbracket i < \text{length } xs; \bigwedge x. g (f x) = g x \rrbracket$
 $\implies (\bigcup x \in \text{set } (xs [i := f (xs ! i)]). g x) = (\bigcup x \in \text{set } xs. g x)$
by (*metis Union-list-update-id*)

lemma *Union-subset*:

$\llbracket \bigwedge x. x \in A \implies (f x) \subseteq (g x) \rrbracket \implies (\bigcup x \in A. f x) \subseteq (\bigcup x \in A. g x)$
by (*metis UN-mono order-refl*)

lemma *UN-sub-empty*:

$\llbracket \text{list-all } P \text{ } xs; \bigwedge x. P x \implies f x = g x \rrbracket \implies (\bigcup x \in \text{set } xs. f x) - (\bigcup x \in \text{set } xs. g x)$
 $= \{\}$
by (*simp add: Ball-set-list-all[symmetric] Union-subset*)

lemma *bij-betw-fun-updI*:

$\llbracket x \notin A; y \notin B; \text{bij-betw } f \text{ } A \text{ } B \rrbracket \implies \text{bij-betw } (f(x := y)) \text{ } (\text{insert } x \text{ } A) \text{ } (\text{insert } y \text{ } B)$
by (*clarsimp simp: bij-betw-def fun-upd-image inj-on-fun-updI split: if-split-asm; blast*)

definition

$\text{bij-betw-map } f \text{ } A \text{ } B \equiv \text{bij-betw } f \text{ } A \text{ } (\text{Some } ' B)$

lemma *bij-betw-map-fun-updI*:

$\llbracket x \notin A; y \notin B; \text{bij-betw-map } f \text{ } A \text{ } B \rrbracket$
 $\implies \text{bij-betw-map } (f(x \mapsto y)) \text{ } (\text{insert } x \text{ } A) \text{ } (\text{insert } y \text{ } B)$

unfolding *bij-betw-map-def* **by** *clarsimp* (*erule* *bij-betw-fun-updI*; *clarsimp*)

lemma *bij-betw-map-imp-inj-on*:

bij-betw-map f A $B \implies \text{inj-on } f$ A

by (*simp* *add*: *bij-betw-map-def* *bij-betw-imp-inj-on*)

lemma *bij-betw-empty-dom-exists*:

$r = \{\}$ $\implies \exists t. \text{bij-betw } t$ $\{\}$ r

by (*clarsimp* *simp*: *bij-betw-def*)

lemma *bij-betw-map-empty-dom-exists*:

$r = \{\}$ $\implies \exists t. \text{bij-betw-map } t$ $\{\}$ r

by (*clarsimp* *simp*: *bij-betw-map-def* *bij-betw-empty-dom-exists*)

lemma *funpow-add* [*simp*]:

fixes $f :: 'a \Rightarrow 'a$

shows $(f \hat{\sim} a) ((f \hat{\sim} b) s) = (f \hat{\sim} (a + b)) s$

by (*metis* *comp-apply* *funpow-add*)

lemma *funpow-unfold*:

fixes $f :: 'a \Rightarrow 'a$

assumes $n > 0$

shows $f \hat{\sim} n = (f \hat{\sim} (n - 1)) \circ f$

by (*metis* *Suc-diff-1* *assms* *funpow-Suc-right*)

lemma *relpow-unfold*: $n > 0 \implies S \hat{\sim} n = (S \hat{\sim} (n - 1)) \circ S$

by (*cases* n , *auto*)

definition

equiv-of $:: ('s \Rightarrow 't) \Rightarrow ('s \times 's) \text{ set}$

where

equiv-of *proj* $\equiv \{(a, b). \text{proj } a = \text{proj } b\}$

lemma *equiv-of-is-equiv-relation* [*simp*]:

equiv *UNIV* (*equiv-of* *proj*)

by (*auto* *simp*: *equiv-of-def* *intro!*: *equivI* *refl-onI* *symI* *transI*)

lemma *in-equiv-of* [*simp*]:

$((a, b) \in \text{equiv-of } f) \iff (f a = f b)$

by (*clarsimp* *simp*: *equiv-of-def*)

lemma *equiv-relation-to-projection*:

```

fixes  $R :: ('a \times 'a) \text{ set}$ 
assumes equiv: equiv UNIV R
shows  $\exists f :: 'a \Rightarrow 'a \text{ set. } \forall x y. f x = f y \iff (x, y) \in R$ 
apply (rule exI [of -  $\lambda x. \{y. (x, y) \in R\}$ ])
apply clarsimp
subgoal for  $x y$ 
  apply (cases  $(x, y) \in R$ )
  apply clarsimp
  apply (rule set-eqI)
  apply clarsimp
  apply (metis equivE sym-def trans-def equiv)
  apply (clarsimp)
  apply (metis UNIV-I equiv equivE mem-Collect-eq refl-on-def)
done
done

```

```

lemma range-constant [simp]:
   $\text{range } (\lambda-. k) = \{k\}$ 
by (clarsimp simp: image-def)

```

```

lemma dom-unpack:
   $\text{dom } (\text{map-of } (\text{map } (\lambda x. (f x, g x)) xs)) = \text{set } (\text{map } (\lambda x. f x) xs)$ 
by (simp add: dom-map-of-conv-image-fst image-image)

```

```

lemma fold-to-disj:
   $\text{fold } (++) \text{ ms } a x = \text{Some } y \implies (\exists b \in \text{set ms. } b x = \text{Some } y) \vee a x = \text{Some } y$ 
by (induct ms arbitrary:a x y; clarsimp) blast

```

```

lemma fold-ignore1:
   $a x = \text{Some } y \implies \text{fold } (++) \text{ ms } a x = \text{Some } y$ 
by (induct ms arbitrary:a x y; clarsimp)

```

```

lemma fold-ignore2:
   $\text{fold } (++) \text{ ms } a x = \text{None} \implies a x = \text{None}$ 
by (metis fold-ignore1 option.collapse)

```

```

lemma fold-ignore3:
   $\text{fold } (++) \text{ ms } a x = \text{None} \implies (\forall b \in \text{set ms. } b x = \text{None})$ 
by (induct ms arbitrary:a x; clarsimp) (meson fold-ignore2 map-add-None)

```

```

lemma fold-ignore4:
   $b \in \text{set ms} \implies b x = \text{Some } y \implies \exists y. \text{fold } (++) \text{ ms } a x = \text{Some } y$ 
using fold-ignore3 by fastforce

```

```

lemma dom-unpack2:
   $\text{dom } (\text{fold } (++) \text{ ms } \text{Map.empty}) = \bigcup (\text{set } (\text{map dom ms}))$ 
apply (induct ms; clarsimp simp:dom-def)
apply (rule equalityI; clarsimp)
apply (drule fold-to-disj)

```

```

apply (erule disjE)
apply clarsimp
apply (rename-tac b)
apply (erule-tac x=b in ballE; clarsimp)
apply clarsimp
apply (rule conjI)
apply clarsimp
subgoal for - - - y
  apply (rule exI[where x = y])
  apply (erule fold-ignore1)
  done
apply clarsimp
apply (rename-tac y)
apply (erule-tac y=y in fold-ignore4; clarsimp)
done

```

lemma *fold-ignore5*: $\text{fold } (++) \text{ ms } a \ x = \text{Some } y \implies a \ x = \text{Some } y \vee (\exists b \in \text{set } \text{ms}. b \ x = \text{Some } y)$
by (induct ms arbitrary:a x y; clarsimp) blast

lemma *dom-inter-nothing*: $\text{dom } f \cap \text{dom } g = \{\} \implies \forall x. f \ x = \text{None} \vee g \ x = \text{None}$
by auto

lemma *fold-ignore6*:
 $f \ x = \text{None} \implies \text{fold } (++) \text{ ms } f \ x = \text{fold } (++) \text{ ms } \text{Map.empty } x$
apply (induct ms arbitrary:f x; clarsimp simp:map-add-def)
by (metis (no-types, lifting) fold-ignore1 option.collapse option.simps(4))

lemma *fold-ignore7*:
 $m \ x = m' \ x \implies \text{fold } (++) \text{ ms } m \ x = \text{fold } (++) \text{ ms } m' \ x$
apply (cases m x)
apply (frule-tac ms=ms in fold-ignore6)
apply (cut-tac f=m' and ms=ms and x=x in fold-ignore6)
apply clarsimp+
apply (rename-tac a)
apply (cut-tac ms=ms and a=m and x=x and y=a in fold-ignore1, clarsimp)
apply (cut-tac ms=ms and a=m' and x=x and y=a in fold-ignore1; clarsimp)
done

lemma *fold-ignore8*:
 $\text{fold } (++) \text{ ms } [x \mapsto y] = (\text{fold } (++) \text{ ms } \text{Map.empty})(x \mapsto y)$
apply (rule ext)
subgoal for xa
apply (cases xa = x)
apply clarsimp
apply (rule fold-ignore1)
apply clarsimp
apply (subst fold-ignore6; clarsimp)
done

done

lemma *fold-ignore9*:

$\llbracket \text{fold } (++) \text{ ms } [x \mapsto y] \ x' = \text{Some } z; \ x = x \rrbracket \implies y = z$
by (*subst (asm) fold-ignore8*) *clarsimp*

lemma *fold-to-map-of*:

$\text{fold } (++) \ (\text{map } (\lambda x. [f \ x \mapsto \ g \ x]) \ xs) \ \text{Map.empty} = \text{map-of } (\text{map } (\lambda x. (f \ x, \ g \ x)) \ xs)$

apply (*rule ext*)

subgoal for *x*

apply (*cases fold (++) (map (λx. [f x ↦ g x]) xs) Map.empty x*)

apply *clarsimp*

apply (*drule fold-ignore3*)

apply (*clarsimp split:if-split-asm*)

apply (*rule sym*)

apply (*subst map-of-eq-None-iff*)

apply *clarsimp*

apply (*rename-tac xa*)

apply (*erule-tac x=xa in ballE; clarsimp*)

apply *clarsimp*

apply (*frule fold-ignore5; clarsimp split:if-split-asm*)

apply (*subst map-add-map-of-foldr[where m=Map.empty, simplified]*)

apply (*induct xs arbitrary:f g; clarsimp split:if-split*)

apply (*rule conjI; clarsimp*)

apply (*drule fold-ignore9; clarsimp*)

apply (*cut-tac ms=map (λx. [f x ↦ g x]) xs and f=[f a ↦ g a] and x=f b in fold-ignore6, clarsimp*)

apply *auto*

done

done

lemma *if-n-0-0*:

$((\text{if } P \text{ then } n \text{ else } 0) \neq 0) = (P \wedge n \neq 0)$

by (*simp split: if-split*)

lemma *insert-dom*:

assumes *fx: f x = Some y*

shows $\text{insert } x \ (\text{dom } f) = \text{dom } f$

unfolding *dom-def using fx by auto*

lemma *map-comp-subset-dom*:

$\text{dom } (\text{prj } \circ_m \ f) \subseteq \text{dom } f$

unfolding *dom-def*

by (*auto simp: map-comp-Some-iff*)

lemmas *map-comp-subset-domD = subsetD [OF map-comp-subset-dom]*

lemma *dom-map-comp*:

$x \in \text{dom } (\text{prj} \circ_m f) = (\exists y z. f x = \text{Some } y \wedge \text{prj } y = \text{Some } z)$
by (*fastforce simp: dom-def map-comp-Some-iff*)

lemma *map-option-Some-eq2*:

$(\text{Some } y = \text{map-option } f x) = (\exists z. x = \text{Some } z \wedge f z = y)$
by (*metis map-option-eq-Some*)

lemma *map-option-eq-dom-eq*:

assumes *ome*: $\text{map-option } f \circ g = \text{map-option } f \circ g'$

shows $\text{dom } g = \text{dom } g'$

proof (*rule set-eqI*)

fix x

{

assume $x \in \text{dom } g$

hence $\text{Some } (f (\text{the } (g x))) = (\text{map-option } f \circ g) x$

by (*auto simp: map-option-case split: option.splits*)

also have $\dots = (\text{map-option } f \circ g') x$ **by** (*simp add: ome*)

finally have $x \in \text{dom } g'$

by (*auto simp: map-option-case split: option.splits*)

} **moreover**

{

assume $x \in \text{dom } g'$

hence $\text{Some } (f (\text{the } (g' x))) = (\text{map-option } f \circ g') x$

by (*auto simp: map-option-case split: option.splits*)

also have $\dots = (\text{map-option } f \circ g) x$ **by** (*simp add: ome*)

finally have $x \in \text{dom } g$

by (*auto simp: map-option-case split: option.splits*)

} **ultimately show** $(x \in \text{dom } g) = (x \in \text{dom } g')$ **by** *auto*

qed

lemma *cart-singleton-image*:

$S \times \{s\} = (\lambda v. (v, s)) \text{ ` } S$

by *auto*

lemma *singleton-eq-o2s*:

$(\{x\} = \text{set-option } v) = (v = \text{Some } x)$

by (*cases v, auto*)

lemma *option-set-singleton-eq*:

$(\text{set-option } \text{opt} = \{v\}) = (\text{opt} = \text{Some } v)$

by (*cases opt, simp-all*)

lemmas *option-set-singleton-eqs*

$= \text{option-set-singleton-eq}$

trans[OF eq-commute option-set-singleton-eq]

lemma *map-option-comp2*:

$\text{map-option } (f \circ g) = \text{map-option } f \circ \text{map-option } g$

by (simp add: option.map-comp fun-eq-iff)

lemma compD:

$\llbracket f \circ g = f \circ g'; g x = v \rrbracket \implies f (g' x) = f v$
by (metis comp-apply)

lemma map-option-comp-eqE:

assumes $om: \text{map-option } f \circ mp = \text{map-option } f \circ mp'$
and $p1: \llbracket mp x = \text{None}; mp' x = \text{None} \rrbracket \implies P$
and $p2: \bigwedge v v'. \llbracket mp x = \text{Some } v; mp' x = \text{Some } v'; f v = f v' \rrbracket \implies P$
shows P

proof (cases mp x)

case None

hence $x \notin \text{dom } mp$ by (simp add: domIff)

hence $mp' x = \text{None}$ by (simp add: map-option-eq-dom-eq [OF om] domIff)

with None show ?thesis by (rule p1)

next

case (Some v)

hence $x \in \text{dom } mp$ by clarsimp

then obtain v' where $\text{Some}' : mp' x = \text{Some } v'$ by (clarsimp simp add: map-option-eq-dom-eq [OF om])

with Some show ?thesis

proof (rule p2)

show $f v = f v'$ using Some' compD [OF om, OF Some] by simp

qed

qed

lemma Some-the:

$x \in \text{dom } f \implies f x = \text{Some } (\text{the } (f x))$

by clarsimp

lemma map-comp-update:

$f \circ_m (g(x \mapsto v)) = (f \circ_m g)(x := f v)$

apply (intro ext)

subgoal for y by (cases g y; simp)

done

lemma restrict-map-eqI:

assumes $req: A \mid^{\iota} S = B \mid^{\iota} S$

and $mem: x \in S$

shows $A x = B x$

proof –

from mem have $A x = (A \mid^{\iota} S) x$ by simp

also have $\dots = (B \mid^{\iota} S) x$ using req by simp

also have $\dots = B x$ using mem by simp

finally show ?thesis .

qed

lemma map-comp-eqI:

assumes $dm: \text{dom } g = \text{dom } g'$
and $fg: \bigwedge x. x \in \text{dom } g' \implies f (\text{the } (g' x)) = f (\text{the } (g x))$
shows $f \circ_m g = f \circ_m g'$
apply (*rule ext*)
subgoal for x
apply (*cases* $x \in \text{dom } g$)
apply (*frule subst* [*OF dm*])
apply (*clarsimp split: option.splits*)
apply (*frule domI* [**where** $m = g'$])
apply (*drule fg*)
apply (*simp*)
apply (*frule subst* [*OF dm*])
apply (*clarsimp*)
apply (*drule not-sym*)
apply (*clarsimp simp: map-comp-Some-iff*)
done
done

definition

$\text{modify-map } m \ p \ f \equiv m \ (p := \text{map-option } f \ (m \ p))$

lemma *modify-map-id*:

$\text{modify-map } m \ p \ \text{id} = m$

by (*auto simp add: modify-map-def map-option-case split: option.splits*)

lemma *modify-map-addr-com*:

assumes $\text{com}: x \neq y$

shows $\text{modify-map} (\text{modify-map } m \ x \ g) \ y \ f = \text{modify-map} (\text{modify-map } m \ y \ f) \ x \ g$

by (*rule ext*) (*simp add: modify-map-def map-option-case com split: option.splits*)

lemma *modify-map-dom* :

$\text{dom} (\text{modify-map } m \ p \ f) = \text{dom } m$

unfolding *modify-map-def* **by** (*auto simp: dom-def*)

lemma *modify-map-None*:

$m \ x = \text{None} \implies \text{modify-map } m \ x \ f = m$

by (*rule ext*) (*simp add: modify-map-def*)

lemma *modify-map-ndom* :

$x \notin \text{dom } m \implies \text{modify-map } m \ x \ f = m$

by (*rule modify-map-None*) *clarsimp*

lemma *modify-map-app*:

$(\text{modify-map } m \ p \ f) \ q = (\text{if } p = q \text{ then } \text{map-option } f \ (m \ p) \text{ else } m \ q)$

unfolding *modify-map-def* **by** *simp*

lemma *modify-map-apply*:

$m p = \text{Some } x \implies \text{modify-map } m p f = m (p \mapsto f x)$
by (*simp add: modify-map-def*)

lemma *modify-map-com*:

assumes *com*: $\bigwedge x. f (g x) = g (f x)$

shows $\text{modify-map } (\text{modify-map } m x g) y f = \text{modify-map } (\text{modify-map } m y f) x g$

using *assms* **by** (*auto simp: modify-map-def map-option-case split: option.splits*)

lemma *modify-map-comp*:

$\text{modify-map } m x (f o g) = \text{modify-map } (\text{modify-map } m x g) x f$

by (*rule ext*) (*simp add: modify-map-def option.map-comp*)

lemma *modify-map-exists-eq*:

$(\exists cte. \text{modify-map } m p' f p = \text{Some } cte) = (\exists cte. m p = \text{Some } cte)$

by (*auto simp: modify-map-def split: if-splits*)

lemma *modify-map-other*:

$p \neq q \implies (\text{modify-map } m p f) q = (m q)$

by (*simp add: modify-map-app*)

lemma *modify-map-same*:

$\text{modify-map } m p f p = \text{map-option } f (m p)$

by (*simp add: modify-map-app*)

lemma *next-update-is-modify*:

$\llbracket m p = \text{Some } cte'; cte = f cte' \rrbracket \implies (m(p \mapsto cte)) = \text{modify-map } m p f$

unfolding *modify-map-def* **by** *simp*

lemma *nat-power-minus-less*:

$a < 2 \wedge (x - n) \implies (a :: nat) < 2 \wedge x$

by (*erule order-less-le-trans*) *simp*

lemma *neg-rtranclI*:

$\llbracket x \neq y; (x, y) \notin R^+ \rrbracket \implies (x, y) \notin R^*$

by (*meson rtranclD*)

lemma *neg-rtrancl-into-trancl*:

$\neg (x, y) \in R^* \implies \neg (x, y) \in R^+$

by (*erule contrapos-nn, erule trancl-into-rtrancl*)

lemma *set-neqI*:

$\llbracket x \in S; x \notin S' \rrbracket \implies S \neq S'$

by *clarsimp*

lemma *set-pair-UN*:

$\{x. P x\} = \bigcup ((\lambda xa. \{xa\} \times \{xb. P (xa, xb)\}) \text{ ` } \{xa. \exists xb. P (xa, xb)\})$

by *fastforce*

lemma singleton-elemD: $S = \{x\} \implies x \in S$
by *simp*

lemma singleton-eqD: $A = \{x\} \implies x \in A$
by *blast*

lemma ball-ran-fun-updI:
 $\llbracket \forall v \in \text{ran } m. P v; \forall v. y = \text{Some } v \longrightarrow P v \rrbracket \implies \forall v \in \text{ran } (m (x := y)). P v$
by (*auto simp add: ran-def*)

lemma ball-ran-eq:
 $(\forall y \in \text{ran } m. P y) = (\forall x y. m x = \text{Some } y \longrightarrow P y)$
by (*auto simp add: ran-def*)

lemma cart-helper:
 $(\{\} = \{x\} \times S) = (S = \{\})$
by *blast*

lemmas converse-trancl-induct' = *converse-trancl-induct* [*consumes 1, case-names base step*]

lemma disjCI2: $(\neg P \implies Q) \implies P \vee Q$ **by** *blast*

lemma insert-UNIV :
 $\text{insert } x \text{ UNIV} = \text{UNIV}$
by *blast*

lemma not-singletonE:
 $\llbracket \forall p. S \neq \{p\}; S \neq \{\}; \bigwedge p p'. \llbracket p \neq p'; p \in S; p' \in S \rrbracket \implies R \rrbracket \implies R$
by *blast*

lemma not-singleton-oneE:
 $\llbracket \forall p. S \neq \{p\}; p \in S; \bigwedge p'. \llbracket p \neq p'; p' \in S \rrbracket \implies R \rrbracket \implies R$
using *not-singletonE* **by** *fastforce*

lemma ball-ran-modify-map-eq:
 $\llbracket \forall v. m x = \text{Some } v \longrightarrow P (f v) = P v \rrbracket$
 $\implies (\forall v \in \text{ran } (\text{modify-map } m x f). P v) = (\forall v \in \text{ran } m. P v)$
by (*auto simp: modify-map-def ball-ran-eq*)

lemma eq-singleton-redux:
 $\llbracket S = \{x\} \rrbracket \implies x \in S$
by *simp*

lemma if-eq-elem-helperE:
 $\llbracket x \in (\text{if } P \text{ then } S \text{ else } S'); \llbracket P; x \in S \rrbracket \implies a = b; \llbracket \neg P; x \in S' \rrbracket \implies a = c \rrbracket$
 $\implies a = (\text{if } P \text{ then } b \text{ else } c)$
by *fastforce*

lemma *if-option-Some*:

$((\text{if } P \text{ then None else Some } x) = \text{Some } y) = (\neg P \wedge x = y)$
by *simp*

lemma *insert-minus-eq*:

$x \notin A \implies A - S = (A - (S - \{x\}))$
by *auto*

lemma *modify-map-K-D*:

$\text{modify-map } m \ p \ (\lambda x. \ y) \ p' = \text{Some } v \implies (m \ (p \mapsto y)) \ p' = \text{Some } v$
by (*simp add: modify-map-def split: if-split-asm*)

lemma *tranclE2*:

assumes *trancl*: $(a, b) \in r^+$
and *base*: $(a, b) \in r \implies P$
and *step*: $\bigwedge c. [(a, c) \in r; (c, b) \in r^+] \implies P$
shows *P*
using *trancl base step*

proof –

note *rl* = *converse-trancl-induct* [**where** $P = \lambda x. x = a \longrightarrow P$]

from *trancl* **have** $a = a \longrightarrow P$

by (*rule rl, (iprover intro: base step)+*)

thus *?thesis* **by** *simp*

qed

lemmas *tranclE2' = tranclE2* [*consumes 1, case-names base trancl*]

lemma *weak-imp-cong*:

$[P = R; Q = S] \implies (P \longrightarrow Q) = (R \longrightarrow S)$
by *simp*

lemma *Collect-Diff-restrict-simp*:

$T - \{x \in T. Q \ x\} = T - \{x. Q \ x\}$
by (*auto intro: Collect-cong*)

lemma *Collect-Int-pred-eq*:

$\{x \in S. P \ x\} \cap \{x \in T. P \ x\} = \{x \in (S \cap T). P \ x\}$
by (*simp add: Collect-conj-eq [symmetric] conj-comms*)

lemma *Collect-restrict-predR*:

$\{x. P \ x\} \cap T = \{\} \implies \{x. P \ x\} \cap \{x \in T. Q \ x\} = \{\}$
by (*fastforce simp: disjoint-iff-not-equal*)

lemma *Diff-Un2*:

assumes *emptyad*: $A \cap D = \{\}$

and *emptybc*: $B \cap C = \{\}$

shows $(A \cup B) - (C \cup D) = (A - C) \cup (B - D)$

proof –

have $(A \cup B) - (C \cup D) = (A \cup B - C) \cap (A \cup B - D)$
by (*rule Diff-Un*)
also have $\dots = ((A - C) \cup B) \cap (A \cup (B - D))$ **using** *emptyad emptybc*
by (*simp add: Un-Diff Diff-triv*)
also have $\dots = (A - C) \cup (B - D)$
proof –
have $(A - C) \cap (A \cup (B - D)) = A - C$ **using** *emptyad emptybc*
by (*metis Diff-Int2 Diff-Int-distrib2 inf-sup-absorb*)
moreover
have $B \cap (A \cup (B - D)) = B - D$ **using** *emptyad emptybc*
by (*metis Int-Diff Un-Diff Un-Diff-Int Un-commute Un-empty-left inf-sup-absorb*)
ultimately show *?thesis*
by (*simp add: Int-Un-distrib2*)
qed
finally show *?thesis* .
qed

lemma *balleI*:
 $\llbracket \forall x \in S. Q\ x; \bigwedge x. \llbracket x \in S; Q\ x \rrbracket \implies P\ x \rrbracket \implies \forall x \in S. P\ x$
by *auto*

lemma *dom-if-None*:
 $\text{dom } (\lambda x. \text{if } P\ x \text{ then None else } f\ x) = \text{dom } f - \{x. P\ x\}$
by (*simp add: dom-def*) *fastforce*

lemma *restrict-map-Some-iff*:
 $((m \mid' S)\ x = \text{Some } y) = (m\ x = \text{Some } y \wedge x \in S)$
by (*cases x \in S, simp-all*)

lemma *context-case-bools*:
 $\llbracket \bigwedge v. P\ v \implies R\ v; \llbracket \neg P\ v; \bigwedge v. P\ v \implies R\ v \rrbracket \implies R\ v \rrbracket \implies R\ v$
by (*cases P v, simp-all*)

lemma *inj-on-fun-upd-strongerI*:
 $\llbracket \text{inj-on } f\ A; y \notin f\ ' (A - \{x\}) \rrbracket \implies \text{inj-on } (f(x := y))\ A$
by (*fastforce simp: inj-on-def*)

lemma *less-handy-casesE*:
 $\llbracket m < n; m = 0 \implies R; \bigwedge m' n'. \llbracket n = \text{Suc } n'; m = \text{Suc } m'; m < n \rrbracket \implies R \rrbracket$
 $\implies R$
by (*cases n; simp*) (*cases m; simp*)

lemma *subset-drop-Diff-strg*:
 $(A \subseteq C) \longrightarrow (A - B \subseteq C)$
by *blast*

lemma *inj-case-bool*:
 $\text{inj } (\text{case-bool } a\ b) = (a \neq b)$
by (*auto dest: inj-onD* [**where** $x = \text{True}$ **and** $y = \text{False}$] *intro: inj-onI split: bool.split-asm*)

lemma *foldl-fun-upd*:

$foldl (\lambda s r. s (r := g r)) f rs = (\lambda x. \text{if } x \in \text{set } rs \text{ then } g x \text{ else } f x)$
by (*induct rs arbitrary: f*) (*auto simp: fun-eq-iff*)

lemma *all-rv-choice-fn-eq-pred*:

$\llbracket \bigwedge rv. P rv \implies \exists fn. f rv = g fn \rrbracket \implies \exists fn. \forall rv. P rv \longrightarrow f rv = g (fn rv)$
apply (*rule exI[where x= $\lambda rv. \text{SOME } h. f rv = g h$]*)
apply (*clarsimp split: if-split*)
by (*meson someI-ex*)

lemma *ex-const-function*:

$\exists f. \forall s. f (f' s) = v$
by *force*

lemma *if-Const-helper*:

$\text{If } P (\text{Con } x) (\text{Con } y) = \text{Con } (\text{If } P x y)$
by (*simp split: if-split*)

lemmas *if-Some-helper = if-Const-helper[where Con=Some]*

lemma *expand-restrict-map-eq*:

$(m \upharpoonright' S = m' \upharpoonright' S) = (\forall x. x \in S \longrightarrow m x = m' x)$
by (*simp add: fun-eq-iff restrict-map-def split: if-split*)

lemma *disj-imp-rhs*:

$(P \implies Q) \implies (P \vee Q) = Q$
by *blast*

lemma *remove1-filter*:

$\text{distinct } xs \implies \text{remove1 } x xs = \text{filter } (\lambda y. x \neq y) xs$
by (*induct xs*) (*auto intro!: filter-True [symmetric]*)

lemma *Int-Union-empty*:

$(\bigwedge x. x \in S \implies A \cap P x = \{\}) \implies A \cap (\bigcup x \in S. P x) = \{\}$
by *auto*

lemma *UN-Int-empty*:

$(\bigwedge x. x \in S \implies P x \cap T = \{\}) \implies (\bigcup x \in S. P x) \cap T = \{\}$
by *auto*

lemma *disjointI*:

$\llbracket \bigwedge x y. \llbracket x \in A; y \in B \rrbracket \implies x \neq y \rrbracket \implies A \cap B = \{\}$
by *auto*

lemma *UN-disjointI*:

assumes *rl*: $\bigwedge x y. \llbracket x \in A; y \in B \rrbracket \implies P x \cap Q y = \{\}$
shows $(\bigcup x \in A. P x) \cap (\bigcup x \in B. Q x) = \{\}$
by (*auto dest: rl*)

lemma *UN-set-member*:

assumes *sub*: $A \subseteq (\bigcup x \in S. P x)$
and $nz: A \neq \{\}$
shows $\exists x \in S. P x \cap A \neq \{\}$

proof –

from *nz* **obtain** *z* **where** *zA*: $z \in A$ **by** *fastforce*
with *sub* **obtain** *x* **where** $x \in S$ **and** $z \in P x$ **by** *auto*
hence $P x \cap A \neq \{\}$ **using** *zA* **by** *auto*
thus *?thesis* **using** *sub nz* **by** *auto*

qed

lemma *append-Cons-cases* [*consumes 1, case-names pre mid post*]:

$\llbracket (x, y) \in \text{set } (as @ b \# bs);$
 $(x, y) \in \text{set } as \implies R;$
 $\llbracket (x, y) \notin \text{set } as; (x, y) \notin \text{set } bs; (x, y) = b \rrbracket \implies R;$
 $(x, y) \in \text{set } bs \implies R \rrbracket \implies R$
by *auto*

lemma *cart-singletons*:

$\{a\} \times \{b\} = \{(a, b)\}$
by *blast*

lemma *disjoint-subset-neg1*:

$\llbracket B \cap C = \{\}; A \subseteq B; A \neq \{\} \rrbracket \implies \neg A \subseteq C$
by *auto*

lemma *disjoint-subset-neg2*:

$\llbracket B \cap C = \{\}; A \subseteq C; A \neq \{\} \rrbracket \implies \neg A \subseteq B$
by *auto*

lemma *iffE2*:

$\llbracket P = Q; \llbracket P; Q \rrbracket \implies R; \llbracket \neg P; \neg Q \rrbracket \implies R \rrbracket \implies R$
by *blast*

lemma *list-case-If*:

$(\text{case } xs \text{ of } [] \Rightarrow P \mid - \Rightarrow Q) = (\text{if } xs = [] \text{ then } P \text{ else } Q)$
by (*rule list.case-eq-if*)

lemma *remove1-Nil-in-set*:

$\llbracket \text{remove1 } x \text{ } xs = []; xs \neq [] \rrbracket \implies x \in \text{set } xs$
by (*induct xs*) (*auto split: if-split-asm*)

lemma *remove1-empty*:

$(\text{remove1 } v \text{ } xs = []) = (xs = [v] \vee xs = [])$
by (*cases xs; simp*)

lemma *set-remove1*:

$x \in \text{set } (\text{remove1 } y \text{ } xs) \implies x \in \text{set } xs$

by (*induct xs*) (*auto split: if-split-asm*)

lemma *If-rearrange*:

$(\text{if } P \text{ then if } Q \text{ then } x \text{ else } y \text{ else } z) = (\text{if } P \wedge Q \text{ then } x \text{ else if } P \text{ then } y \text{ else } z)$

by *simp*

lemma *disjI2-strg*:

$Q \longrightarrow (P \vee Q)$

by *simp*

lemma *eq-imp-strg*:

$P t \longrightarrow (t = s \longrightarrow P s)$

by *clarsimp*

lemma *if-both-strengthen*:

$P \wedge Q \longrightarrow (\text{if } G \text{ then } P \text{ else } Q)$

by *simp*

lemma *if-both-strengthen2*:

$P s \wedge Q s \longrightarrow (\text{if } G \text{ then } P \text{ else } Q) s$

by *simp*

lemma *if-swap*:

$(\text{if } P \text{ then } Q \text{ else } R) = (\text{if } \neg P \text{ then } R \text{ else } Q)$ **by** *simp*

lemma *imp-consequent*:

$P \longrightarrow Q \longrightarrow P$ **by** *simp*

lemma *list-case-helper*:

$xs \neq [] \implies \text{case-list } f g xs = g (\text{hd } xs) (\text{tl } xs)$

by (*cases xs, simp-all*)

lemma *list-cons-rewrite*:

$(\forall x xs. L = x \# xs \longrightarrow P x xs) = (L \neq [] \longrightarrow P (\text{hd } L) (\text{tl } L))$

by (*auto simp: neq-Nil-conv*)

lemma *list-not-Nil-manip*:

$[[xs = y \# ys; \text{case } xs \text{ of } [] \Rightarrow \text{False} \mid (y \# ys) \Rightarrow P y ys]] \implies P y ys$

by *simp*

lemma *ran-ball-triv*:

$\bigwedge P m S. [[\forall x \in (\text{ran } S). P x ; m \in (\text{ran } S)]] \implies P m$

by *blast*

lemma *singleton-tuple-cartesian*:

$(\{(a, b)\} = S \times T) = (\{a\} = S \wedge \{b\} = T)$

$(S \times T = \{(a, b)\}) = (\{a\} = S \wedge \{b\} = T)$

by *blast+*

lemma *strengthen-ignore-if*:

$A\ s \wedge B\ s \longrightarrow (\text{if } P \text{ then } A \text{ else } B)\ s$

by *clarsimp*

lemma *case-sum-True* :

$(\text{case } r \text{ of } \text{Inl } a \Rightarrow \text{True} \mid \text{Inr } b \Rightarrow f\ b) = (\forall b. r = \text{Inr } b \longrightarrow f\ b)$

by (*cases r*) *auto*

lemma *sym-ex-elim*:

$F\ x = y \Longrightarrow \exists x. y = F\ x$

by *auto*

lemma *tl-drop-1* :

$\text{tl } xs = \text{drop } 1\ xs$

by (*simp add: drop-Suc*)

lemma *upt-lhs-sub-map*:

$[x \dots y] = \text{map } ((+) x) [0 \dots y - x]$

by (*induct y*) (*auto simp: Suc-diff-le*)

lemma *upto-0-to-4*:

$[0 \dots 4] = 0 \# [1 \dots 4]$

by (*subst upt-rec*) *simp*

lemma *disjEI*:

$\llbracket P \vee Q; P \Longrightarrow R; Q \Longrightarrow S \rrbracket$

$\Longrightarrow R \vee S$

by *fastforce*

lemma *dom-fun-upd2*:

$s\ x = \text{Some } z \Longrightarrow \text{dom } (s\ (x \mapsto y)) = \text{dom } s$

by (*simp add: insert-absorb domI*)

lemma *foldl-True* :

$\text{foldl } (\vee) \text{ True } bs$

by (*induct bs*) *auto*

lemma *image-set-comp*:

$f\ ' \{g\ x \mid x. Q\ x\} = (f \circ g)\ ' \{x. Q\ x\}$

by *fastforce*

lemma *mutual-exE*:

$\llbracket \exists x. P\ x; \bigwedge x. P\ x \Longrightarrow Q\ x \rrbracket \Longrightarrow \exists x. Q\ x$

by *blast*

lemma *nat-diff-eq*:

fixes $x :: \text{nat}$

shows $\llbracket x - y = x - z; y < x \rrbracket \Longrightarrow y = z$

by *arith*

lemma *comp-upd-simp*:

$(f \circ (g (x := y))) = ((f \circ g) (x := f y))$
by (*rule fun-upd-comp*)

lemma *dom-option-map*:

$\text{dom} (\text{map-option } f \circ m) = \text{dom } m$
by (*rule dom-map-option-comp*)

lemma *drop-imp*:

$P \implies (A \longrightarrow P) \wedge (B \longrightarrow P)$ **by** *blast*

lemma *inj-on-fun-updI2*:

$\llbracket \text{inj-on } f \ A; y \notin f' (A - \{x\}) \rrbracket \implies \text{inj-on } (f(x := y)) \ A$
by (*rule inj-on-fun-upd-strongerI*)

lemma *inj-on-fun-upd-elsewhere*:

$x \notin S \implies \text{inj-on } (f (x := y)) \ S = \text{inj-on } f \ S$
by (*simp add: inj-on-def*) *blast*

lemma *not-Some-eq-tuple*:

$(\forall y \ z. x \neq \text{Some } (y, z)) = (x = \text{None})$
by (*cases x, simp-all*)

lemma *ran-option-map*:

$\text{ran} (\text{map-option } f \circ m) = f' \ \text{ran } m$
by (*auto simp add: ran-def*)

lemma *All-less-Ball*:

$(\forall x < n. P \ x) = (\forall x \in \{.. < n\}. P \ x)$
by *fastforce*

lemma *Int-image-empty*:

$\llbracket \bigwedge x \ y. f \ x \neq g \ y \rrbracket$
 $\implies f' \ S \cap g' \ T = \{\}$
by *auto*

lemma *Max-prop*:

$\llbracket \text{Max } S \in S \implies P (\text{Max } S); (S :: ('a :: \{\text{finite}, \text{linorder}\}) \text{ set}) \neq \{\} \rrbracket \implies P$
 $(\text{Max } S)$
by *auto*

lemma *Min-prop*:

$\llbracket \text{Min } S \in S \implies P (\text{Min } S); (S :: ('a :: \{\text{finite}, \text{linorder}\}) \text{ set}) \neq \{\} \rrbracket \implies P$
 $(\text{Min } S)$
by *auto*

lemma *findSomeD*:

$\text{find } P \ xs = \text{Some } x \implies P \ x \wedge x \in \text{set } xs$

by (*induct xs*) (*auto split: if-split-asm*)

lemma *findNoneD*:

find P xs = None $\implies \forall x \in \text{set } xs. \neg P x$

by (*induct xs*) (*auto split: if-split-asm*)

lemma *dom-upd*:

dom ($\lambda x. \text{if } x = y \text{ then None else } f x$) = dom f - {y}

by (*rule set-eqI*) (*auto split: if-split-asm*)

definition

is-inv :: ('a \rightarrow 'b) \Rightarrow ('b \rightarrow 'a) \Rightarrow bool **where**

is-inv f g $\equiv \text{ran } f = \text{dom } g \wedge (\forall x y. f x = \text{Some } y \longrightarrow g y = \text{Some } x)$

lemma *is-inv-NoneD*:

assumes *g x = None*

assumes *is-inv f g*

shows *x \notin ran f*

proof –

from *assms*

have *x \notin dom g* **by** (*auto simp: ran-def*)

moreover

from *assms*

have *ran f = dom g*

by (*simp add: is-inv-def*)

ultimately

show *?thesis* **by** *simp*

qed

lemma *is-inv-SomeD*:

[f x = Some y; is-inv f g] $\implies g y = \text{Some } x$

by (*simp add: is-inv-def*)

lemma *is-inv-com*:

is-inv f g $\implies is-inv g f$

apply (*unfold is-inv-def*)

apply *safe*

apply (*clarsimp simp: ran-def dom-def set-eq-iff*)

apply (*rename-tac a*)

apply (*erule-tac x=a in allE*)

apply *clarsimp*

apply (*clarsimp simp: ran-def dom-def set-eq-iff*)

apply *blast*

apply (*clarsimp simp: ran-def dom-def set-eq-iff*)

apply (*rename-tac x y*)

apply (*erule-tac x=x in allE*)

apply *clarsimp*

done

lemma *is-inv-inj*:
 $is\text{-}inv\ f\ g \implies inj\text{-}on\ f\ (dom\ f)$
apply (*frule is-inv-com*)
apply (*clarsimp simp: inj-on-def*)
apply (*drule (1) is-inv-SomeD*)
apply (*auto dest: is-inv-SomeD*)
done

lemma *ran-upd'*:
 $\llbracket inj\text{-}on\ f\ (dom\ f); f\ y = Some\ z \rrbracket \implies ran\ (f\ (y := None)) = ran\ f - \{z\}$
by (*force simp: ran-def inj-on-def dom-def intro!: set-eqI*)

lemma *is-inv-None-upd*:
 $\llbracket is\text{-}inv\ f\ g; g\ x = Some\ y \rrbracket \implies is\text{-}inv\ (f\ (y := None))\ (g\ (x := None))$
apply (*subst is-inv-def*)
apply (*clarsimp simp: dom-upd*)
apply (*drule is-inv-SomeD, erule is-inv-com*)
apply (*frule is-inv-inj*)
apply (*auto simp: ran-upd' is-inv-def dest: is-inv-SomeD is-inv-inj*)
done

lemma *is-inv-inj2*:
 $is\text{-}inv\ f\ g \implies inj\text{-}on\ g\ (dom\ g)$
using *is-inv-com is-inv-inj by blast*

Map inversion (implicitly assuming injectivity).

definition
 $the\text{-}inv\text{-}map\ m = (\lambda s. if\ s \in ran\ m\ then\ Some\ (THE\ x. m\ x = Some\ s)\ else\ None)$

Map inversion can be expressed by function inversion.

lemma *the-inv-map-def2*:
 $the\text{-}inv\text{-}map\ m = (Some\ \circ\ the\text{-}inv\text{-}into\ (dom\ m)\ (the\ \circ\ m))\ |'\ (ran\ m)$
apply (*rule ext*)
apply (*clarsimp simp: the-inv-map-def the-inv-into-def dom-def*)
apply (*rule arg-cong[where f=The]*)
apply (*rule ext*)
apply *auto*
done

The domain of a function composition with Some is the universal set.

lemma *dom-comp-Some[simp]*: $dom\ (comp\ Some\ f) = UNIV$ **by** (*simp add: dom-def*)

Assuming injectivity, map inversion produces an inversive map.

lemma *is-inv-the-inv-map*:
 $inj\text{-}on\ m\ (dom\ m) \implies is\text{-}inv\ m\ (the\text{-}inv\text{-}map\ m)$
apply (*simp add: is-inv-def*)
apply (*intro conjI allI impI*)
apply (*simp add: the-inv-map-def2*)

apply (*auto simp add: the-inv-map-def inj-on-def dom-def intro: ranI*)
done

lemma *the-the-inv-mapI*:

inj-on m (dom m) \implies m x = Some y \implies the (the-inv-map m y) = x
by (*auto simp: the-inv-map-def ran-def inj-on-def dom-def*)

lemma *eq-restrict-map-None*:

restrict-map m A x = None \longleftrightarrow x \sim : (A \cap dom m)
by (*auto simp: restrict-map-def split: if-split-asm*)

lemma *eq-the-inv-map-None[simp]*: *the-inv-map m x = None \longleftrightarrow x \notin ran m*

by (*simp add: the-inv-map-def2 eq-restrict-map-None*)

lemma *is-inv-unique*:

is-inv f g \implies is-inv f h \implies g=h

apply (*rule ext*)

subgoal for *x*

apply (*clarsimp simp: is-inv-def dom-def Collect-eq ran-def*)

apply (*drule-tac x=x in spec*)**+**

apply (*cases g x, clarsimp+*)

done

done

lemma *range-convergence1*:

$\llbracket \forall z. x < z \wedge z \leq y \longrightarrow P z; \forall z > y. P (z :: 'a :: linorder) \rrbracket \implies \forall z > x. P z$
using *not-le by blast*

lemma *range-convergence2*:

$\llbracket \forall z. x < z \wedge z \leq y \longrightarrow P z; \forall z. z > y \wedge z < w \longrightarrow P (z :: 'a :: linorder) \rrbracket$
 $\implies \forall z. z > x \wedge z < w \longrightarrow P z$

using *range-convergence1[where P= $\lambda z. z < w \longrightarrow P z$ and $x=x$ and $y=y$]*

by *auto*

lemma *zip-upt-Cons*:

*a < b \implies zip [a..**b**] (x # xs) = (a, x) # zip [Suc a..**b**] xs*

by (*simp add: upt-conv-Cons*)

lemma *map-comp-eq*:

f \circ_m g = case-option None f \circ g

apply (*rule ext*)

subgoal for *x by (cases g x, auto)*

done

lemma *dom-If-Some*:

dom ($\lambda x. \text{if } x \in S \text{ then Some } v \text{ else } f x) = (S \cup \text{dom } f)$

by (*auto split: if-split*)

lemma *foldl-fun-upd-const*:

foldl $(\lambda s x. s(f x := v)) s xs$
 $= (\lambda x. \text{if } x \in f \text{ ' set } xs \text{ then } v \text{ else } s x)$
by $(\text{induct } xs \text{ arbitrary: } s) \text{ auto}$

lemma foldl-id:
foldl $(\lambda s x. s) s xs = s$
by $(\text{induct } xs) \text{ auto}$

lemma SucSucMinus: $2 \leq n \implies \text{Suc } (\text{Suc } (n - 2)) = n$ **by arith**

lemma ball-to-all:
 $(\bigwedge x. (x \in A) = (P x)) \implies (\forall x \in A. B x) = (\forall x. P x \longrightarrow B x)$
by blast

lemma case-option-If:
case-option $P (\lambda x. Q) v = (\text{if } v = \text{None} \text{ then } P \text{ else } Q)$
by clarsimp

lemma case-option-If2:
case-option $P Q v = \text{If } (v \neq \text{None}) (Q \text{ (the } v)) P$
by (simp split: option.split)

lemma if3-fold:
 $(\text{if } P \text{ then } x \text{ else if } Q \text{ then } y \text{ else } x) = (\text{if } P \vee \neg Q \text{ then } x \text{ else } y)$
by simp

lemma rtrancl-insert:
assumes $x\text{-new: } \bigwedge y. (x,y) \notin R$
shows $R^* \text{ `` insert } x S = \text{insert } x (R^* \text{ `` } S)$
proof $-$
have $R^* \text{ `` insert } x S = R^* \text{ `` } (\{x\} \cup S)$ **by simp**
also
have $R^* \text{ `` } (\{x\} \cup S) = R^* \text{ `` } \{x\} \cup R^* \text{ `` } S$
by (subst Image-Un) simp
also
have $R^* \text{ `` } \{x\} = \{x\}$
by (meson Image-closed-trancl Image-singleton-iff subsetI x-new)
finally
show $?thesis$ **by simp**
qed

lemma ran-del-subset:
 $y \in \text{ran } (f (x := \text{None})) \implies y \in \text{ran } f$
by (auto simp: ran-def split: if-split-asm)

lemma trancl-sub-lift:
assumes $\text{sub: } \bigwedge p p'. (p,p') \in r \implies (p,p') \in r'$
shows $(p,p') \in r^+ \implies (p,p') \in r'^+$
by (fastforce intro: trancl-mono sub)

lemma *trancl-step-lift*:

assumes *x-step*: $\bigwedge p p'. (p,p') \in r' \implies (p,p') \in r \vee (p = x \wedge p' = y)$
assumes *y-new*: $\bigwedge p'. \neg(y,p') \in r$
shows $(p,p') \in r'^{\wedge+} \implies (p,p') \in r^{\wedge+} \vee ((p,x) \in r^{\wedge+} \wedge p' = y) \vee (p = x \wedge p' = y)$
apply (*erule trancl-induct*)
apply (*drule x-step*)
apply *fastforce*
apply (*erule disjE*)
apply (*drule x-step*)
apply (*erule disjE*)
apply (*drule trancl-trans, drule r-into-trancl, assumption*)
apply *blast*
apply *fastforce*
apply (*fastforce simp: y-new dest: x-step*)
done

lemma *rtrancl-simulate-weak*:

assumes *r*: $(x,z) \in R^*$
assumes *s*: $\bigwedge y. (x,y) \in R \implies (y,z) \in R^* \implies (x,y) \in R' \wedge (y,z) \in R'^*$
shows $(x,z) \in R'^*$
apply (*rule converse-rtranclE[OF r]*)
apply *simp*
apply (*frule (1) s*)
apply *clarsimp*
by (*rule converse-rtrancl-into-rtrancl*)

lemma *list-case-If2*:

case-list f g xs = If (xs = []) f (g (hd xs) (tl xs))
by (*simp split: list.split*)

lemma *length-ineq-not-Nil*:

$length\ xs > n \implies xs \neq []$
 $length\ xs \geq n \implies n \neq 0 \longrightarrow xs \neq []$
 $\neg length\ xs < n \implies n \neq 0 \longrightarrow xs \neq []$
 $\neg length\ xs \leq n \implies xs \neq []$
by *auto*

lemma *numeral-eqs*:

$2 = Suc\ (Suc\ 0)$
 $3 = Suc\ (Suc\ (Suc\ 0))$
 $4 = Suc\ (Suc\ (Suc\ (Suc\ 0)))$
 $5 = Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))$
 $6 = Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))))$
by *simp+*

lemma *psubset-singleton*:

$(S \subset \{x\}) = (S = \{x\})$

by *blast*

lemma *length-takeWhile-ge*:

$length\ (takeWhile\ f\ xs) = n \implies length\ xs = n \vee (length\ xs > n \wedge \neg f\ (xs\ !\ n))$
by (*induct xs arbitrary: n*) (*auto split: if-split-asm*)

lemma *length-takeWhile-le*:

$\neg f\ (xs\ !\ n) \implies length\ (takeWhile\ f\ xs) \leq n$
apply (*induct xs arbitrary: n; simp*)
subgoal for $x\ xs\ n$ **by** (*cases n; simp*)
done

lemma *length-takeWhile-gt*:

$n < length\ (takeWhile\ f\ xs)$
 $\implies (\exists\ ys\ zs.\ length\ ys = Suc\ n \wedge xs = ys\ @\ zs \wedge takeWhile\ f\ xs = ys\ @\ takeWhile\ f\ zs)$
apply (*induct xs arbitrary: n; simp split: if-split-asm*)
subgoal for $a\ xs\ n$
apply (*cases n; simp*)
apply (*rule exI[where x=[a]]*)
apply *simp*
apply (*erule meta-allE, drule(1) meta-mp*)
apply *clarsimp*
subgoal for $- ys\ zs$
apply (*rule exI[where x=a # ys]*)
apply *simp*
done
done
done

lemma *hd-drop-conv-nth2*:

$n < length\ xs \implies hd\ (drop\ n\ xs) = xs\ !\ n$
by (*rule hd-drop-conv-nth*) *clarsimp*

lemma *map-upt-eq-vals-D*:

$\llbracket map\ f\ [0\ ..<\ n] = ys; m < length\ ys \rrbracket \implies f\ m = ys\ !\ m$
by *clarsimp*

lemma *length-le-helper*:

$\llbracket n \leq length\ xs; n \neq 0 \rrbracket \implies xs \neq [] \wedge n - 1 \leq length\ (tl\ xs)$
by (*cases xs, simp-all*)

lemma *all-ex-eq-helper*:

$(\forall v.\ (\exists v'. v = f\ v' \wedge P\ v\ v') \longrightarrow Q\ v)$
 $= (\forall v'. P\ (f\ v')\ v' \longrightarrow Q\ (f\ v'))$
by *auto*

lemma *nat-less-cases'*:

$(x::nat) < y \implies x = y - 1 \vee x < y - 1$

by *auto*

lemma *less-numeral-nat-iff-disj*:

$(n::\text{nat}) < \text{numeral } m \iff n = \text{numeral } m - 1 \vee n < \text{numeral } m - 1$

apply *clarsimp*

using *less-SucE numeral-eq-Suc* **by** *presburger*

lemma *filter-to-shorter-upto*:

$n \leq m \implies \text{filter } (\lambda x. x < n) [0 ..< m] = [0 ..< n]$

by (*induct m*) (*auto elim: le-SucE*)

lemma *in-emptyE*: $\llbracket A = \{\}; x \in A \rrbracket \implies P$ **by** *blast*

lemma *Ball-emptyI*:

$S = \{\} \implies (\forall x \in S. P x)$

by *simp*

lemma *allfEI*:

$\llbracket \forall x. P x; \bigwedge x. P (f x) \rrbracket \implies \forall x. Q x$

by *fastforce*

lemma *cart-singleton-empty2*:

$(\{x\} \times S = \{\}) = (S = \{\})$

$(\{\} = S \times \{e\}) = (S = \{\})$

by *auto*

lemma *cases-simp-conj*:

$((P \longrightarrow Q) \wedge (\neg P \longrightarrow Q) \wedge R) = (Q \wedge R)$

by *fastforce*

lemma *domE* :

$\llbracket x \in \text{dom } m; \bigwedge r. \llbracket m x = \text{Some } r \rrbracket \implies P \rrbracket \implies P$

by *clarsimp*

lemma *dom-eqD*:

$\llbracket f x = \text{Some } v; \text{dom } f = S \rrbracket \implies x \in S$

by *clarsimp*

lemma *exception-set-finite1*:

$\text{finite } \{x. P x\} \implies \text{finite } \{x. (x = y \longrightarrow Q x) \wedge P x\}$

by (*simp add: Collect-conj-eq*)

lemma *exception-set-finite2*:

$\text{finite } \{x. P x\} \implies \text{finite } \{x. x \neq y \longrightarrow P x\}$

by (*simp add: imp-conv-disj*)

lemmas *exception-set-finite = exception-set-finite1 exception-set-finite2*

lemma *exfEI*:

$\llbracket \exists x. P x; \bigwedge x. P x \implies Q (f x) \rrbracket \implies \exists x. Q x$
by *fastforce*

lemma *Collect-int-vars*:

$\{s. P \text{ rv } s\} \cap \{s. \text{rv} = x f s\} = \{s. P (x f s) s\} \cap \{s. \text{rv} = x f s\}$
by *auto*

lemma *if-0-1-eq*:

$((\text{if } P \text{ then } 1 \text{ else } 0) = (\text{case } Q \text{ of } \text{True} \Rightarrow \text{of-nat } 1 \mid \text{False} \Rightarrow \text{of-nat } 0)) = (P = Q)$
by (*simp split: if-split bool.split*)

lemma *modify-map-exists-cte* :

$(\exists \text{cte. modify-map } m \text{ } p \text{ } f \text{ } p' = \text{Some cte}) = (\exists \text{cte. } m \text{ } p' = \text{Some cte})$
by (*simp add: modify-map-def*)

lemma *dom-eqI*:

assumes *c1*: $\bigwedge x y. P x = \text{Some } y \implies \exists y. Q x = \text{Some } y$
and *c2*: $\bigwedge x y. Q x = \text{Some } y \implies \exists y. P x = \text{Some } y$
shows $\text{dom } P = \text{dom } Q$
unfolding *dom-def* **by** (*auto simp: c1 c2*)

lemma *dvd-reduce-multiple*:

fixes *k* :: *nat*
shows $(k \text{ dvd } k * m + n) = (k \text{ dvd } n)$
by (*induct m*) (*auto simp: add-ac*)

lemma *image-iff2*:

$\text{inj } f \implies f x \in f ' S = (x \in S)$
by (*rule inj-image-mem-iff*)

lemma *map-comp-restrict-map-Some-iff*:

$((g \circ_m (m \mid ' S)) x = \text{Some } y) = ((g \circ_m m) x = \text{Some } y \wedge x \in S)$
by (*auto simp add: map-comp-Some-iff restrict-map-Some-iff*)

lemma *range-subsetD*:

fixes *a* :: '*a* :: *order*
shows $\llbracket \{a..b\} \subseteq \{c..d\}; a \leq b \rrbracket \implies c \leq a \wedge b \leq d$
by *simp*

lemma *case-option-dom*:

$(\text{case } f x \text{ of } \text{None} \Rightarrow a \mid \text{Some } v \Rightarrow b v) = (\text{if } x \in \text{dom } f \text{ then } b (\text{the } (f x)) \text{ else } a)$
by (*auto split: option.split*)

lemma *contrapos-imp*:

$P \longrightarrow Q \implies \neg Q \longrightarrow \neg P$
by *clarsimp*

lemma *filter-eq-If*:

distinct xs \implies *filter* ($\lambda v. v = x$) *xs* = (if $x \in$ set *xs* then $[x]$ else $[\]$)
by (*induct xs*) *auto*

lemma (*in semigroup-add*) *foldl-assoc*:
shows *foldl* (+) ($x+y$) *zs* = $x +$ (*foldl* (+) *y zs*)
by (*induct zs arbitrary: y*) (*simp-all add:add.assoc*)

lemma (*in monoid-add*) *foldl-absorb0*:
shows $x +$ (*foldl* (+) $0 zs$) = *foldl* (+) $x zs$
by (*induct zs*) (*simp-all add:foldl-assoc*)

lemma *foldl-conv-concat*:
foldl (@) *xs xss* = x @ *concat xss*
proof (*induct xss arbitrary: xs*)
case Nil show ?case by simp
next
interpret monoid-add (@) [] proof qed simp-all
case Cons then show ?case by (simp add: foldl-absorb0)
qed

lemma *foldl-concat-concat*:
foldl (@) [] (x @ *ys*) = *foldl* (@) [] x @ *foldl* (@) [] *ys*
by (*simp add: foldl-conv-concat*)

lemma *foldl-does-nothing*:
 $[\ \wedge x. x \in$ set *xs* $\implies f s x = s] \implies$ *foldl* $f s xs = s$
by (*induct xs*) *auto*

lemma *foldl-use-filter*:
 $[\ \wedge v x. [\ \neg g x; x \in$ set *xs*] $\implies f v x = v] \implies$ *foldl* $f v xs =$ *foldl* $f v$ (*filter g xs*)
by (*induct xs arbitrary: v*) *auto*

lemma *map-comp-update-lift*:
assumes *fv: f v = Some v'*
shows ($f \circ_m$ ($g(ptr \mapsto v)$)) = ($(f \circ_m g)(ptr \mapsto v')$)
by (*simp add: fv map-comp-update*)

lemma *restrict-map-cong*:
assumes *sv: S = S'*
and *rl: $\wedge p. p \in S' \implies mp p = mp' p$*
shows $mp \upharpoonright S = mp' \upharpoonright S'$
using *expand-restrict-map-eq rl sv by auto*

lemma *case-option-over-if*:
case-option P Q (if *G* then *None* else *Some v*)
= (if *G* then *P* else *Q v*)
case-option P Q (if *G* then *Some v* else *None*)
= (if *G* then *Q v* else *P*)

by (*simp split: if-split*)⁺

lemma *map-length-cong*:

$\llbracket \text{length } xs = \text{length } ys; \bigwedge x y. (x, y) \in \text{set } (\text{zip } xs \text{ } ys) \implies f x = g y \rrbracket$
 $\implies \text{map } f \text{ } xs = \text{map } g \text{ } ys$

apply *atomize*

apply (*erule rev-mp, erule list-induct2*)

apply *auto*

done

lemma *take-min-len*:

$\text{take } (\text{min } (\text{length } xs) \text{ } n) \text{ } xs = \text{take } n \text{ } xs$

by (*simp add: min-def*)

lemmas *interval-empty = atLeastatMost-empty-iff*

lemma *fold-and-false*[*simp*]:

$\neg(\text{fold } (\wedge) \text{ } xs \text{ } \text{False})$

apply *clarsimp*

apply (*induct xs*)

apply *simp*

apply *simp*

done

lemma *fold-and-true*:

$\text{fold } (\wedge) \text{ } xs \text{ } \text{True} \implies \forall i < \text{length } xs. xs ! i$

apply *clarsimp*

apply (*induct xs*)

apply *simp*

subgoal for *a xs i*

by (*cases i = 0; cases a; auto*)

done

lemma *fold-or-true*[*simp*]:

$\text{fold } (\vee) \text{ } xs \text{ } \text{True}$

by (*induct xs, simp+*)

lemma *fold-or-false*:

$\neg(\text{fold } (\vee) \text{ } xs \text{ } \text{False}) \implies \forall i < \text{length } xs. \neg(xs ! i)$

apply (*induct xs, simp+*)

subgoal for *a xs*

apply (*cases a, simp+*)

apply (*rule allI*)

subgoal for *i* **by** (*cases i = 0, simp+*)

done

done

2.1 Take, drop, zip, *list-all* etc rules

method *two-induct* for $xs\ ys =$
 $((\text{induct } xs \text{ arbitrary: } ys; \text{ simp?}), (\text{case-tac } ys; \text{ simp})?)$

lemma *map-fst-zip-prefix*:
 $\text{map fst } (\text{zip } xs\ ys) \leq xs$
by (*two-induct* $xs\ ys$)

lemma *map-snd-zip-prefix*:
 $\text{map snd } (\text{zip } xs\ ys) \leq ys$
by (*two-induct* $xs\ ys$)

lemma *nth-upt-0* [*simp*]:
 $i < \text{length } xs \implies [0..<\text{length } xs] ! i = i$
by *simp*

lemma *take-insert-nth*:
 $i < \text{length } xs \implies \text{insert } (xs ! i) (\text{set } (\text{take } i\ xs)) = \text{set } (\text{take } (\text{Suc } i)\ xs)$
by (*subst take-Suc-conv-app-nth*, *assumption*, *fastforce*)

lemma *zip-take-drop*:
 $\llbracket n < \text{length } xs; \text{length } ys = \text{length } xs \rrbracket \implies$
 $\text{zip } xs (\text{take } n\ ys @ a \# \text{drop } (\text{Suc } n)\ ys) =$
 $\text{zip } (\text{take } n\ xs) (\text{take } n\ ys) @ (xs ! n, a) \# \text{zip } (\text{drop } (\text{Suc } n)\ xs) (\text{drop } (\text{Suc } n)$
 $ys)$
by (*subst id-take-nth-drop*, *assumption*, *simp*)

lemma *take-nth-distinct*:
 $\llbracket \text{distinct } xs; n < \text{length } xs; xs ! n \in \text{set } (\text{take } n\ xs) \rrbracket \implies \text{False}$
by (*fastforce simp: distinct-conv-nth in-set-conv-nth*)

lemma *take-drop-append*:
 $\text{drop } a\ xs = \text{take } b (\text{drop } a\ xs) @ \text{drop } (a + b)\ xs$
by (*metis append-take-drop-id drop-drop add commute*)

lemma *drop-take-drop*:
 $\text{drop } a (\text{take } (b + a)\ xs) @ \text{drop } (b + a)\ xs = \text{drop } a\ xs$
by (*metis add commute take-drop take-drop-append*)

lemma *not-prefixI*:
 $\llbracket xs \neq ys; \text{length } xs = \text{length } ys \rrbracket \implies \neg xs \leq ys$
by (*auto elim: prefixE*)

lemma *map-fst-zip'*:
 $\text{length } xs \leq \text{length } ys \implies \text{map fst } (\text{zip } xs\ ys) = xs$
by (*metis length-map length-zip map-fst-zip-prefix min-absorb1 not-prefixI*)

lemma *zip-take-triv*:

$n \geq \text{length } bs \implies \text{zip } (\text{take } n \text{ as}) \text{ bs} = \text{zip as bs}$
apply (*induct bs arbitrary: n as; simp*)
subgoal for a bs n as by (*cases n; cases as; simp*)
done

lemma zip-take-triv2:
 $\text{length as} \leq n \implies \text{zip as } (\text{take } n \text{ bs}) = \text{zip as bs}$
apply (*induct as arbitrary: n bs; simp*)
subgoal for a bs n as by (*cases n; cases as; simp*)
done

lemma zip-take-length:
 $\text{zip xs } (\text{take } (\text{length xs}) \text{ ys}) = \text{zip xs ys}$
by (*metis order-refl zip-take-triv2*)

lemma zip-singleton:
 $ys \neq [] \implies \text{zip [a] ys} = [(a, \text{ys} ! 0)]$
by (*cases ys, simp-all*)

lemma zip-append-singleton:
 $[[i = \text{length xs}; \text{length xs} < \text{length ys}] \implies \text{zip } (xs @ [a]) \text{ ys} = (\text{zip xs ys}) @ [(a, \text{ys} ! i)]$
by (*induct xs; cases ys; simp*)
(clarsimp simp: zip-append1 zip-take-length zip-singleton)

lemma ranE:
 $[[v \in \text{ran } f; \bigwedge x. f x = \text{Some } v \implies R] \implies R]$
by (*auto simp: ran-def*)

lemma ran-map-option-restrict-eq:
 $[[x \in \text{ran } (\text{map-option } f \circ g); x \notin \text{ran } (\text{map-option } f \circ (g |' (- \{y\})))] \implies \exists v. g y = \text{Some } v \wedge f v = x]$
apply (*clarsimp simp: elim!: ranE*)
subgoal for w z
apply (*cases w = y*)
apply *clarsimp*
apply (*erule notE, rule ranI[where a=w]*)
apply (*simp add: restrict-map-def*)
done
done

lemma map-of-zip-range:
 $[[\text{length xs} = \text{length ys}; \text{distinct xs}] \implies (\lambda x. (\text{the } (\text{map-of } (\text{zip xs ys}) x))) \text{' set xs} = \text{set ys}]$
apply (*clarsimp simp: image-def*)
apply (*subst ran-map-of-zip [symmetric, where xs=xs and ys=ys]; simp?*)
apply (*clarsimp simp: ran-def*)
apply (*rule equalityI*)
apply *clarsimp*

apply (*rename-tac x*)
apply (*frule-tac x=x in map-of-zip-is-Some; fastforce*)
apply (*clarsimp simp: set-zip*)
by (*metis domI dom-map-of-zip nth-mem ranE ran-map-of-zip option.sel*)

lemma *map-zip-fst*:
 $length\ xs = length\ ys \implies map\ (\lambda(x, y). f\ x)\ (zip\ xs\ ys) = map\ f\ xs$
by (*two-induct xs ys*)

lemma *map-zip-fst'*:
 $length\ xs \leq length\ ys \implies map\ (\lambda(x, y). f\ x)\ (zip\ xs\ ys) = map\ f\ xs$
by (*metis length-map map-fst-zip' map-zip-fst zip-map-fst-snd*)

lemma *map-zip-snd*:
 $length\ xs = length\ ys \implies map\ (\lambda(x, y). f\ y)\ (zip\ xs\ ys) = map\ f\ ys$
by (*two-induct xs ys*)

lemma *map-zip-snd'*:
 $length\ ys \leq length\ xs \implies map\ (\lambda(x, y). f\ y)\ (zip\ xs\ ys) = map\ f\ ys$
by (*two-induct xs ys*)

lemma *map-of-zip-tuple-in*:
 $\llbracket (x, y) \in set\ (zip\ xs\ ys); distinct\ xs \rrbracket \implies map-of\ (zip\ xs\ ys)\ x = Some\ y$
by (*two-induct xs ys*) (*auto intro: in-set-zipE*)

lemma *in-set-zip1*:
 $(x, y) \in set\ (zip\ xs\ ys) \implies x \in set\ xs$
by (*erule in-set-zipE*)

lemma *in-set-zip2*:
 $(x, y) \in set\ (zip\ xs\ ys) \implies y \in set\ ys$
by (*erule in-set-zipE*)

lemma *map-zip-snd-take*:
 $map\ (\lambda(x, y). f\ y)\ (zip\ xs\ ys) = map\ f\ (take\ (length\ xs)\ ys)$
apply (*subst map-zip-snd' [symmetric, where xs=xs and ys=take (length xs) ys], simp*)
apply (*subst zip-take-length [symmetric], simp*)
done

lemma *map-of-zip-is-index*:
 $\llbracket length\ xs = length\ ys; x \in set\ xs \rrbracket \implies \exists i. (map-of\ (zip\ xs\ ys))\ x = Some\ (ys\ !\ i)$
apply (*induct rule: list-induct2; simp*)
apply (*rule conjI; clarsimp*)
apply (*metis nth-Cons-0*)
apply (*metis nth-Cons-Suc*)
done

lemma *map-of-zip-take-update*:

$\llbracket i < \text{length } xs; \text{length } xs \leq \text{length } ys; \text{distinct } xs \rrbracket$
 $\implies (\text{map-of } (\text{zip } (\text{take } i \text{ } xs) \text{ } ys))(xs ! i \mapsto (ys ! i)) = \text{map-of } (\text{zip } (\text{take } (\text{Suc } i) \text{ } xs) \text{ } ys)$
apply (*rule ext*)
subgoal for *x*
apply (*cases x=xs ! i; clarsimp*)
apply (*rule map-of-is-SomeI[symmetric]*)
apply (*simp add: map-fst-zip'*)
apply (*force simp add: set-zip*)
apply (*clarsimp simp: take-Suc-conv-app-nth zip-append-singleton map-add-def*
split: option.splits)
done
done

lemma *map-of-zip-is-Some'*:

$\text{length } xs \leq \text{length } ys \implies (x \in \text{set } xs) = (\exists y. \text{map-of } (\text{zip } xs \text{ } ys) \text{ } x = \text{Some } y)$
apply (*subst zip-take-length[symmetric]*)
apply (*rule map-of-zip-is-Some*)
by (*metis length-take min-absorb2*)

lemma *map-of-zip-inj*:

$\llbracket \text{distinct } xs; \text{distinct } ys; \text{length } xs = \text{length } ys \rrbracket$
 $\implies \text{inj-on } (\lambda x. (\text{the } (\text{map-of } (\text{zip } xs \text{ } ys) \text{ } x))) (\text{set } xs)$
apply (*clarsimp simp: inj-on-def*)
apply (*subst (asm) map-of-zip-is-Some, assumption*)
apply *clarsimp*
apply (*clarsimp simp: set-zip*)
by (*metis nth-eq-iff-index-eq*)

lemma *map-of-zip-inj'*:

$\llbracket \text{distinct } xs; \text{distinct } ys; \text{length } xs \leq \text{length } ys \rrbracket$
 $\implies \text{inj-on } (\lambda x. (\text{the } (\text{map-of } (\text{zip } xs \text{ } ys) \text{ } x))) (\text{set } xs)$
apply (*subst zip-take-length[symmetric]*)
apply (*erule map-of-zip-inj, simp*)
by (*metis length-take min-absorb2*)

lemma *list-all-nth*:

$\llbracket \text{list-all } P \text{ } xs; i < \text{length } xs \rrbracket \implies P (xs ! i)$
by (*metis list-all-length*)

lemma *list-all-update*:

$\llbracket \text{list-all } P \text{ } xs; i < \text{length } xs; \bigwedge x. P \text{ } x \implies P (f \text{ } x) \rrbracket$
 $\implies \text{list-all } P \text{ } (xs [i := f (xs ! i)])$
by (*metis length-list-update list-all-length nth-list-update*)

lemma *list-allI*:

$\llbracket \text{list-all } P \text{ } xs; \bigwedge x. P \text{ } x \implies P' \text{ } x \rrbracket \implies \text{list-all } P' \text{ } xs$

by (*metis list-all-length*)

lemma *list-all-imp-filter*:

$list\text{-}all (\lambda x. f\ x \longrightarrow g\ x)\ xs = list\text{-}all (\lambda x. g\ x)\ [x \leftarrow xs . f\ x]$

by (*fastforce simp: Ball-set-list-all[symmetric]*)

lemma *list-all-imp-filter2*:

$list\text{-}all (\lambda x. f\ x \longrightarrow g\ x)\ xs = list\text{-}all (\lambda x. \neg f\ x)\ [x \leftarrow xs . (\lambda x. \neg g\ x)\ x]$

by (*fastforce simp: Ball-set-list-all[symmetric]*)

lemma *list-all-imp-chain*:

$\llbracket list\text{-}all (\lambda x. f\ x \longrightarrow g\ x)\ xs; list\text{-}all (\lambda x. f'\ x \longrightarrow f\ x)\ xs \rrbracket$

$\implies list\text{-}all (\lambda x. f'\ x \longrightarrow g\ x)\ xs$

by (*clarsimp simp: Ball-set-list-all [symmetric]*)

lemma *inj-Pair*:

$inj\text{-}on\ (Pair\ x)\ S$

by (*rule inj-onI, simp*)

lemma *inj-on-split*:

$inj\text{-}on\ f\ S \implies inj\text{-}on\ (\lambda x. (z, f\ x))\ S$

by (*auto simp: inj-on-def*)

lemma *split-state-strg*:

$(\exists x. f\ s = x \wedge P\ x\ s) \longrightarrow P\ (f\ s)\ s$ **by** *clarsimp*

lemma *theD*:

$\llbracket the\ (f\ x) = y; x \in dom\ f \rrbracket \implies f\ x = Some\ y$

by (*auto simp add: dom-def*)

lemma *bspec-split*:

$\llbracket \forall (a, b) \in S. P\ a\ b; (a, b) \in S \rrbracket \implies P\ a\ b$

by *fastforce*

lemma *set-zip-same*:

$set\ (zip\ xs\ xs) = Id \cap (set\ xs \times set\ xs)$

by (*induct xs*) *auto*

lemma *ball-ran-updI*:

$(\forall x \in ran\ m. P\ x) \implies P\ v \implies (\forall x \in ran\ (m\ (y \mapsto v)). P\ x)$

by (*auto simp add: ran-def*)

lemma *not-psubset-eq*:

$\llbracket \neg A \subset B; A \subseteq B \rrbracket \implies A = B$

by *blast*

lemma *set-as-imp*:

$(A \cap P \cup B \cap \neg P) = \{s. (s \in P \longrightarrow s \in A) \wedge (s \notin P \longrightarrow s \in B)\}$
by *auto*

lemma *in-image-op-plus*:

$(x + y \in (+) x \text{ ' } S) = ((y :: 'a :: \text{ring}) \in S)$
by (*simp add: image-def*)

lemma *insert-subtract-new*:

$x \notin S \implies (\text{insert } x \ S - S) = \{x\}$
by *auto*

lemmas *zip-is-empty = zip-eq-Nil-iff*

lemma *minus-Suc-0-lt*:

$a \neq 0 \implies a - \text{Suc } 0 < a$
by *simp*

lemma *fst-last-zip-upt*:

$\text{zip } [0 ..< m] \ xs \neq [] \implies$
 $\text{fst } (\text{last } (\text{zip } [0 ..< m] \ xs)) = (\text{if } \text{length } \ xs < m \ \text{then } \text{length } \ xs - 1 \ \text{else } m - 1)$
apply (*subst last-conv-nth, assumption*)
apply (*simp only: One-nat-def*)
apply (*subst nth-zip; simp*)
apply (*rule order-less-le-trans[OF minus-Suc-0-lt]; simp*)
apply (*rule order-less-le-trans[OF minus-Suc-0-lt]; simp*)
done

lemma *neq-into-nprefix*:

$[x \neq \text{take } (\text{length } x) \ y] \implies \neg x \leq y$
by (*clarsimp simp: prefix-def less-eq-list-def*)

lemma *suffix-eqI*:

$[\text{suffix } \ xs \ as; \ \text{suffix } \ xs \ bs; \ \text{length } \ as = \text{length } \ bs;$
 $\text{take } (\text{length } \ as - \text{length } \ xs) \ as \leq \text{take } (\text{length } \ bs - \text{length } \ xs) \ bs] \implies as = bs$
by (*clarsimp elim!: prefixE suffixE*)

lemma *suffix-Cons-mem*:

$\text{suffix } (x \# \ xs) \ as \implies x \in \text{set } as$
by (*metis in-set-conv-decomp suffix-def*)

lemma *distinct-imply-not-in-tail*:

$[\text{distinct } list; \ \text{suffix } (y \# \ ys) \ list] \implies y \notin \text{set } ys$
by (*clarsimp simp: suffix-def*)

lemma *list-induct-suffix* [*case-names Nil Cons*]:

assumes *nilr: P []*

and $constr: \bigwedge x xs. \llbracket P xs; suffix (x \# xs) as \rrbracket \implies P (x \# xs)$
shows $P as$
proof –
define as' **where** $as' == as$

have $suffix as as'$ **unfolding** as' -*def* **by** *simp*
then show *?thesis*
proof (*induct as*)
case *Nil* **show** *?case* **by** *fact*
next
case (*Cons x xs*)

show *?case*
proof (*rule constr*)
from *Cons.prem*s **show** $suffix (x \# xs) as$ **unfolding** as' -*def* .
then have $suffix xs as'$ **by** (*auto dest: suffix-ConsD simp: as'-def*)
then show $P xs$ **using** *Cons.hyps* **by** *simp*
qed
qed
qed

Parallel etc. and lemmas for list prefix

lemma *prefix-induct* [*consumes 1, case-names Nil Cons*]:
fixes *prefix*
assumes $np: prefix \leq lst$
and $base: \bigwedge xs. P [] xs$
and $rl: \bigwedge x xs y ys. \llbracket x = y; xs \leq ys; P xs ys \rrbracket \implies P (x \# xs) (y \# ys)$
shows $P prefix lst$
using np
proof (*induct prefix arbitrary: lst*)
case *Nil* **show** *?case* **by** *fact*
next
case (*Cons x xs*)

have $prem: (x \# xs) \leq lst$ **by** *fact*
then obtain $y ys$ **where** $lv: lst = y \# ys$
by (*rule prefixE, auto*)

have $ih: \bigwedge lst. xs \leq lst \implies P xs lst$ **by** *fact*

show *?case* **using** $prem$
by (*auto simp: lv intro!: rl ih*)
qed

lemma *not-prefix-cases*:
fixes *prefix*
assumes $pf: \neg prefix \leq lst$
and $c1: \llbracket prefix \neq []; lst = [] \rrbracket \implies R$
and $c2: \bigwedge a as x xs. \llbracket prefix = a \# as; lst = x \# xs; x = a; \neg as \leq xs \rrbracket \implies R$

```

    and c3:  $\bigwedge a \text{ as } x \text{ xs. } \llbracket \text{prefix} = a\#as; \text{lst} = x\#xs; x \neq a \rrbracket \implies R$ 
  shows R
proof (cases prefix)
  case Nil then show ?thesis using pfx by simp
next
  case (Cons a as)

  have c:  $\text{prefix} = a\#as$  by fact

  show ?thesis
proof (cases lst)
  case Nil then show ?thesis
    by (intro c1, simp add: Cons)
next
  case (Cons x xs)
  show ?thesis
proof (cases x = a)
  case True
  show ?thesis
proof (intro c2)
  show  $\neg as \leq xs$  using pfx c Cons True
    by simp
  qed fact+
next
  case False
  show ?thesis by (rule c3) fact+
  qed
qed
qed

lemma not-prefix-induct [consumes 1, case-names Nil Neq Eq]:
  fixes prefix
  assumes np:  $\neg \text{prefix} \leq \text{lst}$ 
  and base:  $\bigwedge x \text{ xs. } P (x\#xs)$  []
  and r1:  $\bigwedge x \text{ xs } y \text{ ys. } x \neq y \implies P (x\#xs) (y\#ys)$ 
  and r2:  $\bigwedge x \text{ xs } y \text{ ys. } \llbracket x = y; \neg xs \leq ys; P \text{ xs } ys \rrbracket \implies P (x\#xs) (y\#ys)$ 
  shows  $P \text{ prefix } \text{lst}$ 
  using np
proof (induct lst arbitrary: prefix)
  case Nil then show ?case
    by (auto simp: neq-Nil-conv elim!: not-prefix-cases intro!: base)
next
  case (Cons y ys)

  have npfx:  $\neg \text{prefix} \leq (y \# ys)$  by fact
  then obtain x xs where pv:  $\text{prefix} = x \# xs$ 
    by (rule not-prefix-cases) auto

  have ih:  $\bigwedge \text{prefix. } \neg \text{prefix} \leq ys \implies P \text{ prefix } ys$  by fact

```

show *?case using npfx*
by (*simp only: pv*) (*erule not-prefix-cases, auto intro: r1 r2 ih*)
qed

lemma *rsubst*:
 $\llbracket P\ s; s = t \rrbracket \implies P\ t$
by *simp*

lemma *ex-impE*: $(\exists x. P\ x) \longrightarrow Q \implies P\ x \implies Q$
by *blast*

lemma *option-Some-value-independent*:
 $\llbracket f\ x = \text{Some}\ v; \bigwedge v'. f\ x = \text{Some}\ v' \rrbracket \implies f\ y = \text{Some}\ v$
by *blast*

Some int bitwise lemmas. Helpers for proofs about `NatBitwise.thy`

lemma *int-2p-eq-shiffl*:
 $(2::\text{int}) \hat{x} = 1 \lll x$
by (*simp add: shiffl-int-def*)

lemma *nat-int-mul*:
 $\text{nat}\ (\text{int}\ a * b) = a * \text{nat}\ b$
by (*simp add: nat-mult-distrib*)

lemma *int-shiffl-less-cancel*:
 $n \leq m \implies ((x :: \text{int}) \lll n < y \lll m) = (x < y \lll (m - n))$
apply (*drule le-Suc-ex*)
apply (*clarsimp simp: shiffl-int-def power-add*)
done

lemma *int-shiffl-lt-2p-bits*:
 $0 \leq (x::\text{int}) \implies x < 1 \lll n \implies \forall i \geq n. \neg x !! i$
apply (*clarsimp simp: shiffl-int-def*)
by (*metis bit-take-bit-iff not-less take-bit-int-eq-self-iff*)
— TODO: The converse should be true as well, but seems hard to prove.

lemmas *int-eq-test-bit = bin-eq-iff*
lemmas *int-eq-test-bitI = int-eq-test-bit[THEN iffD2, rule-format]*

lemma *le-nat-shrink-left*:
 $y \leq z \implies y = \text{Suc}\ x \implies x < z$
by *simp*

lemma *length-ge-split*:
 $n < \text{length}\ xs \implies \exists x\ xs'. xs = x \# xs' \wedge n \leq \text{length}\ xs'$
by (*cases xs*) *auto*

Support for defining enumerations on datatypes derived from enumerations

lemma *distinct-map-enum*:

$$\llbracket (\forall x y. (F x = F y \longrightarrow x = y)) \rrbracket$$

$$\implies \text{distinct } (\text{map } F \text{ (enum-class.enum :: 'a :: enum list)})$$
by (*simp add: distinct-map inj-onI*)

lemma *if-option-None-eq*:

$$((\text{if } P \text{ then None else Some } x) = \text{None}) = P$$
by (*auto split: if-splits*)

lemma *not-in-ran-None-upd*:

$$x \notin \text{ran } m \implies x \notin \text{ran } (m(y := \text{None}))$$
by (*auto simp: ran-def split: if-split*)

Prevent clarsimp and others from creating Some from not None by folding this and unfolding again when safe.

definition

$$\text{not-None } x = (x \neq \text{None})$$

lemma *sorted-wrt-hd-min*: $\llbracket \bigwedge x. P x x; \text{sorted-wrt } P \text{ } xs \rrbracket \implies (\forall x \in \text{set } xs. P (\text{hd } xs) x)$
by (*induction xs auto*)

lemma *disjoint-no-subset*: $A \cap B = \{\} \implies A \neq \{\} \implies A \subseteq B \implies \text{False}$
by *blast*

lemma *map-prod-split-case*: $\text{map-prod } f g x = (\text{case } x \text{ of } (a, b) \implies (f a, g b))$
by (*cases x auto*)

lemma *map-prod-split-prj*: $\text{map-prod } f g x = (f (\text{fst } x), g (\text{snd } x))$
by (*cases x auto*)

end

Chapter 3

ML Antiquotations

3.1 Building terms: *mk-term*

```
theory MkTermAntiquote  
imports  
  Pure  
begin
```

mk-term: ML antiquotation for building and splicing terms.
See `MkTermAntiquote_Tests.thy` for examples and tests.

ML-file *mkterm-antiquote.ML*

```
end
```

mk-term: ML antiquotation for building and splicing terms.

```
theory MkTermAntiquote-Tests  
imports  
  MkTermAntiquote  
  Main  
begin
```

Basic usage: `@{mk-term pattern... (vars, ...)} (args, ...)`

The vars should match schematic vars in the pattern, and they are substituted with the given arguments.

Template vars can include type vars, and they should be applied to type arguments.

```
ML-val ‹  
@{assert} (@{mk-term ?x (x)} @term x::'a)  
  = @term x::'a);  
  
@{assert} (@{mk-term 0::'a::zero ('a)} @typ nat)  
  = @term 0::nat);
```

(* Tuples are used to pass multiple arguments *)

```
@{assert} (@{mk-term ?xs @ ?ys (ys, xs)} (@term [True]}, @term [False]))
```

```

    = @term [False] @ [True]);
  >

```

Note that *mk-term* does not perform full type inference automatically. If some argument types are mismatched or too general, the output may be inconsistent. This may change in future versions.

```

ML-val <
@assert (@mk-term ?x = ?y (x, y)) (@term 0::'a::zero, @term Suc 0)
    = (Const (@const-name HOL.eq), @typ 'a::zero => 'a => bool) $
      @term 0::'a::zero $ @term Suc 0))
  >

```

Besides static checking, using *mk-term* also gives a reusable template:

```

ML-val <
let
  val mk-pair: term*term -> term = @mk-term (?a, ?b) (a, b);
in
  @assert (mk-pair (@term ()), @term True) = @term ((), True));
  @assert (mk-pair (@term int 1), @term int 2)
    = @term (int 1, int 2))
end;
  >

```

end

3.2 Term pattern: *term-pat*

theory *TermPatternAntiquote* **imports**

Pure

begin

term-pat: ML antiquotation for pattern matching on terms.

See *TermPatternAntiquote_Tests.thy* for examples and tests.

ML <

structure *Term-Pattern-Antiquote* = *struct*

val *quote-string* = *quote*

(* *typ* matching; doesn't support matching on named TVars.

* This is because each TVar is likely to appear many times in the pattern. *)

fun *gen-typ-pattern* (TVar -) = -

| *gen-typ-pattern* (TFree (v, sort)) =

Term.TFree (^ quote-string v ^, [^ commas (map quote-string sort) ^])

| *gen-typ-pattern* (Type (typ-head, args)) =

Term.Type (^ quote-string typ-head ^, [^ commas (map gen-typ-pattern args) ^])

(* *term* matching; does support matching on named (non-dummy) Vars.

```

* The ML var generated will be identical to the Var name except in
* indexed names like ?v1.2, which creates the var v12. *)
fun gen-term-pattern (Var ((-dummy-, -), -)) = -
| gen-term-pattern (Var ((v, 0), -)) = v
| gen-term-pattern (Var ((v, n), -)) = v ^ string-of-int n
| gen-term-pattern (Const (n, typ)) =
  Term.Const ( ^ quote-string n ^, ^ gen-typ-pattern typ ^ )
| gen-term-pattern (Free (n, typ)) =
  Term.Free ( ^ quote-string n ^, ^ gen-typ-pattern typ ^ )
| gen-term-pattern (t as f $ x) =
  (* (read-term-pattern -) helpfully generates a dummy var that is
  * applied to all bound vars in scope. We go back and remove them. *)
  let fun default () = ( ^ gen-term-pattern f ^ $ ^ gen-term-pattern x ^ );
  in case strip-comb t of
    (h as Var ((-dummy-, -), -), bs) =>
      if forall is-Bound bs then gen-term-pattern h else default ()
    | - => default () end
| gen-term-pattern (Abs (-, typ, t)) =
  Term.Abs (-, ^ gen-typ-pattern typ ^, ^ gen-term-pattern t ^ )
| gen-term-pattern (Bound n) = Bound ^ string-of-int n

(* Create term pattern. All Var names must be distinct in order to generate ML
variables. *)
fun term-pattern-antiquote ctxt s =
  let val pat = Proof-Context.read-term-pattern ctxt s
      val add-var-names' = fold-aterms (fn Var (v, -) => curry (:) v | - => I);
      val vars = add-var-names' pat [] |> filter (fn (n, -) => n <> -dummy-)
      val - = if vars = distinct (=) vars then () else
        raise TERM (Pattern contains duplicate vars, [pat])
  in ( ^ gen-term-pattern pat ^ ) end

end;
val - = Context.>> (Context.map-theory (
  ML-Antiquotation.inline @{binding term-pat}
  ((Args.context -- Scan.lift Parse.embedded-inner-syntax)
   >> uncurry Term-Pattern-Antiquote.term-pattern-antiquote)))
>

end

theory TermPatternAntiquote-Tests
imports
  TermPatternAntiquote
  Main
begin
  Term pattern matching utility.
  Instead of writing monstrosities such as

```



```

case t of
  Const ("Pure.imp", _) $
    P $
    Const (@{const_name Trueprop}, _) $
      (Const ("HOL.eq", _) $
        (Const (@{const_name "my_func"}, _) $ x) $ y)
=> (P, x, y)

```

simply use a term pattern with variables:

```

case t of
  @{term_pat "PROP ?P \<Longrightarrow> my_func ?x = ?y"}
=> (P, x, y)

```

Each *term-pat* generates an ML pattern that can be used in any case-expression or name binding. The ML pattern matches directly on the term datatype; it does not perform matching in the Isabelle logic.

Schematic variables in the pattern generate ML variables. The variables must obey ML's pattern matching rules, i.e. each can appear only once.

Due to the difficulty of enforcing this rule for type variables, schematic type variables are ignored and not checked.

Example: evaluate arithmetic expressions in ML.

```

ML-val <
fun eval-num @{term_pat numeral ?n} = HOLogic.dest-numeral n
| eval-num @{term_pat Suc ?n} = eval-num n + 1
| eval-num @{term_pat 0} = 0
| eval-num @{term_pat 1} = 1
| eval-num @{term_pat ?x + ?y} = eval-num x + eval-num y
| eval-num @{term_pat ?x - ?y} = eval-num x - eval-num y
| eval-num @{term_pat ?x * ?y} = eval-num x * eval-num y
| eval-num @{term_pat ?x div ?y} = eval-num x div eval-num y
| eval-num t = raise TERM (eval-num, [t]);

```

```

eval-num @{term (1 + 2) * 3 - 4 div 5}
>

```

Regression test: backslash handling

```

ML-val <
val @{term_pat  $\alpha$ } = @{term  $\alpha$ }
>

```

Regression test: special-casing for dummy vars

```

ML-val <
val @{term_pat  $\lambda x y. \_$ } = @{term  $\lambda x y. z$ }
>

```

end

theory *Print-Annotated*

imports

Main

keywords

print-annotated-thm :: *diag*

begin

ML <

signature ANNOTATE-TERM =

sig

val string-of-term:

Proof.context -> int option -> term -> string

val string-of-thm:

Proof.context -> int option -> thm -> string

val print-thm: Proof.context -> int option -> thm -> unit

val print-thm-cmd:

int option ->

*Facts.ref * Token.src list ->*

Toplevel.transition -> Toplevel.transition

end

structure Annotate-Term : ANNOTATE-TERM =

struct

local

open Sledgehammer-Util

in

fun string-of-term ctxt margin term =

let

val keywords = Thy-Header.get-keywords' ctxt

val ctxt' = ctxt

|> Config.put show-markup false

|> Config.put Printer.show-type-emphasis false

|> Config.put show-types false

|> Config.put show-sorts false

|> Config.put show-consts false

in

term

|> singleton (Syntax.uncheck-terms ctxt')

|> Sledgehammer-Isar-Annotate.annotate-types-in-term ctxt'

|> Print-Mode.setmp []

(Syntax.unparse-term ctxt')

#> the-default Pretty.string-of (Option.map Pretty.string-of-margin

margin))

end

```

end

fun string-of-thm ctxt margin thm = string-of-term ctxt margin (thm |> Thm.concl-of)

fun print-thm ctxt margin thm =
  writeln (Active.sendback-markup-command (lemma \ ^
    string-of-thm ctxt margin (Variable.import-vars ctxt thm) ^\))

fun print-thm-cmd margin arg = Toplevel.keep (fn state =>
  let
    val ctxt = Toplevel.context-of state
    val [thm] = Attrib.eval-thms ctxt [arg]
  in print-thm ctxt margin thm end
)

val - =
  Outer-Syntax.command command-keyword <print-annotated-thm>
  print theorems with (minimal) complete type annotations
  ((Scan.option (Args.parens Parse.nat) -- Parse.thm) >> (fn (margin, facts)
=>
  print-thm-cmd margin facts));

end
>

end

```

```

theory ML-Fun-Cache
imports Pure
begin

```

Extension of the basic cache in **Cache**:

- Include some statistics on cache hits / misses
- Retrieval of existing caches
- Support garbage collected caches (via weak references)

Combination of synchronized variables and weak references. Stored value might get garbage collected. Note the difference in the signature (option value at some positions) compared to **Synchronized**.

```

ML <
signature SYNCHRONIZED-WEAK =
sig
  type 'a var
  val var: string -> 'a -> 'a var

```

```

val value: 'a var -> 'a option
val assign: 'a var -> 'a -> unit

val timed-access: 'a var -> ('a option -> Time.time option) -> ('a option ->
('b * 'a) option) -> 'b option
val guarded-access: 'a var -> ('a option -> ('b * 'a) option) -> 'b
val change-result: 'a var -> ('a option -> 'b * 'a) -> 'b
val change: 'a var -> ('a option -> 'a) -> unit

end;

structure Synchronized-Weak: SYNCHRONIZED-WEAK =
struct
  abstype 'a var = Weak-Var of ('a Unsynchronized.ref option Unsynchronized.ref)
  Synchronized.var
  with

(* basic operations on weak references *)

  fun mk-weak x = Weak.weak (SOME (Unsynchronized.ref x))

  fun get-val w = case !w of
    SOME r => SOME (!r)
  | NONE => NONE

  fun map-val f w = f (get-val w)

(* the main interface *)

  fun var name x =
    Weak-Var (Synchronized.var name (mk-weak x))

  fun value (Weak-Var x) =
    get-val (Synchronized.value x)

  fun assign (Weak-Var x) v = Synchronized.assign x (mk-weak v)

  fun apply-access f w =
    case f (get-val w) of
      SOME (res, x) => SOME (res, mk-weak x)
    | NONE => NONE

  fun timed-access (Weak-Var x) time-limit f =
    Synchronized.timed-access x (map-val time-limit) (apply-access f)

  fun guarded-access var f = the (timed-access var (fn - => NONE) f);

(* unconditional change *)

```

```

fun change-result var f = guarded-access var (SOME o f);
fun change var f = change-result var (fn x => ((), f x));
end
end
>

```

ML <

```

structure More-Binding = struct

```

— cf. `Position.here`

```

fun here b =
  let
    val pos = Binding.pos-of b
    val text = Binding.print b
    val props = Position.properties-of pos;
    val (s1, s2) =
      (case (Position.line-of pos, Position.file-of pos) of
        (SOME i, NONE) => ( , (line ^ Value.print-int i ^))
      | (SOME i, SOME name) => ( , (line ^ Value.print-int i ^ of ^ quote name
      ^))
      | (NONE, SOME name) => ( , (file ^ quote name ^))
      | - => if Position.is-reported pos then ( , \092<^here>) else ( , ));
  in
    Markup.markup (Markup.properties props Markup.position) (text ^ s1 ^ s2)
  end;

```

end

```

signature FUN-CACHE =

```

```

sig

```

```

  (* non garbage collected variant of cache *)
  val create: binding -> ('table -> string) -> 'table -> ('table -> 'key ->
'value lazy option) ->
  ('key * 'value lazy -> 'table -> 'table) -> ('key -> 'value) -> 'key ->
'value

```

```

  (* garbage collected variant of cache *)
  val create-gc: binding -> ('table -> string) -> 'table -> ('table -> 'key ->
'value lazy option) ->
  ('key * 'value lazy -> 'table -> 'table) -> ('key -> 'value) -> 'key ->
'value

```

```

  datatype 'content handler = Handler of {name: binding, hits: unit -> int, misses:
unit -> int,
    reset: unit -> unit, content: unit -> 'content, mk-string: 'content ->

```

```

string}

(* non garbage collected variant of cache *)
val create-handler: binding -> ('table -> string) -> 'table -> ('table -> 'key
-> 'value lazy option) ->
  ('key * 'value lazy -> 'table -> 'table) -> ('key -> 'value) -> (('key ->
'value) * 'table handler)

(* garbage collected variant of cache *)
val create-gc-handler: binding -> ('table -> string) -> 'table -> ('table ->
'key -> 'value lazy option) ->
  ('key * 'value lazy -> 'table -> 'table) -> ('key -> 'value) -> (('key ->
'value) * 'table handler)

val get-handler: string -> string handler option
val delete-handler: string -> unit
val all-handlers: unit -> (string * string handler) list
val cache-statistics: unit -> (binding * {hits: int, misses: int, content-size: int})
list

val reset-cache: string -> unit
val reset-all-caches: unit -> unit

val get-info: 'content handler -> {name: binding, hits: int, misses: int, content:
'content, mk-string: 'content -> string}

val dummy-handler : 'content -> 'content handler;

val pretty-handler: 'content handler -> Pretty.T
val string-of-handler: 'content handler -> string
end;

structure Fun-Cache: FUN-CACHE =
struct

datatype 'content handler = Handler of {name: binding, hits: unit -> int, misses:
unit -> int,
  reset: unit -> unit, content: unit -> 'content, mk-string: 'content -> string};

fun dummy-handler c = Handler {name = Binding.name , hits = K 0, misses =
K 0, reset = K (), content = K c, mk-string = K ()};

val caches = Synchronized.var caches (Symtab.empty)

fun add-handler name handler = Synchronized.change caches (fn tab =>
  let
    val raw-name = Binding.name-of name
    val - = if Symtab.defined tab raw-name then tracing (overwriting cache: ^
More-Binding.here name) else ()
  end

```

```

in Syntab.update (raw-name, handler) tab end)

fun cached-apply name lookup update f x (tab, hits, misses) =
  case lookup tab x of
    SOME y => (y, (tab, hits + 1, misses))
  | NONE =>
    let val y = Lazy.lazy-name name (fn () => f x)
    in (y, (update (x, y) tab, hits, misses + 1)) end

fun gen-apply cached-apply change-result cache =
  change-result cache cached-apply
|> Lazy.force;

fun create-handler name mk-string empty lookup update f =
  let
    val raw-name = Binding.name-of name
    val empty-cache = (empty, 0, 0)
    val cache = Synchronized.var raw-name empty-cache;
    fun apply x = gen-apply (cached-apply raw-name lookup update f x) Synchron-
      ized.change-result cache

    fun hits () = #2 (Synchronized.value cache)
    fun misses () = #3 (Synchronized.value cache)
    fun content () = #1 (Synchronized.value cache)
    fun string () = mk-string (#1 (Synchronized.value cache))
    fun reset () = Synchronized.change cache (K empty-cache)

    val handler = Handler {name = name, hits = hits, misses = misses, reset =
      reset, content = content, mk-string = mk-string}
    val handler' = Handler {name = name, hits = hits, misses = misses, reset =
      reset, content = string, mk-string = I}
    val - = add-handler name handler'
  in (apply, handler) end;

fun create name mk-string empty lookup update f = fst (create-handler name mk-string
  empty lookup update f)

fun map-default-trace msg default f v =
  case v of SOME x => f x | NONE => (tracing msg; f default)

fun the-default-trace msg default v =
  case v of
    SOME x => x
  | NONE => (tracing msg; default)

fun create-gc-handler name mk-string empty lookup update f =
  let
    val raw-name = Binding.name-of name
    val empty-cache = (empty, 0, 0)

```

```

    val cache = Synchronized-Weak.var raw-name empty-cache;
    val the-default = the-default-trace (cache: ^ More-Binding.here name ^ was
garbage collected)
    val map-default = map-default-trace (cache: ^ More-Binding.here name ^ was
garbage collected - starting with empty cache again.)

    fun apply x = gen-apply (map-default empty-cache (cached-apply raw-name
lookup update f x)) Synchronized-Weak.change-result cache

    fun get cache = the-default empty-cache (Synchronized-Weak.value cache)
    fun hits () = #2 (get cache)
    fun misses () = #3 (get cache)
    fun content () = #1 (get cache)
    fun string () = mk-string (#1 (get cache))
    fun reset () = Synchronized-Weak.change cache (K empty-cache)

    val handler = Handler {name = name, hits = hits, misses = misses, reset =
reset, content = content, mk-string = mk-string}
    val handler' = Handler {name = name, hits = hits, misses = misses, reset =
reset, content = string, mk-string = I}
    val - = add-handler name handler'
    in (apply, handler) end;

fun create-gc name mk-string empty lookup update f = fst (create-gc-handler name
mk-string empty lookup update f)

fun get-handler name =
    Syntab.lookup (Synchronized.value caches) name

fun delete-handler name = Synchronized.change caches (Syntab.delete name)

fun all-handlers () =
    Syntab.dest (Synchronized.value caches)

fun cache-statistics () = all-handlers () |>
    map (fn (-, Handler h) =>
        (#name h, {hits = #hits h (), misses = #misses h (), content-size =
ML-Heap.obj-size (#content h ())}))

fun reset-cache name =
    case get-handler name of
        SOME (Handler h) => #reset h ()
    | NONE => warning (reset-handler: no handler with name ^ name)

fun reset-all-caches () =
    all-handlers () |> map (apsnd (fn Handler h => #reset h ())) |> K ()

fun get-info (Handler handler:'content handler) =
    {name = #name handler, hits = (#hits handler) (), misses = (#misses handler)

```



```

(), content = (#content handler) (), mk-string = #mk-string handler}

fun pretty-elem name value = Pretty.block (Pretty.breaks [Pretty.str name, Pretty.str
=, Pretty.str value])
fun pretty-info {name, content, hits, misses, mk-string, ...} =
  Pretty.list { } (
    [pretty-elem name (More-Binding.here name),
     pretty-elem hits (string-of-int hits),
     pretty-elem misses (string-of-int misses),
     pretty-elem content (mk-string content)])

fun pretty-handler x = (pretty-info o get-info) x
fun string-of-info x = (Pretty.string-of o pretty-info) x
fun string-of-handler x = (string-of-info o get-info) x

val - =
  ML-system-pp (fn depth => fn pretty => fn handler =>
    Pretty.to-polyml (pretty-handler handler));

end;
>

```

3.2.1 Example

Cache without garbage collection

```

ML-val <
fun fib n =
  if n < 3 then 1
  else fib (n-1) + fib (n-2)

val (fib, handler) = Fun-Cache.create-handler
  @{binding fib}
  (fn n => @{make-string} n) (* as the type of n is fixed @{make-string} will pick
a proper print function *)
  Inttab.empty
  Inttab.lookup
  Inttab.update
  fib

val - = tracing (trace 1: ^ Fun-Cache.string-of-handler handler)
val tests1 = map fib (1 upto 6)
val - = (tracing enforcing garbage collection; ML-Heap.full-gc()); (* does not affect
cache *)
val - = tracing (trace 2: ^ Fun-Cache.string-of-handler handler)
val tests2 = map fib (1 upto 6)
val - = tracing (trace 3: ^ Fun-Cache.string-of-handler handler)
>

```

Cache with garbage collection

```

ML-val <
  fun fib n =
    if n < 3 then 1
    else fib (n-1) + fib (n-2)

  val (fib, handler) = Fun-Cache.create-gc-handler
    @{binding fib}
    (fn n => @{make-string} n) (* as the type of n is fixed @{make-string} will pick
    a proper print function *)
    Inttab.empty
    Inttab.lookup
    Inttab.update
    fib

  val - = tracing (trace 1: ^ Fun-Cache.string-of-handler handler)
  val tests1 = map fib (1 upto 6)
  val - = tracing (trace 2: ^ Fun-Cache.string-of-handler handler)
  val - = (tracing enforcing garbage collection; ML-Heap.full-gc()); (* should empty
  cache *)
  val - = tracing (trace 3: ^ Fun-Cache.string-of-handler handler)
  val tests2 = map fib (1 upto 6)
  val - = tracing (trace 4: ^ Fun-Cache.string-of-handler handler)
  val tests3 = map fib (1 upto 6)
  val - = tracing (trace 5: ^ Fun-Cache.string-of-handler handler)
  >

```

end

theory *AutoCorres-Utills*

imports

Print-Annotated

ML-Fun-Cache

keywords *lazy-named-theorems::thy-decl*

begin

definition *CONV-ID:: 'a::{} ⇒ 'a* **where**

CONV-ID x ≡ x

lemma *CONV-ID-intro: (x::'a::{}) ≡ CONV-ID x*

by (*simp add: CONV-ID-def*)

ML-file *utils.ML*

definition *FALSE ≡ (∧ P. PROP P)*

lemma *ex-falso-quodlibet: PROP FALSE ⇒ PROP P*

by (*simp add: FALSE-def*)

ML-file *<context-tactical.ML>*
ML-file *lazy-named-theorems.ML*
ML-file *interpretation-data.ML*

definition *sim-set* :: ('a ⇒ 'b ⇒ bool) ⇒ 'a set ⇒ 'b set ⇒ bool **where**
sim-set R A B ⇔ (∀ a∈A. ∃ b∈B. R a b)

lemma *sim-set-empty*: *sim-set* R {} B
by (*auto simp: sim-set-def*)

lemma *sim-set-insert*: R a b ⇒ *sim-set* R A B ⇒ *sim-set* R (insert a A) (insert b B)
by (*auto simp: sim-set-def*)

lemma *sim-set-Collect-Ex*:
(∧ a. *sim-set* R {x. P a x} {x. Q a x}) ⇒ *sim-set* R {x. ∃ a. P a x} {x. ∃ a. Q a x}
by (*fastforce simp: sim-set-def*)

lemma *sim-set-Collect-conj*:
(A ⇒ *sim-set* R {x. P x} {x. Q x}) ⇒ *sim-set* R {x. A ∧ P x} {x. A ∧ Q x}
by (*fastforce simp: sim-set-def*)

lemma *sim-set-Collect-eq*:
R a b ⇒ *sim-set* R {x. x = a} {x. x = b}
by (*fastforce simp: sim-set-def*)

lemma *sim-set-eq-rel-set*: *sim-set* R = *rel-set* R OO (⊆)
apply (*auto simp: sim-set-def rel-set-def relcomp.simps fun-eq-iff OO-def*)[1]
subgoal for A B
by (*intro exI[of - {b∈B. ∃ a∈A. R a b}] blast*)
apply *blast*
done

end

3.3 ML Antiquotation to Match and Instantiate (recombine) cterms and terms

theory *Match-Cterm*
imports
AutoCorres-Utils
TermPatternAntiquote
begin

ML *<*
signature MATCH-CTERM =

```

sig
  val try-match: ('a -> 'b) -> 'a -> 'b option (* handles Match | Pattern.MATCH *)
  val switch: ('a -> 'b) list -> 'a -> 'b (* raises Match *)

  val cterm-match: cterm * cterm -> ctyp TVars.table * cterm Vars.table
  val cterm-first-order-match: cterm * cterm -> ctyp TVars.table * cterm Vars.table

  val match: theory -> term * term -> typ TVars.table * term Vars.table
  val first-order-match: theory -> term * term -> typ TVars.table * term Vars.table

  val pattern-vars: term -> term list
  val cterm-pattern-vars: cterm -> cterm list

  val cterm-instantiate: Position.T ->
    ctyp TVars.table * cterm Vars.table ->
    cterm list -> cterm list -> cterm -> cterm
  val instantiate: Position.T ->
    typ TVars.table * term Vars.table ->
    term list -> term list -> term -> term

  datatype mode = Higher-Order | First-Order | ML-Pattern
  val parse-mode: Token.T list -> mode * Token.T list
  val parse-morph-mode: Token.T list -> mode * Token.T list
  val ml-match-term: {morph: bool, cert: bool} -> mode -> Proof.context ->
    Position.T -> string -> string
end

structure Match-Cterm: MATCH-CTERM =
struct

fun try-match f x = SOME (f x) handle Match => NONE | Pattern.MATCH =>
NONE

fun switch (c::cs) = (fn x => (* to allow partial evaluation on list elements *)
  (case try-match c x of
    SOME v => v
  | NONE => switch cs x))
| switch [] = raise Match

(* ML code composition *)
fun ml-val-binding (name, value) = space-implode [val, name, =, value];
fun ml-indent n str = String.concat (replicate n " ") ^ str

fun ml-let-binding vals body =
  [let] @
  (map (ml-indent 2) vals) @ [
  in,
  ml-indent 2 body,

```

```

end]

fun ml-list elems = enclose [ ] (space-implode , elems)
fun ml-tuple elems = enclose ( ) (space-implode , elems)
fun ml-record elems = enclose { } (space-implode , elems)
val ml-atomic = enclose ( )

fun triv-record-field name = space-implode [name, =, name]
val triv-record-inst = map triv-record-field #> ml-record

fun ml-fun name args body =
  space-implode ([fun, name] @ args @ [=]) ::
  (map (ml-indent 2) body)

fun ml-fn args body =
  space-implode (map (fn arg => space-implode [fn, arg, =>]) args)::
  (map (ml-indent 2) body)

fun ml-apply f args = space-implode (f :: args)

fun ml-check-match term-pat args ct = (* apply to args to ommit warning on unused
  identifiers *)
  ml-apply
  (ml-atomic (space-implode [fn, term-pat, =>, ml-tuple args, | - => raise Match
  ]))
  [ml-atomic (ml-apply Thm.term-of [ct])]

(* Interface used by antiquotation *)
fun cterm-match (pat, obj) = Thm.match (pat, obj)
  handle Pattern.MATCH => Exn.reraise Pattern.MATCH

fun cterm-first-order-match (pat, obj) = Thm.first-order-match (pat, obj)
  handle Pattern.MATCH => Exn.reraise Pattern.MATCH

fun gen-match match thy (pat, obj) =
  let
    val (Tinsts, tinsts) = match thy (pat, obj) (Vartab.empty, Vartab.empty)
    fun mk-Tinst ((a, i), (S, T)) =
      (((a, i), S), T);
    fun mk-tinst ((x, i), (U, t)) =
      let val T = Envir.subst-type Tinsts U in
        (((x, i), T), t)
      end
  in
    (TVars.build (Vartab.fold (TVars.add o mk-Tinst) Tinsts),
     Vars.build (Vartab.fold (Vars.add o mk-tinst) tinsts))
  end
end

```

```

fun match thy (pat, obj) = gen-match Pattern.match thy (pat, obj)
  handle Pattern.MATCH => Exn.reraise Pattern.MATCH

fun first-order-match thy (pat, obj) = gen-match Pattern.first-order-match thy (pat,
obj)
  handle Pattern.MATCH => Exn.reraise Pattern.MATCH

fun cterm-instantiate pos (tenv, env) vars vals pattern =
  let
    val insts = map (Term.dest-Var o Thm.term-of) vars ~~ vals
    fun upd var value = the-default value (AList.lookup (op =) insts var)
    val env' = Vars.map upd env
  in
    Thm.instantiate-cterm (tenv, env') pattern
  end
  handle
    CTERM (msg, args) =>
      Exn.reraise (CTERM (msg ^\n from antiquotation Match-Cterm.cterm-instantiate
here: ^ Position.here pos, args));

fun instantiate pos (tenv, env) vars vals pattern =
  let
    val insts = map Term.dest-Var vars ~~ vals
    fun upd var value = the-default value (AList.lookup (op =) insts var)
    val env' = Vars.map upd env
  in
    Term-Subst.instantiate (tenv, env') pattern
  end
  handle
    TERM (msg, args) =>
      Exn.reraise (TERM (msg ^\n from antiquotation Match-Cterm.instantiate
here: ^ Position.here pos, args));

(* Antiquotation *)

fun collect-vars t = fold-aterms (fn Var v => cons v | - => I) t [] |> rev

fun pattern-vars pat = pat
  |> collect-vars
  |> filter-out (fn ((-dummy-, -), -) => true | - => false)
  |> (distinct (op =)) |> map Var

fun cterm-pattern-vars cpat =
  let
    val ctxt = cpat |> Thm.theory-of-cterm |> Proof-Context.init-global
  in
    cpat |> Thm.term-of |> pattern-vars |> map (Thm.cterm-of ctxt)
  end
end

```

```

datatype mode = Higher-Order | First-Order | ML-Pattern

fun ml-match-term (kind as {morph:bool, cert:bool}) (mode:mode) ctxt pos pat-
term-str =
  let
    val phiN = phi-
    val thyN = thy-

    val match = case (mode, kind) of
      (Higher-Order, {cert = true ,...}) => Match-Cterm.cterm-match
    | (Higher-Order, {cert = false,...}) => ml-apply Match-Cterm.match [thyN]
    | (- , {cert = true ,...}) => Match-Cterm.cterm-first-order-match
    | (- , {cert = false,...}) => ml-apply Match-Cterm.first-order-match
    [thyN]

    val cterm-of = case kind of
      {morph = false, cert = true } => Thm.cterm-of ML-context
    | {morph = false, cert = false} =>
      | {morph = true , cert = true } => (Morphism.cterm ^ phiN ^ o
      Thm.cterm-of ML-context)
    | {morph = true , cert = false} => Morphism.term ^ phiN

    val ctN = ct- val envN = env- val instN = inst- val patN = pat- val posN =
    pos-

    val instantiateN = instantiate
    val reserved = [ctN, envN, instN, instantiateN, patN, posN, instantiateN]
    val pat = Proof-Context.read-term-pattern ctxt pattern-str
    val vars = collect-vars pat |> filter-out (fn ((-dummy-, -), -) => true | - =>
    false)
    |> (mode <> ML-Pattern) ? (distinct (op =))
    val names = map (#1 o #1) vars
    val dups = duplicates (op =) names
    val - = if null dups then ()
      else error (ml-match-cterm: duplicate variables in pattern: ^ Position.here
    pos ^ (commas (map quote dups)))
    val clashes = inter (op =) names reserved
    val - = if null clashes then ()
      else error (ml-match-cterm: avoid internal name(s) in pattern: ^ (commas
    (map quote clashes)))

    val names-var = map (suffix -var) names

    val ml-pat = ml-apply cterm-of [ml-atomic (ML-Syntax.print-term pat)]
    val ml-pattern-check =
      if mode = ML-Pattern then
        [ml-val-binding (-,
          ml-check-match (Term-Pattern-Antiquote.term-pattern-antiquote ctxt
          pattern-str) names ctN)]
  end

```

```

else []

val pat-vars = case cert of
  true => Match-Cterm.cterm-pattern-vars
| false => Match-Cterm.pattern-vars

val inst = case cert of
  true => Thm.instantiate-cterm
| false => Term-Subst.instantiate
val instantiate = case cert of
  true => Match-Cterm.cterm-instantiate
| false => Match-Cterm.instantiate

val vals = map ml-val-binding [
  (posN, ML-Syntax.print-position pos),
  (ml-list names-var, ml-apply pat-vars [patN]),
  (envN, ml-apply match [ml-tuple [patN, ctN]]),
  (ml-list names,
   ml-apply map
    [ml-atomic (ml-apply inst [envN]), ml-list names-var]),
  (ml-list names-var,
   ml-apply map
    [ml-atomic (ml-apply inst [ml-tuple [ml-apply fst [envN], Vars.empty]],
    ml-list names-var])]
val instantiate = ml-fun instantiateN [ml-record names]
  [ml-apply instantiate [posN, envN, ml-list names-var, ml-list names, patN]]
val inst = ml-val-binding (instN, triv-record-inst (names @ [instantiateN, ctN]))
val pat-binding = map ml-val-binding [(patN, ml-pat)]
val outer-args = if morph then [phiN] else []

val body = ml-let-binding (ml-pattern-check @ vals @ instantiate @ [inst]) instN
val args = (if cert then [] else [thyN]) @ [ctN]
val res = ml-fn args body |> space-implode \n
val outer-body = ml-let-binding pat-binding res (* partial application of phi to
pattern *)
val outer-res = ml-fn outer-args outer-body |> space-implode \n
val - = Utils.verbose-msg 5 ctxt (fn - => ml-match-cterm: ^ outer-res)
in
  outer-res
end

val parse-mode = Scan.optional
  (Args.parens (
    Args.$$$ ho >> K Higher-Order ||
    Args.$$$ fo >> K First-Order ||
    Args.$$$ ml >> K ML-Pattern))
  Higher-Order;

```



```

val parse-morph-mode = Scan.optional
  (Args.parens (
    Args.$$$ ho >> K Higher-Order ||
    Args.$$$ fo >> K First-Order))
  Higher-Order;

end

val - = Theory.setup
  (ML-Antiquotation.value-embedded @{binding cterm-match}
    ((Args.context -- Scan.lift (Match-Cterm.parse-mode -- Parse.position
  Parse.embedded-inner-syntax))
    >> (fn (ctxt, (mode, (pattern-str, pos))) =>
      Match-Cterm.ml-match-term {morph = false, cert = true} mode ctxt
  pos pattern-str)) #>
    (ML-Antiquotation.value-embedded @{binding cterm-morph-match}
    ((Args.context -- Scan.lift (Match-Cterm.parse-morph-mode -- Parse.position
  Parse.embedded-inner-syntax))
    >> (fn (ctxt, (mode, (pattern-str, pos))) =>
      Match-Cterm.ml-match-term {morph = true, cert = true} mode ctxt
  pos pattern-str))) #>
    (ML-Antiquotation.value-embedded @{binding match}
    ((Args.context -- Scan.lift (Match-Cterm.parse-mode -- Parse.position
  Parse.embedded-inner-syntax))
    >> (fn (ctxt, (mode, (pattern-str, pos))) =>
      Match-Cterm.ml-match-term {morph = false, cert = false} mode ctxt
  pos pattern-str))) #>
    (ML-Antiquotation.value-embedded @{binding morph-match}
    ((Args.context -- Scan.lift (Match-Cterm.parse-morph-mode -- Parse.position
  Parse.embedded-inner-syntax))
    >> (fn (ctxt, (mode, (pattern-str, pos))) =>
      Match-Cterm.ml-match-term {morph = true, cert = false} mode ctxt
  pos pattern-str))))
  )

```

The antiquotation yields a function that matches the schematic variables of a pattern with a `cterm`. The matched parts are returned as `cterm` in a record, where the field name is derived from the original schematic variable name. Moreover a function `instantiate` is returned that can be used to instantiate the matched pattern with other `cterms`.

The antiquotation makes use of the kernel operations `Thm.match / Thm.first_order_match`. These operations are efficient in the sense that no costly re-certification of subterms has to be performed.

```

ML-val <
val {f, g, instantiate, ct-} = @{cterm-match λx. ?f x + ?g x} @{cterm λx. p x +
q x}
val twisted = instantiate {f = g, g = f}
  >

```

Dummy pattern can also be supplied. The matched values for the dummy patterns are not part of the matching result but still considered in *instantiate*.

```

ML-val <
val {f, g, instantiate, ...} = @{\cterm-match \lambda x. ?f x + ?g x + -} @{\cterm \lambda x. p
x + q + r x}
val twisted = instantiate {f = g, g = f}
>

```

```

ML-val <
val {f, g, ...} = @{\cterm-match \lambda x. ?f x + ?g x} @{\cterm \lambda x. p x + q}
>

```

```

ML-val <
val {f1, f2, ...} = @{\cterm-match \lambda x. ?f1.0 x + ?f2.0 x} @{\cterm \lambda x. p x + q}
>

```

```

ML-val <
val {f, ...} = @{\cterm-match \lambda x. ?f x} @{\cterm <\lambda x. f>}
>

```

There is also a variant for first-order-matching.

```

ML-val <
let
  val - = @{\cterm-match (fo) \lambda x. ?f x} @{\cterm <\lambda x. f>}
in
  ()
end
handle Pattern.MATCH => tracing (this is not first order)
>

```

There is also the corresponding antiquotations for plain terms.

```

ML-val <
val {f, g, instantiate, ...} = @{\match \lambda x. ?f x + ?g x} @{\theory} @{\term \lambda x. p x
+ q x}
val twisted = instantiate {f = g, g = f}
>

```

There you can also see the different workings of first-order-matching. Note the eta expanded match result in the higher-order variant before.

```

ML-val <
val {f, g, instantiate, ...} = @{\match (fo) \lambda x. ?f x + ?g x} @{\theory} @{\term \lambda x.
p x + q x}
val twisted = instantiate {f = g, g = f}
>

```

There is a variant for ML-pattern-matching. This means, that before using first-order-matching it is tested whether the term actually matches the underlying ML-pattern with ML-pattern matching.

```

ML-val <
val {f, g, instantiate, ...} = @{cterm-match (ml) λx. ?f x + ?g x} @{cterm λx. p
x + q x}
val twisted = instantiate {f = g, g = f}
>

```

Note that as with all ML-Patterns a variable might only appear once. So the following example is a valid first-order or higher-order pattern but not a valid ML-pattern

```

ML-val <
val {f, ...} = @{cterm-match ?f + ?f} @{cterm a + a}
>

```

This is what happens conceptually in the expanded antiquotation.

```

ML-val <
val ctxt = @{context}
val pat = Proof-Context.read-term-pattern ctxt λx. ?f x + ?g x |> Thm.cterm-of
ctxt
val [g-var, f-var] = Term.add-vars (Thm.term-of pat) [] |> map (Thm.cterm-of
ctxt o Var)
val env = Thm.match (pat, @{cterm λx. a x + b})
val [f, g] = map (Thm.instantiate-cterm env) [f-var, g-var]
val tenv = fst env
val [f-var, g-var] = map (Thm.instantiate-cterm (tenv, Vars.empty)) [f-var, g-var]
fun instantiate {f, g} =
  Match-Cterm.cterm-instantiate Position.none env [f-var, g-var] [f, g]
val res = {f = f, g = g, instantiate = instantiate}
val x = instantiate {f=g, g=f}
>

```

With the verbose flag you can see the generated function.

```

declare [[verbose=5]]

```

```

ML-val <
val X = @{cterm-match ?X + -} @{cterm <a + a>}
>

```

```

ML-val <
val X = @{cterm-morph-match ?X + -} Morphism.identity @{cterm <a + a>}
>

```

```

ML-val <
val X = @{morph-match ?X + -} Morphism.identity @{theory} @{term <a + a>}
>

```

```
ML-val <
val X = @{cterm-match <?X + ->} @{cterm <a + a>}
>
```

```
declare [[verbose=0]]
```

```
end
```

3.4 Record Antiquotation

```
theory ML-Record-Antiquotation
```

```
  imports Main
```

```
begin
```

3.4.1 Motivation

A shortcoming of ML records is the lack of proper update / map functions for the record fields. Manually defining those update functions is considered as painful, as it requires 'quadratic' editing when adding a new field: - add the new field to every record pattern (in every already defined update / map function) - add a new update / map function for the field.

Various workarounds have been proposed. E.g. - Using mutable references as fields to allow for destructive updates - Fancy higher order function-combinators (Fold): <http://mlton.org/FunctionalRecordUpdate>

Here we develop yet another solution. We provide a ML-Antiquotation that generates the definitions.

For a record specification as datatype `datatype 'a foo = Foo {f1:'a, f2:bool}` it generates the datatype as specified with one constructor `Foo` wrapping the record and additionally all the 'canonical' functions:

- `make_foo`
- `dest_foo`
- `get_f1`
- `get_f2`
- `map_f1`
- `map_f2`

3.4.2 Example

```
ML-val <
datatype 'a foo = Foo of {f1:'a, f2:bool};

val make-foo = Foo;
fun dest-foo (Foo r) = r;

fun get-f1 (Foo r) = #f1 r;
fun get-f2 (Foo r) = #f2 r;

fun map-f1 f (Foo {f1, f2}) =
  Foo {f1 = f f1, f2 = f2};
fun map-f2 f (Foo {f1, f2}) =
  Foo {f1 = f1, f2 = f f2};
>
```

3.4.3 Implementation

```
ML <

structure ML-Record-Antiquotation =
struct

fun remove-comments s =
  let
    val unbalanced-comments-msg = unbalanced ML comments (* ... *)
    fun err-unbalanced () = error unbalanced-comments-msg

    fun rem n [] = if n = 0 then [] else err-unbalanced ()
      | rem n (::*::xs) = rem (n + 1) xs
      | rem n (*::)xs = if n > 0 then rem (n - 1) xs else err-unbalanced ()
      | rem n (x::xs) = if n = 0 then x::rem n xs else rem n xs
  in
    s |> Symbol.explode |> rem 0 |> implode
  end

fun to-upper-first s =
  case String.explode s of
    [] =>
  | c::cs => String.implode ((Char.toUpper c)::cs)

fun split-first [] = raise List.Empty
  | split-first (x :: xs) = (x, xs);
```

— We rely on the ML compiler to check the syntax and only provide a hands-on parser here.

```
fun parse-record-declaration s =
  let
    fun is-whitespace c = member (op =) [ , \t, \n] (str c)
```

```

fun white-space-explode = []
  | white-space-explode s = String.fields (is-whitespace) s |> filter-out (fn x =>
x = )

fun remove p s
  = String.explode s |> filter-out p |> String.implode

val normalize-spaces = space-implode o white-space-explode;

val remove-delimiter = remove (fn c => member (op =) [{(, },),;] (str c))
val trim = remove (fn c => str c = )

fun strip-word word s =
  let
    val (first, rest) = s |> white-space-explode |> split-first
  in
    if first = word then space-implode rest else error (expecting ' ^ word ^ ')
  end

val [lhs, rhs] = s |> remove-comments |> normalize-spaces |> space-explode =
|> map normalize-spaces

fun split-first-word s = s |> white-space-explode |> split-first ||> space-implode

fun split-last-word s = s |> white-space-explode |> split-last |>> space-implode

fun split-type s =
  let
    val (params, type-name) = s
      |> strip-word datatype
      |> split-last-word
      |>> remove-delimiter
      |>> space-explode ,
  in (params, type-name) end

val (params, rec-name) = split-type lhs

fun is-type-parameter s = String.isPrefix ' s orelse String.isPrefix " s

fun get-params s = s
  |> white-space-explode
  |> map remove-delimiter
  |> map (space-explode ,)
  |> flat
  |> filter is-type-parameter

val (constr, field-names-params) =

```

```

    rhs
  |> split-first-word
  ||> strip-word of
  ||> remove-delimiter
  ||> space-explode ,
  ||> map (space-explode :)
  ||> map (fn [name, typ] => (trim name, get-params typ))

  in ((lhs, rhs), (params, rec-name), constr, field-names-params) end

fun mk-record fields = enclose { } (commas fields);

fun make-name record = make- ^ record;
fun dest-name record = dest- ^ record;
fun map-name field = map- ^ field;
fun get-name field = get- ^ field;

fun mk-record-ass fields values =
  fields ~ values
  |> map (fn (f,v) => f ^ = ^ v)
  |> commas
  |> enclose { }

fun replace p q = map (fn x => if x = p then q else x)

fun mk-map constr fields n =
  let
    val (field, params) = nth fields n
    val field-names = map fst fields
    val [f] = Name.variant-list field-names [f]
  in
    fun ^ map-name field ^
      f ( ^ constr ^ ^ mk-record field-names ^ ) ^ = \n ^
      ^ constr ^ ^ mk-record-ass field-names (replace field (f ^ ^ field) field-names)
  ^;
  end

fun mk-maps constr fields =
  0 upto (length fields) - 1
  |> map (mk-map constr fields)
  |> cat-lines

fun mk-get constr field-name =
  fun ^ get-name field-name ^ ( ^ constr ^ r) = # ^ field-name ^ r;;

fun mk-gets constr fields =
  fields |> map fst |> map (mk-get constr) |> cat-lines

fun mk-record-datatype-declaration add-decl s =

```

```

let
  val ((lhs, rhs), record as (params, rec-name), constr, fields) = parse-record-declaration
s;
  val decl = lhs ^ = ^ rhs ^ ;;
  val make = val ^ make-name rec-name ^ = ^ constr ^ ;;
  val dest = fun ^ dest-name rec-name ^ ( ^ constr ^ r) = r;;
  val gets = mk-gets constr fields;
  val maps = mk-maps constr fields;
  fun cond-cons P a xs = if P then a::xs else xs
in
  cat-lines (cond-cons add-decl decl [make, dest, gets, maps])
end

end
>

```

Test the parser and its output.

```

ML-val <
let
  val test = datatype 'a foo = Foo of {fld1:int, fld2:'a}
in writeln (ML-Record-Antiquotation.mk-record-datatype-declaration true test) end
>

```

```

setup <
ML-Antiquotation.inline-embedded (Binding.make (record,here))
(Scan.lift Parse.embedded-input >> (fn source =>
  let
    val - = ML-Lex.read-source source; — provide some markup as side effect
    val sanitized-str = fst (Input.source-content source) |> ML-Lex.tokenize |>
ML-Lex.flatten
  in ML-Record-Antiquotation.mk-record-datatype-declaration true sanitized-str
end))
>

```

3.4.4 Examples

```

ML-val <

structure JustAStruct = struct
datatype bar = Bar of int;

@{record <datatype 'a foo =
  Foo of {f1:'a (* comments are removed *),
    f2:bar }
}>}

val my-foo = Foo {f1 = 1, f2 = Bar 42};
val my-foo1 = map-f1 (K (Bar 44)) my-foo;
val t = get-f1 my-foo1

```



```

val t1 = make-foo {f1=3, f2=Bar 4}
val t2 = dest-foo t1

@{record <datatype foo2 = Foo2 of {f3:int foo}>}

end
>

```

Note that the markup of the datatype declaration is limited to basic Lex-markup. Experiments with explicitly evaluating the datatype with e.g. the `ML` function failed. The issue there is that the `ML` function is not evaluated at the point (aka context) exactly in the text position, but with the context of the surrounding `ML` command.

The following examples try to illustrate the issues.

Works as expected, as every type referred to in the datatype spec is already known at the beginning of the **ML-val** command.

```

ML-val <
val - = ML <datatype foo = Foo of int>;
>

```

The following fails as 'bar' is not known at the beginning of the context.

```

ML-val <
datatype bar = Bar of int
val - = ML <datatype foo = Foo of bar>
handle ERROR x => warning x
>

```

Adding a semicolon after the definition of bar helps. There the semicolon marks the end of an "evaluation / compilation chunk". So `ML` is evaluated within an already augmented `ML`-context that knows about `bar`.

```

ML-val <
datatype bar = Bar of int;
val - = ML <datatype foo = Foo of bar>
>

```

In the context of a structure the semicolon is not enough.

```

ML-val <

structure InAStruct =
struct
datatype bar = Bar of int;

val - = ML <datatype foo = Foo of bar>
handle ERROR x => warning x
end

>

```

end

```
theory Misc-Antiquotation  
  imports Main  
begin
```

3.5 Various Antiquotations

3.5.1 Antiquotations for terms with schematic variables

```
setup <  
  let  
    val parser = Args.context -- Scan.lift Parse.embedded-inner-syntax  
    fun pattern (ctxt, str) =  
      str |> Proof-Context.read-term-pattern ctxt  
      |> ML-Syntax.print-term  
      |> ML-Syntax.atomic  
  in  
    ML-Antiquotation.inline @{binding pattern} (parser >> pattern)  
  end  
>
```

```
ML <@{pattern  $\lambda(a,b,c). ?g a b c$ }>
```

```
setup <  
  let  
    type style = term -> term;  
  
    fun pretty-term-style ctxt (style: style, t) =  
      Document-Output.pretty-term ctxt (style t);  
  
    val basic-entity = Document-Output.antiquotation-pretty-source;  
  
  in  
    (* Document antiquotation that allows schematic variables *)  
    (basic-entity binding <pattern> (Term-Style.parse -- Args.term-pattern) pretty-term-style)  
  end  
>
```

3.5.2 Antiquotation for types with schematic variables

```
setup <  
  let  
    val parser = Args.context -- Scan.lift Parse.embedded-inner-syntax  
    fun typ-pattern (ctxt, str) =  
      let  
        val ctxt' = Proof-Context.set-mode Proof-Context.mode-schematic ctxt  
      in
```

```

    str |> Proof-Context.read-typ ctxt'
        |> ML-Syntax.print-typ
        |> ML-Syntax.atomic
  end
in
  ML-Antiquotation.inline @{binding typ-pattern} (parser >> typ-pattern)
end
›

ML ‹@{typ-pattern ?'a list}›
ML ‹Sign.typ-instance @{theory} (@{typ 'a list}, @{typ-pattern ?'a list})›

end

```

Chapter 4

Proof Tools

```
theory Tuple-Tools
```

```
imports Main
  Misc-Antiquotation
  TermPatternAntiquote
  ML-Fun-Cache
```

```
begin
```

4.1 Tools for handling Tuples

The tools are supposed to help simplifying expressions containing *case-prod* constructions, like $\lambda(w, x, y, z). f w x y z$.

- Simprocs for single step splitting and simplification of tuples / *case-prods* instead of repeated application of the built-in variants for pairs
- Looper for the simplifier to instantiate f in $f (a, b, c)$ with $\lambda(a, b, c). g a b c$

4.1.1 Antiquotations for terms with schematic variables

```
setup <
let
  val parser = Args.context -- Scan.lift Parse.embedded-inner-syntax
  fun pattern (ctxt, str) =
    str |> Proof-Context.read-term-pattern ctxt
        |> ML-Syntax.print-term
        |> ML-Syntax.atomic
in
  ML-Antiquotation.inline @{binding pattern} (parser >> pattern)
end
```

```

>
ML <@{pattern  $\lambda(a,b,c). ?g a b c$ }>

setup <
  let
  type style = term -> term;

  fun pretty-term-style ctxt (style: style, t) =
    Document-Output.pretty-term ctxt (style t);

  val basic-entity = Document-Output.antiquotation-pretty-source;

  in
  (* Document antiquotation that allows schematic variables *)
  (basic-entity binding <pattern> (Term-Style.parse -- Args.term-pattern) pretty-term-style)
  end
>

```

Unification only works modulo ordinary beta reduction but does not succeed on tupled-beta reduction. For example the simplifier will not find an instantiation for the variable $?c$

```

schematic-goal  $\bigwedge a b. (\text{case } (a, b) \text{ of } (x, y) \Rightarrow f x y) \equiv ?c (a, b)$ 
  apply simp
  oops

```

Such equations can typically occur in congruence rules when a pair is splitted by $(\bigwedge x. PROP ?P x) \equiv (\bigwedge a b. PROP ?P (a, b))$.

```

lemma tuple-up-eq-trivial:  $f r \equiv (\lambda x. f x) r$ 
  by simp

```

```

thm tuple-up-eq-trivial

```

```

lemma tuple-up-eq2:  $f (\text{fst } r) (\text{snd } r) \equiv (\text{case } r \text{ of } (x1, x2) \Rightarrow f x1 x2)$ 
  by (simp add: split-def)

```

Constant to explicitly trigger splitting by `simproc SPLIT_simproc` below

```

definition SPLIT :: 'a::{}  $\Rightarrow$  'a
  where SPLIT P  $\equiv$  P

```

```

lemma SPLIT-cong:  $PROP \text{SPLIT } P \equiv PROP \text{SPLIT } P$ 
  by simp

```

```

lemma case-prod-out:  $(\lambda r. f ((\lambda(a,b). (g a b)) r)) = (\lambda(a, b). f (g a b))$ 
  apply (rule ext)
  apply (simp add: split-paired-all)
  done

```

```

lemma case-prod-eta-contract:  $(\lambda x. (\text{case-prod } s) x) \equiv (\text{case-prod } s)$ 
  by simp

```

definition *ETA-TUPLED* :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b)
where *ETA-TUPLED* f ≡ f

lemma *ETA-TUPLED-trans*: f ≡ g ⇒ *ETA-TUPLED* f ≡ g
by (*simp add: ETA-TUPLED-def*)

Add the congruence rule *SPLIT PROP ?P ≡ SPLIT PROP ?P* to the simpset together with the simproc to avoid descending into *P* before the split.

ML <

structure Bin =
struct

type Bin = bool list

fun bin-of-int n =
 if n < 0 then error (bin: ^ cannot convert negativ number ' ^ string-of-int n ^')
 else map (fn 0 => false | - => true) (radixpand (2,n));

fun int-of-bin bs =
 let
fun int-of pos [] = 0
 | int-of pos (false::bs) = int-of (2 * pos) bs
 | int-of pos (true::bs) = pos + int-of (2 * pos) bs
 in int-of 1 (rev bs) end

fun string-of-bin bs =
 let
fun bit true = #1
 | bit false = #0
 in
 0b ^ String.implode (map bit bs)
 end

val bin-ord = list-ord bool-ord;
structure Bintab = Table(type key = Bin val ord = bin-ord);

val - = @{assert} (int-of-bin (bin-of-int 1) = 1)
val - = @{assert} (int-of-bin (bin-of-int 22) = 22)
val - = @{assert} (int-of-bin (bin-of-int 43) = 43)
val - = @{assert} (is-none (try bin-of-int ~1))
val - = @{assert} (string-of-bin (bin-of-int 0) = 0b0)
val - = @{assert} (string-of-bin (bin-of-int 1) = 0b1)
val - = @{assert} (string-of-bin (bin-of-int 7) = 0b111)
val - = @{assert} (string-of-bin (bin-of-int 23) = 0b10111)
 end
 >

ML \langle

```
structure Tuple-Tools = struct
```

— We document the effect of functions using assertions

— t has to be type correct and be equivalent to ct modulo beta/eta reduction and typing

```
fun assert-cterm' ctxt t ct =
```

```
  let
```

```
    val t1 = Envir.beta-eta-contract (Thm.term-of (Thm.cterm-of ctxt t))
```

```
    val t2 = Envir.beta-eta-contract (Thm.term-of ct)
```

```
  in @ {assert} (Term.aconv-untyped (t1, t2)) end
```

```
val assert-cterm = assert-cterm' @ {context}
```

```
fun assert-term t1 t2 = @ {assert} (t1 = t2)
```

```
fun gen-mk-prod (t1, T1) (t2, T2) =
```

```
  (HOLogic.pair-const T1 T2 $ t1 $ t2, HOLogic.mk-prodT (T1, T2))
```

```
fun gen-mk-tuple [] = (@ {term ()}, @ {typ unit})
```

```
  | gen-mk-tuple [(t, T)] = (t, T)
```

```
  | gen-mk-tuple (t::ts) = gen-mk-prod t (gen-mk-tuple ts)
```

— Functions to create various terms on tuples. * The primed variants take a list of element types. * The unprimed variants only take a number (arity of the tuple) and generate default element types.

```
fun mk-elT-named n i = TFree (n ^ string-of-int i, @ {sort type});
```

```
val mk-elT = mk-elT-named 'a
```

```
fun mk-elT' Ts i = nth Ts (i-1);
```

```
fun mk-tupleT' Ts i = HOLogic.mk-tupleT (drop (i - 1) Ts);
```

```
fun mk-tupleT n i = mk-tupleT' (map mk-elT (1 upto n)) i;
```

```
fun mk-el-name-named x i = x ^ string-of-int i;
```

```
val mk-el-name = mk-el-name-named x
```

```
fun mk-el' Ts i = Free (mk-el-name i, mk-elT' Ts i);
```

```
fun mk-el-named' x Ts i = Free (mk-el-name-named x i, mk-elT' Ts i);
```

```
fun mk-el-named x i = Free (mk-el-name-named x i, mk-elT i);
```

```
fun mk-el i = Free (mk-el-name i, mk-elT i);
```

```
fun mk-tuple-bounds n = (1 upto n) |> map (fn i => (Bound (n - i), mk-elT i))
```

```
|> gen-mk-tuple |> fst
```

```
val mk-tuple-packed-name = r;
```

```

fun mk-tuple n i = HOLogic.mk-tuple (map mk-el (i upto n));
fun mk-tuple-named x n i = HOLogic.mk-tuple (map (mk-el-named x) (i upto n));
fun mk-tuple' Ts n i = HOLogic.mk-tuple (map (mk-el' Ts) (i upto n))
fun mk-tuple-named' x Ts n i = HOLogic.mk-tuple (map (mk-el-named' x Ts) (i
upto n))

fun mk-tuple-packed n = Free (mk-tuple-packed-name, mk-tupleT n 1);

fun mk-P n = Free (P, map mk-elT (1 upto n) ----> @{typ bool});
fun mk-P-frees n = list-comb (mk-P n, map mk-el (1 upto n));
fun mk-P-bounds n = list-comb (mk-P n, map Bound (n-1 downto 0));

fun mk-Prop-P n = Free (P, mk-tupleT n 1 --> @{typ prop});

fun mk-fun (name,resT) i n = Free (name, map mk-elT (i upto (i+n-1)) ---->
resT);
fun mk-fun-bounds (name, resT) i n = list-comb (mk-fun (name, resT) i n, map
Bound (n-1 downto 0));

fun mk-fun-with-types Ts (name,resT) = Free (name, map (mk-elT' Ts) (1 upto
(length Ts)) ----> resT);
fun mk-fun-bounds-with-types Ts (name, resT) = list-comb (mk-fun-with-types Ts
(name, resT), map Bound ((length Ts - 1) downto 0));

val mk-f-resT = @{typ 'b};
val mk-f-name = f;
fun mk-f n = Free (mk-f-name, map mk-elT (1 upto n) ----> mk-f-resT);
fun mk-f-frees n = list-comb (mk-f n, map mk-el (1 upto n));
fun mk-f-bounds n = list-comb (mk-f n, map Bound (n-1 downto 0));

fun mk-f-tupled n = Free (mk-f-name, mk-tupleT n 1 --> mk-f-resT)

fun mk-selT n i = mk-tupleT n 1 --> mk-elT i;
fun mk-selT' Ts i = mk-tupleT' Ts i --> mk-elT' Ts i;
fun mk-snd n i = Const (@{const-name <snd>}, mk-tupleT n i --> mk-tupleT n
(i+1));
fun mk-snd' Ts i = Const (@{const-name <snd>}, mk-tupleT' Ts i --> mk-tupleT'
Ts (i+1));
fun mk-fst n i = Const (@{const-name <fst>}, mk-tupleT n i --> mk-elT i);
fun mk-fst' Ts i = Const (@{const-name <fst>}, mk-tupleT' Ts i --> mk-elT' Ts
i);

fun mk-snds' Ts i r =
  let
    val n = length Ts;
  in
    if n = 1 then r
  end

```



```

else
  if i = (n-1) then mk-snd' Ts 1 $ r
  else mk-snd' Ts (n - i) $ (mk-snds' Ts (i+1) r)
end;

fun mk-snds n i r =
  if i = (n-1) then mk-snd n 1 $ r
  else mk-snd n (n - i) $ (mk-snds n (i+1) r)

fun mk-sel' Ts r i =
  let
    val n = length Ts;
  in
    if n = 1 then r
    else
      if i = 1 then mk-fst' Ts 1 $ r
      else if i = n then mk-snds' Ts 1 r
      else mk-fst' Ts i $ mk-snds' Ts (n - i + 1) r
  end;

val - = assert-term (mk-sel' ([@{typ 'a}, @{typ 'b}, @{typ 'c}, @{typ 'd}]) (Bound
0) 4)
(Const (@{const-name snd}, @{typ 'c × 'd ⇒ 'd}) $
(Const (@{const-name snd}, @{typ 'b × 'c × 'd ⇒ 'c × 'd'}) $
(Const (@{const-name snd}, @{typ 'a × 'b × 'c × 'd ⇒ 'b × 'c × 'd'}) $ Bound
0)))

val - = assert-term (mk-sel' ([@{typ 'a}, @{typ 'b}, @{typ 'c}, @{typ 'd}]) (Bound
0) 3)
(Const (@{const-name fst}, @{typ 'c × 'd ⇒ 'c'}) $
(Const (@{const-name snd}, @{typ 'b × 'c × 'd ⇒ 'c × 'd'}) $
(Const (@{const-name snd}, @{typ 'a × 'b × 'c × 'd ⇒ 'b × 'c × 'd'}) $ Bound
0)))

fun mk-sel n r i =
  if i = 1 then mk-fst n 1 $ r
  else if i = n then mk-snds n 1 r
  else mk-fst n i $ mk-snds n (n - i + 1) r

fun mk-eq n r i =
  HOLogic.mk-eq (mk-sel n r i, mk-el i);

fun mk-case-prod-from T t i n =
  let
    fun mk-cp n i =
      if (n - i) <= 1 then
        HOLogic.case-prod-const (mk-elT i, mk-elT n, T) $
          (Abs (mk-el-name i, mk-elT i, Abs (mk-el-name n, mk-elT n, t)))
      else

```

```

      HOLogic.case-prod-const (mk-elT i, mk-tupleT n (i+1), T) $
        (Abs (mk-el-name i, mk-elT i, mk-cp n (i+1)))
    in mk-cp (n+i-1) i end;

```

```

fun mk-case-prod T t n =
  mk-case-prod-from T t 1 n

```

```

fun mk-case-prod-with-types Ts T t =
  let
    val n = length Ts
    fun mk-cp n i =
      if (n - i) <= 1 then
        HOLogic.case-prod-const (mk-elT' Ts i, mk-elT' Ts n, T) $
          (Abs (mk-el-name i, mk-elT' Ts i, Abs (mk-el-name n, mk-elT' Ts n, t)))
      else
        HOLogic.case-prod-const (mk-elT' Ts i, mk-tupleT' Ts (i+1), T) $
          (Abs (mk-el-name i, mk-elT' Ts i, mk-cp n (i+1)))
    in mk-cp n 1 end;

```

```

fun mk-case-prod-P n = mk-case-prod @{typ bool} (mk-P-bounds n) n;
fun mk-case-prod-f n = mk-case-prod mk-f-resT (mk-f-bounds n) n;
fun mk-case-prod-fun (name, resT) i n = mk-case-prod-from resT (mk-fun-bounds
(name, resT) i n) i n;
fun mk-case-prod-fun-with-types Ts (name, resT) = mk-case-prod-with-types Ts
resT (mk-fun-bounds-with-types Ts (name, resT));
fun mk-case-prod-f-tupled-bounds n = mk-case-prod mk-f-resT (mk-f-tupled n $
mk-tuple-bounds n) n

```

```

fun argTs-of-TVar n [(TVar ((x,-,-))] = map (mk-elT-named x) (1 upto n)
| argTs-of-TVar - Ts = Ts

```

```

fun mk-case-prod' name T n =
  if n <= 1 then Free (name, T)
  else
    let
      val (domT, rangeT) = dest-funT T
      val argTs = HOLogic.strip-tupleT domT |> argTs-of-TVar n
    in
      if length argTs = n
      then mk-case-prod-fun-with-types argTs (name, rangeT) (* Type was already
instantiated so we take those types *)
      else mk-case-prod-fun (name, rangeT) 1 n
    end

```

```

fun mk-tuple-from-type name T n =
  if n <= 1 then Free (name, T)
  else

```

```

let
  val Ts = HOLogic.strip-tupleT T
in
  if length Ts = n
  then mk-tuple-named' name Ts n 1
  else mk-tuple-named name n 1
end

fun mk-tuple-or-case-prod name T n =
  if is-some (try dest-funT T)
  then mk-case-prod' name T n
  else mk-tuple-from-type name T n

fun mk-f-sels n r = list-comb (mk-f n, map (mk-sel n r) (1 upto n));

fun mk-tuple-up-eq n =
  let
    val r = mk-tuple-packed n;
    val lhs = mk-f-sels n r;
    val rhs = mk-case-prod-f n $ r;
  in Logic.mk-equals (lhs, rhs) end

val - = assert-cterm (mk-tuple-up-eq 3)
  @{cterm f (fst r) (fst (snd r)) (snd (snd r)) ≡ case r of (x1, x2, x3) ⇒ f x1 x2
x3}

fun mk-tuple-up-eq-thm ctxt n =
  let
    val fixes = [mk-tuple-packed n, mk-f n]
    |> map (fn (Free (x, -)) => x);
  in
    Goal.prove ctxt fixes [] (mk-tuple-up-eq n) (fn - =>
      simp-tac (put-simpset HOL-basic-ss ctxt addsimps [mk-meta-eq @{thm split-def}]))
  1)
  end

fun mk-tuple-case-eq n =
  let
    val lhs = mk-case-prod-f n $ mk-tuple n 1
    val rhs = mk-f-frees n
  in
    Logic.mk-equals (lhs, rhs)
  end

val - = assert-cterm (mk-tuple-case-eq 3)
  @{cterm case (x1, x2, x3) of (x1, x2, x3) ⇒ f x1 x2 x3 ≡ f x1 x2 x3}

```

```

fun mk-tuple-case-eq-thm ctxt n =
  let
    val fixes = mk-f n::map mk-el (1 upto n)
      |> map (fn (Free (x, -)) => x)
  in
    Goal.prove ctxt fixes [] (mk-tuple-case-eq n) (fn - =>
      simp-tac (put-simpset HOL-basic-ss ctxt addsimps [@{thm Product-Type.prod.case}]))
  1)
  end

```

```

fun num-case-prod t =
  case strip-comb t of
    (Const (@{const-name case-prod},-), (Abs (-, -, cp))::-) => 1 + num-case-prod
  cp
  | (Const (@{const-name case-prod},-), -) => 1
  | - => 0;

```

```

val strip-uu = prefix strip- Name.uu-

```

```

fun strip-case-prod t =
  case strip-comb t of
    (Const (@{const-name case-prod},-), Abs (x, xT, (Abs (y, yT, -))::-) =>
      [(x,xT), (y, yT)])
  | (Const (@{const-name case-prod},cpT), Abs (x, xT, cp)::-) =>
    (case strip-comb cp of
      (Const (@{const-name case-prod},-), -) => (x,xT)::strip-case-prod cp
    | - => (* eta expand lost abstraction in case-prod *)
      [(x,xT), (strip-uu, nth (binder-types cpT) 1)]))
  | (Const (@{const-name case-prod}, cpT), cp::-) =>
    (* eta expand lost abstractions in case-prod *)
    let
      val [xT, yT] = take 2 (binder-types (domain-type cpT))
    in [(strip-uu, xT), (strip-uu, yT)] end
  | - => [];

```

```

fun strip-case-prod' (Abs (x, xT, -)) = [(x, xT)]
  | strip-case-prod' t = strip-case-prod t

```

```

fun mk-split-tupled-all-lhs n =
  Logic.list-all ([ (mk-tuple-packed-name, mk-tupleT n 1)], (mk-Prop-P n) $ Bound
  0)

```

```

fun mk-prod-bound (i,iT) (t,T) =
  HOLogic.pair-const iT T $ Bound i $ t

```

```

fun mk-prod-bounds [(i, iT), (j, jT)] = mk-prod-bound (i,iT) (Bound j, jT)
  | mk-prod-bounds ((i,iT)::xs) =

```

```

let
  val t = mk-prod-bounds xs
in mk-prod-bound (i,iT) (t, fastype-of t) end

fun mk-split-tupled-all-rhs n =
let
  val - = @{assert} (n >= 2)
in
  Logic.list-all (map (fn i => (mk-el-name i, mk-elT i)) (1 upto n),
    (mk-Prop-P n) $ mk-prod-bounds (map (fn i => (n-i, mk-elT i)) (1 upto
n)))
end

fun mk-split-tupled-all n =
  Logic.mk-equals (mk-split-tupled-all-lhs n, mk-split-tupled-all-rhs n)

val - = assert-cterm (mk-split-tupled-all 3)
@{cterm ( $\wedge r. PROP P r \equiv (\wedge x1\ x2\ x3. PROP P (x1, x2, x3))$ )}

fun mk-split-tupled-all-thm ctxt n =
  Goal.prove ctxt [P] [] (mk-split-tupled-all n) (fn - => simp-tac
    (put-simpset HOL-basic-ss ctxt addsimps @{thms split-paired-all}) 1)

fun mk-eta-tupled-eq n =
let
  val lhs = mk-f-tupled n
  val rhs = mk-case-prod-f-tupled-bounds n
  val eq = Logic.mk-equals (lhs, rhs)
in
  eq
end

fun mk-eta-tupled-eq-thm ctxt n =
  Goal.prove @{context} [mk-f-name] [] (mk-eta-tupled-eq n) (fn - =>
    simp-tac @{context} 1) |> Thm.transfer' ctxt

structure Data = Generic-Data(
  type T = (thm * thm * thm * thm) list;
  val empty = [];
  fun merge (thms1, thms2) = if length thms1 >= length thms2 then thms1 else
thms2;
);

fun get-tuple-up-eq-thm ctxt n =
  if n <= 1 then @{thm tuple-up-eq-trivial}
  else
    Thm.transfer' ctxt (#1 (nth (Data.get (Context.Proof ctxt)) (n - 2)))

```

```

    handle Subscript => mk-tuple-up-eq-thm ctxt n

fun get-split-tupled-all-thm ctxt n =
  if n <= 1 then error (get-split-tupled-all only makes sense for n >= 2)
  else
    Thm.transfer' ctxt (#2 (nth (Data.get (Context.Proof ctxt)) (n - 2)))
    handle Subscript => mk-split-tupled-all-thm ctxt n

fun get-tuple-case-eq-thm ctxt n =
  if n <= 1 then error (get-tuple-case-eq-thm only makes sense for n >= 2)
  else
    Thm.transfer' ctxt (#3 (nth (Data.get (Context.Proof ctxt)) (n - 2)))
    handle Subscript => mk-tuple-case-eq-thm ctxt n

fun get-eta-tupled-eq-thm ctxt n =
  if n <= 1 then @{thm reflexive}
  else
    Thm.transfer' ctxt (#4 (nth (Data.get (Context.Proof ctxt)) (n - 2)))
    handle Subscript => mk-eta-tupled-eq-thm ctxt n

fun liberal-zip (x::xs) (y::ys) = (x,y)::liberal-zip xs ys
  | liberal-zip _ [] = []
  | liberal-zip [] _ = []

fun get-first-subterm P t =
  if P (Term.head-of t)
  then SOME t
  else
    case t of
      (t1 $ t2) => (case get-first-subterm P t1 of
        NONE => get-first-subterm P t2
        | some => some)
    | Abs (-, -, t) => get-first-subterm P t
    | - => NONE

— Split a tuple completely in one step, instead of repeated applications of  $(\bigwedge x. PROP ?P x) \equiv (\bigwedge a b. PROP ?P (a, b))$ 
val split-tupled-all-simproc =
  Simplifier.make-simproc @context {name = split-tupled-all-simproc, identifier
= [],
lhss = [Proof-Context.read-term-pattern @context  $\bigwedge r. PROP ?P r$ ],
proc = fn - => fn ctxt => fn ct =>
  let
    fun get-tuple-arity (Const (@{const-name Pure.all},-) $ (Abs (-, T, -))) =
      length (HOLogic.flatten-tuple T)
    | get-tuple-arity - = 0;
    val t = Thm.term-of ct
    val n = get-tuple-arity t
  in

```

```

    if n >= 2
    then
      let
        fun is-case-prod (Const (@{const-name case-prod}, -)) = true
          | is-case-prod - = false

        fun guess-names t = case get-first-subterm is-case-prod t of
          SOME t' => map fst (strip-case-prod t') |> filter-out (fn n => n
= strip-uu)
          | - => []

        val thm = get-split-tupled-all-thm ctxt n
          |> Drule.rename-bvars (liberal-zip (map mk-el-name (1 upto n))
(guess-names t))
          in SOME thm end
        else NONE
      end
    }

```

— Simplify a *case-prod* applied to a tuple constructed from *Pair* in one step instead of repeated application of $(\text{case } (?x1.0, ?x2.0) \text{ of } (x, xa) \Rightarrow ?f \ x \ xa) = ?f \ ?x1.0 \ ?x2.0$ or $(\text{case } (?a, ?b) \text{ of } (c, d) \Rightarrow ?f \ c \ d) = ?f \ ?a \ ?b$

```

val tuple-case-simproc =
  Simplifier.make-simproc @{} {name = tuple-simproc, identifier = [],
  lhss = [Proof-Context.read-term-pattern @{} case-prod ?X (?x, ?y)],
  proc = fn - => fn ctxt => fn ct =>
    let
      fun get-tuple-arity
        (t as (Const (@{const-name Product-Type.prod.case-prod}, -) $ - $
          (p as (Const (@{const-name Pair}, -) $ - $ -)))) =
        Int.min (num-case-prod t + 1, length (HOLogic.strip-tuple p))
      | get-tuple-arity - = 0;
      val n = get-tuple-arity (Thm.term-of ct)
    in
      if n >= 2
      then SOME (get-tuple-case-eq-thm ctxt n)
      else NONE
    end
  }

```

— Instantiaten for tupled-beta reduction

```

local
fun strip-all (Const (@{const-name Pure.all}, -) $ Abs (x, T, t)) =
  let
    val (vTs, bdy) = strip-all t
  in ((x, T)::vTs, bdy) end
| strip-all t = ([], t)

```

```

fun dest-pair-bound (Const (@{const-name Product-Type.Pair}, -) $ Bound i $ p)
=
  i::dest-pair-bound p
  | dest-pair-bound (Bound i) = [i]

fun lookup env i = nth env i

fun mk-name env (Bound i) = fst (lookup env i)
  | mk-name env (Var ((x,-,-)) = x
  | mk-name env (Free (x,-)) = x
  | mk-name env - = Name.uu-;

fun mk-var env (Bound i) = SOME (lookup env i)
  | mk-var - = NONE

fun is-Pair (Const (@{const-name Product-Type.Pair}, -) $ - $ -) = true
  | is-Pair - = false;

fun dest-tuple env (Const (@{const-name Product-Type.Pair}, -) $ t $ p)
  = mk-name env t :: (if is-Pair p then dest-tuple env p else [mk-name env p])

fun dest-tuple' env (Const (@{const-name Product-Type.Pair}, -) $ t $ p)
  = mk-var env t :: (if is-Pair p then dest-tuple' env p else [mk-var env p])

fun mk-case-prod' T seed [(n1, T1), (n2, T2)] =
  (Const (@{const-name case-prod}, (T1 --> T2 --> T) --> HOLogic.mk-prodT
  (T1, T2) --> T)$
  Abs (n1, T1, Abs (n2, T2, seed)), HOLogic.mk-prodT (T1, T2))
  | mk-case-prod' T seed ((n1, T1)::bs) =
    let
      val (t', T2) = mk-case-prod' T seed bs
    in (Const (@{const-name case-prod}, (T1 --> T2 --> T) --> HO-
    Logic.mk-prodT (T1, T2) --> T)$
      Abs (n1, T1, t'), HOLogic.mk-prodT (T1, T2))
    end

fun get-tuple-var env (Var v $
  (p as (Const (@{const-name Product-Type.Pair}, -)$ -$ -))) =
  [(env, v, dest-tuple env p)]
  | get-tuple-var env (t1 $ t2) =
  (case get-tuple-var env t1 of
    [] => get-tuple-var env t2
  | xs => xs)
  | get-tuple-var env (Abs (x, T, t)) =
  get-tuple-var ((x, T)::env) t
  | get-tuple-var env - = [];

```



```

fun dest-tuple'' env t = case try (dest-tuple' env) t of
  SOME xs => xs
  | NONE => [NONE]

fun flatten-upto-first-tuple xxs =
  let
    val (upto-one, more) = chop-prefix (fn xs => length xs <= 1) xxs
  in
    if null more then ([],[])
    else (flat (upto-one), hd more)
  end

(* finds function vars that are applied to a tuple in some argument e.g.
 * ?f x y (a, b)
 *)
fun get-tuple-vars env (t as (- $ -)) =
  (case strip-comb t of
    (Var v, ts) =>
      let
        val (args, tuple-args) = map (dest-tuple'' env) ts |> flatten-upto-first-tuple
        val rest = map (get-tuple-vars env) ts |> flat
      in
        if null tuple-args then rest
        else (env, v, (args, tuple-args)) :: rest
      end
    | (t, ts) => map (get-tuple-vars env) (t::ts) |> flat
  | get-tuple-vars env (Abs (x, T, t)) =
    get-tuple-vars ((x,T)::env) t
  | get-tuple-vars - - = []);

fun get-distinct-tuple-vars env t =
  t |> get-tuple-vars env |> distinct (fn ((-,v1,-),(-,v2,-)) => v1 = v2)

in

val list-abs = fold-rev Term.abs

fun map-filter2 - [] [] = []
  | map-filter2 f (x :: xs) (y :: ys) =
    let
      val vs = map-filter2 f xs ys
    in case f x y of SOME v => v :: vs | - => vs end
  | map-filter2 - - - = raise ListPair.UnequalLengths;

fun gen-calc-inst bound get-tuple-vars ctxt max-idx t = try <
  get-tuple-vars [] t
  |> map (fn (-,((x,i), pT), (args, bs)) =>

```

```

let
  val (all-argTs, finalT) = strip-type pT
  val (argTs, ([tupleT], rest-argTs)) = all-argTs |> chop (length args) ||> chop
1

  fun mk-vars bound vs Ts = (vs ~~ Ts)
    |> map (fn (SOME (x, -), T) => (true, (x, T))
          | (NONE, T) => if bound then (false, (Name.uu-, T)) else (true,
(Name.uu-, T)))

  fun filter-tagged tags xs = map-filter2 (fn b => fn x => if b then SOME x
else NONE) tags xs

  val tagged-args = mk-vars false args argTs
  val args = map snd tagged-args
  val tagged-tuple-args = mk-vars bound bs (HOLogic.flatten-tupleT tupleT)
  val tuple-args = map snd tagged-tuple-args
  val tagged-vars = tagged-args @ tagged-tuple-args
  val tags = map fst tagged-vars
  val vars = map snd tagged-vars

  val bnds = length vars - 1 downto 0 |> filter-tagged tags
  val Ts = map snd vars |> filter-tagged tags
  val rT = rest-argTs ----> finalT
  val newVar = Var ((x,max-idx+1), Ts ----> rT)
  val seed = Term.list-comb (newVar, map Bound bnds)
  val case-prod = fst (mk-case-prod' rT seed tuple-args)
  val inst = list-abs args case-prod
  in ((x,i), Thm.ctrm-of ctxt inst) end
) catch - => []

fun calc-first-inst bound = gen-calc-inst bound (fn env => take 1 o get-tuple-vars
env)
val calc-inst = calc-first-inst true
val calc-inst' = calc-first-inst false
val calc-insts = gen-calc-inst false get-distinct-tuple-vars

end

```

(* For a goal with a schematic variable of the form

$?f(x_1, x_2, \dots, x_n)$

as it might occur in an congruence rule

$\bigwedge \dots x_1 \dots x_n \dots . ?X \equiv ?f(x_1, x_2, \dots, x_n) \dots$

instantiate $?f$ to $\lambda(x_1, x_2, \dots, x_n). ?f' x_1 x_2 \dots x_n \dots$

Currently only instantiates the first such $?f$ it finds by a canonical traversal of the term.

The tupled argument does not have to be the first argument, also works for the more general case

like $?f\ t1\ t2\ (x1,x2,\dots,xn)\ t3\ t4$

`tuple-inst-tac` can be used as a looper.

*)

```
fun cond-trace ctxt s =
  if Config.get ctxt Simplifier.simp-trace then tracing s else ()

fun tuple-inst-tac ctxt i = SUBGOAL (fn (trm, -) => fn st =>
  let
    val max-idx = Thm.maxidx-of-cterm (Thm.cprop-of st)
  in
    case calc-inst ctxt max-idx trm of
      [] => (cond-trace ctxt tuple-inst-tac: no instantiation found; no-tac st)
    | inst => let
        val - = cond-trace ctxt (tuple-inst-tac: ^@{make-string} inst);
        in PRIMITIVE (Drule.infer-instantiate ctxt inst) st end
      end) i;
```

```
fun tuple-insts ctxt thm =
  let
    val max-idx = Thm.maxidx-of-cterm (Thm.cprop-of thm)
  in
    case calc-insts ctxt max-idx (Thm.prop-of thm) of
      [] => thm
    | inst => let
        val - = cond-trace ctxt (tuple-inst-tac: ^@{make-string} inst);
        in Drule.infer-instantiate ctxt inst thm end
      end;
```

(* case-prod $?f$ (Pair ...) ... *)

```
fun is-case-prod-app-Pair t =
  case Term.strip-comb t of
    (Const (@{const-name case-prod}, -), - :: maybe-pair :: -) => (case Term.strip-comb
  maybe-pair of
    (Const (@{const-name Pair}, -), -) => true | - => false)
  | - => false
```

(* apply beta reduction for Pairs *)

```
fun mksimps ctxt thm =
  let
    val thm' = (if exists-subterm is-case-prod-app-Pair (Thm.prop-of thm)
    then
      let
```

```

      val thm' = Simplifier.simplify (put-simpset HOL-ss ctxt addsimps @ {thms
Product-Type.prod.case}) thm
      val - = cond-trace ctxt (Tuple-Tools.mksimps: ^ @ {make-string} thm')
      in thm' end (* (%(x,y). f x y) (a,b) = f a b *)
    else thm)
  in (Simpdata.mksimps Simpdata.mksimps-pairs) ctxt thm'
end

val SPLIT-simproc =
  Simplifier.make-simproc @ {context} {name = SPLIT-simproc, identifier = [],
  lhs = [Proof-Context.read-term-pattern @ {context} PROP SPLIT ?P,
    Proof-Context.read-term-pattern @ {context} SPLIT ?P],
  proc = fn - => fn ctxt => fn ct =>
    let
      fun get-tuple-arity (Const (@ {const-name SPLIT}, -))$
        (Const (@ {const-name Pure.all}, -) $ (Abs (-, T, -))) =
        length (HOLogic.flatten-tuple T T)
      | get-tuple-arity - = 0;
      val t = Thm.term-of ct
      val n = get-tuple-arity t
    in
      if n >= 2
      then
        let
          fun is-case-prod (Const (@ {const-name case-prod}, -)) = true
          | is-case-prod - = false

          fun guess-names t = case get-first-subterm is-case-prod t of
            SOME t' => map fst (strip-case-prod t') |> filter-out (fn n => n
= strip-uu)
            | - => []

          val split-thm = get-split-tupled-all-thm ctxt n
          |> Drule.rename-bvars (liberal-zip (map mk-el-name (1 upto n))
(guess-names t))

          val case-eq-thm = get-tuple-case-eq-thm ctxt n

          val conv = Conv.rewr-conv @ {thm SPLIT-def}
            then-conv Conv.rewr-conv split-thm
            then-conv (Conv.bottom-conv (K (Conv.try-conv (Conv.rewr-conv
case-eq-thm)))) ctxt)

          in SOME (conv ct) end
          else SOME @ {thm SPLIT-def}
        end
      }
  fun asserts [] x = x

```

```

| asserts ((cond, msg)::xs) x = if cond x then asserts xs x else error (msg x)

structure Intlisttab = Table(type key = int list val ord = list-ord int-ord);

fun comb-product [] = [[]]
  | comb-product (xs::yss) =
    let
      val prods = comb-product yss
      val prodss = map (fn x => map (fn ps => x::ps) prods) xs
    in
      flat prodss
    end

val - = @{assert} (comb-product [[1,2],[3,4],[5,6]] =
  [[1, 3, 5], [1, 3, 6], [1, 4, 5], [1, 4, 6], [2, 3, 5], [2, 3, 6], [2, 4, 5], [2, 4, 6]])

(*
 * Generates a custom rule with the tuple arity of an abstracted variable. The
 * position of the
 * abstraction is specified in the pattern (which is also used as simproc pattern).
 * Every real variable
 * in that pattern (besides the dummy pattern -) will be considered as an abstraction
 * to split. E.g.
 * thm:  $\bigwedge v. P (?y v) \implies \text{seq } ?x (\lambda x. ?y x)$  with pattern  $\text{seq } - ?Y$ 
 * When this is applied to a term  $\text{seq } x (\lambda(a,b,c). ?y a b c)$  a rule of arity 3 will be
 * generated:
 *  $\bigwedge a b c. P (?y a b c) \implies \text{seq } ?x (\lambda(a, b, c). ?y a b c)$  which can be matched with
 * the term.
 *)
fun split-rule-simproc ctxt name pattern rule =
  let
    val pat = Proof-Context.read-term-pattern ctxt pattern
    fun dest-eq thm =
      let
        val @{term-pat Trueprop (?lhs = ?rhs)} = thm |> Thm.concl-of
        in (lhs, rhs) end
  in
    val lhs = rule |> dest-eq |> fst

    val (tenv, term-env) = Pattern.match (Proof-Context.theory-of ctxt) (pat, lhs)
  (Vartab.empty, Vartab.empty)
  fun get-vars env (Abs (x, T, t)) = get-vars (env @ [(x, T)]) t
    | get-vars env t =
      (case Term.strip-comb t of
        (Var x, args) => (env, x, args) :: flat (map (get-vars env) args)
        | (-, args) => flat (map (get-vars env) args))
  end

```

```

fun msg s x = split-rule-simproc: ^ s

fun is-dummy ((n, -) = (n = -dummy-))
val rule-subterms = Vartab.dest term-env |> filter-out (is-dummy o fst) |> map
(snd o snd)
val rule-vars = flat (map (get-vars []) rule-subterms)
|> map (asserts [
  (fn (env, var, args) => not (null env), fn - => matched subterm for
splitting has to be abstraction, use dummy pattern '-' to skip subterms),
  (fn (env, var, args) => not (null args), fn (env, var, args) =>
  (Variable ' ^ Term.string-of-vname' (fst var) ^
  ' has to be applied to bound variable, about to be splitted)),
  (fn (env, var, args) => hd args = Bound (length env - 1), fn (env, var,
args) =>
  (first argument of variable ' ^ Term.string-of-vname' (fst var) ^
  ' expected to be bound variable ' ^ (fst (hd env)) ^ '))
])
|> map (#2)

fun split-rule arities =
let
  val (names, ctxt') = Variable.add-fixes (map (fst o fst) rule-vars) ctxt
  fun mk-inst ((var, T), (n, name)) i = ((var, Thm.cterm-of ctxt (mk-case-prod-fun
(name, range-type T) i n)), i+n)
  val insts = fold-map mk-inst (rule-vars ~ (arities ~ names)) 1 |> fst

  val [rule'] = Drule.infer-instantiate ctxt' insts rule
  |> Simplifier.simplify (put-simpset HOL-basic-ss ctxt' addsimps @ {thms
case-prod-out})
  |> mk-meta-eq
  |> single
  |> Proof-Context.export ctxt' ctxt

in
  rule'
end
val (split-rule, handler) = Fun-Cache.create-handler (Binding.map-name (fn
name => name ^ -cache) name)
(fn tab => @ {make-string} (Intlisttab.dest tab))
Intlisttab.empty Intlisttab.lookup Intlisttab.update split-rule
fun split-rule-names arity-names =
let
  val arities = map fst arity-names
  val names = map snd arity-names
  val rule = split-rule arities
  fun sum ns = fold (fn n => fn m => n + m) ns 0
  val bound-names = map (mk-el-name) (1 upto (sum arities))
  val renamings = (bound-names ~ flat (names)) |> filter-out (fn (-, n) =>

```

```

n = strip-uu)
  in
    Drule.rename-bvars renamings rule
  end

fun trace-cache-info () =
  tracing (@{make-string} handler)

(* populate

cache, and as a side-effect check if splitting works *)
val - = map split-rule (comb-product (map (K (1 upto 3)) rule-vars))
val - = trace-cache-info ()
in
  Simplifier.make-simproc ctxt {name = Binding.name-of name, identifier = [],
    lhs = [pat],
    proc = fn - => fn ctxt => fn ct =>
      let
        val t = Thm.term-of ct
        val (tenv, term-env) = Pattern.match (Proof-Context.theory-of ctxt) (pat,
t) (Vartab.empty, Vartab.empty)
        val subterms = Vartab.dest term-env |> filter-out (is-dummy o fst) |> map
(snd o snd)
        val arity-names = map ((fn vars => (length vars, map fst vars)) o
strip-case-prod) subterms
        val - = asserts [(fn - => length arity-names = length rule-vars, fn - =>
split-rule-simproc: unexpected number of matched subterms)]
      in
        if exists (fn (n,-) => n > 1) arity-names
        then
          let
            val splitted-rule = split-rule-names arity-names
            in SOME splitted-rule end
          else NONE
        end
      }
end

fun tuple-inst-simp-tac ctxt i =
  tuple-inst-tac ctxt i THEN
  simp-tac (put-simpset HOL-ss ctxt addsimps @{thms Product-Type.prod.case}) i

fun tuple-refl-tac ctxt i =
  resolve-tac ctxt @{thms refl} i
  ORELSE (
    tuple-inst-simp-tac ctxt i THEN
    resolve-tac ctxt @{thms refl} i
  )

```

```
fun split-rule-simprocs ctxt = map (fn (name, rule, pattern) => split-rule-simproc
  ctxt name pattern rule)
```

```
fun gen-split-rule ctxt0 name-arities thm =
  let
    val ctxt = Variable.set-body false ctxt0
    val case-prod-eta-contract-thm = @{thm case-prod-eta-contract}
    val case-prod-conv = Conv.bottom-conv (
      K (Conv.try-conv (Conv.rewr-conv case-prod-eta-contract-thm)))
    val name-arities = name-arities |> sort-by fst
    val names = map fst name-arities
    val arities = map snd name-arities
    fun eta-contract ctxt thm = Conv.fconv-rule (case-prod-conv ctxt) thm
    val vars = Term.add-vars (Thm.prop-of thm) [] |>
      filter (fn ((n,-),-) => member (op =) names n) |> distinct (op =) |> sort-by
      (fst o fst)
    val names' = map (fst o fst) vars
    val insts = ((names' ~~ arities) ~~ vars)
      |> map (fn ((f, n), (var, T)) =>
        (var, mk-tuple-or-case-prod f T n))
    val new-names = [] |> fold Term.add-frees (map snd insts) |> map fst
    val (new-names', ctxt') = Variable.add-fixes new-names ctxt
    val - = @{assert} (new-names = new-names')
    val (-, ctxt'') = ctxt' |> Variable.import-terms false (map snd insts)
      |> apsnd (Context-Position.set-visible false)
    val insts = map (apsnd (Thm.cterm-of ctxt'')) insts
    val split-arities = arities |> filter (fn n => n > 1)
    val splits = map (get-split-tupled-all-thm ctxt) split-arities
    val case-eqs = map (get-tuple-case-eq-thm ctxt) split-arities
  in
    thm
    |> Drule.infer-instantiate ctxt'' insts
    |> Simplifier.asm-full-simplify (put-simpset HOL-basic-ss ctxt'' addsimps
      (@{thms case-prod-out} @ case-eqs @ splits))
    |> eta-contract ctxt''
    |> single
    |> Proof-Context.export ctxt'' ctxt
    |> hd
    |> Drule.zero-var-indexes
  end
```

```
fun split-rule ctxt names thm n =
  if n <= 1 then thm
  else gen-split-rule ctxt (map (fn name => (name, n)) names) thm
```

```
fun split-rule-bin ctxt names thm bin =
  split-rule ctxt names thm (Bin.int-of-bin bin)
```



```

val case-prod-conv = mk-meta-eq @ {thm case-prod-conv}

fun print-conv msg (ct:cterm) =
  let
    val - = tracing (msg ^ : ^ @ {make-string} ct);
  in Conv.all-conv ct end

fun bounded-top-conv i conv ctxt ct =
  if i <= 0 then Conv.all-conv ct
  else
    (conv ctxt then-conv Conv.sub-conv (bounded-top-conv (i - 1) conv) ctxt) ct;

fun bounded-bottom-conv i conv ctxt ct =
  if i <= 0 then Conv.all-conv ct
  else
    (Conv.sub-conv (bounded-bottom-conv (i - 1) conv) ctxt then-conv conv ctxt)
  ct;

fun bounded-top-rewrs-conv i rewrs = bounded-top-conv i (K (Conv.try-conv (Conv.rewrs-conv
rewrs)));

fun bounded-bottom-rewrs-conv i rewrs = bounded-bottom-conv i (K
(Conv.try-conv (Conv.rewrs-conv rewrs)));

fun eta-expand-tupled-conv ctxt ct =
  let
    val arity-of = HOLogic.flatten-tupleT #> length
    val domT = Thm.typ-of-cterm ct |> domain-type
    val arity = domT |> arity-of
    val name-types = strip-case-prod' (Thm.term-of ct)
    val case-prod-arity = length name-types
    val renamings = map (fn (n, T) => if arity-of T = 1 then SOME n else NONE)
  name-types
    val base-conv =
      if case-prod-arity > 0 andalso case-prod-arity < arity then
        let
          val eta-contract-rule = get-eta-tupled-eq-thm ctxt case-prod-arity
            |> Thm.symmetric
          in fn ct => Conv.rewr-conv eta-contract-rule ct handle CTERM - =>
Conv.all-conv ct end
        else Conv.all-conv
    val conv =
      if arity > 1 andalso case-prod-arity < arity then
        base-conv then-conv

```

```

      Conv.rewr-conv (Drule.rename-bvars' renamings (get-eta-tupled-eq-thm
ctxt arity)) then-conv
      bounded-bottom-rewrs-conv (arity * 2) [case-prod-conv] ctxt
      else Conv.all-conv
    in
      conv ct
    end
  handle Match => Conv.all-conv ct

fun eta-expand-tuple ctxt = eta-expand-tupled-conv ctxt #> Thm.rhs-of

end
>

```

```

simproc-setup ETA-TUPLED (⟨ETA-TUPLED f⟩) = ⟨fn - => fn ctxt => fn ct
=>
  let
    val arity = Thm.typ-of-cterm ct |> domain-type |> HOLogic.flatten-tupleT |>
length
  in
    if arity <= 1 then
      SOME @ {thm ETA-TUPLED-def}
    else
      SOME (@ {thm ETA-TUPLED-trans} OF [Tuple-Tools.get-eta-tupled-eq-thm
ctxt arity])
    end
  end
declare [[simproc del: ETA-TUPLED]]

```

Set up the theorem cache. We could use "dynamic programming" here and rewrite "thm n" with one step of "thm (n - 1)" and *case-prod* = $(\lambda c p. c (fst p) (snd p))$

```

setup ⟨
Context.theory-map (Tuple-Tools.Data.map (fn - => map (fn i =>
(Thm.trim-context (Tuple-Tools.mk-tuple-up-eq-thm @ {context} i),
Thm.trim-context (Tuple-Tools.mk-split-tupled-all-thm @ {context} i),
Thm.trim-context (Tuple-Tools.mk-tuple-case-eq-thm @ {context} i),
Thm.trim-context (Tuple-Tools.mk-eta-tupled-eq-thm @ {context} i))) (2 upto
50)))
>

```

```

attribute-setup split-tuple = ⟨
let
  val parse-arity = Args.$$$ arity |-- Args.colon |-- Parse.nat
  val parse-names = Parse.and-list1 Parse.short-ident
in
  Scan.lift (parse-names -- parse-arity) >> (fn (names, arity) =>
let
  in
    Thm.rule-attribute [] (fn context => fn thm =>

```

```

    Tuple-Tools.split-rule (Context.proof-of context) names thm arity)
  end)
end
>

```

```

thm HOL.ext [split-tuple f and g arity: 3]

```

```

ML-val <
val test = @{pattern  $\wedge a b c s. f a b c \equiv ?c (a,b,k) s$ }
val x = Tuple-Tools.calc-inst' @{context} 2 test
>

```

Especially in the case of congruence rules we prefer the more restrictive instantiation, where the freshly introduced variable does not depend on 'non-bound' tuple components

```

ML-val <
val test = @{pattern  $\wedge a b c s. f a b c \equiv ?c (a,b,k) s$ }
val x = Tuple-Tools.calc-inst @{context} 2 test
>

```

```

ML-val <
val thm = Tuple-Tools.get-tuple-up-eq-thm @{context} 1
>

```

```

ML-val <
val thm = Tuple-Tools.get-tuple-up-eq-thm @{context} 0
>

```

```

ML-val <
val thm = Tuple-Tools.get-tuple-up-eq-thm @{context} 3
>

```

```

ML-val <
val thm = Tuple-Tools.get-split-tupled-all-thm @{context} 30
>

```

```

ML-val <
val thm = Tuple-Tools.get-split-tupled-all-thm @{context} 2
>

```

```

ML-val <
val thm = Tuple-Tools.get-tuple-case-eq-thm @{context} 2
>

```

```

ML-val <
val thm = Tuple-Tools.get-eta-tupled-eq-thm @{context} 2
>

```

>

declare $[[simp-trace=false]]$

Variable names should be preserved here, since all abstractions stay in place.

```
lemma  $\bigwedge r. (\lambda(x,y). (x::nat) < y \wedge y < 200) r$   
  apply (tactic  $\langle$   
    simp-tac (put-simpset HOL-basic-ss @{context} addsimprocs [Tuple-Tools.split-tupled-all-simproc,  
    Tuple-Tools.tuple-case-simproc]  
    delsimps @{thms Product-Type.prod.case Product-Type.case-prod-conv}) 1  
   $\rangle$ )  
  oops
```

Variable names are not preserved due to eta contraction: $(\lambda(x, y). x < y) = (\lambda(x, y). x < y)$

```
lemma  $\bigwedge r. (\lambda(x,y). (x::nat) < y) r$   
  apply (tactic  $\langle$   
    simp-tac (put-simpset HOL-basic-ss @{context} addsimprocs [Tuple-Tools.split-tupled-all-simproc,  
    Tuple-Tools.tuple-case-simproc]  
    delsimps @{thms Product-Type.prod.case Product-Type.case-prod-conv}) 1  
   $\rangle$ )  
  oops
```

```
lemma  $(\lambda(x,y,p,z,f). (x::nat) < y) (a,b,c) = (case\ c\ of\ (p,z,f) \Rightarrow a < b)$   
  apply (tactic  $\langle$   
    simp-tac (put-simpset HOL-basic-ss @{context} addsimprocs [Tuple-Tools.split-tupled-all-simproc,  
    Tuple-Tools.tuple-case-simproc]  
    delsimps @{thms Product-Type.prod.case Product-Type.case-prod-conv}) 1  
   $\rangle$ )  
  done
```

```
lemma  $\bigwedge r. (\lambda(x,y,p,z,f). (p::nat) < f) r$   
  apply (tactic  $\langle$   
    simp-tac (put-simpset HOL-basic-ss @{context} addsimprocs [Tuple-Tools.split-tupled-all-simproc,  
    Tuple-Tools.tuple-case-simproc]  
    delsimps @{thms Product-Type.prod.case Product-Type.case-prod-conv}) 1  
   $\rangle$ )  
  oops
```

```
lemma  $(\lambda(x,y,p). (x::nat) < y) (a,b,c,d,e) = (a < b)$   
  apply (tactic  $\langle$   
    simp-tac (put-simpset HOL-basic-ss @{context} addsimprocs [Tuple-Tools.split-tupled-all-simproc,  
    Tuple-Tools.tuple-case-simproc]  
    delsimps @{thms Product-Type.prod.case Product-Type.case-prod-conv}) 1  
   $\rangle$ )  
  done
```

ML-val \langle

```

val test = @{\pattern  $\wedge a b c d e s. f a b c \equiv ?c a b (c,d,e) s$ }
val x = Tuple-Tools.calc-inst @{\context} 2 test
>

```

```

ML-val <
val test = @{\pattern  $\wedge a b c s. P x \implies f a b c \equiv ?c (a,b,c) s$ }
val x = Tuple-Tools.calc-inst @{\context} 2 test
>

```

```

ML-val <
val test = @{\pattern  $\wedge a b c s. P x =simp=> f a b c \equiv ?c (a,b,c) s$ }
val x = Tuple-Tools.calc-inst @{\context} 2 test
>

```

```

ML-val <
val test = @{\pattern  $\wedge a b c s. f a b c \equiv ?c (a,b,k) s$ }
val x = Tuple-Tools.calc-inst @{\context} 2 test
>

```

```

ML-val <
val test = @{\pattern  $\wedge a b c s. f a b c \equiv ?c (a,b,k) s$ }
val x = Tuple-Tools.calc-inst @{\context} 2 test
>

```

```

ML-val <
val test = @{\pattern  $\wedge a b c s. f a b c \equiv ?c (a,x+y,x + y + z) s$ }
val x = Tuple-Tools.calc-inst @{\context} 2 test
>

```

```

ML-val <
val test = @{\pattern  $\wedge a b c s. f a b c \equiv ?c (a,x+y,x + y + z) s$ }
val x = Tuple-Tools.calc-inst @{\context} 2 test
>

```

```

ML-val <
val test = @{\pattern  $(\wedge a b c s. P x \implies f a b c \equiv ?c (a,b,c) s) \implies (\wedge a b c s. Q x)$ }
val x = Tuple-Tools.calc-inst @{\context} 2 test
>

```

```

lemma SPLIT ( $\wedge r. ((\lambda z. Q z \wedge (Q z \longrightarrow P z)) r)$ )
   $\equiv$  XXX
  apply (tactic <
simp-tac ( @{\context}
addsimprocs [Tuple-Tools.SPLIT-simproc, Tuple-Tools.tuple-case-simproc]

```

```

|> Simplifier.add-cong @{thm SPLIT-cong}
|> Simplifier.add-cong @{thm conj-cong} 1
)
oops

lemma PROP SPLIT ( $\bigwedge r. ((\lambda(x,y,z). y < z \wedge z=s) r) \implies P r$ )
 $\equiv (\bigwedge x y z. y < z \wedge z = s \implies P (x, y, s))$ 
apply (tactic <
asm-full-simp-tac (put-simpset HOL-basic-ss @{context}
addsimprocs [Tuple-Tools.SPLIT-simproc, Tuple-Tools.tuple-case-simproc] |> Simplifier.add-cong @{thm SPLIT-cong}) 1
)
done

schematic-goal ETA-TUPLED ( $\lambda x::('a \times 'b \times 'c). f x = ?XXX$ )
supply [[simp-trace]]
thm Product-Type.cond-case-prod-eta
supply [[simproc add: ETA-TUPLED]]
apply (simp)
done

context
fixes f::('a  $\times$  'b  $\times$  'c  $\times$  'd)  $\Rightarrow$  nat
fixes c::('a  $\times$  'b  $\times$  'c  $\times$  'd)  $\Rightarrow$  's  $\Rightarrow$  bool
begin

ML-val <
val t1 = Tuple-Tools.eta-expand-tupled-conv @{context} @{cterm f}
val t2 = Tuple-Tools.eta-expand-tupled-conv @{context} @{cterm  $\lambda(a, b). f (a, b)$ }
val t3 = Tuple-Tools.eta-expand-tupled-conv @{context} @{cterm  $\lambda(a, b, c). f (a, b, c)$ }

val c1 = Tuple-Tools.eta-expand-tupled-conv @{context} @{cterm c}
val c2 = Tuple-Tools.eta-expand-tupled-conv @{context} @{cterm  $\lambda(a, b) s. c (a, b) s$ }
val d1 = Tuple-Tools.eta-expand-tupled-conv @{context} @{cterm  $\lambda(a, (b::nat \times nat)) s. a < 2$ }
val d2 = Tuple-Tools.eta-expand-tupled-conv @{context} @{cterm  $\lambda(a, (b::nat \times nat \times nat)) s. a < 2$ }
val d3 = Tuple-Tools.eta-expand-tupled-conv @{context} @{cterm  $\lambda(a, (b::nat \times nat \times nat \times nat)) s. a < 2$ }
>

end

end

```

```

theory Subgoal-Methods
imports Main
begin
ML ‹
signature SUBGOAL-METHODS =
sig
  val fold-subgoals: Proof.context -> bool -> thm -> thm
  val unfold-subgoals-tac: Proof.context -> tactic
  val distinct-subgoals: Proof.context -> thm -> thm
end;

structure Subgoal-Methods: SUBGOAL-METHODS =
struct

fun max-common-prefix eq (ls :: lss) =
  let
    val ls' = tag-list 0 ls;
    fun all-prefix (i, a) =
      forall (fn ls' => if length ls' > i then eq (a, nth ls' i) else false) lss
    val ls'' = take-prefix all-prefix ls'
    in map snd ls'' end
  | max-common-prefix [] = [];

fun push-outer-params ctxt th =
  let
    val ctxt' = ctxt
    |> Simplifier.empty-simpset
    |> Simplifier.add-simp Drule.norm-hhf-eq;
  in
    Conv.fconv-rule
    (Raw-Simplifier.rewrite-cterm (true, false, false) (K (K NONE)) ctxt') th
  end;

fun fix-schematics ctxt raw-st =
  let
    val ((schematic-types, [st']), ctxt1) = Variable.importT [raw-st] ctxt;
    val ((-, inst), ctxt2) =
      Variable.import-inst true [Thm.prop-of st'] ctxt1;

    val schematic-terms = Vars.map (K (Thm.cterm-of ctxt2)) inst;
    val schematics = (schematic-types, schematic-terms);

  in (Thm.instantiate schematics st', ctxt2) end

val strip-params = Term.strip-all-vars;
val strip-prems = Logic.strip-imp-prems o Term.strip-all-body;

```

```

val strip-concl = Logic.strip-imp-concl o Term.strip-all-body;

fun fold-subgoals ctxt prefix raw-st =
  if Thm.nprems-of raw-st < 2 then raw-st
  else
    let
      val (st, inner-ctxt) = fix-schematics ctxt raw-st;

      val subgoals = Thm.prems-of st;
      val paramss = map strip-params subgoals;
      val common-params = max-common-prefix (eq-snd (op =)) paramss;

      fun strip-shift subgoal =
        let
          val params = strip-params subgoal;
          val diff = length common-params - length params;
          val prems = strip-prems subgoal;
          in map (Term.incr-boundvars diff) prems end;

      val prems = map (strip-shift) subgoals;

      val common-prems = max-common-prefix (op aconv) prems;

      val common-params = if prefix then common-params else [];
      val common-prems = if prefix then common-prems else [];

      fun mk-concl subgoal =
        let
          val params = Term.strip-all-vars subgoal;
          val local-params = drop (length common-params) params;
          val prems = strip-prems subgoal;
          val local-prems = drop (length common-prems) prems;
          val concl = strip-concl subgoal;
          in Logic.list-all (local-params, Logic.list-implies (local-prems, concl)) end;

      val goal =
        Logic.list-all (common-params,
          (Logic.list-implies (common-prems, Logic.mk-conjunction-list (map mk-concl
            subgoals))));

      val chyp = Thm.cterm-of inner-ctxt goal;

      val (common-params', inner-ctxt') =
        Variable.add-fixes (map fst common-params) inner-ctxt
        |>> map2 (fn (-, T) => fn x => Thm.cterm-of inner-ctxt (Free (x, T)))
        common-params;

```



```

fun try-dest rule =
  try (fn () => (@{thm conjunctionD1} OF [rule], @{thm conjunctionD2}
OF [rule])) ();

fun solve-headgoal rule =
  let
    val rule' = rule
      |> Drule.forall-intr-list common-params'
      |> push-outer-params inner-ctxt';
  in
    (fn st => Thm.implies-elim st rule')
  end;

fun solve-subgoals rule' st =
  (case try-dest rule' of
   SOME (this, rest) => solve-subgoals rest (solve-headgoal this st)
  | NONE => solve-headgoal rule' st);

val rule = Drule.forall-elim-list common-params' (Thm.assume chyp);
in
  st
  |> push-outer-params inner-ctxt
  |> solve-subgoals rule
  |> Thm.implies-intr chyp
  |> singleton (Variable.export inner-ctxt' ctxt)
end;

fun distinct-subgoals ctxt raw-st =
  let
    val (st, inner-ctxt) = fix-schematics ctxt raw-st;
    val subgoals = Drule.cprems-of st;
    val atomize = Conv.fconv-rule (Object-Logic.atomize-prems inner-ctxt);

    val rules =
      map (atomize o Raw-Simplifier.norm-hhf inner-ctxt o Thm.assume) subgoals
      |> sort (int-ord o apply2 Thm.nprems-of);

    val st' = st
      |> ALLGOALS (fn i =>
        Object-Logic.atomize-prems-tac inner-ctxt i THEN solve-tac inner-ctxt rules
      i)
      |> Seq.hd;

    val subgoals' = subgoals
      |> inter (op aconv) (Thm.chyps-of st')
      |> distinct (op aconv);
  in
    Drule.implies-intr-list subgoals' st'
  |> singleton (Variable.export inner-ctxt ctxt)
end;

```

```

end;

(* Variant of filter-prems-tac that recovers premise order *)
fun filter-prems-tac' ctxt pred =
  let
    fun Then NONE tac = SOME tac
      | Then (SOME tac) tac' = SOME (tac THEN' tac');
    fun thins H (tac, n, i) =
      (if pred H then (tac, n + 1, i)
       else (Then tac (rotate-tac n THEN' eresolve-tac ctxt [thin-rl]), 0, i + n));
  in
    SUBGOAL (fn (goal, i) =>
      let val Hs = Logic.strip-assums-hyp goal in
        (case fold thins Hs (NONE, 0, 0) of
          (NONE, -, -) => no-tac
          | (SOME tac, -, n) => tac i THEN rotate-tac (~ n) i)
        end)
      end;
  end;

fun trim-prems-tac ctxt rules =
  let
    fun matches (prem, rule) =
      let
        val ((-, prem'), ctxt') = Variable.focus NONE prem ctxt;
        val rule-prop = Thm.prop-of rule;
      in is-some (Unify.matcher (Context.Proof ctxt') [rule-prop] [prem']) end;
  in filter-prems-tac' ctxt (not o member matches rules) end;

val adhoc-conjunction-tac = REPEAT-ALL-NEW
  (SUBGOAL (fn (goal, i) =>
    if can Logic.dest-conjunction (Logic.strip-imp-concl goal)
    then resolve0-tac [Conjunction.conjunctionI] i
    else no-tac));

fun unfold-subgoals-tac ctxt =
  TRY (adhoc-conjunction-tac 1)
  THEN (PRIMITIVE (Raw-Simplifier.norm-hhf ctxt));

val - =
  Theory.setup
  (Method.setup @ {binding fold-subgoals}
    (Scan.lift (Args.mode prefix) >> (fn prefix => fn ctxt =>
      SIMPLE-METHOD (PRIMITIVE (fold-subgoals ctxt prefix))))
    lift all subgoals over common premises/params #>
  Method.setup @ {binding unfold-subgoals}
    (Scan.succeed (fn ctxt => SIMPLE-METHOD (unfold-subgoals-tac ctxt)))
    recover subgoals after folding #>
  Method.setup @ {binding distinct-subgoals}

```

```

    (Scan.succeed (fn ctxt => SIMPLE-METHOD (PRIMITIVE (distinct-subgoals
    ctxt))))
    trim all subgoals to be (logically) distinct #>
    Method.setup @{binding trim}
    (Attrib.thms >> (fn thms => fn ctxt =>
    SIMPLE-METHOD (HEADGOAL (trim-prems-tac ctxt thms))))
    trim all premises that match the given rules);

end;
>

end

```

4.2 Tools for intro-rule based term synthesis *synthesize-rules*

```

theory Synthesize
imports
  Tuple-Tools
  AutoCorres-Utils
  MkTermAntiquote
keywords
  synthesize-rules::thy-decl and
  add-synthesize-pattern::thy-decl % ML and
  print-synthesize-rules::diag
begin

```

ML-file *<synthesize-rules.ML>*

4.2.1 Commands

```

ML <
val - = Outer-Syntax.local-theory command-keyword <synthesize-rules>
  declare named collection of synthesize rules (pattern indexed intro rules)
  (Parse.and-list1 Parse.binding >>
    (fold (fn b => (snd o Synthesize-Rules.declare b))))
>

ML <
val - =
  Outer-Syntax.local-theory command-keyword <add-synthesize-pattern>
  add pattern schemes via ML or declarations
  (Parse.and-list1 Parse.binding :|-- (fn rules-names =>
    (keyword <where> |-- Synthesize-Rules.pattern-decls >> (fn decls =>
      fold (fn rules-name => Synthesize-Rules.add-pattern-decls rules-name
decls) rules-names))

```

```

    ||
    (keyword <=> |-- Parse.ML-source >> (fn src =>
      fold (fn rules-name => Synthesize-Rules.add-pattern-ml rules-name src)
        rules-names))))
  >

```

```

ML <
  val - = Outer-Syntax.local-theory command-keyword <print-synthesize-rules>
    print named collection of synthesize rules (optionally matching a given term)
    (Parse.name-position -- Scan.option Parse.term >>
      (fn (name, term-opt) => Synthesize-Rules.print-rules-cmd name term-opt))
  >

```

4.2.2 ML Antiquotations

```

ML < Theory.setup
  (ML-Antiquotation.inline-embedded binding <synthesize-rules-name>
    (Args.context -- Scan.lift Parse.embedded-position >>
      (fn (ctxt, name) => ML-Syntax.print-string (Synthesize-Rules.check (Context.Proof
        ctxt) name |> fst))))
  >

```

```

ML < Theory.setup
  (ML-Antiquotation.value-embedded binding <synthesize-rules>
    (Args.context -- Scan.lift Parse.embedded-position >>
      (fn (ctxt, name) => Synthesize-Rules.get-rules ML-context ^
        ML-Syntax.print-string (Synthesize-Rules.check (Context.Proof ctxt) name
        |> fst))))
  >

```

```

ML <
  Theory.setup
  (ML-Antiquotation.inline @{binding mk-synthesize-pattern}
    ((Args.context -- Scan.lift Parse.embedded-inner-syntax --
      (Scan.optional (Scan.lift ((Synthesize-Rules.comma-list Args.name))) []))
      >> (fn ((ctxt, pattern-str), synth-args) => Synthesize-Rules.gen-pattern-fun
        ctxt (pattern-str, synth-args) )))
  >

```

4.2.3 Attributes

```

attribute-setup synthesize-rule = <
  let
    val priority = Args.$$$ priority |-- Args.colon |-- Parse.int
    val opt-priority = Scan.optional (Scan.lift (priority)) 10
    val only-schematic-goal = Args.$$$ only-schematic-goal >> (fn - => true)
    val opt-only-schematic-goal = Scan.optional (Scan.lift only-schematic-goal) false
    val del = Args.del >> (fn - => true)
    val opt-del = Scan.optional (Scan.lift del) false
    val split = Args.$$$ split |-- Args.colon |-- Parse.and-list Parse.short-ident

```

```

    val opt-split = Scan.optional (Scan.lift (split)) []
  in
    Scan.lift (Parse.and-list1 Parse.name-position)
      -- opt-priority -- opt-split -- opt-only-schematic-goal -- opt-del >> (fn
      (((rules-names, prio), splits), only-schematic-goal), del) =>
      let
        in
          Thm.declaration-attribute (fn thm => fn context =>
            let
              val ctxt = Context.proof-of context
              val thm-binding1 = Utils.guess-binding-of-thm ctxt thm
              val thm-binding2 = if Binding.is-empty thm-binding1 then Binding.make
              (??, (snd (hd rules-names))) else thm-binding1
              val rules-names = map (Synthesize-Rules.check context #> fst) rules-names
              fun add-rule rules-name =
                if null splits
                then Synthesize-Rules.add-rule rules-name {only-schematic-goal = only-schematic-goal}
              thm-binding2 prio thm
                else Synthesize-Rules.add-split-rule rules-name {only-schematic-goal =
              only-schematic-goal} thm-binding2 prio splits thm #> snd
              fun del-rule rules-name =
                if null splits
                then Synthesize-Rules.del-rule rules-name {only-schematic-goal = only-schematic-goal}
              thm-binding2 prio thm
                else Synthesize-Rules.del-split-rule rules-name {only-schematic-goal =
              only-schematic-goal} thm-binding2 prio splits thm
              in
                context |> fold (if del then del-rule else add-rule) rules-names
              end)
            end)
          end
        end
      >

```

end

Chapter 5

Rule by Method

```
theory Rule-By-Method
imports
  Main
  HOL-Eisbach.Eisbach-Tools
begin

ML <
signature RULE-BY-METHOD =
sig
  val rule-by-tac: Proof.context -> {vars: bool, prop: bool} ->
    (Proof.context -> tactic) -> (Proof.context -> tactic) list -> Position.T
-> thm
end;

fun atomize ctxt = Conv.fconv-rule (Object-Logic.atomize ctxt);

fun fix-schematics ctxt raw-st =
  let
    val ((schematic-types, [st^]), ctxt1) = Variable.importT [raw-st] ctxt;
    fun certify-inst ((-, terms), ctxt) = (Vars.map (K (Thm.cterm-of ctxt)) terms,
      ctxt);
    val (schematic-terms, ctxt2) =
      Variable.import-inst true [Thm.prop-of st^] ctxt1
      |> certify-inst;
    val schematics = (schematic-types, schematic-terms);
  in (Thm.instantiate schematics st', ctxt2) end

fun curry-asm ctxt st = if Thm.nprems-of st = 0 then Seq.empty else
  let
    val prems = Thm.cprem-of st 1 |> Thm.term-of |> Logic.strip-imp-prems;
    val (thesis :: xs, ctxt') = Variable.variant-fixes (thesis :: replicate (length prems)
```

```

P) ctxt;

val rl =
  xs
  |> map (fn x => Thm.ctrm-of ctxt' (Free (x, propT)))
  |> Conjunction.mk-conjunction-balanced
  |> (fn xs => Thm.apply (Thm.apply @ {ctrm Pure.imp} xs) (Thm.ctrm-of
ctxt' (Free (thesis,propT))))
  |> Thm.assume
  |> Conjunction.curry-balanced (length prems)
  |> Drule.implies-intr-hyps

val rl' = singleton (Variable.export ctxt' ctxt) rl;

in Thm.bicompose (SOME ctxt) {flatten = false, match = false, incremented =
false}
  (false, rl', 1) 1 st end;

val drop-trivial-imp =
let
  val asm =
    Thm.assume (Drule.protect @ {cprop (PROP A ==> PROP A) ==> PROP A})
  |> Goal.conclude;

in
  Thm.implies-elim asm (Thm.trivial @ {cprop PROP A})
  |> Drule.implies-intr-hyps
  |> Thm.generalize (Names.empty, Names.make-set [A]) 1
  |> Drule.zero-var-indexes
end

val drop-trivial-imp' =
let
  val asm =
    Thm.assume (Drule.protect @ {cprop (PROP P ==> A) ==> A})
  |> Goal.conclude;

  val asm' = Thm.assume @ {cprop PROP P == Trueprop A}

in
  Thm.implies-elim asm (asm' COMP Drule.equal-elim-rule1)
  |> Thm.implies-elim (asm' COMP Drule.equal-elim-rule2)
  |> Drule.implies-intr-hyps
  |> Thm.permute-prems 0 ~ 1
  |> Thm.generalize (Names.empty, Names.make-set [A,P]) 1
  |> Drule.zero-var-indexes
end

fun atomize-equiv-tac ctxt i =

```

```

Object-Logic.full-atomize-tac ctxt i
THEN PRIMITIVE (fn st' =>
let val (-,[A,-]) = Drule.strip-comb (Thm.cprem-of st' i) in
if Object-Logic.is-judgment ctxt (Thm.term-of A) then st'
else error (Failed to fully atomize result:\n ^ (Syntax.string-of-term ctxt (Thm.term-of
A))) end)

```

```

structure Data = Proof-Data

```

```

(
  type T = thm list * bool;
  fun init - = ([],false);
);

```

```

val empty-rule-prems = Data.map (K ([],true));

```

```

fun add-rule-prem thm = Data.map (apfst (Thm.add-thm thm));

```

```

fun with-rule-prems enabled parse =
  Scan.state :|--- (fn context =>
  let
    val context' = Context.proof-of context |> Data.map (K ([Drule.free-dummy-thm],enabled))
    |> Context.Proof
  in Scan.lift (Scan.pass context' parse) end)

```

```

fun get-rule-prems ctxt =
  let
    val (thms,b) = Data.get ctxt
  in if (not b) then [] else thms end

```

```

fun zip-subgoal assume tac (ctxt,st : thm) = if Thm.nprems-of st = 0 then Seq.single
(ctxt,st) else

```

```

let
  fun bind-prems st' =
    let
      val prems = Drule.cprems-of st';
      val (asms, ctxt') = Assumption.add-assumes prems ctxt;
      val ctxt'' = fold add-rule-prem asms ctxt';
      val st'' = Goal.conclude (Drule.implies-elim-list st' (map Thm.assume prems));
    in (ctxt'',st'') end

```

```

fun defer-prems st' =
  let
    val nprems = Thm.nprems-of st';
    val st'' = Thm.permute-prems 0 nprems (Goal.conclude st');
  in (ctxt,st'') end;

```



```

in
  tac ctxt (Goal.protect 1 st)
  |> Seq.map (if assume then bind-prems else defer-prems) end

fun zip-subgoals assume tacs pos ctxt st =
let
  val nprems = Thm.nprems-of st;
  val - = nprems < length tacs andalso error (More tactics than rule assumptions
  ^ Position.here pos);
  val tacs' = map (zip-subgoal assume) (tacs @ (replicate (nprems - length tacs)
  (K all-tac)));
  val ctxt' = empty-rule-prems ctxt;
in Seq.EVERY tacs' (ctxt',st) end;

fun rule-by-tac' ctxt {vars,prop} tac asm-tacs pos raw-st =
let
  val (st,ctxt1) = if vars then (raw-st,ctxt) else fix-schematics ctxt raw-st;

  val ([x],ctxt2) = Proof-Context.add-fixes [(Binding.name Auto-Bind.thesisN,NONE,
  NoSyn)] ctxt1;

  val thesis = if prop then Free (x,propT) else Object-Logic.fixed-judgment ctxt2
  x;

  val cthesis = Thm.cterm-of ctxt thesis;

  val revcut-rl' = Thm.instantiate' [] ([NONE,SOME cthesis]) @ {thm revcut-rl};

  fun is-thesis t = Logic.strip-assums-concl t aconv thesis;

  fun err thm str = error (str ^ Position.here pos ^ \n ^
  (Pretty.string-of (Goal-Display.pretty-goal ctxt thm)));

  fun pop-thesis st =
let
  val prems = Thm.premis-of st |> tag-list 0;
  val (i,-) = (case filter (is-thesis o snd) prems of
  [] => err st Lost thesis
  | [x] => x
  | - => err st More than one result obtained);
in st |> Thm.permute-prems 0 i end

  val asm-st =
  (revcut-rl' OF [st])
  |> (fn st => Goal.protect (Thm.nprems-of st - 1) st)

```

```

    val (ctxt3,concl-st) = case Seq.pull (zip-subgoals (not vars) asm-tacs pos ctxt2
asm-st) of
      SOME (x,-) => x
    | NONE => error (Failed to apply tactics to rule assumptions. ^ (Position.here
pos));

    val concl-st-prepped =
      concl-st
    |> Goal.conclude
    |> (fn st => Goal.protect (Thm.nprems-of st) st |> Thm.permute-prems 0
~1 |> Goal.protect 1)

    val concl-st-result = concl-st-prepped
    |> (tac ctxt3
      THEN (PRIMITIVE pop-thesis)
      THEN curry-asm ctxt
      THEN PRIMITIVE (Goal.conclude #> Thm.permute-prems 0 1 #>
Goal.conclude))

    val result = (case Seq.pull concl-st-result of
      SOME (result,-) => singleton (Proof-Context.export ctxt3 ctxt) result
    | NONE => err concl-st-prepped Failed to apply tactic to rule conclusion:)

    val drop-rule = if prop then drop-trivial-imp else drop-trivial-imp'

    val result' = ((Goal.protect (Thm.nprems-of result -1) result) RS drop-rule)
    |> (if prop then all-tac else
      (atomize-equiv-tac ctxt (Thm.nprems-of result)
      THEN resolve-tac ctxt @_{thms Pure.reflexive} (Thm.nprems-of result)))
    |> Seq.hd
    |> Raw-Simplifier.norm-hhf ctxt

    in Drule.zero-var-indexes result' end;

fun rule-by-tac is-closed ctxt args tac asm-tacs pos raw-st =
  let val f = rule-by-tac' ctxt args tac asm-tacs pos
  in
    if is-closed orelse Context-Position.is-really-visible ctxt then SOME (f raw-st)
  else try f raw-st
  end

fun pos-closure (scan : 'a context-parser) :
  (('a * (Position.T * bool)) context-parser) = (fn (context,toks) =>
  let
    val (((context',x),tr-toks),toks') = Scan.trace (Scan.pass context (Scan.state --
scan)) toks;
    val pos = Token.range-of tr-toks;
    val is-closed = exists (fn t => is-some (Token.get-value t)) tr-toks
  in ((x,(Position.range-position pos, is-closed)),(context',toks')) end)

```

```
val parse-flags = Args.mode schematic -- Args.mode raw-prop >> (fn (b,b') =>
{vars = b, prop = b'})
```

```
fun tac m ctxt =
  NO-CONTEXT-TACTIC ctxt
  (Method.evaluate-runtime m ctxt []);
```

(* Declare as a mixed attribute to avoid any partial evaluation *)

```
fun handle-dummy f (context, thm) =
  case (f context thm) of SOME thm' => (NONE, SOME thm')
  | NONE => (SOME context, SOME Drule.free-dummy-thm)
```

```
val (rule-prems-by-method : attribute context-parser) = Scan.lift parse-flags :--
(fn flags =>
  pos-closure (Scan.repeat1
    (with-rule-prems (not (#vars flags)) Method.text-closure ||
      Scan.lift (Args.$$$ - >> (K Method.succeed-text)))) >>
    (fn (flags,(ms,(pos, is-closed))) => handle-dummy (fn context =>
      rule-by-tac is-closed (Context.proof-of context) flags (K all-tac) (map tac
ms) pos))
```

```
val (rule-concl-by-method : attribute context-parser) = Scan.lift parse-flags :-- (fn
flags =>
  pos-closure (with-rule-prems (not (#vars flags)) Method.text-closure)) >>
  (fn (flags,(m,(pos, is-closed))) => handle-dummy (fn context =>
    rule-by-tac is-closed (Context.proof-of context) flags (tac m) [] pos))
```

```
val - = Theory.setup
  (Global-Theory.add-thms-dynamic (@{binding rule-prems},
    (fn context => get-rule-prems (Context.proof-of context))) #>
  Attrib.setup @{binding #} rule-prems-by-method
    transform rule premises with method #>
  Attrib.setup @{binding @} rule-concl-by-method
    transform rule conclusion with method #>
  Attrib.setup @{binding atomized}
    (Scan.succeed (Thm.rule-attribute []
      (fn context => fn thm =>
        Conv.fconv-rule (Object-Logic.atomize (Context.proof-of context)) thm
        |> Drule.zero-var-indexes)))
  atomize rule)
>
```

experiment begin

```
ML <
  val [att] = @{attributes [@<erule thin-rl, cut-tac TrueI, fail>]}
  val k = Attrib.attribute @{context} att
```

```

    val - = case (try k (Context.Proof @{context}, Drule.dummy-thm)) of
      SOME - => error Should fail
    | - => ()
  ›

lemmas baz = [[@⟨erule thin-rl, rule revcut-rl[of P → P ∧ P], simp⟩]] for P

lemmas bazz[THEN impE] = TrueI[@⟨erule thin-rl, rule revcut-rl[of P → P ∧ P], simp⟩] for P

lemma Q → Q ∧ Q by (rule baz)

method silly-rule for P :: bool uses rule =
  (rule [[@⟨erule thin-rl, cut-tac rule, drule asm-rl[of P]⟩]])

lemma assumes A shows A by (silly-rule A rule: ⟨A⟩)

lemma assumes A[simp]: A shows A
apply (match conclusion in P for P ⇒
  ⟨rule [[@⟨erule thin-rl, rule revcut-rl[of P], simp⟩]]⟩)
done

end

end

theory Option-Scanner
imports ML-Record-Antiquotation
begin

  The purpose of the option scanner is to provide an interface to scan lists
  of options (key / value) for toplevel commands. As values may have different
  types the idea is that the collection of options is represented by an record of
  optional values. This record can be provided individually for each command.
  The initial option-record "empty" has every component assigned to NONE. To
  be able to specify all options in a single list, we represent a single parser
  / scanner for a field as an update function for the option-record, which is
  composed by the map-functions for the record fields. So we can preserve
  different types for parsing individual fields (and do not need an universal
  value type), while the specification list is a monotype.

  ML ‹

  signature OPTION-SCANNER =
    sig
      type 'opt parse = ('opt -> 'opt) parser
      type 'opt field = string * 'opt parse * 'opt parse option (* name, parser, default
      value parser *)
    end
  end

```

```

type ('opt, 'a) map-field = ('a option -> 'a option) -> ('opt -> 'opt)

val mk-opt: string -> 'a parser -> (('opt, 'a) map-field) -> 'a option ->
'opt field

val bool-opt: ('opt, bool) map-field -> bool option -> 'opt field
val int-opt: ('opt, int) map-field -> int option -> 'opt field
val real-opt: ('opt, real) map-field -> real option -> 'opt field
val string-opt: ('opt, string) map-field -> string option -> 'opt field
val path-opt: ('opt, string * Position.T) map-field -> (string * Position.T)
option -> 'opt field
val string-list-opt: ('opt, string list) map-field -> string list option -> 'opt
field
val path-list-opt: ('opt, (string * Position.T) list) map-field -> (string * Posi-
tion.T) list option -> 'opt field

val scan-bool: bool parser
val scan-list: 'a parser -> 'b parser -> 'c list parser -> 'c list parser
val scan-string-list: string list parser

val get-options: 'opt -> (string * ('opt field)) list -> 'opt parser

end

structure Option-Scanner:OPTION-SCANNER =
struct

type 'opt parse = ('opt -> 'opt) parser
type 'opt field = string * 'opt parse * 'opt parse option
type ('opt, 'a) map-field = ('a option -> 'a option) -> ('opt -> 'opt)

val equals = keyword <=>

val scan-bool = Parse.group (fn - => bool (true / false))
(Parse.reserved false >> K false || Parse.reserved true >> K true)

val scan-path = Scan.ahead Parse.not-eof -- Parse.path >> (fn (tok, name)
=> (name, Token.pos-of tok))

fun scan-list scan-open scan-close scan-entries =
(scan-open -- scan-close >> K [] ) ||
(scan-open |-- scan-entries --| scan-close)

val scan-string-list = scan-list keyword <[> keyword <]> (Parse.enum , Parse.embedded)
val scan-path-list = scan-list keyword <[> keyword <]> (Parse.enum , scan-path)

fun scan-field scan map-field = scan >> (map-field o K o SOME)
fun scan-default default map-field = Option.map (fn v => Scan.succeed (map-field
(K (SOME v)))) default

```

```

fun mk-opt (kind:string) scan map-field default = (kind, scan-field scan map-field,
scan-default default map-field)

fun bool-opt map-field default      = mk-opt bool scan-bool map-field default
fun int-opt map-field default       = mk-opt int Parse.int map-field default
fun real-opt map-field default      = mk-opt real Parse.real map-field default
fun string-opt map-field default    = mk-opt string Parse.embedded map-field
default
fun path-opt map-field default      = mk-opt path scan-path map-field default
fun string-list-opt map-field default = mk-opt string list scan-string-list map-field
default
fun path-list-opt map-field default = mk-opt path list scan-path-list map-field
default

fun scan-config-value eq scan-value maybe-default =
  (eq |-- Parse.!!! scan-value) ||
  (case maybe-default of SOME d => d | - => Scan.fail)

fun scan-option (name, (-, scan-value, maybe-default)) =
  Parse.reserved name -- Parse.!!! (scan-config-value equals scan-value maybe-default)

fun scan-options opts =
  let
    fun string-of-opt (name, (kind, parser, maybe-default)) = name ^ : ^ kind;
    val option-description = map (Pretty.str o string-of-opt) opts |> Pretty.list (
) |> Pretty.string-of;
    val msg = fn () => options ^ option-description
    val one-of = Parse.group msg (Scan.first (map scan-option opts));
  in
    Scan.optional (scan-list keyword <[] keyword > (Parse.enum , (Parse.!!!
one-of))) []
  end

fun no-duplicates xs =
  case duplicates (op =) xs of
    [] => ()
  | ds => error (duplicate option(s) defined: ^ Pretty.string-of (Pretty.list [ ]
(map Pretty.str ds)))

fun applies fs init =
  let
    fun apply f x = f x
  in fold apply fs init end

fun eval-options init opts =

```

```

let
  val - = no-duplicates (map fst opts)
in
  applies (map snd opts) init
end

fun get-options init opts =
  scan-options opts >> eval-options init

end

>

```

ML-val <

— Setup step 1: define the options record. By using the record antiquotation the map-functions for the fields will be automatically generated.

```

@{record
  <datatype opts = Opts of {i:int option, b:bool option, str:string list option}>
}

```

— Setup step 2: Define the "empty" record.

```

val empty-opts = make-opts {i=NONE, b=NONE, str = NONE}

```

— Setup step 3: Define the options you want to support.

```

val opts = [(i-opt, Option-Scanner.int-opt map-i NONE),
            (b-opt, Option-Scanner.bool-opt map-b (SOME true)),
            (flags, Option-Scanner.string-list-opt map-str NONE)]

```

— Aux functions to showcase the parsing

```

fun filtered-input b =
  filter Token.is-proper (Token.explode (Thy-Header.get-keywords' @context)) (Binding.pos-of
  b) (Binding.name-of b)

```

```

fun do-parse parser = Scan.error parser o filtered-input
fun do-parse-error parser input =
  do-parse parser input
  handle ERROR str => (warning str; (empty-opts, []))

```

— Examples for successful parsing.

```

val (opts1, -) = do-parse (Option-Scanner.get-options empty-opts opts)
  @binding <[i-opt=22, b-opt=true]>

```

```
val (opts2, -) = do-parse (Option-Scanner.get-options empty-opts opts)
  @ {binding <[i-opt=22, b-opt]>}
```

```
val (opts3, -) = do-parse (Option-Scanner.get-options empty-opts opts)
  @ {binding <[flags=[hallo, echo, otto]]>}
```

— Examples for error reporting.

```
val - = do-parse-error (Option-Scanner.get-options empty-opts opts)
  @ {binding <[b-opt=ddfsf]>}
```

```
val - = do-parse-error (Option-Scanner.get-options empty-opts opts)
  @ {binding <[foo=42]>}
```

```
val - = do-parse-error (Option-Scanner.get-options empty-opts opts)
  @ {binding <[i-opt=22, i-opt=23]>}
```

```
>
```

end

```
theory Named-Rules
  imports Main
  keywords named-rules::thy-decl
begin
ML-file named-rules.ML
end
```

```
theory Subgoals
  imports Main
  keywords prefers :: prf-script % proof and subgoals :: prf-script-goal % proof
begin
```

```
definition protected-conjunction :: prop  $\Rightarrow$  prop  $\Rightarrow$  prop (infix &  $\hat{\&}$  2) where
  protected-conjunction A B  $\equiv$  (PROP A &&& PROP B)
```

```
definition
  protected-prop A  $\equiv$  PROP A
```

```
lemma protect-prop:
  PROP A if PROP protected-prop A
  using that unfolding protected-prop-def .
```

```
ML <
fun filter-subgoals (test : cterm  $\rightarrow$  bool) thm =
  let
```



```

    val indexed-subgoals = tag-list 0 (Thm.cprems-of thm);
  in
    map fst (Library.filter (fn (-, subgoal) => test subgoal) indexed-subgoals)
  end

fun match-cterm pattern cterm =
  (Thm.match (pattern, cterm); true) handle Pattern.MATCH => false

datatype match-kind = Match-Concl | Match-Prem

fun match-cterm-rec pattern cterm =
  if match-cterm pattern cterm then
    true
  else
    match-cterm-rec pattern (Thm.dest-fun cterm)
  handle
    Thm.CTERM - => false

fun dest-all-cterm-all ctxt ct =
  let
    val ((-, ct), ctxt) = Variable.dest-all-cterm ct ctxt
  in
    dest-all-cterm-all ctxt ct
  end
  handle CTERM - => ct

fun match-subgoal (kind : match-kind) (no-match : bool) ctxt (pattern : cterm)
(subgoal : cterm) =
  let
    val cterms = case kind of
      Match-Concl => [dest-all-cterm-all ctxt subgoal |> Drule.strip-imp-concl] |
      Match-Prem => (dest-all-cterm-all ctxt subgoal |> Drule.strip-imp-prems);
    val match = fold (fn cterm => fn b => b orelse match-cterm-rec pattern
      (Thm.dest-arg cterm)) cterms false
  in
    if no-match then not match else match
  end

fun prefer-and-uncurry-subgoals-tac pred ctxt : tactic = fn thm =>
  let
    val indices = filter-subgoals pred thm
  in
    if indices = [] then
      Seq.empty
    else if length indices = 1 then
      CONVERSION (
        Conv.top-conv (K (Conv.rewr-conv @ {thm protected-prop-def[symmetric]}))
      )
    else
      1 (Drule.rearrange-prems indices thm)
  end

```

```

else
  Seq.single (Conjunction.uncurry-balanced
    (length indices)
    (Drule.rearrange-prems indices thm))
end

fun prefer-and-protect-subgoals-tac pred ctxt =
  prefer-and-uncurry-subgoals-tac pred ctxt
  THEN (REPEAT-DETERM (CHANGED-PROP (CONVERSION ((Conv.top-sweep-conv
(K (Conv.rewr-conv
  @{thm protected-conjunction-def[symmetric]})) ctxt)) 1)))

fun prefer-and-protect-subgoals-tac-pat (kind : match-kind) (no-match : bool) (pattern
: cterm) =
  fn ctxt => prefer-and-protect-subgoals-tac (match-subgoal kind no-match ctxt pat-
tern) ctxt

fun unprotect-subgoals-tac thms ctxt : tactic =
  REPEAT-DETERM (CHANGED-PROP (CONVERSION
  (Conv.top-sweep-conv (K (Conv.rewrs-conv thms)) ctxt) 1))

fun construct-pattern ctxt pattern =
  let
    val pattern = Proof-Context.read-term-pattern ctxt pattern;
    val pattern =
      if Term.fastype-of pattern = @{typ prop} then
        HOLogic.dest-Trueprop pattern
      else
        pattern
  in
    Thm.cterm-of ctxt pattern end

val parse-match-kind = Scan.optional (
  Args.parens (
    Args.$$$ concl >> K (Match-Concl, false) ||
    Args.$$$ not-concl >> K (Match-Concl, true) ||
    Args.$$$ prems >> K (Match-Prems, false) ||
    Args.$$$ not-prems >> K (Match-Prems, true))) (Match-Concl, false);

fun unprotect-and-finish thms =
  Seq.make-results o
  Seq.single o
  Proof.refine-singleton
  (Method.Basic (fn - => Method.succeed)) o
  Proof.refine-singleton
  (Method.Basic (fn ctxt => SIMPLE-METHOD
  (unprotect-subgoals-tac thms ctxt))
  )

```

```

val - = Outer-Syntax.command command-keyword <prefers>
  select subgoals that match a given pattern
  (parse-match-kind -- Parse.embedded-inner-syntax >> (fn ((kind, invert), pat-
tern) =>
    Toplevel.proofs (fn state => (
      let
        val ctxt = Proof.context-of state;
        val pattern = construct-pattern ctxt pattern
      in
        unprotect-and-finish @{thms protected-conjunction-def protected-prop-def}
      o
        Proof.refine-singleton
        (Method.Basic (fn ctxt => SIMPLE-METHOD
          (prefer-and-protect-subgoals-tac-pat kind invert pattern ctxt))) end)
state)));

val - =
  Outer-Syntax.command command-keyword <subgoals>
  focus on all subgoals that match a given pattern within backward refinement
  (parse-match-kind -- Parse.embedded-inner-syntax >> (fn ((kind, invert),
pattern) =>
    Toplevel.proofs (fn state => (
      let
        val ctxt = Proof.context-of state;
        val pattern = construct-pattern ctxt pattern
      in
        unprotect-and-finish @{thms protected-conjunction-def protected-prop-def}
      o
        #2 o
        Subgoal.subgoal Binding.empty-atts NONE (false, []) o
        Proof.refine-singleton
        (Method.Basic (fn ctxt => SIMPLE-METHOD
          (prefer-and-protect-subgoals-tac-pat kind invert pattern ctxt))) end)
state)))
  >

end

```

5.1 Tagging

```

theory Tagging
  imports Main Subgoals HOL-Eisbach.Eisbach
  keywords preferT prefersT :: prf-script % proof
  and subgoalT subgoalsT :: prf-script-goal % proof
begin

```

5.1.1 Basic Definitions and Theorems

```

definition ASM-TAG (¶) where

```

ASM-TAG $t \equiv \text{True}$

definition *TAG* :: 'b ⇒ 'a :: {} ⇒ 'a (- | - [13, 13] 14) **where**
 ⟨*TAG* $t\ x \equiv x$ ⟩

abbreviation *tag-prop* (- || - [0, 0] 0) **where**
tag-prop $t\ x \equiv \text{PROP } \text{TAG } t\ x$

lemma *TAG-cong[cong]*: $x \equiv y \implies (\text{tag } | x) \equiv \text{tag } | (y::'a::\{\})$
by *simp*

lemma *ASM-TAG-cong[cong]*: $\ulcorner \text{tag} \longleftrightarrow \ulcorner \text{tag}$
by *simp*

lemma *ASM-TAG-I[intro!]*: $\ulcorner x$ **by** (*simp add: ASM-TAG-def*)

lemma *TAG-TrueI[intro!, simp]*: $\text{tag } | \text{True}$
by (*simp add: TAG-def*)

lemma *TAG-False[simp]*: $(\text{tag } | \text{False}) \longleftrightarrow \text{False}$
by (*simp add: TAG-def*)

lemma *TAG-false[elim!]*: $(\text{tag } | \text{False}) \implies P$
by (*simp add: TAG-def*)

lemma *ASM-TAG-aux1*:
 $\text{PROP } P \equiv (\text{True} \implies \text{PROP } P)$
by *auto*

lemma *ASM-TAG-CONV1*:
 $\text{PROP } \text{TAG } t\ P \equiv (\text{ASM-TAG } t \implies \text{PROP } P)$
unfolding *TAG-def ASM-TAG-def* **by** *auto*

lemma *disjE-tagged*:
 $(P \vee Q) \implies (\ulcorner "l" \implies P \implies R) \implies (\ulcorner "r" \implies Q \implies R) \implies R$
by *blast*

lemma *conjI-tagged*:
 $(\ulcorner "l" \implies P) \implies (\ulcorner "r" \implies Q) \implies P \wedge Q$
by *blast*

— We will use the syntax $\ulcorner t \mid A$ for tags that should end up in assumptions and conclusions:

lemma *assm-tagE[elim!]*:
assumes $\ulcorner t \mid A$
assumes $\ulcorner t \implies t \mid A \implies P$
shows P
using *assms unfolding TAG-def* **by** *auto*

— Consider $(\ulcorner "l" \mid P) \vee (\ulcorner "r" \mid Q)$

```

ML <
fun unfold-tags ctxt = Local-Defs.unfold ctxt @ { thms TAG-def }
val untagged-attr = Thm.rule-attribute [] (fn context => unfold-tags (Context.proof-of context))

fun unfold-tac' ctxt rewr =
  let
    val rewr = map (Local-Defs.abs-def-rule ctxt) rewr
  in
    rewrite-goal-tac ctxt rewr
  end

fun thin-asm-tag-tac ctxt = REPEAT o eresolve-tac ctxt @ { thms thin-rl [ of ¶ t for t ] }

fun untag-tac ctxt = thin-asm-tag-tac ctxt THEN' unfold-tac' ctxt @ { thms TAG-def }
>

method-setup untag =
  <Args.context >> (fn - => fn ctxt => SIMPLE-METHOD' (untag-tac ctxt))
remove tags

```

Subgoals Test

```

lemma
  assumes False
  shows
    ∧ a b. A ⇒ B ⇒ C ⇒ "foo" | AA a b
    A ⇒ B ⇒ C2 ⇒ "bar" | B
    A ⇒ B ⇒ C ⇒ "bar" | C
    A2 ⇒ B ⇒ C22 ⇒ "bar" | C
    ∧ b. A2 ⇒ B ⇒ C22 ⇒ "foo" | CC b
    ∧ a b. A2 ⇒ B ⇒ C22 ⇒ f a b
    ∧ c d. A2 ⇒ ¶ "foobar" ⇒ B ⇒ C22 ⇒ D c d
  subgoals (concl) f -
    using <False> by simp
  subgoals (concl) "foo" | -
    using <False> by simp+
  subgoals "bar" | -
    using <False> by simp+
  subgoals (prems) ¶ "foobar"
oops

```

5.1.2 Conversions

```

ML <
fun extract-tag-trm trm =
  case trm of
    Const < Trueprop for Const < TAG - - for tag -> > => SOME tag

```

```

| Const ⟨TAG - - for tag -⟩ => SOME tag
| - => NONE

val tag-of-goal = extract-tag-trm o Logic.strip-assums-concl o Thm.prop-of
>

ML ⟨
fun prems-conv cv ct =
  let
    val n = Logic.count-prems (Thm.term-of ct)
  in
    Conv.prems-conv n cv ct
  end

fun goal-conv cv ct =
  let
    val n = Logic.count-prems (Thm.term-of ct)
  in
    Conv.concl-conv n cv ct
  end
>

ML ⟨
fun rm-head-tag-conv ctx = Conv.top-rewrs-conv @ {thms TAG-def} ctx |> goal-conv
>

ML ⟨
fun get-add-tag-rewrs - tag-ct =
  let
    val ctyp = Thm.ctyp-of-cterm tag-ct
  in [
    instantiate ⟨'t = ⟨ctyp⟩ and t1 = tag-ct in
      lemma (schematic) ⟨(PROP P) ≡ (ASM-TAG t1 ⇒ PROP P)⟩ for t1 :: 't
    and P
      by (unfold ASM-TAG-def, rule ASM-TAG-aux1)⟩
    ]
  end
val it = get-add-tag-rewrs @ {context} @ {cterm 1}
>

lemma norm-tag:
  Trueprop (t | P) ≡ (t || P)
unfolding TAG-def by auto

ML ⟨
fun push-concl-tag-to-assms ctxt thm =
  let
    val tag-opt = tag-of-goal thm
    val tag = the-default @ {term '$'} tag-opt
  end

```

```

val ctag = Thm.ctrm-of ctxt tag
val push-thms = get-add-tag-rewrs ctxt ctag
val rewr-conv = Conv.rewrs-conv push-thms
val push-conv = prems-conv rewr-conv
val push = Conv.fconv-rule push-conv
val norm-conv = Conv.rewrs-conv @ { thms norm-tag } |> goal-conv
val norm-conv = Conv.top-sweep-conv (fn - => norm-conv) ctxt
val assms-conv = Conv.rewrs-conv @ { thms ASM-TAG-CONV1 }
  |> goal-conv
val assms-conv = Conv.top-sweep-conv (fn - => assms-conv) ctxt
val rm = Conv.fconv-rule (rm-head-tag-conv ctxt)
in
  thm
  |> Conv.fconv-rule (prems-conv (norm-conv /goal-conv))
  |> Conv.fconv-rule assms-conv
  |> (case tag-opt of NONE => I | - => push)
  |> rm
end
>

```

```

ML <
val push-tags-attr =
  Thm.rule-attribute []
  (fn context => push-concl-tag-to-assms (Context.proof-of context))
>

```

```

attribute-setup push-tags =
  <Scan.succeed () >> (fn - => push-tags-attr)

```

```

ML <
fun extract-asm-tag-trm trm =
  case trm of
    Const <Trueprop for Const <ASM-TAG - for tag> => SOME tag
  | - => NONE

```

```

fun mk-add-tag-thms t = [
  instantiate <t = <Thm.ctyp-of-ctrm t> and t = t in
    lemma (schematic) <Trueprop y ≡ Trueprop (t | y)> for t :: 't by (unfold
TAG-def)>,
  instantiate <t = <Thm.ctyp-of-ctrm t> and t = t in
    lemma (schematic) <Trueprop y ≡ (t | Trueprop y)> for t :: 't by (unfold
TAG-def)>
]

```

```

fun get-list-tag t =
  let
    val tags = Logic.strip-assums-hyp t |> List.mapPartial extract-asm-tag-trm
    val tag = HOLogic.mk-list @ { typ string } tags
  in

```

```

    if tags = [] then NONE else SOME tag
  end

fun add-tag-conv assm t ctxt = case get-list-tag t of
  NONE => Conv.all-conv
| SOME tag =>
  let
    val ctag = Thm.ctrm-of ctxt tag
    val push-thms =
      if assm then
        get-add-tag-rewrs ctxt ctag
      else
        mk-add-tag-thms ctag
    val rewr-conv = Conv.rewrs-conv push-thms |> goal-conv
  in
    rewr-conv
  end

fun tidy-tags-tac0 assm ctxt i thm =
  let
    val t = Thm.cprem-of thm i |> Thm.term-of
    val cconv = add-tag-conv assm t |> Conv.top-sweep-conv
    val conv = cconv ctxt |> Conv.try-conv
  in
    (CONVERSION conv THEN' thin-asm-tag-tac ctxt) i thm
  end
  handle THM - => Seq.empty

fun tidy-tags-tac assm ctxt =
  REPEAT o eresolve-tac ctxt @ {thms assm-tagE} THEN' tidy-tags-tac0 assm ctxt

val tidy-tags-meth =      fn - => fn ctxt => SIMPLE-METHOD' (tidy-tags-tac
false ctxt)
val tidy-tags-assm-meth = fn - => fn ctxt => SIMPLE-METHOD' (tidy-tags-tac
true  ctxt)
>

method-setup tidy-tags-tac =
  <Args.context >> tidy-tags-meth>
  compress tags into list of tags and tag goals

method-setup tidy-tags-assm-tac =
  <Args.context >> tidy-tags-assm-meth>
  compress tags into list of tags and tag assumptions

method tidy-tags = changed <tidy-tags-tac>

method tidy-tags-assm = changed <tidy-tags-assm-tac>

```



```

lemma
   $n > 0$  if  $n > 1 \vee n > 2$  for  $n :: nat$ 
  using that
  apply (rule disjE-tagged; tidy-tags)
  oops

```

```

lemma
   $n > 0$  if  $(\llbracket "l" \rrbracket | n > 1) \vee (\llbracket "r" \rrbracket | n > 2)$  for  $n :: nat$ 
  using that
  apply standard
  apply (all tidy-tags)
  oops

```

5.1.3 Globbing

```

ML <
  datatype 't glob = WILDCARD | ANY | TOKEN of 't

```

```

fun
  matches-glob [] [] = true
| matches-glob [] - = false
| matches-glob (TOKEN t :: ps) (t' :: ts) = t' = t andalso matches-glob ps ts
| matches-glob (ANY :: ps) (- :: ts) = matches-glob ps ts
| matches-glob (WILDCARD :: ps) (t :: ts) =
  matches-glob ps (t :: ts) orelse matches-glob (WILDCARD :: ps) ts
| matches-glob (WILDCARD :: ps) [] = matches-glob ps []
| matches-glob - [] = false

```

```

fun assert b = (if b then () else error assert)

```

```

val p1 = [WILDCARD, TOKEN a, WILDCARD, TOKEN c]
val p2 = [ANY, TOKEN a, WILDCARD, TOKEN c]
val it1 = assert ([
  matches-glob p1 [b, a, c],
  matches-glob p1 [b, a, c, d],
  matches-glob p1 [a, c],
  matches-glob p1 [b, a, d]
]) = [true, false, true, false]
val it2 = assert ([
  matches-glob p2 [b, a, c],
  matches-glob p2 [b, a, c, d],
  matches-glob p2 [a, c],
  matches-glob p2 [b, a, d]
]) = [true, false, false, false]

```

```

ML <
  fun tag-list-of goal =
    case extract-tag-trm goal of

```

```

    NONE => []
  | SOME t => map HOLogic.dest-string (HOLogic.dest-list t)
    handle TERM (dest-list, -) => [HOLogic.dest-string t]

fun matches-glob-term glob = matches-glob glob o tag-list-of
>

```

```

ML <
val parse-glob-token = (Parse.sym-ident || Parse.name)
  >> (fn
    * => WILDCARD
  | - => ANY
  | s => TOKEN s
  )
val parse-glob = Scan.repeat1 parse-glob-token
>

```

5.1.4 Reordering Subgoals

```

ML <
val empty = (Vartab.empty, Vartab.empty)

fun matches thy pat trm = (Pattern.match thy (pat, trm) empty; true)
  handle Pattern.MATCH => false

fun matches-tag thy tag goal =
  case extract-tag-trm goal of
    NONE => false
  | SOME t => matches thy tag t

fun filter-thms thy pat = List.filter (fn thm => matches thy pat (Thm.prop-of
  thm))

(* Find index of first subgoal in thm whose conclusion matches pred *)
fun find-subgoal pred thm =
  let
    val subgoals = Thm.premis-of thm
  in
    Library.find-index (fn goal => pred (Logic.strip-assums-concl goal)) subgoals
  end

fun prefer-by-pred-tac pred: tactic = fn thm =>
  let
    val i = find-subgoal pred thm + 1
  in
    if i > 0 then Tactic.prefer-tac i thm else no-tac thm
  end

fun prefer-by-tag-tac thy tag: tactic =

```

```

    prefer-by-pred-tac (matches-tag thy tag)

fun prefer-pred pred st =
  let
    val st = Proof.assert-no-chain st
    val thy = Proof.theory-of st
  in
    Proof.refine-singleton
      (Method.Basic (fn - => METHOD (fn - => prefer-by-pred-tac (pred thy))))
  st
  end

fun prefer-glob glob = prefer-pred (fn - => matches-glob-term glob)
>

```

```

ML <Outer-Syntax.command command-keyword <preferT> select subgoal by tag
  (parse-glob >> (Toplevel.proof o prefer-glob))
>

```

```

lemma
  TAG "p" P TAG "q" Q PROP TAG "r" R TAG "p" P2
preferT r
oops

```

```

lemma
   $\wedge t. TAG "p" P \wedge TAG "q" Q$ 
apply (rule conjI)
preferT q
oops

```

5.1.5 subgoalT and subgoalsT, and prefersT

```

ML <
fun orelseOpt f g x = case f x of NONE => g x | y => y
fun assem-name-of-tags trm =
  extract-tag-trm trm |>
    Option.mapPartial (
      orelseOpt
        (try (HOLogic.dest-string o hd o HOLogic.dest-list))
        (try HOLogic.dest-string)
    )
  |> the-default
>

```

```

ML <
fun chop-by - [] = []
  | chop-by eq (x :: xs) =
  let
    val (g, rest) = chop-prefix (eq x) xs

```

```

    in
      (x :: g) :: chop-by eq rest
    end
  fun group-by k = chop-by (fn a => fn b => k a = k b) o sort-by k
  >

```

```

ML <
  val tag-name-of-prop = assem-name-of-tags o Thm.prop-of
  fun note-of-thms (name, thms) =
    let
      val binding = Binding.qualified-name name
    in
      (((binding, []), [(thms, [untagged-attr])])
    end
  fun note-thms thms ctx =
    let
      val thms-tags = map (fn thm => (tag-name-of-prop thm, thm)) thms
      val grouped = group-by fst thms-tags |> map (fn g => (fst (hd g), map snd g))
    in
      Proof-Context.note-thmss (map note-of-thms grouped) ctx |> snd
    end
  fun note-thms-tac thms (ctx, thm) = (note-thms thms ctx, thm) |> Seq.succeed |>
  Seq.make-results
  fun note-thms-meth x = (Scan.succeed () >>
    (fn () => fn - => CONTEXT-METHOD (fn thms => note-thms-tac thms))) x
  >

```

method-setup *note-thms* = <*note-thms-meth*>

```

ML <
  local
    val opt-fact-binding =
      Scan.optional (Parse.binding -- Parse.opt-attribs || Parse.attribs >> pair Bind-
        ing.empty)
        Binding.empty-attribs;

    val for-params =
      Scan.optional
        (keyword <for> |--
          Parse.!!! ((Scan.option Parse.dots >> is-some) --
            (Scan.repeat1 (Parse.maybe-position Parse.name-position))))
        (false, []);

    fun subgoal-cmd facts-name-opt param-specs st =
      let
        val (subgoal-focus, st) = Subgoal.subgoal-cmd (Binding.empty, []) facts-name-opt
          param-specs st
        val prems = #prems subgoal-focus
        val st = Proof.map-context (note-thms prems) st
      end

```

```

in
  st
end

val - =
  Outer-Syntax.command command-keyword ⟨subgoalT⟩
  subgoal for tags
  ((Scan.option parse-glob)
   -- (Scan.option (keyword ⟨premises⟩ |-- Parse!!! opt-fact-binding)) --
   for-params >> (fn ((glob-opt, b), c) =>
     Toplevel.proofs (
       Seq.make-results
       o Seq.single
       o Proof.refine-singleton (Method.Basic (fn ctxt => SIMPLE-METHOD'
(untag-tac ctxt)))
       o subgoal-cmd b c
       o (case glob-opt of SOME x => prefer-glob x | - => fn x => x)
       o Proof.refine-singleton (Method.Basic (tidy-tags-meth []))
     )))

val parse-keep =
  Scan.optional (Args.parens (Args.$$$ keep >> K true)) false;

fun tidy-then-protect-meth glob =
  let
    val pred = matches-glob-term glob o Logic.strip-assums-concl o Thm.term-of
    fun tidy-then-protect-tac ctxt =
      TRYALL (tidy-tags-tac false ctxt)
      THEN prefer-and-protect-subgoals-tac pred ctxt
    val tidy-then-protect = Method.Basic
      (SIMPLE-METHOD o tidy-then-protect-tac)
  in tidy-then-protect end

fun get-unprotect-thms keep =
  @{thms protected-conjunction-def protected-prop-def} @
  (if keep then [] else @{thms TAG-def})

val - =
  Outer-Syntax.command command-keyword ⟨subgoalsT⟩
  focus on all subgoals that match a given pattern within backward refinement
  (parse-keep -- parse-glob >> (fn (keep, glob) =>
    Toplevel.proofs (
      unprotect-and-finish (get-unprotect-thms keep) o
      #2 o
      Subgoal.subgoal Binding.empty-atts NONE (false, []) o
      Proof.refine-singleton (tidy-then-protect-meth glob)
    )))

val - =

```

```

Outer-Syntax.command command-keyword <prefersT>
  focus on all subgoals that match a given pattern within backward refinement
  (parse-keep -- parse-glob >> (fn (keep, glob) =>
    Toplevel.proofs (
      unprotect-and-finish (get-unprotect-thms keep) o
      Proof.refine-singleton (tidy-then-protect-meth glob)
    )))
in end
>

```

```

lemma
  assumes TAG "p" P Q False
  shows TAG ["x"] True TAG ["y"] False TAG ["x", "y"] False TAG ["a", "b"]
True
  using assms
  apply -
  subgoal premises prems
    using prems
    apply note-thms
    thm p
  oops

```

```

lemma
  assumes TAG "p" P Q False
  shows TAG ["x"] True TAG ["y"] False TAG ["x", "y"] False TAG ["a", "b"]
True
  using assms
  apply -
  subgoal premises prems
    using prems
    apply note-thms
    thm p
  oops

```

```

lemma
  assumes TAG "p" P TAG ["q", "r"] Q TAG ["q"] QQ False
  shows TAG ["x"] True TAG ["y"] False TAG ["x", "y"] False TAG ["a", "b"]
True
  using assms
  apply -

```

```

subgoalsT (keep) x *

```

```

subgoalT x premises prems
  thm p q
  using <False> by simp
subgoalT x - premises prems
  thm p q
  using <False> by simp

```

```

done
subgoalT y premises prems
  thm p q
  using <False> by simp
subgoalsT *
  using <False> by simp+
oops

lemma
  assumes TAG "p" P TAG ["q", "r"] Q False
  shows
    TAG ["x"] True TAG ["y"] False P TAG ["x", "y"] False TAG ["a", "b"]
True
  using assms
  apply -
  subgoalT x premises prems
    using prems apply -
    using <False> by simp

  subgoalT y premises prems
    using prems apply -
    using <False> by simp

  subgoalT premises prems
    using prems apply -
    using <False> by simp

  subgoalT x y premises prems
    using prems apply -
    using <False> by simp

  subgoalT a b premises prems
    using prems apply -
    thm p q
    using <False> by simp
done

lemma
  n > 0 if (⊢ "l" | n > 1) ∨ (⊢ "r" | n > 2) for n :: nat
  using that
  apply standard
  subgoalT l premises
    using l by simp
  subgoalT r premises
    using r by simp
done

lemma
  n > 0 if (⊢ "l" | n > 9) ∨ (⊢ "s" | n > 2) ∨ (⊢ "s" | n > 0) for n :: nat

```

```
using that
apply (elim disjE)
prefersT s — tidies all goals before tags are matched
  apply simp+
subgoalT l premises
  using l by simp
done

end
```


Chapter 6

Verification Condition Generator *runs-to-vcg*

```
theory Basic-Runs-To-VCG
  imports
    Named-Rules
    HOL-Eisbach.Eisbach-Tools
    Tagging
begin
```

6.1 Marked Assumptions

```
ML <
```

```
structure Marked-Assumptions =
  struct
```

```
  structure Data = Theory-Data
```

```
  (
    type T = (thm -> Proof.context -> Proof.context) Symtab.table;
    val empty = Symtab.empty;
    val merge = Symtab.merge (K true);
  );
```

```
  fun import-assm ctxt thm =
    ( case head-of (HOLogic.dest-Trueprop (Thm.prop-of thm)) of
      Const (name, _) =>
        (case Symtab.lookup (Data.get (Proof-Context.theory-of ctxt)) name of
          SOME add-thm => SOME (add-thm thm ctxt)
          | NONE => NONE)
      | - => NONE )
    handle TERM - => NONE
```

```
  fun with-assms ctxt old-prems tac =
```

```

Subgoal.FOCUS (fn { context = ctxt1, prems, ... } =>
  let
    val (ctxt4, prems') = (ctxt1, old-prems) |> fold (fn thm => fn (ctxt2, prems')
=>
      case import-assm ctxt2 thm of
        SOME ctxt3 => (ctxt3, prems')
      | NONE => (ctxt2, thm::prems') prems
    in
      tac ctxt4 prems'
    end) ctxt 1

fun add-marker name f = Data.map (Symtab.update (name, f))

end ;

>

```

```

attribute-setup parse-ASSMs =
  <Scan.succeed (Thm.declaration-attribute (fn thm => Context.map-proof (fn ctxt
=>
  case Marked-Assumptions.import-assm ctxt thm of SOME ctxt => ctxt | NONE
=> ctxt))))>

```

named-theorems *remove-ASSMs* Remove the markers for marked assumptions

method *cleanup-ASSMs* = *simp-all only: remove-ASSMs*

definition *SIMP-ASSM* :: *bool* \Rightarrow *bool* **where** [*remove-ASSMs*]: *SIMP-ASSM* *P*
 \longleftrightarrow *P*

lemma *SIMP-ASSM-D*: *SIMP-ASSM* *P* \Longrightarrow *P* **by** (*simp add: SIMP-ASSM-def*)
setup < *Marked-Assumptions.add-marker* @{*const-name SIMP-ASSM*} (fn thm
 \Rightarrow fn ctxt \Rightarrow
 ctxt *addsimps* [thm *RS* @{thm *SIMP-ASSM-D*}]>

6.2 THEN-ALL-NEW-FORWARD

ML <

```

fun each-protected (t : int -> tactic) : tactic =
  let
    fun work-at i st =
      if Thm.nprems-of st = i then Seq.single st
      else
        rotate-prems i st
        |> Goal.protect 1
        |> t i
        |> Seq.maps (fn st =>
          Goal.conclude st
          |> rotate-prems (~ i)

```

```

    |> work-at (i + Thm.nprems-of st) )

in work-at 0 end

(* THEN-ALL-NEW-FORWARD m1 m2, applies first m1 and then all all new
subgoals m2 is applied.

m1 THEN-ALL-NEW m2 (i.e. m1; m2) applies m2 in the reverse order. This is
a problem when schematic
variables need to be resolved *)

fun THEN-ALL-NEW-FORWARD' (t1 : tactic) (t2 : int -> tactic) (st : thm) =
  Goal.protect 1 st
  |> t1
  |> Seq.maps (each-protected t2)
  |> Seq.map Goal.conclude

fun THEN-ALL-NEW-FORWARD (t1 : tactic) (t2 : tactic) =
  THEN-ALL-NEW-FORWARD' t1 (K t2)

>

method-setup all-new = <
(Method.text-closure -- Method.text-closure) >> (fn (m1, m2) => fn ctxt =>
fn facts =>
  let
    val tac1 = method-evaluate m1 ctxt facts
    val tac2 = method-evaluate m2 ctxt facts
  in Method.RUNTIME (SIMPLE-METHOD (THEN-ALL-NEW-FORWARD tac1
tac2) facts) end)
>

```

6.3 Basic VCG

```

named-theorems runs-to-vcg-cong-state-only
named-theorems runs-to-vcg-weaken
named-theorems runs-to-vcg-cong-program-only
named-rules (intro) runs-to-vcg

named-theorems runs-to-vcg-post-elim
named-theorems runs-to-vcg-post-intros

declare conjI[runs-to-vcg-post-intros]
declare disjE[runs-to-vcg-post-elim]

ML <
signature RUNS-TO-VCG =
sig
  val apply-runs-to:

```

```

(Proof.context -> int -> tactic) option ->
  (Proof.context -> (unit -> string) -> tactic) ->
    term -> Proof.context -> tactic
val prepare:
  {do-nosplit: bool, no-unsafe-hyp-subst: bool} -> Proof.context -> tactic
val runs-to-vcg-tac:
  (Proof.context -> int -> tactic) option ->
    int -> (Proof.context -> (unit -> string) -> tactic) ->
      {do-nosplit: bool, no-unsafe-hyp-subst: bool} ->
        (Proof.context -> tactic) -> Proof.context -> tactic
val split-paired-all: Proof.context -> int -> tactic
val trace-tac: 'a -> (unit -> string) -> 'b -> 'b Seq.seq
val no-trace-tac: Proof.context -> (unit -> string) -> tactic
val trace-print-tac: Proof.context -> (unit -> string) -> thm -> thm Seq.seq
end

structure Runs-To-VCG : RUNS-TO-VCG =
struct

(*Print the current proof state and pass it on.*)
fun trace-tac - msg st = (tracing (msg ()); Seq.single st);
fun trace-print-tac ctxt msg st = print-tac ctxt (msg ()) st;
val no-trace-tac = K (K (all-tac)): Proof.context -> (unit -> string) -> tactic

fun split-paired-all ctxt =
  simp-tac (put-simpset HOL-basic-ss ctxt addsimps @{thms split-paired-all})

fun prepare {do-nosplit, no-unsafe-hyp-subst} ctxt =
let
  val post-intros = Named-Theorems.get ctxt @{named-theorems runs-to-vcg-post-intros}
  val post-elim = Named-Theorems.get ctxt @{named-theorems runs-to-vcg-post-elim}
in
  REPEAT-DETERM-FIRST (fn n =>
    CHANGED (split-paired-all ctxt n)
  ORELSE
    CHANGED (eresolve-tac ctxt @{thms exE bexE conjE} n)
  ORELSE
    CHANGED (resolve-tac ctxt @{thms allI ballI impI} n)
  ORELSE
    CHANGED (full-simp-tac
      ((fold Simplifier.add-cong
        (Named-Theorems.get ctxt @{named-theorems runs-to-vcg-cong-state-only})
      ctxt)) n)
  ORELSE
    (if no-unsafe-hyp-subst
      then CHANGED (asm-full-simp-tac (put-simpset HOL-basic-ss ctxt) n)
      else CHANGED (bound-hyp-subst-tac ctxt n))
  ORELSE
    (if do-nosplit then no-tac else CHANGED (eresolve-tac ctxt post-elim n))
end

```

```

    ORELSE
    CHANGED (resolve-tac ctxt post-intros n)
    ORELSE
    all-tac
  end

local
fun parse-program1 ctxt g cong-program-thm =
  let
    fun lhs t = t |> HOLogic.dest-Trueprop |> HOLogic.dest-eq |> #1
    val pat = Thm.concl-of cong-program-thm |> lhs
    val concl = HOLogic.dest-Trueprop (Logic.strip-assums-concl g)
    val (key, -) = case Thm.prem-of cong-program-thm of [prem] =>
      lhs prem |> Term.dest-Var
      | - => error Runs-To-VCG parse-program: wrong format of rule
  in
    case Unify.unifiers (Context.Proof ctxt, Envir.init, [(pat, concl)]) |> Seq.pull
    of SOME ((env,-), -) =>
      (case Vartab.lookup (Envir.term-env env) key of NONE => NONE
      | SOME (-, t) => SOME t)
    | NONE => NONE
  end
in
fun parse-program ctxt t = get-first (parse-program1 ctxt t)
  (Named-Theorems.get ctxt @{named-theorems runs-to-vcg-cong-program-only})
end

fun with-rules tac ctxt =
  let
    fun tac' ctxt thms i = tac ctxt thms
  in
    Named-Rules.with-rules @{named-rules runs-to-vcg} tac' ctxt 1
  end

fun apply-runs-to splitter-opt trace prg = with-rules (fn ctxt => fn rules =>
  let
    val n = length rules
    fun t i thm msg () =
      let
        val d = Thm.derivation-name thm
      in
        At ^ Syntax.string-of-term ctxt (head-of prg) ^ apply ^ msg ^ : ^
        space-implode [d, ( ^ @ {make-string} (i + 1) ^ of ^ @ {make-string} n
        ^ )]
      end
    fun MAP f = FIRST (map-index f rules)
  in
    TRY (DETERM (
      MAP (fn (i, thm) => resolve-tac ctxt [thm] 1 THEN trace ctxt (t i thm rule))
    ))
  end

```

```

    ORELSE (case splitter-opt of
      SOME splitter =>
        trace ctxt (fn () => Trying splitter on  $\hat{\text{Syntax.string-of-term}}\text{ ctxt prg})
    THEN
      splitter ctxt 1
      | - => no-tac)
    ORELSE
      MAP (fn (i, thm) =>
        resolve-tac ctxt
        (Named-Theorems.get ctxt @{named-theorems Basic-Runs-To-VCG.runs-to-vcg-weaken})
      1 THEN
        resolve-tac ctxt [thm] 1 THEN
          trace ctxt (t i thm weakened rule))))
  end)$ 
```

```

fun runs-to-vcg-tac splitter-opt n trace options solver ctxt =
let
  fun flat-trace f () =  $\hat{f}$  ()
  fun runs-to-vcg-loop n trace ctxt prems =
    if n = 0 then all-tac else
      SUBGOAL (fn (g, -) =>
        case parse-program ctxt g of
          SOME prg =>
            THEN-ALL-NEW-FORWARD
              (CHANGED (apply-runs-to splitter-opt trace prg ctxt))
              (THEN-ALL-NEW-FORWARD
                (prepare options ctxt)
                (Marked-Assumptions.with-assms ctxt prems
                  (runs-to-vcg-loop (n - 1) (fn ctxt => fn f => trace ctxt (flat-trace
f))))))
            ORELSE trace ctxt (fn () => No rule applicable for  $\hat{\text{Syntax.string-of-term}}\text{ ctxt prg})
          | NONE =>
            TRY (SOLVED' (Method.insert-tac ctxt prems THEN'
              (K (solver ctxt) ORELSE' assume-tac ctxt)) 1))
            1
        in
          runs-to-vcg-loop n trace ctxt []
        end

  end
end$ 
```

```

method-setup runs-to-vcg =
< (Scan.lift (Args.mode trace) --
  Scan.lift (Args.mode nosplit) --
  Scan.lift (Args.mode no-unsafe-hyp-subst) --
  Scan.optional (Scan.lift (Args.parens Parse.nat)) (~1) --
  Scan.option Method.text-closure) >>

```

```

(fn (((do-trace, do-nosplit), do-no-unsafe-hyp-subst), n), text) => fn ctxt =>
  let val tac = (case text of
    NONE => (fn - => no-tac)
  | SOME txt => fn ctxt => NO-CONTEXT-TACTIC ctxt (Method.evaluate
txt ctxt []))
  in SIMPLE-METHOD (Runs-To-VCG.runs-to-vcg-tac NONE n
    (if do-trace then Runs-To-VCG.trace-tac else Runs-To-VCG.no-trace-tac)
    {do-nosplit=do-nosplit, no-unsafe-hyp-subst=do-no-unsafe-hyp-subst}
    tac ctxt) end) ›

```

6.4 Setup for Tagging

lemma *push-tag-to-assm*:

$t \mid P$ if $\nabla t \implies P$

using that **unfolding** *ASM-TAG-def TAG-def* by *simp*

lemma *tagE*:

$tag \mid Q \implies (Q \implies P) \implies P$

by (*auto simp: TAG-def*)

bundle *basic-vcg-tagging-setup*

begin

lemmas [*runs-to-vcg-post-intros*] = *push-tag-to-assm TAG-TrueI ASM-TAG-I*

lemmas [*runs-to-vcg-post-elim*s] = *tagE*

end

end

theory *Runs-To-VCG*

imports

Basic-Runs-To-VCG

begin

lemma *transfer-conj-imp*: *rel-fun* (=) (*rel-fun* (\longrightarrow) (\longrightarrow)) (\wedge) (\wedge)

by (*auto simp: rel-fun-def*)

lemma *transfer-all-imp*: *rel-fun* (*rel-set* *R*) (*rel-fun* (*rel-fun* *R* (\longrightarrow)) (\longrightarrow)) (*Ball*)

(*Ball*)

by (*auto simp: rel-fun-def rel-set-def*)

6.5 *runs-to-vcg*

```
method-setup trace-goals = ⟨  
  Scan.lift Parse.string >>  
  (fn msg => fn ctxt => fn using => Method.RUNTIME (CONTEXT-TACTIC  
    (print-tac ctxt msg)))  
⟩
```

```
method-setup trace = ⟨  
  (Scan.lift Parse.string) >>  
  (fn msg => fn ctxt => fn using => Method.RUNTIME (CONTEXT-TACTIC  
    (fn st => (tracing msg; Seq.single st))))  
⟩
```

6.6 Check

```
thm runs-to-vcg [no-vars]  
end
```


Chapter 7

Proof Methods

```
theory Eisbach-Methods
imports
  Subgoal-Methods
  HOL-Eisbach.Eisbach-Tools
  Rule-By-Method
begin
```

7.1 Debugging methods

```
method print-concl = (match conclusion in P for P  $\Rightarrow$   $\langle$ print-term P $\rangle$ )
```

```
method-setup print-raw-goal =  $\langle$ Scan.succeed (fn ctxt  $\Rightarrow$  fn facts  $\Rightarrow$ 
  (fn (ctxt, st)  $\Rightarrow$  (Output.writeln (Thm.string-of-thm ctxt st);
    Seq.make-results (Seq.single (ctxt, st)))))) $\rangle$ 
```

```
ML  $\langle$ fn method-evaluate text ctxt facts =
  NO-CONTEXT-TACTIC ctxt
  (Method.evaluate-runtime text ctxt facts) $\rangle$ 
```

```
method-setup print-headgoal =
   $\langle$ Scan.succeed (fn ctxt  $\Rightarrow$ 
    fn -  $\Rightarrow$  fn (ctxt', thm)  $\Rightarrow$ 
      ((SUBGOAL (fn (t,-)  $\Rightarrow$ 
        (Output.writeln
          (Pretty.string-of (Syntax.pretty-term ctxt t)); all-tac)) 1 thm);
        Seq.make-results (Seq.single (ctxt', thm)))))) $\rangle$ 
```

7.2 Simple Combinators

```
method-setup defer-tac =  $\langle$ Scan.succeed (fn -  $\Rightarrow$  SIMPLE-METHOD (defer-tac
  1)) $\rangle$ 
```

```
method-setup prefer-last =  $\langle$ Scan.succeed (fn -  $\Rightarrow$  SIMPLE-METHOD (PRIMITIVE
```

(*Thm.permute-prems 0 ~ 1*))>

```
method-setup all =  
  <Method.text-closure >> (fn m => fn ctxt => fn facts =>  
    let  
      fun tac i st' =  
        Goal.restrict i 1 st'  
        |> method-evaluate m ctxt facts  
        |> Seq.map (Goal.unrestrict i)  
  
      in SIMPLE-METHOD (ALLGOALS tac) facts end)  
>
```

```
method-setup determ =  
  <Method.text-closure >> (fn m => fn ctxt => fn facts =>  
    let  
      fun tac st' = method-evaluate m ctxt facts st'  
  
      in SIMPLE-METHOD (DETERM tac) facts end)  
> <Run the given method, but only yield the first result>
```

```
ML <  
  fun require-determ (method : Method.method) facts st =  
    case method facts st |> Seq.filter-results |> Seq.pull of  
      NONE => Seq.empty  
    | SOME (r1, rs) =>  
      (case Seq.pull rs of  
        NONE => Seq.single r1 |> Seq.make-results  
      | - => Method.fail facts st);  
  
  fun require-determ-method text ctxt =  
    require-determ (Method.evaluate-runtime text ctxt);  
>
```

```
method-setup require-determ =  
  <Method.text-closure >> require-determ-method  
> <Run the given method, but fail if it returns more than one result>
```

```
method-setup changed =  
  <Method.text-closure >> (fn m => fn ctxt => fn facts =>  
    let  
      fun tac st' = method-evaluate m ctxt facts st'  
  
      in SIMPLE-METHOD (CHANGED tac) facts end)  
>
```

```
method-setup timeit =  
  <Method.text-closure >> (fn m => fn ctxt => fn facts =>
```

```

let
  fun timed-tac st seq = Seq.make (fn () => Option.map (apsnd (timed-tac st))
    (timeit (fn () => (Seq.pull seq)))));

  fun tac st' =
    timed-tac st' (method-evaluate m ctxt facts st');

in SIMPLE-METHOD tac [] end)
>

```

```

method-setup timeout =
  <Scan.lift Parse.int -- Method.text-closure >> (fn (i,m) => fn ctxt => fn facts
=>
  let
    fun str-of-goal th = Pretty.string-of (Goal-Display.pretty-goal ctxt th);

    fun limit st f x = Timeout.apply (Time.fromSeconds i) f x
      handle Timeout.TIMEOUT - => error (Method timed out:\n ^ (str-of-goal
st));

    fun timed-tac st seq = Seq.make (limit st (fn () => Option.map (apsnd
(timed-tac st))
      (Seq.pull seq)));

    fun tac st' =
      timed-tac st' (method-evaluate m ctxt facts st');

in SIMPLE-METHOD tac [] end)
>

```

```

method repeat-new methods m = (m ; (repeat-new <m>)?)

```

The following *fails* and *succeeds* methods protect the goal from the effect of a method, instead simply determining whether or not it can be applied to the current goal. The *fails* method inverts success, only succeeding if the given method would fail.

```

method-setup fails =
  <Method.text-closure >> (fn m => fn ctxt => fn facts =>
  let
    fun fail-tac st' =
      (case Seq.pull (method-evaluate m ctxt facts st') of
        SOME - => Seq.empty
        | NONE => Seq.single st')

in SIMPLE-METHOD fail-tac facts end)
>

```

```

method-setup succeeds =

```

```

<Method.text-closure >> (fn m => fn ctxt => fn facts =>
  let
    fun can-tac st' =
      (case Seq.pull (method-evaluate m ctxt facts st') of
        SOME (st'',-) => Seq.single st'
        | NONE => Seq.empty)

    in SIMPLE-METHOD can-tac facts end)
>

```

This method wraps up the "focus" mechanic of match without actually doing any matching. We need to consider whether or not there are any assumptions in the goal, as premise matching fails if there are none.

If the *fails* method is removed here, then backtracking will produce a set of invalid results, where only the conclusion is focused despite the presence of subgoal premises.

```

method focus-concl methods m =
  ((fails <erule thin-rl>, match conclusion in - => <m>)
  | match premises (local) in H:- (multi) => <m>)

```

repeat applies a method a specific number of times, like a bounded version of the '+' combinator.

usage: apply (repeat n *text*)

- Applies the method *text* to the current proof state n times. - Fails if *text* can't be applied n times.

```

ML <
  fun repeat-tac count tactic =
    if count = 0
    then all-tac
    else tactic THEN (repeat-tac (count - 1) tactic)
>

```

```

method-setup repeat = <
  Scan.lift Parse.nat -- Method.text-closure >> (fn (count, text) => fn ctxt =>
  fn facts =>
    let val tactic = method-evaluate text ctxt facts
        in SIMPLE-METHOD (repeat-tac count tactic) facts end)
>

```

notepad begin

fix A B C

assume *assms*: A B C

repeat: simple repeated application.

have A ∧ B ∧ C ∧ True

repeat: fails if method can't be applied the specified number of times.

apply (fails <repeat 4 <rule conjI, rule assms>>)

```
apply (repeat 3 ⟨rule conjI, rule assms⟩)
by (rule TrueI)
```

repeat: application with subgoals.

```
have A ∧ A B ∧ B C ∧ C
apply –
```

We have three subgoals. This *repeat* call consumes two of them.

```
apply (repeat 2 ⟨rule conjI, (rule assms)+⟩)
```

One subgoal remaining...

```
apply (rule conjI, (rule assms)+)
done
```

end

Literally a copy of the parser for *subgoal-tac* composed with an analogue of **prefer**.

Useful if you find yourself introducing many new facts via *subgoal-tac*, but prefer to prove them immediately rather than after they're used.

```
setup ⟨
  Method.setup binding ⟨prop-tac⟩
  (Args.goal-spec -- Scan.lift (Scan.repeat1 Parse.embedded-inner-syntax --
Parse.for-fixes) >>
  (fn (quant, (props, fixes)) => fn ctxt =>
    (SIMPLE-METHOD'' quant
      (EVERY' (map (fn prop => Rule-Insts.subgoal-tac ctxt prop fixes) props)
        THEN'
          (K (prefer-tac 2))))))
  insert prop (dynamic instantiation), introducing prop subgoal first
  ⟩
```

```
notepad begin {
  fix xs
  assume assms: list-all even (xs :: nat list)
```

```
  from assms have even (sum-list xs)
  apply (induct xs)
  apply simp
```

Inserts the desired proposition as the current subgoal.

```
apply (prop-tac list-all even xs)
subgoal by simp
```

The prop *list-all even xs* is now available as an assumption. Let's add another one.

```
apply (prop-tac even (sum-list xs))
subgoal by simp
```

Now that we've proven our introduced props, use them!

```

apply clarsimp
done
}
end

```

7.3 Advanced combinators

7.3.1 Protecting goal elements (assumptions or conclusion) from methods

```

context
begin

```

```

private definition protect-concl  $x \equiv \neg x$ 
private definition protect-false  $\equiv \text{False}$ 

```

```

private lemma protect-start: (protect-concl  $P \implies \text{protect-false}$ )  $\implies P$ 
by (simp add: protect-concl-def protect-false-def) (rule ccontr)

```

```

private lemma protect-end: protect-concl  $P \implies P \implies \text{protect-false}$ 
by (simp add: protect-concl-def protect-false-def)

```

```

method only-asm methods  $m =$ 
  (match premises in  $H[\textit{thin}]$ :- (multi, cut)  $\Rightarrow$ 
     $\langle \textit{rule protect-start},$ 
      match premises in  $H'[\textit{thin}]$ :protect-concl -  $\Rightarrow$ 
         $\langle \textit{insert } H, m; \textit{rule protect-end}[OF H'] \rangle \rangle$ )

```

```

method only-concl methods  $m = (\textit{focus-concl } \langle m \rangle)$ 

```

```

end

```

```

notepad begin

```

```

fix  $D C$ 
assume  $DC:D \implies C$ 
have  $D \wedge D \implies C \wedge C$ 
apply (only-asm  $\langle \textit{simp} \rangle$ ) — stash conclusion before applying method
apply (only-concl  $\langle \textit{simp add: DC} \rangle$ ) — hide premises from method
by (rule DC)

```

```

end

```

7.3.2 Safe subgoal folding (avoids expanding meta-conjuncts)

Isabelle's goal mechanism wants to aggressively expand meta-conjunctions if they are the top-level connective. This means that *fold-subgoals* will immediately be unfolded if there are no common assumptions to lift over.

To avoid this we simply wrap conjunction inside of conjunction' to hide it from the usual facilities.

context begin

definition

conjunction' :: *prop* ⇒ *prop* ⇒ *prop* (**infixr** &~& 2) **where**
conjunction' *A B* ≡ (*PROP A* &&& *PROP B*)

In general the context antiquotation does not work in method definitions. Here it is fine because `Conv.top_sweep_conv` is just over-specified to need a `Proof.context` when anything would do.

method *safe-meta-conjuncts* =

raw-tactic
 ⟨*REPEAT-DETERM*
 (*CHANGED-PROP*
 (*PRIMITIVE*
 (*Conv.gconv-rule* ((*Conv.top-sweep-conv* (*K* (*Conv.rewr-conv* @{*thm conjunction'-def*[*symmetric*]})) @{*context*})) 1)))⟩

method *safe-fold-subgoals* = (*fold-subgoals* (*prefix*), *safe-meta-conjuncts*)

lemma *atomize-conj'* [*atomize*]: (*A* &~& *B*) == *Trueprop* (*A* & *B*)
 by (*simp add: conjunction'-def*, *rule atomize-conj*)

lemma *context-conjunction'I*:

PROP P ⇒ (*PROP P* ⇒ *PROP Q*) ⇒ *PROP P* &~& *PROP Q*

apply (*simp add: conjunction'-def*)

apply (*rule conjunctionI*)

apply *assumption*

apply (*erule meta-mp*)

apply *assumption*

done

lemma *conjunction'I*:

PROP P ⇒ *PROP Q* ⇒ *PROP P* &~& *PROP Q*

by (*rule context-conjunction'I*; *simp*)

lemma *conjunction'E*:

assumes *PQ*: *PROP P* &~& *PROP Q*

assumes *PQR*: *PROP P* ⇒ *PROP Q* ⇒ *PROP R*

shows

PROP R

apply (*rule PQR*)

apply (*rule PQ*[*simplified conjunction'-def*, *THEN conjunctionD1*])

by (*rule PQ*[*simplified conjunction'-def*, *THEN conjunctionD2*])

end

notepad begin

```

fix  $D C E$ 

assume  $DC: D \wedge C$ 
have  $D C \wedge C$ 
apply –
apply (safe-fold-subgoals, simp, atomize (full))
apply (rule DC)
done

end

```

7.4 Utility methods

7.4.1 Finding a goal based on successful application of a method

```

method-setup find-goal =
  <Method.text-closure >> (fn m => fn ctxt => fn facts =>
    let
      fun prefer-first i = SELECT-GOAL
        (fn st' =>
          (case Seq.pull (method-evaluate m ctxt facts st') of
            SOME (st'',-) => Seq.single st''
          | NONE => Seq.empty)) i THEN prefer-tac i

    in SIMPLE-METHOD (FIRSTGOAL prefer-first) facts end)

```

Ensure that the proof state is in a certain case of a case distinction:

```

method in-case for  $x = \text{match premises in } t = x \text{ for } t \Rightarrow \text{succeed}$ 

```

Focus on a case in a case distinction:

```

method find-case for  $x = \text{find-goal } \langle \text{in-case } x \rangle$ 

```

notepad begin

```

fix  $A B$ 
assume  $A: A$  and  $B: B$ 

have  $A A B$ 
  apply (find-goal <match conclusion in  $B \Rightarrow \langle - \rangle$ >)
  apply (rule B)
  by (rule A)+

have  $A \wedge A A \wedge A B$ 
  apply (find-goal <fails <simp>>) — find the first goal which cannot be simplified
  apply (rule B)
  by (simp add: A)+

have  $B A A \wedge A$ 

```


apply (*find-goal* \langle succeeds \langle simp $\rangle\rangle$) — find the first goal which can be simplified (without doing so)

apply (*rule conjI*)
by (*rule A B*) $+$

fix $x::'a$ **and** $S :: nat \Rightarrow 'a$ **and** $T R$

have

$x = T \Longrightarrow A$
 $x = S 10 \Longrightarrow B$
 $x = R \Longrightarrow B$

apply –

apply (*find-case S ?n*)
apply (*fails* \langle in-case $R\rangle$)
apply (*in-case S ?n*)
by (*rule A B*) $+$

end

7.4.2 Remove redundant subgoals

Tries to solve subgoals by assuming the others and then using the given method. Backtracks over all possible re-orderings of the subgoals.

context begin

definition *protect* (*PROP P*) $\equiv P$

lemma *protectE*: *PROP protect P* \Longrightarrow (*PROP P* \Longrightarrow *PROP R*) \Longrightarrow *PROP R* **by** (*simp add: protect-def*)

private lemmas *protect-thin* = *thin-rl*[**where** $V=PROP$ *protect P for P*]

private lemma *context-conjunction'I-protected*:

assumes P : *PROP P*
assumes PQ : *PROP protect (PROP P)* \Longrightarrow *PROP Q*
shows
 $PROP P \ \&\ \& \ PROP Q$
apply (*simp add: conjunction'-def*)
apply (*rule P*)
apply (*rule PQ*)
apply (*simp add: protect-def*)
by (*rule P*)

private lemma *conjunction'-sym*: $PROP P \ \&\ \& \ PROP Q \Longrightarrow PROP Q \ \&\ \& \ PROP P$

apply (*simp add: conjunction'-def*)
apply (*frule conjunctionD1*)
apply (*drule conjunctionD2*)
apply (*rule conjunctionI*)
by *assumption+*

```

private lemmas context-conjuncts'I =
  context-conjunction'I-protected
  context-conjunction'I-protected[THEN conjunction'-sym]

method distinct-subgoals-strong methods m =
  (safe-fold-subgoals,
   (intro context-conjuncts'I;
    (((elim protectE conjunction'E)?, solves ⟨m⟩)
     | (elim protect-thin)?)))?

end

method forward-solve methods fwd m =
  (fwd, prefer-last, fold-subgoals, safe-meta-conjuncts, rule conjunction'I,
   defer-tac, ((intro conjunction'I)?, solves ⟨m⟩))[I]

method frule-solve methods m uses rule = (forward-solve ⟨frule rule⟩ ⟨m⟩)
method drule-solve methods m uses rule = (forward-solve ⟨drule rule⟩ ⟨m⟩)

notepad begin
  {
  fix A B C D E
  assume ABCD:  $A \implies B \implies C \implies D$ 
  assume ACD:  $A \implies C \implies D$ 
  assume DE:  $D \implies E$ 
  assume B C

  have  $A \implies D$ 
  apply (frule-solve ⟨simp add: ⟨B⟩ ⟨C⟩⟩ rule: ABCD)
  apply (drule-solve ⟨simp add: ⟨B⟩ ⟨C⟩⟩ rule: ACD)
  apply (match premises in  $A \implies \langle fail \rangle \mid - \implies \langle - \rangle$ )
  apply assumption
  done
  }
end

notepad begin
  {
  fix A B C
  assume A: A
  have  $A B \implies A$ 
  apply –
  apply (distinct-subgoals-strong ⟨assumption⟩)
  by (rule A)
  }

```

```

have  $B \implies A$   $A$ 
by (distinct-subgoals-strong  $\langle$ assumption $\rangle$ , rule A) — backtracking required here
}

{
fix  $A B C$ 

assume  $B: B$ 
assume  $BC: B \implies C B \implies A$ 
have  $A B \longrightarrow (A \wedge C) B$ 
apply (distinct-subgoals-strong  $\langle$ simp $\rangle$ , rule B) — backtracking required here
by (simp add: BC)

}
end

```

7.5 Attribute methods (for use with *rule-by-method* attributes)

```

method prove-prop-raw for  $P :: prop$  methods  $m =$ 
  (erule thin-rl, rule revcut-rl[of PROP P],
   solves  $\langle$ match conclusion in -  $\implies$   $\langle$ m $\rangle$  $\rangle$ )

method prove-prop for  $P :: prop =$  (prove-prop-raw PROP P  $\langle$ auto $\rangle$ )

experiment begin

lemma assumes  $A$ [simp]: $A$  shows  $A$  by (rule [[ $\langle$ @prove-prop  $A$  $\rangle$ ]])

end

```

7.6 Shortcuts for *prove-prop*.

Note these are less efficient than using the raw syntax because the facts are re-proven every time.

```

method ruleP for  $P :: prop =$  (catch  $\langle$ rule [[ $\langle$ @prove-prop PROP P $\rangle$ ]] $\rangle$   $\langle$ fail $\rangle$ )
method insertP for  $P :: prop =$  (catch  $\langle$ insert [[ $\langle$ @prove-prop PROP P $\rangle$ ]] $\rangle$   $\langle$ fail $\rangle$ ) $[1]$ 

experiment begin

lemma assumes  $A$ [simp]: $A$  shows  $A$  by (ruleP False | ruleP  $A$ )
lemma assumes  $A:A$  shows  $A$  by (ruleP  $\wedge P. P \implies P \implies P$ , rule A, rule A)

end

context begin

```

private definition *bool-protect* (*b::bool*) \equiv *b*

lemma *bool-protectD*:
bool-protect P \implies *P*
unfolding *bool-protect-def* **by** *simp*

lemma *bool-protectI*:
P \implies *bool-protect P*
unfolding *bool-protect-def* **by** *simp*

When you want to apply a rule/tactic to transform a potentially complex goal into another one manually, but want to indicate that any fresh emerging goals are solved by a more brutal method. E.g. *apply (solves-emerging frule x=... in my-rulefastforce simp: ... intro! ...*

method *solves-emerging* **methods** *m1 m2* = (*rule bool-protectD*, (*m1* ; (*rule bool-protectI* | (*m2*; *fail*))))

end

end

theory *Less-Monad-Syntax*
imports *HOL-Library.Monad-Syntax*
begin

no-syntax
-thenM :: [*'a*, *'b*] \Rightarrow *'c* (**infixr** >> 54)

no-notation
Monad-Syntax.bind (**infixr** >>= 54)

notation (**output**)
bind-do (**infixl** >>= 54)

translations
CONST bind-do <= *CONST bind*

end

Chapter 8

Option Monad (State Reader)

```
theory Reader-Monad
imports
  More-Lib
  Less-Monad-Syntax
begin
```

```
type-synonym ('s,'a) lookup = 's ⇒ 'a option
```

Similar to *map-option* but the second function returns option as well

definition

```
opt-map :: ('s,'a) lookup ⇒ ('a ⇒ 'b option) ⇒ ('s,'b) lookup (infixl |> 54)
```

where

```
f |> g ≡ λs. case f s of None ⇒ None | Some x ⇒ g x
```

```
abbreviation opt-map-Some :: ('s → 'a) ⇒ ('a ⇒ 'b) ⇒ 's → 'b (infixl ||> 54)
```

where

```
f ||> g ≡ f |> (Some ∘ g)
```

```
lemmas opt-map-Some-def = opt-map-def
```

lemma *opt-map-cong* [*fundef-cong*]:

```
[[ f = f'; ∧ v s. f s = Some v ⇒ g v = g' v ]] ⇒ f |> g = f' |> g'
```

```
by (rule ext) (simp add: opt-map-def split: option.splits)
```

lemma *in-opt-map-eq*:

```
((f |> g) s = Some v) = (∃ v'. f s = Some v' ∧ g v' = Some v)
```

```
by (simp add: opt-map-def split: option.splits)
```

lemma *opt-mapE*:

```
[[ (f |> g) s = Some v; ∧ v'. [[f s = Some v'; g v' = Some v]] ⇒ P ]] ⇒ P
```

```
by (auto simp: in-opt-map-eq)
```

lemma *opt-map-upd-None*:

$f(x := None) |> g = (f |> g)(x := None)$
by (*auto simp: opt-map-def*)

lemma *opt-map-upd-Some*:
 $f(x \mapsto v) |> g = (f |> g)(x := g v)$
by (*auto simp: opt-map-def*)

lemmas *opt-map-upd[simp] = opt-map-upd-None opt-map-upd-Some*

declare *None-upd-eq[simp]*

lemma $[(f |> g) x = None; g v = None] \implies f(x \mapsto v) |> g = f |> g$
by *simp*

definition
 $obind :: ('s, 'a) lookup \Rightarrow ('a \Rightarrow ('s, 'b) lookup) \Rightarrow ('s, 'b) lookup$ (**infixl** $|>>$ 53)
where
 $f |>> g \equiv \lambda s. \text{case } f \text{ s of } None \Rightarrow None \mid Some\ x \Rightarrow g\ x\ s$

adhoc-overloading
Monad-Syntax.bind obind

definition
 $ofail = K\ None$

definition
 $oreturn = K\ o\ Some$

definition
 $oassert\ P \equiv \text{if } P \text{ then } oreturn\ () \text{ else } ofail$

definition $oapply :: 'a \Rightarrow ('a \Rightarrow 'b\ option) \Rightarrow 'b\ option$
where
 $oapply\ x \equiv \lambda s. s\ x$

If the result can be an exception. Corresponding `bindE` would be analogous to lifting in `NonDetMonad`.

definition
 $oreturnOk\ x = K\ (Some\ (Inr\ x))$

definition
 $othrow\ e = K\ (Some\ (Inl\ e))$

definition
 $oguard\ G \equiv (\lambda s. \text{if } G\ s \text{ then } Some\ () \text{ else } None)$

definition

condition $c L R \equiv (\lambda s. \text{if } c \ s \ \text{then } L \ s \ \text{else } R \ s)$

definition

oskip $\equiv \text{oreturn } ()$

Monad laws

lemma *oreturn-bind* [*simp*]: $(\text{oreturn } x \ |>> \ f) = f \ x$
by (*auto simp add: oreturn-def obind-def K-def*)

lemma *obind-return* [*simp*]: $(m \ |>> \ \text{oreturn}) = m$
by (*auto simp add: oreturn-def obind-def K-def split: option.splits*)

lemma *obind-assoc*:

$(m \ |>> \ f) \ |>> \ g = m \ |>> \ (\lambda x. f \ x \ |>> \ g)$
by (*auto simp add: oreturn-def obind-def K-def split: option.splits*)

Binding fail

lemma *obind-fail* [*simp*]:
 $f \ |>> \ (\lambda -. \ \text{ofail}) = \text{ofail}$
by (*auto simp add: ofail-def obind-def K-def split: option.splits*)

lemma *ofail-bind* [*simp*]:
 $\text{ofail} \ |>> \ m = \text{ofail}$
by (*auto simp add: ofail-def obind-def K-def split: option.splits*)

Function package setup

lemma *opt-bind-cong* [*fundef-cong*]:
 $\llbracket f = f'; \bigwedge v \ s. f' \ s = \text{Some } v \implies g \ v \ s = g' \ v \ s \rrbracket \implies f \ |>> \ g = f' \ |>> \ g'$
by (*rule ext*) (*simp add: obind-def split: option.splits*)

lemma *opt-bind-cong-apply* [*fundef-cong*]:
 $\llbracket f \ s = f' \ s; \bigwedge v. f' \ s = \text{Some } v \implies g \ v \ s = g' \ v \ s \rrbracket \implies (f \ |>> \ g) \ s = (f' \ |>> \ g') \ s$
by (*simp add: obind-def split: option.splits*)

lemma *oassert-bind-cong* [*fundef-cong*]:
 $\llbracket P = P'; P' \implies m = m' \rrbracket \implies \text{oassert } P \ |>> \ m = \text{oassert } P' \ |>> \ m'$
by (*auto simp: oassert-def*)

lemma *oassert-bind-cong-apply* [*fundef-cong*]:
 $\llbracket P = P'; P' \implies m \ () \ s = m' \ () \ s \rrbracket \implies (\text{oassert } P \ |>> \ m) \ s = (\text{oassert } P' \ |>> \ m') \ s$
by (*auto simp: oassert-def*)

lemma *oreturn-bind-cong* [*fundef-cong*]:
 $\llbracket x = x'; m \ x' = m' \ x' \rrbracket \implies \text{oreturn } x \ |>> \ m = \text{oreturn } x' \ |>> \ m'$
by *simp*

lemma *oreturn-bind-cong-apply* [*fundef-cong*]:

$\llbracket x = x'; m \ x' \ s = m' \ x' \ s \rrbracket \Longrightarrow (\text{oreturn } x \ |>> \ m) \ s = (\text{oreturn } x' \ |>> \ m') \ s$
by *simp*

lemma *oreturn-bind-cong2* [*fundef-cong*]:

$\llbracket x = x'; m \ x' = m' \ x' \rrbracket \Longrightarrow (\text{oreturn } \$ \ x) \ |>> \ m = (\text{oreturn } \$ \ x') \ |>> \ m'$
by *simp*

lemma *oreturn-bind-cong2-apply* [*fundef-cong*]:

$\llbracket x = x'; m \ x' \ s = m' \ x' \ s \rrbracket \Longrightarrow ((\text{oreturn } \$ \ x) \ |>> \ m) \ s = ((\text{oreturn } \$ \ x') \ |>> \ m') \ s$
by *simp*

lemma *ocondition-cong* [*fundef-cong*]:

$\llbracket c = c'; \bigwedge s. c' \ s \Longrightarrow l \ s = l' \ s; \bigwedge s. \neg c' \ s \Longrightarrow r \ s = r' \ s \rrbracket$
 $\Longrightarrow \text{ocondition } c \ l \ r = \text{ocondition } c' \ l' \ r'$
by (*auto simp: ocondition-def*)

Decomposition

lemma *ocondition-K-true* [*simp*]:

$\text{ocondition } (\lambda-. \ \text{True}) \ T \ F = T$
by (*simp add: ocondition-def*)

lemma *ocondition-K-false* [*simp*]:

$\text{ocondition } (\lambda-. \ \text{False}) \ T \ F = F$
by (*simp add: ocondition-def*)

lemma *ocondition-False*:

$\llbracket \bigwedge s. \neg P \ s \rrbracket \Longrightarrow \text{ocondition } P \ L \ R = R$
by (*rule ext, clarsimp simp: ocondition-def*)

lemma *ocondition-True*:

$\llbracket \bigwedge s. P \ s \rrbracket \Longrightarrow \text{ocondition } P \ L \ R = L$
by (*rule ext, clarsimp simp: ocondition-def*)

lemma *in-oreturn* [*simp*]:

$(\text{oreturn } x \ s = \text{Some } v) = (v = x)$
by (*auto simp: oreturn-def K-def*)

lemma *oreturnE*:

$\llbracket \text{oreturn } x \ s = \text{Some } v; v = x \Longrightarrow P \rrbracket \Longrightarrow P$
by *simp*

lemma *in-ofail* [*simp*]:

$\text{ofail } s \neq \text{Some } v$
by (*auto simp: ofail-def K-def*)

lemma *ofailE*:

$\text{ofail } s = \text{Some } v \Longrightarrow P$
by *simp*

lemma *in-oassert-eq* [*simp*]:
 $(oassert\ P\ s = Some\ v) = P$
by (*simp add: oassert-def*)

lemma *oassert-True* [*simp*]:
 $oassert\ True = oreturn\ ()$
by (*simp add: oassert-def*)

lemma *oassert-False* [*simp*]:
 $oassert\ False = ofail$
by (*simp add: oassert-def*)

lemma *oassertE*:
 $\llbracket oassert\ P\ s = Some\ v; P \implies Q \rrbracket \implies Q$
by *simp*

lemma *in-obind-eq*:
 $((f\ |>>\ g)\ s = Some\ v) = (\exists\ v'. f\ s = Some\ v' \wedge g\ v'\ s = Some\ v)$
by (*simp add: obind-def split: option.splits*)

lemma *obind-eqI*:
 $\llbracket f\ s = f\ s'; \bigwedge x. f\ s = Some\ x \implies g\ x\ s = g'\ x\ s' \rrbracket \implies obind\ f\ g\ s = obind\ f\ g'$
 s'
by (*simp add: obind-def split: option.splits*)

lemma *obind-eqI-full*:
 $\llbracket f\ s = f\ s'; \bigwedge x. \llbracket f\ s = Some\ x; f\ s' = f\ s \rrbracket \implies g\ x\ s = g'\ x\ s' \rrbracket$
 $\implies obind\ f\ g\ s = obind\ f\ g'\ s'$
by (*drule sym[where s=f s]*)
(clarsimp simp: obind-def split: option.splits)

lemma *obindE*:
 $\llbracket (f\ |>>\ g)\ s = Some\ v;$
 $\bigwedge v'. \llbracket f\ s = Some\ v'; g\ v'\ s = Some\ v \rrbracket \implies P \rrbracket \implies P$
by (*auto simp: in-obind-eq*)

lemma *in-othrow-eq* [*simp*]:
 $(othrow\ e\ s = Some\ v) = (v = Inl\ e)$
by (*auto simp: othrow-def K-def*)

lemma *othrowE*:
 $\llbracket othrow\ e\ s = Some\ v; v = Inl\ e \implies P \rrbracket \implies P$
by *simp*

lemma *in-oreturnOk-eq* [*simp*]:
 $(oreturnOk\ x\ s = Some\ v) = (v = Inr\ x)$
by (*auto simp: oreturnOk-def K-def*)

lemma *oreturnOkE*:
 $\llbracket \text{oreturnOk } x \ s = \text{Some } v; v = \text{Inr } x \implies P \rrbracket \implies P$
by *simp*

lemmas *omonadE* [*elim!*] =
opt-mapE obindE oreturnE ofailE othrowE oreturnOkE oassertE

lemma *in-opt-map-Some-eq*:
 $((f \ ||> \ g) \ x = \text{Some } y) = (\exists v. f \ x = \text{Some } v \wedge g \ v = y)$
by (*simp add: in-opt-map-eq*)

lemma *in-opt-map-None-eq*[*simp*]:
 $((f \ ||> \ g) \ x = \text{None}) = (f \ x = \text{None})$
by (*simp add: opt-map-def split: option.splits*)

lemma *oreturn-comp*[*simp*]:
 $\text{oreturn } x \circ f = \text{oreturn } x$
by (*simp add: oreturn-def K-def o-def*)

lemma *ofail-comp*[*simp*]:
 $\text{ofail} \circ f = \text{ofail}$
by (*auto simp: ofail-def K-def*)

lemma *oassert-comp*[*simp*]:
 $\text{oassert } P \circ f = \text{oassert } P$
by (*simp add: oassert-def*)

lemma *fail-apply*[*simp*]:
 $\text{ofail } s = \text{None}$
by (*simp add: ofail-def K-def*)

lemma *oassert-apply*[*simp*]:
 $\text{oassert } P \ s = (\text{if } P \ \text{then } \text{Some } () \ \text{else } \text{None})$
by (*simp add: oassert-def*)

lemma *oreturn-apply*[*simp*]:
 $\text{oreturn } x \ s = \text{Some } x$
by *simp*

lemma *oapply-apply*[*simp*]:
 $\text{oapply } x \ s = s \ x$
by (*simp add: oapply-def*)

lemma *obind-comp-dist*:
 $\text{obind } f \ g \circ h = \text{obind } (f \circ h) \ (\lambda x. g \ x \circ h)$
by (*auto simp: obind-def split: option.splits*)

lemma *if-comp-dist*:

(if P then f else g) o h = (if P then $f o h$ else $g o h$)
by *auto*

8.1 "While" loops over option monad.

This is an inductive definition of a while loop over the plain option monad
(without passing through a state)

inductive-set

option-while' :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a option) \Rightarrow 'a option rel
for C B

where

final: $\neg C r \Longrightarrow (Some\ r, Some\ r) \in option_while'\ C\ B$
| *fail*: $\llbracket C\ r; B\ r = None \rrbracket \Longrightarrow (Some\ r, None) \in option_while'\ C\ B$
| *step*: $\llbracket C\ r; B\ r = Some\ r'; (Some\ r', sr') \in option_while'\ C\ B \rrbracket$
 $\Longrightarrow (Some\ r, sr') \in option_while'\ C\ B$

definition

option-while C B $r \equiv$
(if $(\exists s. (Some\ r, s) \in option_while'\ C\ B)$ then
(*THE* $s. (Some\ r, s) \in option_while'\ C\ B$) else *None*)

lemma *option-while'-inj*:

assumes $(s, s') \in option_while'\ C\ B$ $(s, s'') \in option_while'\ C\ B$
shows $s' = s''$

using *assms* **by** (*induct* rule: *option-while'.induct*) (*auto elim*: *option-while'.cases*)

lemma *option-while'-inj-step*:

$\llbracket C\ s; B\ s = Some\ s'; (Some\ s, t) \in option_while'\ C\ B; (Some\ s', t') \in option_while'\ C\ B \rrbracket \Longrightarrow t = t'$

by (*metis option-while'.step option-while'-inj*)

lemma *option-while'-THE*:

assumes $(Some\ r, sr') \in option_while'\ C\ B$
shows (*THE* $s. (Some\ r, s) \in option_while'\ C\ B$) = sr'
using *assms* **by** (*blast dest*: *option-while'-inj*)

lemma *option-while'-simps*:

$\neg C\ s \Longrightarrow option_while\ C\ B\ s = Some\ s$
 $C\ s \Longrightarrow B\ s = None \Longrightarrow option_while\ C\ B\ s = None$
 $C\ s \Longrightarrow B\ s = Some\ s' \Longrightarrow option_while\ C\ B\ s = option_while\ C\ B\ s'$
 $(Some\ s, ss') \in option_while'\ C\ B \Longrightarrow option_while\ C\ B\ s = ss'$
using *option-while'-inj-step*[of $C\ s\ B\ s'$]

by (*auto simp*: *option-while-def option-while'-THE*

intro: *option-while'.intros*

dest: *option-while'-inj*

elim: *option-while'.cases*)

lemma *option-while-rule*:

assumes *option-while* $C B s = \text{Some } s'$
assumes $I s$
assumes *istep*: $\bigwedge s s'. C s \implies I s \implies B s = \text{Some } s' \implies I s'$
shows $I s' \wedge \neg C s'$
proof –
{ **fix** $ss ss'$ **assume** $(ss, ss') \in \text{option-while}' C B ss = \text{Some } s ss' = \text{Some } s'$
then have *?thesis* **using** $\langle I s \rangle$
by (*induct arbitrary: s*) (*auto intro: istep*) }
then show *?thesis* **using** *assms(1)*
by (*auto simp: option-while-def option-while'-THE split: if-split-asm*)
qed

lemma *option-while'-term*:
assumes $I r$
assumes *wf M*
assumes *step-less*: $\bigwedge r r'. \llbracket I r; C r; B r = \text{Some } r' \rrbracket \implies (r', r) \in M$
assumes *step-I*: $\bigwedge r r'. \llbracket I r; C r; B r = \text{Some } r' \rrbracket \implies I r'$
obtains sr' **where** $(\text{Some } r, sr') \in \text{option-while}' C B$
apply *atomize-elim*
using *assms(2,1)*
proof *induct*
case (*less r*)
show *?case*
proof (*cases C r B r rule: bool.exhaust[case-product option.exhaust]*)
case (*True-Some r'*)
then have $(r', r) \in M I r'$
by (*auto intro: less step-less step-I*)
then obtain sr' **where** $(\text{Some } r', sr') \in \text{option-while}' C B$
by *atomize-elim (rule less)*
then have $(\text{Some } r, sr') \in \text{option-while}' C B$
using *True-Some* **by** (*auto intro: option-while'.intros*)
then show *?thesis ..*
qed (*auto intro: option-while'.intros*)
qed

lemma *option-while-rule'*:
assumes *option-while* $C B s = ss'$
assumes *wf M*
assumes $I (\text{Some } s)$
assumes *less*: $\bigwedge s s'. C s \implies I (\text{Some } s) \implies B s = \text{Some } s' \implies (s', s) \in M$
assumes *step*: $\bigwedge s s'. C s \implies I (\text{Some } s) \implies B s = \text{Some } s' \implies I (\text{Some } s')$
assumes *final*: $\bigwedge s. C s \implies I (\text{Some } s) \implies B s = \text{None} \implies I \text{None}$
shows $I ss' \wedge (\text{case } ss' \text{ of } \text{Some } s' \Rightarrow \neg C s' \mid - \Rightarrow \text{True})$
proof –
define $ss \equiv \text{Some } s$
obtain $ss1'$ **where** $(\text{Some } s, ss1') \in \text{option-while}' C B$
using *assms(3,2,4,5)* **by** (*rule option-while'-term*)
then have $*$: $(ss, ss') \in \text{option-while}' C B$ **using** $\langle \text{option-while } C B s = ss' \rangle$
by (*auto simp: option-while-simps ss-def*)

```

show ?thesis
proof (cases ss')
  case (Some s') with * ss-def show ?thesis using ⟨I -⟩
  by (induct arbitrary:s) (auto intro: step)
next
  case None with * ss-def show ?thesis using ⟨I -⟩
  by (induct arbitrary:s) (auto intro: step final)
qed
qed

```

8.2 Lift *option-while* to the $(\text{'a}, \text{'s})$ lookup monad

definition

```

owhile :: ('a ⇒ 's ⇒ bool) ⇒ ('a ⇒ ('s,'a) lookup) ⇒ 'a ⇒ ('s,'a) lookup
where
owhile c b a ≡ λs. option-while (λa. c a s) (λa. b a s) a

```

lemma *owhile-unroll*:

```

owhile C B r = ocondition (C r) (B r |>> owhile C B) (oreturn r)
by (auto simp: ocondition-def obind-def oreturn-def owhile-def
  option-while-simps K-def split: option.split)

```

rule for terminating loops

lemma *owhile-rule*:

```

assumes I r s
assumes wf M
assumes less: ∧r r'. [I r s; C r s; B r s = Some r] ⇒ (r',r) ∈ M
assumes step: ∧r r'. [I r s; C r s; B r s = Some r] ⇒ I r' s
assumes fail: ∧r. [I r s; C r s; B r s = None] ⇒ Q None
assumes final: ∧r. [I r s; ¬C r s] ⇒ Q (Some r)
shows Q (owhile C B r s)

```

proof –

```

let ?rs' = owhile C B r s
have (case ?rs' of Some r ⇒ I r s | - ⇒ Q None)
  ∧ (case ?rs' of Some r' ⇒ ¬ C r' s | - ⇒ True)
  by (rule option-while-rule'[where B=λr. B r s and s=r, OF - ⟨wf -⟩])
  (auto simp: owhile-def intro: assms)
then show ?thesis by (auto intro: final split: option.split-asm)
qed

```

end

theory *Option-MonadND*

```

imports
  Reader-Monad
begin

```

definition

$$ogets :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b \text{ option})$$
where

$$ogets f \equiv (\lambda s. \text{Some } (f s))$$
definition

$$ocatch :: ('s, ('e + 'a)) \text{ lookup} \Rightarrow ('e \Rightarrow ('s, 'a) \text{ lookup}) \Rightarrow ('s, 'a) \text{ lookup}$$

$$(\text{infix } <ocatch> 10)$$
where

$$f <ocatch> \text{ handler} \equiv \text{do } \{$$

$$x \leftarrow f;$$

$$\text{case } x \text{ of } \text{Inr } b \Rightarrow \text{oreturn } b \mid \text{Inl } e \Rightarrow \text{handler } e$$

$$\}$$
definition

$$odrop :: ('s, 'e + 'a) \text{ lookup} \Rightarrow ('s, 'a) \text{ lookup}$$
where

$$odrop f \equiv \text{do } \{$$

$$x \leftarrow f;$$

$$\text{case } x \text{ of } \text{Inr } b \Rightarrow \text{oreturn } b \mid \text{Inl } e \Rightarrow \text{ofail}$$

$$\}$$
definition

$$osequence-x :: ('s, 'a) \text{ lookup list} \Rightarrow ('s, \text{unit}) \text{ lookup}$$
where

$$osequence-x xs \equiv \text{foldr } (\lambda x y. \text{do } \{ x; y \}) xs (\text{oreturn } ())$$
definition

$$osequence :: ('s, 'a) \text{ lookup list} \Rightarrow ('s, 'a \text{ list}) \text{ lookup}$$
where

$$osequence xs \equiv \text{let } mcons = (\lambda p q. p |>> (\lambda x. q |>> (\lambda y. \text{oreturn } (x\#y))))$$

$$\text{in foldr } mcons xs (\text{oreturn } [])$$
definition

$$omap :: ('a \Rightarrow ('s, 'b) \text{ lookup}) \Rightarrow 'a \text{ list} \Rightarrow ('s, 'b \text{ list}) \text{ lookup}$$
where

$$omap f xs \equiv osequence (\text{map } f xs)$$
definition

$$opt-cons :: 'a \text{ option} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} (\text{infixr } o\# 65)$$
where

$$opt-cons x xs \equiv \text{case } x \text{ of } \text{None} \Rightarrow xs \mid \text{Some } x' \Rightarrow x' \# xs$$

end

```

theory Apply-Trace
imports
  Main
begin

ML-file <ml-helpers/ThmExtras.ML>

ML <

signature APPLY-TRACE =
sig
  val apply-results :
    {silent-fail : bool} ->
    (Proof.context -> thm -> ((string * int option) * term) list -> unit) ->
    Method.text-range -> Proof.state -> Proof.state Seq.result Seq.seq

  (* Lower level interface. *)
  val can-clear : theory -> bool
  val clear-deps : thm -> thm
  val join-deps : thm -> thm -> thm
  val used-facts : Proof.context -> thm -> ((string * int option) * term) list
  val pretty-deps: bool -> (string * Position.T) option -> Proof.context -> thm
->
  ((string * int option) * term) list -> Pretty.T
end

structure Apply-Trace : APPLY-TRACE =
struct

(*TODO: Add more robust oracle without hyp clearing *)
fun thm-to-cterm keep-hyps thm =
let

  val thy = Thm.theory-of-thm thm
  val pairs = Thm.tpairs-of thm
  val ceqs = map (Thm.global-cterm-of thy o Logic.mk-equals) pairs
  val hyps = Thm.chyps-of thm
  val prop = Thm.cprop-of thm
  val thm' = if keep-hyps then Drule.list-implies (hyps,prop) else prop

in
  Drule.list-implies (ceqs,thm') end

val (-, clear-thm-deps') =
  Context.>>> (Context.map-theory-result (Thm.add-oracle (binding <count-cheat>,
thm-to-cterm false)));

```

```

fun clear-deps thm =
let
  val thm' = try clear-thm-deps' thm
  |> Option.map (fold (fn - => fn t => (@{thm Pure.reflexive} RS t)) (Thm.tpairs-of
thm))

in case thm' of SOME thm' => thm' | NONE => error Can't clear deps here end

fun can-clear thy = Context.subthy(@{theory},thy)

fun join-deps pre-thm post-thm =
let
  val pre-thm' = Thm.flexflex-rule NONE pre-thm |> Seq.hd
  |> Thm.adjust-maxidx-thm (Thm.maxidx-of post-thm + 1)
in
  Conjunction.intr pre-thm' post-thm |> Conjunction.elim |> snd
end

fun get-ref-from-nm' nm =
let
  val exploded = space-explode - nm;
  val base = List.take (exploded, (length exploded) - 1) |> space-implode -
  val idx = List.last exploded |> Int.fromString;
in if is-some idx andalso base <> then SOME (base, the idx) else NONE end

fun get-ref-from-nm nm = Option.join (try get-ref-from-nm' nm);

fun maybe-nth l = try (curry List.nth l)

fun fact-from-derivation ctxt xnm =
let
  val facts = Proof-Context.facts-of ctxt;
  (* TODO: Check that exported local fact is equivalent to external one *)

  val idx-result =
let
  val (name', idx) = get-ref-from-nm xnm |> the;
  val entry = try (Facts.retrieve (Context.Proof ctxt) facts) (name', Posi-
tion.none) |> the;
  val thm = maybe-nth (#thms entry) (idx - 1) |> the;
in SOME (xnm, thm) end handle Option => NONE;

fun non-idx-result () =
let
  val entry = try (Facts.retrieve (Context.Proof ctxt) facts) (xnm, Position.none)

```



```

|> the;
  val thm = try the-single (#thms entry) |> the;
  in SOME (#name entry, thm) end handle Option => NONE;

in
  case idx-result of
    SOME thm => SOME thm
  | NONE => non-idx-result ()
  end

fun most-local-fact-of ctxt xnm =
  let
    val local-name = try (fn xnm => Long-Name.explode xnm |> tl |> tl |> Long-Name.implode)
    xnm |> the;
  in SOME (fact-from-derivation ctxt local-name |> the) end handle Option =>
    fact-from-derivation ctxt xnm;

fun thms-of (PBody {thms,...}) = thms

fun proof-body-descend' f get-fact (ident, thm-node) deptab = let
  val nm = Proofterm.thm-node-name thm-node
  val body = Proofterm.thm-node-body thm-node
in
  (if not (f nm) then
    (Inttab.update-new (ident, SOME (nm, get-fact nm |> the)) deptab handle
    Inttab.DUP - => deptab)
  else raise Option) handle Option =>
    ((fold (proof-body-descend' f get-fact) (thms-of (Future.join body))
    (Inttab.update-new (ident, NONE) deptab)) handle Inttab.DUP - => deptab)
end

fun used-facts' f get-fact thm =
  let
    val body = thms-of (Thm.proof-body-of thm);
  in fold (proof-body-descend' f get-fact) body Inttab.empty end

fun used-pbody-facts ctxt thm =
  let
    val nm = Thm.get-name-hint thm;
    val get-fact = most-local-fact-of ctxt;
  in
    used-facts' (fn nm' => nm' = orelse nm' = nm) get-fact thm
    |> Inttab.dest |> map-filter snd |> map snd |> map (apsnd (Thm.prop-of))
  end

fun raw-primitive-text f = Method.Basic (fn - => ((K (fn (ctxt, thm) => Seq.make-results
(Seq.single (ctxt, f thm))))))

```

```

(*Find local facts from new hyps*)
fun used-local-facts ctxt thm =
  let
    val hyps = Thm.hyps-of thm
    val facts = Proof-Context.facts-of ctxt |> Facts.dest-static true []

    fun match-hyp hyp =
      let
        fun get (nm, thms) =
          case (get-index (fn t => if (Thm.prop-of t) aconv hyp then SOME hyp else
NONE) thms)
          of SOME t => SOME (nm, t)
            | NONE => NONE

        in
          get-first get facts
        end

      in
        map-filter match-hyp hyps end

    fun used-facts ctxt thm =
      let
        val used-from-pbody = used-pbody-facts ctxt thm |> map (fn (nm, t) => ((nm, NONE), t))
        val used-from-hyps = used-local-facts ctxt thm |> map (fn (nm, (i, t)) =>
((nm, SOME i), t))
        in
          in
            (used-from-hyps @ used-from-pbody)
          end
        end

    (* Perform refinement step, and run the given stateful function
    against computed dependencies afterwards. *)
    fun refine args f text state =
      let

        val ctxt = Proof.context-of state

        val thm = Proof.simple-goal state |> #goal

        fun save-deps deps = f ctxt thm deps

      in
        in
          if (can-clear (Proof.theory-of state)) then
            Proof.refine (Method.Combinator (Method.no-combinator-info, Method.Then,
[raw-primitive-text (clear-deps), text,
raw-primitive-text (fn thm' => (save-deps (used-facts ctxt thm^); join-deps thm

```

```

thm')))) state
  else
    (if (#silent-fail args) then (save-deps [];Proof.refine text state) else error
Apply-Trace theory must be imported to trace applies)
end

(* Boilerplate from Proof.ML *)

fun method-error kind pos state =
  Seq.single (Proof-Display.method-error kind pos (Proof.raw-goal state));

fun apply args f text = Proof.assert-backward #> refine args f text #>
  Seq.maps-results (Proof.apply ((raw-primitive-text I),(Position.none, Position.none)));

fun apply-results args f (text, range) =
  Seq.APPEND (apply args f text, method-error (Position.range-position range));

structure Filter-Thms = Named-Thms
(
  val name = @{binding no-trace}
  val description = thms to be ignored from tracing
)

(* Print out the found dependencies. *)
fun pretty-deps only-names query ctxt thm deps =
let
  (* Remove duplicates. *)
  val deps = sort-distinct (prod-ord (prod-ord string-ord (option-ord int-ord)) Term-Ord.term-ord)
  deps

  (* Fetch canonical names and theorems. *)
  val deps = map (fn (ident, term) => ThmExtras.adjust-thm-name ctxt ident
term) deps

  (* Remove boring theorems. *)
  val deps = subtract (fn (a, ThmExtras.FoundName (-, thm)) => Thm.eq-thm
(thm, a)
| - => false) (Filter-Thms.get ctxt) deps

  val deps = case query of SOME (raw-query,pos) =>
let
  val pos' = perhaps (try (Position.shift-offsets {remove-id = false} 1)) pos;
  val q = Find-Theorems.read-query pos' raw-query;
  val results = Find-Theorems.find-theorems-cmd ctxt (SOME thm) (SOME
1000000000) false q
|> snd
|> map ThmExtras.fact-ref-to-name;

```

```

    (* Only consider theorems from our query. *)

    val deps = inter (fn (ThmExtras.FoundName (nmidx,-), ThmExtras.FoundName
(nmidx',-)) => nmidx = nmidx'
                    | - => false) results deps
    in deps end
  | - => deps

in
  if only-names then
    Pretty.block
      (Pretty.separate (map (ThmExtras.pretty-fact only-names ctxt) deps))
    else
      (* Pretty-print resulting theorems. *)
      Pretty.big-list used theorems:
        (map (Pretty.item o single o ThmExtras.pretty-fact only-names ctxt) deps)
  end

val - = Context.>> (Context.map-theory Filter-Thms.setup)

end
>

end

theory Apply-Trace-Cmd
imports Apply-Trace
keywords apply-trace :: prf-script
begin

ML<

val - =
  Outer-Syntax.command @{command-keyword apply-trace} initial refinement step
(unstructured)

  (Args.mode only-names -- (Scan.option (Parse.position Parse.cartouche)) --
Method.parse >>
  (fn ((on,query),text) => Toplevel.proofs (Apply-Trace.apply-results {silent-fail
= false}
  (Pretty.writeln ooo (Apply-Trace.pretty-deps on query)) text)));

>

```

```
lemmas [no-trace] = protectI protectD TrueI Eq-TrueI eq-reflection
```

```
lemma (a ∧ b) = (b ∧ a)
  apply-trace auto
  oops
```

```
lemma (a ∧ b) = (b ∧ a)
  apply-trace <intro> auto
  oops
```

```
lemma
  assumes X: b = a
  assumes Y: b = a
  shows
    b = a
  apply-trace (rule Y)
  oops
```

```
locale Apply-Trace-foo = fixes b a
  assumes X: b = a
begin

  lemma shows b = a b = a
    apply -
    apply-trace (rule Apply-Trace-foo.X)
    prefer 2
    apply-trace (rule X)
    oops
end
```

```
experiment begin
```

Example of trace for grouped lemmas

```
definition ex :: nat set where
  ex = {1,2,3,4}
```

```
lemma v1: 1 ∈ ex by (simp add: ex-def)
```

```
lemma v2: 2 ∈ ex by (simp add: ex-def)
```

```
lemma v3: 3 ∈ ex by (simp add: ex-def)
```

Group several lemmas in a single one

```
lemmas vs = v1 v2 v3
```

```
lemma 2 ∈ ex
```

```
apply-trace (simp add: vs)  
oops  
  
end  
end
```

Chapter 9

Mutual CCPO Recursion

fixed-point

```
theory Mutual-CCPO-Recursion
imports
  HOL-Library.Complete-Partial-Order2
  AutoCorres-Utills
keywords fixed-point :: thy-goal-stmt
begin
```

9.1 Relate orders between locale and type classes

```
lemma gfp-lub-fun: gfp.lub-fun = Inf
unfolding fun-lub-def fun-eq-iff Inf-apply image-def by simp
```

```
lemma gfp-le-fun: gfp.le-fun = ( $\geq$ )
unfolding fun-ord-def le-fun-def fun-eq-iff Inf-apply image-def by simp
```

```
lemma
shows fst-prod-lub: fst (prod-lub luba lubb S) = luba (fst ' S)
and snd-prod-lub: snd (prod-lub luba lubb S) = lubb (snd ' S)
by (auto simp: prod-lub-def)
```

```
lemma fun-lub-app: fun-lub lub S x = lub (( $\lambda$ f. f x) ' S)
by (auto simp: fun-lub-def image-def)
```

```
lemma ccpo-Inf: class.ccpo (Inf :: 'a::complete-lattice set  $\Rightarrow$  -) ( $\geq$ ) (mk-less ( $\geq$ ))
by (rule cont-intro)
```

```
lemma ccpo-Sup: class.ccpo (Sup :: 'a::complete-lattice set  $\Rightarrow$  -) ( $\leq$ ) (mk-less ( $\leq$ ))
by (rule cont-intro)
```

```
lemma ccpo-flat: class.ccpo (flat-lub b) (flat-ord b) (mk-less (flat-ord b))
using Partial-Function.ccpo[OF flat-interpretation] .
```

lemma *monotone-pair*[*partial-function-mono*]:
 $monotone\ R\ orda\ f \implies monotone\ R\ ordb\ g \implies monotone\ R\ (rel\text{-}prod\ orda\ ordb)$
 $(\lambda x. (f\ x, g\ x))$
by (*simp add: monotone-def*)

lemma *monotone-fst'*[*partial-function-mono*]:
 $monotone\ orda\ (rel\text{-}prod\ ordb\ ordc)\ f \implies monotone\ orda\ ordb\ (\lambda x. fst\ (f\ x))$
by (*auto simp add: monotone-def rel-prod-sel*)

lemma *monotone-snd'*[*partial-function-mono*]:
 $monotone\ orda\ (rel\text{-}prod\ ordb\ ordc)\ f \implies monotone\ orda\ ordc\ (\lambda x. snd\ (f\ x))$
by (*auto simp add: monotone-def rel-prod-sel*)

lemma *monotone-id-fun-elim*[*partial-function-mono*]:
 $monotone\ (\ge)\ ord\ (\lambda x. x) \implies monotone\ (\ge)\ (fun\text{-}ord\ ord)\ (\lambda x. x)$
by (*auto simp add: monotone-def le-fun-def fun-ord-def*)

lemma *monotone-id*[*partial-function-mono*]: $monotone\ ord\ ord\ (\lambda x. x)$
by (*auto simp add: monotone-def*)

lemma *monotone-abs*:
 $(\bigwedge x. monotone\ R\ Q\ (\lambda f. F\ f\ x)) \implies monotone\ R\ (fun\text{-}ord\ Q)\ (\lambda f\ x. F\ f\ x)$
by (*simp add: monotone-def fun-ord-def*)

lemma *monotone-fun-ord-applyD*:
 $monotone\ orda\ (fun\text{-}ord\ ordb)\ f \implies monotone\ orda\ ordb\ (\lambda y. f\ y\ x)$
by (*auto simp add: monotone-def fun-ord-def*)

lemma *admissible-snd*:
 $cont\ luba\ orda\ (prod\text{-}lub\ lubb\ Inf)\ (rel\text{-}prod\ ordb\ (\ge))\ F \implies$
 $ccpo.admissible\ luba\ orda\ (\lambda x. snd\ (F\ x))$
unfolding *ccpo.admissible-def cont-def prod-eq-iff*
by (*auto simp: prod-lub-def*)

lemma *cont-apply-gfp*: $cont\ gfp.lub\text{-}fun\ gfp.le\text{-}fun\ Inf\ (\le)\ (\lambda x. x\ c)$
by (*rule cont-applyI*) (*simp add: cont-def*)

lemma *mcont-mem-gfp*:
 $mcont\ gfp.lub\text{-}fun\ gfp.le\text{-}fun\ (Inf)\ (\ge)\ F \implies gfp.admissible\ (\lambda A. x \in F\ A)$
by (*auto simp: mcont-def cont-def ccpo.admissible-def*)

lemma *mcont2mcont-call*:
 $mcont\ luba\ orda\ (fun\text{-}lub\ lubb)\ (fun\text{-}ord\ ordb)\ F \implies mcont\ luba\ orda\ lubb\ ordb$
 $(\lambda x. F\ x\ c)$
by (*auto simp: mcont-def monotone-def cont-def fun-lub-def fun-ord-def le-fun-def fun-eq-iff*)
(simp add: image-def)

lemma *preorder-fun-ord* [*partial-function-mono*]: *class.preorder* R (*mk-less* R) \implies
class.preorder (*fun-ord* R) (*mk-less* (*fun-ord* R))

by (*force simp: class.preorder-def fun-ord-def mk-less-def*)

lemma *preorder-monotone-const'* [*partial-function-mono*]:
class.preorder *leq* (*mk-less* *leq*) \implies *monotone ord leq* ($\lambda\cdot. c$)
by (*rule preorder.monotone-const*)

declare *preorder-rel-prodI* [*partial-function-mono*]
declare *gfp.preorder* [*partial-function-mono*]
declare *lfp.preorder* [*partial-function-mono*]

lemma *option-ord-preorder* [*partial-function-mono*]: *class.preorder* *option-ord* (*mk-less*
option-ord)
by *simp*

9.2 Prove admissibility of *corresXF*

lemma *not-imp-not-iff*: $(\neg A \longrightarrow \neg B) \longleftrightarrow (B \longrightarrow A)$
by *auto*

lemma *mcont-fun-lub-call*:
mcont luba orda (fun-lub lubb) (fun-ord ordb) f \implies
mcont luba orda lubb ordb ($\lambda y. f y x$)
by (*simp add: mcont-fun-lub-apply*)

lemma *chain-disj-of-subsingleton*:
Complete-Partial-Order.chain ord $\{r. P r\}$ \implies
Complete-Partial-Order.chain ord $\{r. Q r\}$ \implies
 $(\bigwedge r q. P r \implies Q q \implies r = q) \implies$
Complete-Partial-Order.chain ord $\{r. P r \vee Q r\}$
by (*auto simp: chain-def*)

definition *subsingleton-set* :: '*a set* \implies *bool* **where**
subsingleton-set $P \longleftrightarrow (\forall a \in P. \forall b \in P. a = b)$

lemma *subsingleton-set-empty*[*iff*]: *subsingleton-set* $\{\}$
by (*simp add: subsingleton-set-def*)

lemma *subsingleton-set-singleton*[*iff*]: *subsingleton-set* $\{x\}$
by (*simp add: subsingleton-set-def*)

context *ccpo*
begin

lemma *subsingleton-sets-imp-chain*:
subsingleton-set $(\bigcup Ps) \implies$ *Complete-Partial-Order.chain* $(\leq) (\bigcup Ps)$
by (*auto simp: subsingleton-set-def Complete-Partial-Order.chain-def Ball-def*)

lemma *Sup-of-subsingleton-sets-eq*:
assumes P_s : subsingleton-set $(\bigcup P_s)$ **and** P : $P \in P_s$ **and** a : $a \in P$
shows $\text{Sup } (\bigcup P_s) = a$
proof (*intro order.antisym ccpo-Sup-least ccpo-Sup-upper subsingleton-sets-imp-chain[OF P_s]*)
fix $x :: 'a$ **assume** $x \in \bigcup P_s$
then obtain Q **where** $Q \in P_s$ $x \in Q$ **by** *auto*
with $P_s P a$ **show** $x \leq a$
by (*auto simp add: subsingleton-set-def Ball-def*)
qed (*use P a in blast*)

lemma *monotone-Sup-of-subsingleton-sets*:
assumes P_s : $\bigwedge F$. subsingleton-set $(\bigcup (P_s F))$
and $*$: $\bigwedge F G$. $\text{ord } F G \implies \text{sim-set } (\text{sim-set } (\leq)) (P_s F) (P_s G)$
shows *monotone* $\text{ord } (\leq) (\lambda F$. $\text{Sup } (\bigcup (P_s F)))$
proof (*intro monotoneI ccpo-Sup-least subsingleton-sets-imp-chain[OF P_s]*)
fix $F G x$ **assume** $\text{ord } F G$ $x \in \bigcup (P_s F)$
with $*$ [*of F G*] **obtain** y **where** $x \leq y$ $y \in \bigcup (P_s G)$
by (*fastforce simp: sim-set-def*)
then show $x \leq \text{Sup } (\bigcup (P_s G))$
apply (*intro order-trans[OF ‹x ≤ y›*)
apply (*intro ccpo-Sup-upper subsingleton-sets-imp-chain[OF P_s]*)
apply *auto*
done
qed

lemma *ccpo-refl*: $x \leq x$..

lemma *fixp-unfold-def*:
fixes $f :: 'a \Rightarrow 'a$
assumes def : $F \equiv \text{ccpo.fixp } \text{Sup } (\leq) f$
assumes f : *monotone* $(\leq) (\leq) f$
shows $F = f F$
by (*unfold def*) (*rule fixp-unfold[OF f]*)

lemma *fixp-induct-def*:
fixes $f :: 'a \Rightarrow 'a$
assumes def : $F \equiv \text{ccpo.fixp } \text{Sup } (\leq) f$
assumes mono : *monotone* $(\leq) (\leq) f$
assumes adm : *ccpo.admissible* $\text{Sup } (\leq) P$
assumes bot : $P (\text{Sup } \{\})$
assumes step : $\bigwedge x$. $P x \implies P (f x)$
shows $P F$
by (*unfold def*) (*rule fixp-induct[OF adm mono bot step]*)

lemma *induct-Sup-of-subsingleton-sets*:
assumes P_s : subsingleton-set $(\bigcup P_s)$
and adm : *ccpo.admissible* $\text{Sup } (\leq) P$
and bot : $P (\text{Sup } \{\})$

and step: $\bigwedge p x. p \in Ps \implies x \in p \implies P x$
shows $P (Sup (\bigcup Ps))$
proof cases
assume $*$: $(\bigcup Ps) = \{\}$ **from bot show** *?thesis unfolding* $*$.
next
assume $*$: $(\bigcup Ps) \neq \{\}$
show *?thesis*
by (*intro ccpo.admissibleD[OF adm subsingleton-sets-imp-chain[OF Ps] *]*)
(blast intro: step)
qed

lemma *induct-Sup-of-subsingleton-sets-def*:
assumes $F: F \equiv Sup (\bigcup Ps)$
assumes $Ps: \text{subsingleton-set } (\bigcup Ps)$
and $adm: \text{ccpo.admissible } Sup (\leq) P$
and $bot: P (Sup \{\})$
and step: $\bigwedge p x. p \in Ps \implies x \in p \implies P x$
shows $P F$
unfolding F **by** (*rule induct-Sup-of-subsingleton-sets; fact*)

end

lemma *flat-lub-empty*: $\text{flat-lub } x \{\} = x$
by (*simp add: flat-lub-def*)

lemma *monotone-bind-option[partial-function-mono]*:
 $\text{monotone ord option-ord } f \implies (\bigwedge a. \text{monotone ord option-ord } (\lambda x. g x a)) \implies$
 $\text{monotone ord option-ord } (\lambda x. \text{Option.bind } (f x) (g x))$
by (*fastforce simp: monotone-def flat-ord-def bind-eq-None-conv*)

lemma (**in** *ccpo*) *admissible-ord*: $\text{ccpo.admissible } Sup (\leq) (\lambda x. x \leq b)$
by (*clarsimp simp add: ccpo.admissible-def intro!: ccpo-Sup-least*)

ML-file $\langle \text{mutual-ccpo-recursion.ML} \rangle$

setup \langle
(Mutual-CCPO-Rec.add-ccpo option Mutual-CCPO-Rec.synth-option
 $\#> \text{Mutual-CCPO-Rec.add-ccpo lfp Mutual-CCPO-Rec.synth-lfp}$
 $\#> \text{Mutual-CCPO-Rec.add-ccpo gfp Mutual-CCPO-Rec.synth-gfp}$
 $|> \text{Context.theory-map } \rangle$

no-notation $top (\top)$
no-notation $bot (\perp)$
no-notation sup (**infixl** \sqcup 65)
no-notation inf (**infixl** \sqcap 70)

hide-const (**open**) *cont*

end

Part II
C-Parser

Chapter 10

Theory Variants for Target Architectures via L_4V -ARCH

```
theory Target-Architecture
imports Main
keywords
  if-architecture-by :: qed-global % proof and
  if-architecture-context :: thy-decl-block

begin

ML ‹
  structure Target-Architecture =
  struct

    datatype arch = ARM | ARM64 | ARM-HYP | RISC64 | X64

    val ARM-N      = ARM
    val ARM64-N   = ARM64
    val ARM-HYP-N = ARM-HYP
    val RISC64-N  = RISC64
    val X64-N     = X64
    val architectures = [(ARM-N, (here, ARM)), (ARM64-N, (here, ARM64)), (ARM-HYP-N,
    (here, ARM-HYP)),
      (RISC64-N, (here, RISC64)), (X64-N, (here, X64))]

    val rev-architectures = map (fn (name, (-, arch)) => (arch, name)) architectures

    val string-of = AList.lookup (op =) rev-architectures #> the

    fun check-architecture thy (name, pos) =
      case AList.lookup (op =) architectures name of
      SOME (def-pos, arch) =>
        let
          val markup = Position.entity-markup architecture-variant (name, def-pos)
```

```

    val - =
      Context-Position.reports (Proof-Context.init-global thy) [(pos, markup),
        (pos, Markup.string)]
      in arch end
    | NONE => error (undefined architecture variant ^ quote name ^ Position.here
      pos ^ \n ^
        known variants: ^ @ {make-string} (map #1 architectures))

    val active = the-default ARM (try (getenv-strict) L4V-ARCH |>
      Option.map (fn n => check-architecture @ {theory} (n, here)))

  end
>

```

ML ‹

```

structure Target-Architecture =
struct
  open Target-Architecture

  fun arch-parser msg p = (Parse.$$$ ( |-- Parse.list (Parse.name-position) --|
    Parse.$$$ ))
    :|-- (fn archs =>
      let
        val archs' = map (check-architecture @ {theory}) archs
        val is-active = member (op =) archs' active
        val - = if not is-active then writeln (active architecture ^ quote (string-of active)
          ^ not in ^ @ {make-string} (map string-of archs') ^ : ^ msg) else ()
      in
        p is-active
      end)

```

```

fun is-active arch = (arch = active);

```

```

fun add-path arch =
  if is-active arch then Sign.add-path (string-of arch) else Sign.mandatory-path
  (string-of arch)

```

```

val - =
  Outer-Syntax.command command-keyword ‹if-architecture-by› conditional ter-
  minal backward proof
  (arch-parser aborting proof (oops) (fn is-active =>
    (Method.parse -- Scan.option Method.parse >> (fn (m1, m2) =>
      (Method.report m1;
        Option.map Method.report m2;
        (if is-active then
          Isar-Cmd.terminal-proof (m1, m2)
        else

```

```

    Toplevel.forget-proof (* oops *))))));

val - =
  Outer-Syntax.command command-keyword <if-architecture-context> conditional
  context or experiment
  (arch-parser experiment only (fn is-active =>
    (Scan.repeat Parse-Spec.context-element --| Parse.begin
      >> (fn elems =>
        if is-active then
          Toplevel.begin-nested-target (Target-Context.context-begin-nested-cmd []
            elems)
          else
            Toplevel.begin-main-target true (Experiment.experiment-cmd elems #>
              snd))))));

val - = Theory.setup
  (ML-Antiquotation.conditional binding <if-ARM> (fn - => is-active ARM) #>
    ML-Antiquotation.conditional binding <if-ARM64> (fn - => is-active ARM64)
    #>
    ML-Antiquotation.conditional binding <if-ARM-HYP> (fn - => is-active ARM-HYP)
    #>
    ML-Antiquotation.conditional binding <if-RISCV64> (fn - => is-active RISCV64)
    #>
    ML-Antiquotation.conditional binding <if-X64> (fn - => is-active X64))

end
>

end

```

Chapter 11

Unified Memory Model (UMM)

11.1 More Word Lemmas

This is a holding area for Word utility lemmas that are too specific or unpolished for the AFP, but which are reusable enough to be collected together for the rest of L4V. New utility lemmas that only prove facts about words should be added here (in preference to being kept where they were first needed).

theory *Word-Lemmas-Internal*

imports

Word-Lib.Word-Lemmas

Word-Lib.More-Word-Operations

Word-Lib.Many-More

Word-Lib.Word-Syntax

Word-Lib.Syntax-Bundles

begin

unbundle *bit-operations-syntax*

unbundle *bit-projection-infix-syntax*

lemmas *shiftr-nat-def* = *push-bit-eq-mult*[of - a **for** *a::nat*, folded *shiftr-def*]

lemmas *shiftr-nat-def* = *drop-bit-eq-div*[of - a **for** *a::nat*, folded *shiftr-def*]

declare *bit-simps*[*simp*]

lemma *signed-ge-zero-scst-eq-ucast*:

$0 \leq x \implies \text{scast } x = \text{ucast } x$

by (*simp add: scast-eq-ucast word-sle-msb-le*)

lemma *disjCI2*:

$(\neg P \implies Q) \implies P \vee Q$

by *blast*

lemma *nat-diff-diff-le-lhs*:

$$a + c - b \leq d \implies a - (b - c) \leq (d :: \text{nat})$$

by *arith*

lemma *is-aligned-obvious-no-wrap'*:

$$\llbracket \text{is-aligned ptr sz; } x = 2 \wedge \text{sz} - 1 \rrbracket$$

$$\implies \text{ptr} \leq \text{ptr} + x$$

by (*fastforce simp: field-simps intro: is-aligned-no-overflow*)

lemmas *add-ge0-weak* = *add-increasing*[**where** 'a=int and b=0]

lemmas *aligned-sub-aligned* = *Aligned.aligned-sub-aligned'*

lemma *minus-minus-swap*:

$$\llbracket a \leq c; b \leq d; b \leq a; d \leq c; (d :: \text{nat}) - b = c - a \rrbracket$$

$$\implies a - b = c - d$$

by *arith*

lemma *minus-minus-swap'*:

$$\llbracket c \leq a; d \leq b; b \leq a; d \leq c; (b :: \text{nat}) - d = a - c \rrbracket$$

$$\implies a - b = c - d$$

by *arith*

lemmas *word-le-mask-eq* = *le-mask-imp-and-mask*

lemma *int-and-leR*:

$$0 \leq b \implies a \text{ AND } b \leq (b :: \text{int})$$

by (*rule AND-upper2*)

lemma *int-and-leL*:

$$0 \leq a \implies a \text{ AND } b \leq (a :: \text{int})$$

by (*rule AND-upper1*)

lemma *if-then-1-else-0*:

$$(\text{if } P \text{ then } 1 \text{ else } 0) = (0 :: 'a :: \text{zero-neq-one}) \iff \neg P$$

by (*simp split: if-split*)

lemma *if-then-0-else-1*:

$$(\text{if } P \text{ then } 0 \text{ else } 1) = (0 :: 'a :: \text{zero-neq-one}) \iff P$$

by *simp*

lemmas *if-then-simps* = *if-then-0-else-1 if-then-1-else-0*

lemma *createNewCaps-guard*:

fixes $x :: 'a :: \text{len word}$

shows $\llbracket \text{unat } x = c; b < 2 \wedge \text{LENGTH}('a) \rrbracket$

$\implies (n < \text{of-nat } b \wedge n < x) = (n < \text{of-nat } (\min (\min b c) c))$
by (*metis* (*no-types*) *min-less-iff-conj* *nat-neq-iff* *unat-less-helper* *unat-ucast-less-no-overflow* *unsigned-less* *word-unat.Rep-inverse*)

lemma *bits-2-subtract-ineq*:

$i < (n :: ('a :: \text{len}) \text{ word})$
 $\implies 2^{\text{bits}} + 2^{\text{bits}} * \text{unat } (n - (1 + i)) = \text{unat } (n - i) * 2^{\text{bits}}$
apply (*simp* *add*: *unat-sub* *word-minus-one-le-leq*)
apply (*subst* *unatSuc*)
apply *clarsimp*
apply *unat-arith*
apply (*simp* *only*: *mult-Suc-right[symmetric]*)
apply (*rule* *trans[OF mult.commute]*, *rule* *arg-cong2[where f=(*)]*, *simp-all*)
apply (*simp* *add*: *word-less-nat-alt*)
done

lemmas *double-neg-mask = neg-mask-combine*

lemmas *int-unat = uint-nat[symmetric]*

lemmas *word-sub-mono3 = word-plus-mcs-4'*

lemma *shift-distinct-helper*:

$\llbracket (x :: 'a :: \text{len} \text{ word}) < \text{bnd}; y < \text{bnd}; x \neq y; x \ll n = y \ll n; n < \text{LENGTH}('a);$
 $\text{bnd} - 1 \leq 2^{\text{LENGTH}('a) - n} - 1 \rrbracket$
 $\implies P$
apply (*cases* $n = 0$)
apply *simp*
apply (*drule* *word-plus-mono-right[where x=1]*)
apply *simp-all*
apply (*subst* *word-le-sub1*)
apply (*rule* *power-not-zero*)
apply *simp*
apply *simp*
apply (*drule*(1) *order-less-le-trans*)+
apply (*clarsimp* *simp*: *bang-eq*)
subgoal for *na*
apply (*drule-tac* $x=na + n$ **in** *spec*)
apply (*simp* *add*: *nth-shiffl*)
apply (*cases* $na + n < \text{LENGTH}('a)$, *simp-all*)
apply *safe*
apply (*drule*(1) *nth-bounded*)
apply *simp*
apply *simp*
apply (*drule*(1) *nth-bounded*)
apply *simp*
apply *simp*
done
done

lemma *of-nat-shift-distinct-helper*:

$$\llbracket x < \text{bnd}; y < \text{bnd}; x \neq y; (\text{of-nat } x :: 'a :: \text{len word}) \lll n = \text{of-nat } y \lll n;$$

$$n < \text{LENGTH}('a); \text{bnd} \leq 2^{\wedge}(\text{LENGTH}('a) - n) \rrbracket$$

$$\implies P$$
apply (*cases* $n = 0$)
apply (*simp add: word-unat.Abs-inject unats-def*)
apply (*subgoal-tac* $\text{bnd} < 2^{\wedge} \text{LENGTH}('a)$)
apply (*erule*(1) *shift-distinct-helper*[*rotated, rotated, rotated*])
defer
apply (*erule*(1) *of-nat-mono-maybe*[*rotated*])
apply (*erule*(1) *of-nat-mono-maybe*[*rotated*])
apply (*simp add: word-unat.Abs-inject unats-def*)
apply (*erule order-le-less-trans*)
apply (*rule power-strict-increasing*)
apply *simp*
apply *simp*
apply (*fastforce simp: unat-of-nat-minus-1 word-less-nat-alt*)
done

lemmas *pre-helper2* = *add-mult-in-mask-range*[*folded add-mask-fold*]

lemma *ptr-add-distinct-helper*:

$$\llbracket \text{ptr-add } (p :: 'a :: \text{len word}) (x * 2^{\wedge} n) = \text{ptr-add } p (y * 2^{\wedge} n); x \neq y;$$

$$x < \text{bnd}; y < \text{bnd}; n < \text{LENGTH}('a);$$

$$\text{bnd} \leq 2^{\wedge}(\text{LENGTH}('a) - n) \rrbracket$$

$$\implies P$$
apply (*clarsimp simp: ptr-add-def word-unat-power*[*symmetric*]
shiftl-t2n[*symmetric, simplified mult.commute*])
using *of-nat-shift-distinct-helper*
by *blast*

lemma *unat-sub-le-strg*:

$$\text{unat } v \leq v2 \wedge x \leq v \wedge y \leq v \wedge y < (x :: ('a :: \text{len}) \text{word})$$

$$\longrightarrow \text{unat } (x + (-1 - y)) \leq v2$$
apply *clarsimp*
apply (*erule order-trans*[*rotated*])
apply (*fold word-le-nat-alt*)
apply (*rule order-trans*[*rotated*], *assumption*)
apply (*rule order-trans*[*rotated*], *rule word-sub-le*[**where** $y=y+1$])
apply (*erule Word.inc-le*)
apply (*simp add: field-simps*)
done

lemma *multi-lessD*:

$$\llbracket (a :: \text{nat}) * b < c; 0 < a; 0 < b \rrbracket$$

$$\implies a < c \wedge b < c$$
by (*cases a, simp-all, cases b, simp-all*)

lemmas *leq-high-bits-shiftr-low-bits-leq-bits* =
leq-high-bits-shiftr-low-bits-leq-bits-mask[*unfolded mask-2pm1*[*of high-bits*]]

lemmas *unat-le-helper* = *word-unat-less-le*

lemmas *word-of-nat-plus* = *of-nat-add*[**where** *'a='a :: len word*]
lemmas *word-of-nat-minus* = *of-nat-diff*[**where** *'a='a :: len word*]

lemma *word-up-bound*:
(*ptr :: 'a :: len word*) $\leq 2^{\text{LENGTH}('a) - 1}$
by *auto*

lemma *base-length-minus-one-inequality*:
assumes *foo*: $wbase \leq 2^{\text{sz} - 1}$
 $1 < wlength :: ('a :: len) \text{ word}$
 $wlength \leq 2^{\text{sz} - wbase}$
 $sz < \text{LENGTH}('a)$
shows $wbase \leq wbase + wlength - 1$
proof –

note *sz-less* = *power-strict-increasing*[*OF foo(4)*, **where** *a=2*]

from *foo* **have** *plus*: $unat\ wbase + unat\ wlength < 2^{\text{LENGTH}('a)}$
apply –
apply (*rule order-le-less-trans*[*rotated*], *rule sz-less*, *simp*)
apply (*simp add: unat-arith-simps split: if-split-asm*)
done

from *foo* **show** *?thesis*
by (*simp add: unat-arith-simps plus*)
qed

lemmas *from-bool-to-bool-and-1* = *from-to-bool-last-bit*[**where** *x=r for r*]

lemmas *max-word-neq-0* = *max-word-not-0*

lemmas *word-le-p2m1* = *word-up-bound*[**where** *ptr=w for w*]

lemma *inj-ucast*:
 $\llbracket uc = ucast; is-up\ uc \rrbracket$
 $\implies inj\ uc$
using *down-ucast-inj is-up-down* **by** *blast*

lemma *ucast-eq-0*[*OF refl*]:
 $\llbracket c = ucast; is-up\ c \rrbracket$
 $\implies (c\ x = 0) = (x = 0)$
by (*metis uint-0-iff uint-up-ucast*)

lemmas *is-up-compose'* = *is-up-compose*

lemma *uint-is-up-compose*:
 fixes *uc* :: 'a :: len word \Rightarrow 'b :: len word
 and *uc'* :: 'b word \Rightarrow 'c :: len sword
 assumes *uc* = *ucast*
 and *uc'* = *ucast*
 and *uuc* = *uc'* \circ *uc*
 shows \llbracket *is-up uc*; *is-up uc'* \rrbracket
 \Rightarrow *uint* (*uuc b*) = *uint b*
 apply (*simp add: assms*)
 apply (*frule is-up-compose*)
 apply *simp-all*
 apply (*simp only: Word.uint-up-ucast*)
 done

lemma *uint-is-up-compose-pred*:
 fixes *uc* :: 'a :: len word \Rightarrow 'b :: len word
 and *uc'* :: 'b word \Rightarrow 'c :: len sword
 assumes *uc* = *ucast* **and** *uc'* = *ucast* **and** *uuc* = *uc'* \circ *uc*
 shows \llbracket *is-up uc*; *is-up uc'* \rrbracket
 \Rightarrow P (*uint* (*uuc b*)) \longleftrightarrow P (*uint b*)
 apply (*simp add: assms*)
 apply (*frule is-up-compose*)
 apply *simp-all*
 apply (*simp only: Word.uint-up-ucast*)
 done

lemma *is-down-up-sword*:
 fixes *uc* :: 'a :: len word \Rightarrow 'b :: len sword
 shows \llbracket *uc* = *ucast*; $\text{LENGTH}('a) < \text{LENGTH}('b)$ \rrbracket
 \Rightarrow *is-up uc* = (\neg *is-down uc*)
 by (*simp add: target-size source-size is-up-def is-down-def*)

lemma *is-not-down-compose*:
 fixes *uc* :: 'a :: len word \Rightarrow 'b :: len word
 and *uc'* :: 'b word \Rightarrow 'c :: len sword
 shows \llbracket *uc* = *ucast*; *uc'* = *ucast*; $\text{LENGTH}('a) < \text{LENGTH}('c)$ \rrbracket
 \Rightarrow \neg *is-down* (*uc'* \circ *uc*)
 unfolding *is-down-def*
 by (*simp add: Word.target-size Word.source-size*)

lemma *sint-ucast-uint*:
 fixes *uc* :: 'a :: len word \Rightarrow 'b :: len word
 and *uc'* :: 'b word \Rightarrow 'c :: len sword
 assumes *uc* = *ucast* **and** *uc'* = *ucast* **and** *uuc* = *uc'* \circ *uc*
 and $\text{LENGTH}('a) < \text{LENGTH}('c \text{ signed})$
 shows \llbracket *is-up uc*; *is-up uc'* \rrbracket
 \Rightarrow *sint* (*uuc b*) = *uint b*

```

apply (simp add: assms)
apply (frule is-up-compose')
apply simp-all
apply (simp add: ucast-ucast-b)
apply (rule sint-ucast-eq-uint)
apply (insert assms)
apply (simp add: is-down-def target-size source-size)
done

```

```

lemma sint-ucast-uint-pred:
  fixes  $uc :: 'a :: \text{len word} \Rightarrow 'b :: \text{len word}$ 
    and  $uc' :: 'b \text{ word} \Rightarrow 'c :: \text{len sword}$ 
    and  $uuc :: 'a \text{ word} \Rightarrow 'c \text{ sword}$ 
  assumes  $uc = \text{ucast}$  and  $uc' = \text{ucast}$  and  $uuc = uc' \circ uc$ 
    and  $\text{LENGTH}('a) < \text{LENGTH}('c)$ 
  shows  $\llbracket \text{is-up } uc; \text{is-up } uc' \rrbracket$ 
     $\Longrightarrow P(\text{uint } b) \longleftrightarrow P(\text{sint } (uuc \ b))$ 
  apply (simp add: assms)
  apply (insert sint-ucast-uint[where  $uc=uc$  and  $uc'=uc'$  and  $uuc=uuc$  and  $b =$ 
b])
  apply (simp add: assms)
done

```

```

lemma sint-uucast-uint-uucast-pred:
  fixes  $uc :: 'a :: \text{len word} \Rightarrow 'b :: \text{len word}$ 
    and  $uc' :: 'b \text{ word} \Rightarrow 'c :: \text{len sword}$ 
  assumes  $uc = \text{ucast}$  and  $uc' = \text{ucast}$  and  $uuc = uc' \circ uc$ 
    and  $\text{LENGTH}('a) < \text{LENGTH}('c)$ 
  shows  $\llbracket \text{is-up } uc; \text{is-up } uc' \rrbracket$ 
     $\Longrightarrow P(\text{uint}(uuc \ b)) \longleftrightarrow P(\text{sint } (uuc \ b))$ 
  apply (simp add: assms)
  apply (insert sint-ucast-uint[where  $uc=uc$  and  $uc'=uc'$  and  $uuc=uuc$  and  $b =$ 
b])
  apply (insert uint-is-up-compose-pred[where  $uc=uc$  and  $uc'=uc'$  and  $uuc=uuc$ 
and  $b=b$ ])
  apply (simp add: assms uint-is-up-compose-pred)
done

```

```

lemma unat-minus':
  fixes  $x :: 'a :: \text{len word}$ 
  shows  $x \neq 0 \Longrightarrow \text{unat } (-x) = 2^{\wedge} \text{LENGTH}('a) - \text{unat } x$ 
  by (simp add: unat-minus wsst-TYs(3))

```

```

lemma word-nth-neq:
   $n < \text{LENGTH}('a) \Longrightarrow (\sim\sim x :: 'a :: \text{len word}) !! n = (\neg x !! n)$ 
  by (simp add: word-size word-ops-nth-size)

```

```

lemma word-wrap-of-natD:
  fixes  $x :: 'a :: \text{len word}$ 

```

```

assumes wraps:  $\neg x \leq x + \text{of-nat } n$ 
shows  $\exists k. x + \text{of-nat } k = 0 \wedge k \leq n$ 
proof -
  show ?thesis
  proof (rule exI [where x = unat (- x)], intro conjI)
    show  $x + \text{of-nat } (\text{unat } (-x)) = 0$ 
      by simp
    next
      show  $\text{unat } (-x) \leq n$ 
      proof (subst unat-minus')
        from wraps show  $x \neq 0$ 
        by (rule contrapos-pn, simp add: not-le)
      next
        show  $2 \wedge \text{LENGTH}'a - \text{unat } x \leq n$  using wraps
        apply (simp add: no-olen-add-nat le-diff-conv not-less)
        apply (erule order-trans)
        apply (simp add: unat-of-nat)
        done
      qed
    qed
  qed

```

```

lemma two-bits-cases:
  [[ LENGTH('a) > 2; (x :: 'a :: len word) && 3 = 0  $\implies$  P; x && 3 = 1  $\implies$  P;
    x && 3 = 2  $\implies$  P; x && 3 = 3  $\implies$  P ]
   $\implies$  P
  apply (frule and-mask-cases[where n=2 and x=x, simplified mask-eq])
  using upt-conv-Cons by auto[1]

```

```

lemma zero-OR-eq:
   $y = 0 \implies (x \parallel y) = x$ 
  by simp

```

```

declare is-aligned-neg-mask-eq[simp]
declare is-aligned-neg-mask-weaken[simp]

```

```

lemmas mask-in-range = neg-mask-in-mask-range[folded add-mask-fold]
lemmas aligned-range-offset-mem = aligned-offset-in-range[folded add-mask-fold]
lemmas range-to-bl' = mask-range-to-bl'[folded add-mask-fold]
lemmas range-to-bl = mask-range-to-bl[folded add-mask-fold]
lemmas aligned-ranges-subset-or-disjoint = aligned-mask-range-cases[folded add-mask-fold]
lemmas aligned-range-offset-subset = aligned-mask-range-offset-subset[folded add-mask-fold]
lemmas aligned-diff = aligned-mask-diff[unfolded mask-2pm1]
lemmas aligned-ranges-subset-or-disjoint-coroll = aligned-mask-ranges-disjoint[folded
  add-mask-fold]
lemmas distinct-aligned-addresses-accumulate = aligned-mask-ranges-disjoint2[folded
  add-mask-fold]

```

```

lemmas bang-big = test-bit-over

```

lemma *unat-and-mask-le*:

$n < \text{LENGTH}('a) \implies \text{unat} (x \&\& \text{mask } n) \leq 2^{\wedge n}$ **for** $x :: 'a :: \text{len word}$
by (*simp add: and-mask-less' order-less-imp-le unat-less-power*)

lemma *sign-extend-less-mask-idem*:

$\llbracket w \leq \text{mask } n; n < \text{size } w \rrbracket \implies \text{sign-extend } n w = w$
by (*simp add: sign-extend-def le-mask-imp-and-mask le-mask-high-bits*)

lemma *word-and-le*:

$a \leq c \implies (a :: 'a :: \text{len word}) \&\& b \leq c$
by (*subst and.commute*) (*erule word-and-le'*)

lemma *le-smaller-mask*:

$\llbracket x \leq \text{mask } n; n \leq m \rrbracket \implies x \leq \text{mask } m$ **for** $x :: 'a :: \text{len word}$
by (*erule (1) order.trans[OF - mask-mono]*)

lemma *upcast-less-unat-less*:

assumes *less*: $\text{UCAST}('a \rightarrow 'b) x < \text{UCAST}('a \rightarrow 'b) (of\text{-nat } y)$
assumes *len*: $\text{LENGTH}('a :: \text{len}) \leq \text{LENGTH}('b :: \text{len})$
assumes *bound*: $y < 2^{\wedge \text{LENGTH}('a)}$
shows $\text{unat } x < y$
by (*rule unat-mono[OF less, simplified unat-ucast-up-simp[OF len] unat-of-nat-eq[OF bound]]*)

lemma *word-ctz-max*:

$\text{word-ctz } w \leq \text{size } w$
unfolding *word-ctz-def*
by (*rule order-trans[OF List.length-takeWhile-le], clarsimp simp: word-size*)

lemma *scast-of-nat-small*:

$x < 2^{\wedge (\text{LENGTH}('a) - 1)} \implies \text{scast} (of\text{-nat } x :: 'a :: \text{len word}) = (of\text{-nat } x :: 'b :: \text{len word})$
apply *transfer*
apply *simp*
by (*metis One-nat-def id-apply int-eq-sint of-int-eq-id signed-of-nat*)

lemmas *casts-of-nat-small = ucast-of-nat-small scast-of-nat-small*

— The conditions under which ‘takeWhile P xs = take n xs’ and ‘dropWhile P xs = drop n xs’

definition *list-while-len where*

$\text{list-while-len } P n xs \equiv (\forall i. i < n \longrightarrow i < \text{length } xs \longrightarrow P (xs ! i))$
 $\wedge (n < \text{length } xs \longrightarrow \neg P (xs ! n))$

lemma *list-while-len-iff-takeWhile-eq-take*:

$\text{list-while-len } P n xs \longleftrightarrow \text{takeWhile } P xs = \text{take } n xs$
unfolding *list-while-len-def*


```

apply (rule iffI[OF takeWhile-eq-take-P-nth], simp+)
apply (intro conjI allI impI)
  apply (rule takeWhile-take-has-property-nth, clarsimp)
apply (drule-tac f=length in arg-cong, simp)
apply (induct xs arbitrary: n; clarsimp split: if-splits)
done

```

```

lemma list-while-len-exists:
   $\exists n. \text{list-while-len } P \ n \ xs$ 
apply (induction xs; simp add: list-while-len-def)
apply (erule exE)
subgoal for  $x \ xs \ n$ 
  apply (cases P x)
    apply (rule exI[where  $x = \text{Suc } n$ ], clarsimp)
  subgoal for  $i$  by (cases i; simp)
  subgoal by (rule exI[where  $x = 0$ ], simp)
done
done

```

```

lemma takeWhile-truncate:
   $\text{length } (\text{takeWhile } P \ xs) \leq m$ 
   $\implies \text{takeWhile } P \ (\text{take } m \ xs) = \text{takeWhile } P \ xs$ 
apply (cut-tac list-while-len-exists[where  $P = P$  and  $xs = xs$ ], clarsimp)
subgoal for  $n$ 
  apply (cases  $n \leq m$ )
    apply (subgoal-tac list-while-len P n (take m xs))
      apply (simp add: list-while-len-iff-takeWhile-eq-take)
      apply (clarsimp simp: list-while-len-def)
      apply (simp add: list-while-len-iff-takeWhile-eq-take)
    done
done

```

```

lemma word-clz-shiftr-1:
  fixes  $z :: 'a :: \text{len word}$ 
  assumes  $\text{wordsize}: 1 < \text{LENGTH } ('a)$ 
  shows  $z \neq 0 \implies \text{word-clz } (z \gg 1) = \text{word-clz } z + 1$ 
  supply word-size [simp]
  apply (clarsimp simp: word-clz-def)
  using wordsize apply (subst bl-shiftr, simp)
  apply (subst takeWhile-append2; simp add: wordsize)
  apply (subst takeWhile-truncate)
  using word-clz-nonzero-max[where  $w = z$ ]
  apply (clarsimp simp: word-clz-def)
  apply simp+
done

```

```

lemma shiftr-Suc:
  fixes  $x :: 'a :: \text{len word}$ 
  shows  $x \gg (\text{Suc } n) = x \gg n \gg 1$ 

```

by (clarsimp simp: shiftr-shiftr)

lemma word-clz-shiftr:

fixes z :: 'a::len word

shows $n < \text{LENGTH}('a) \implies \text{mask } n < z \implies \text{word-clz } (z \gg n) = \text{word-clz } z + n$

apply (simp add: le-mask-iff Not-eq-iff[THEN iffD2, OF le-mask-iff, simplified not-le])

word-neq-0-conv)

apply (induction n; simp)

subgoal for n

apply (subgoal-tac $0 < z \gg n$)

apply (subst shiftr-Suc)

apply (subst word-clz-shiftr-1; simp)

apply (clarsimp simp: word-less-nat-alt shiftr-div-2n' div-mult2-eq)

apply (cases unat z div 2 ^ n = 0; simp)

apply (clarsimp simp: div-eq-0-iff)

done

done

lemma mask-to-bl-exists-True:

$x \ \&\& \ \text{mask } n \neq 0 \implies \exists m. (\text{rev } (\text{to-bl } x)) ! m \wedge m < n$

apply (subgoal-tac $\neg(\forall m. m < n \longrightarrow \neg(\text{rev } (\text{to-bl } x)) ! m)$, fastforce)

apply (intro notI)

apply (subgoal-tac $x \ \&\& \ \text{mask } n = 0$, clarsimp)

apply (clarsimp simp: eq-zero-set-bl in-set-conv-nth)

apply (subst (asm) to-bl-nth, clarsimp simp: word-size)

apply (clarsimp simp: word-size)

by (meson nth-mask test-bit-bl word-and-nth)

lemma word-ctz-shiftr-1:

fixes z :: 'a::len word

assumes wordsize: $1 < \text{LENGTH}('a)$

shows $z \neq 0 \implies 1 \leq \text{word-ctz } z \implies \text{word-ctz } (z \gg 1) = \text{word-ctz } z - 1$

supply word-size [simp]

apply (clarsimp simp: word-ctz-def)

using wordsize apply (subst bl-shiftr, simp)

apply (simp add: rev-take)

apply (subgoal-tac

length (takeWhile Not (rev (to-bl z))) - Suc 0

= length (takeWhile Not (take 1 (rev (to-bl z)) @ drop 1 (rev (to-bl z))))

- Suc 0)

apply (subst (asm) takeWhile-append2)

apply clarsimp

apply (cases rev (to-bl z); simp)

apply clarsimp

apply (subgoal-tac $\exists m. (\text{rev } (\text{to-bl } z)) ! m \wedge m < \text{LENGTH}('a)$, clarsimp)

subgoal for m

apply (cases m)

```

apply (cases rev (to-bl z); simp)
subgoal for nat
  apply (subst takeWhile-append1, subst in-set-conv-nth)
    apply (rule exI[where x=nat])
    apply (intro conjI)
    apply (clarsimp simp: wordsize)
  using wordsize apply linarith
  apply (rule refl, clarsimp)
  apply simp
  done
done
apply (rule mask-to-bl-exists-True, simp)
apply simp
done

```

```

lemma word-ctz-bound-below-helper:
  fixes x :: 'a::len word
  assumes sz: n ≤ LENGTH('a)
  shows x && mask n = 0
     $\implies$  to-bl x = (take (LENGTH('a) - n) (to-bl x) @ replicate n False)
  apply (subgoal-tac replicate n False = drop (LENGTH('a) - n) (to-bl x))
  apply (subgoal-tac True ∉ set (drop (LENGTH('a) - n) (to-bl x)))
    apply (drule list-of-false, clarsimp simp: sz)
  apply (drule-tac sym[where t=drop n x for n x], clarsimp)
  apply (rule sym)
  apply (rule is-aligned-drop; clarsimp simp: is-aligned-mask sz)
  done

```

```

lemma word-ctz-bound-below:
  fixes x :: 'a::len word
  assumes sz[simp]: n ≤ LENGTH('a)
  shows x && mask n = 0  $\implies$  n ≤ word-ctz x
  apply (clarsimp simp: word-ctz-def)
  apply (subst word-ctz-bound-below-helper[OF sz]; simp)
  apply (subst takeWhile-append2; clarsimp)
  done

```

```

lemma word-ctz-bound-above:
  fixes x :: 'a::len word
  shows x && mask n ≠ 0  $\implies$  word-ctz x < n
  apply (cases n ≤ LENGTH('a))
  apply (frule mask-to-bl-exists-True, clarsimp)
  subgoal for m
    apply (clarsimp simp: word-ctz-def)
    apply (subgoal-tac m < length ((rev (to-bl x))))
    apply (subst id-take-nth-drop[where xs=rev (to-bl x)], assumption)
    apply (subst takeWhile-tail, simp)
    apply (rule order.strict-trans1)
    apply (rule List.length-takeWhile-le)

```

```

    apply simp
    apply (erule order.strict-trans2, clarsimp)
  done
  apply (simp add: not-le)
  apply (erule le-less-trans[OF word-ctz-max, simplified word-size])
  done

```

```

lemma word-ctz-shiftr:
  fixes z::'a::len word
  assumes nz: z ≠ 0
  shows n < LENGTH('a) ⇒ n ≤ word-ctz z ⇒ word-ctz (z >> n) = word-ctz
z - n
  apply (induction n; simp)
  subgoal for n
    apply (subst shiftr-Suc)
    apply (subst word-ctz-shiftr-1, simp)
    apply clarsimp
    apply (subgoal-tac word-ctz z < n, clarsimp)
    apply (rule word-ctz-bound-above, clarsimp simp: word-size)
    apply (subst (asm) and-mask-eq-iff-shiftr-0[symmetric], clarsimp simp: nz)
    apply (rule word-ctz-bound-below, clarsimp simp: word-size)
    apply (rule mask-zero)
    apply (rule is-aligned-shiftr, simp add: is-aligned-mask)
    apply (cases z && mask (Suc n) = 0, simp)
    apply (frule word-ctz-bound-above[rotated]; clarsimp simp: word-size)
  apply simp
  done
done

```

— Useful for solving goals of the form `'(w::32 word) <= 0xFFFFFFFF'` by simplification

```

lemma word-less-max-simp:
  fixes w :: 'a::len word
  assumes max-w = -1
  shows w ≤ max-w
  unfolding assms by simp

```

```

lemma word-and-mask-word-ctz-zero:
  assumes l = word-ctz w
  shows w && mask l = 0
  unfolding word-ctz-def assms
  apply (word-eqI)
  apply (drule takeWhile-take-has-property-nth)
  apply (simp add: test-bit-bl)
  done

```

```

lemma word-ctz-len-word-and-mask-zero:
  fixes w :: 'a::len word
  shows word-ctz w = LENGTH('a) ⇒ w = 0

```

by (drule sym, drule word-and-mask-word-ctz-zero, simp)

lemma word-le-1:

fixes $w :: 'a::len$ word

shows $w \leq 1 \iff w = 0 \vee w = 1$

using dual-order.antisym lt1-neq0 word-zero-le by blast

lemma less-ucast-ucast-less':

$x < \text{UCAST}('b \rightarrow 'a) y \implies \text{UCAST}('a \rightarrow 'b) x < y$

for $x :: 'a::len$ word and $y :: 'b::len$ word

by (clarsimp simp: order.strict-iff-order dual-order.antisym le-ucast-ucast-le)

lemma ucast-up-less-bounded-implies-less-ucast-down':

assumes $len: \text{LENGTH}('a::len) < \text{LENGTH}('b::len)$

assumes bound: $y < 2^{\text{LENGTH}('a)}$

assumes less: $\text{UCAST}('a \rightarrow 'b) x < y$

shows $x < \text{UCAST}('b \rightarrow 'a) y$

apply (rule le-less-trans[OF - ucast-mono[OF less bound]])

using len by (simp add: is-down ucast-down-ucast-id)

lemma ucast-up-less-bounded-iff-less-ucast-down':

assumes $len: \text{LENGTH}('a::len) < \text{LENGTH}('b::len)$

assumes bound: $y < 2^{\text{LENGTH}('a)}$

shows $\text{UCAST}('a \rightarrow 'b) x < y \iff x < \text{UCAST}('b \rightarrow 'a) y$

apply (rule iffI)

prefer 2

apply (simp add: less-ucast-ucast-less')

using assms by (rule ucast-up-less-bounded-implies-less-ucast-down')

lemma word-of-int-word-of-nat-eqD:

$\llbracket \text{word-of-int } x = (\text{word-of-nat } y :: 'a :: len \text{ word}); 0 \leq x; x < 2^{\text{LENGTH}('a)}; y < 2^{\text{LENGTH}('a)} \rrbracket$

$\implies \text{nat } x = y$

by (metis nat-eq-numeral-power-cancel-iff of-nat-inj word-of-int-nat zless2p zless-nat-conj)

lemma ucast-down-0:

$\llbracket \text{UCAST}('a::len \rightarrow 'b::len) x = 0; \text{unat } x < 2^{\text{LENGTH}('b)} \rrbracket \implies x = 0$

by (metis Word.of-nat-unat unat-0 unat-eq-of-nat word-unat-eq-iff)

lemma uint-minus-1-eq:

$\langle \text{uint } (- 1 :: 'a \text{ word}) = 2^{\text{LENGTH}('a::len)} - 1 \rangle$

by transfer (simp add: mask-eq-exp-minus-1)

lemma FF-eq-minus-1:

$\langle 0xFF = (- 1 :: 8 \text{ word}) \rangle$

by simp

lemma shiftl-t2n':

$w \ll n = w * (2^{\wedge} n)$ for $w :: 'a::len$ word

```

    by (simp add: shiftl-t2n)

end

theory Word-Lemmas-32-Internal
imports Word-Lib.Word-Lib-Sumo Word-Lib.Machine-Word-32 Word-Lemmas-Internal
begin

lemmas sint-eq-uint-32 = sint-eq-uint-2pl[where 'a=32, simplified]

lemmas sle-positive-32 = sle-le-2pl[where 'a=32, simplified]

lemmas sless-positive-32 = sless-less-2pl[where 'a=32, simplified]

lemma zero-le-sint-32:
  [[ 0 ≤ (a :: word32); a < 0x80000000 ]]
  ⇒ 0 ≤ sint a
  by (clarsimp simp: sint-eq-uint-32 unat-less-helper)

lemmas unat-add-simple = iffD1[OF unat-add-lem[where 'a = 32, folded word-bits-def]]

lemma upto-enum-inc-1:
  a < 2 ^ word-bits - 1
  ⇒ [(0:: 'a :: len word) .e. 1 + a] = [0.e.a] @ [(1+a)]
  using upper-trivial upto-enum-inc-1-len by force

lemmas upt-enum-offset-trivial =
  upt-enum-offset-trivial[where 'a=32, folded word-bits-def]

lemmas unat32-eq-of-nat = unat-eq-of-nat[where 'a=32, folded word-bits-def]

declare mask-32-max-word[simp]

lemma le-32-mask-eq:
  (bits :: word32) ≤ 32 ⇒ bits && mask 6 = bits
  by (fastforce elim: le-less-trans intro: less-mask-eq)

lemmas scast-1-32[simp] = scast-1[where 'a=32]

lemmas mask-32-id[simp] = mask-len-id[where 'a=32, folded word-bits-def]

lemmas t2p-shiftr-32 = t2p-shiftr[where 'a=32, folded word-bits-def]

lemma mask-eq1-nochoice:
  (x :: word32) && 1 = x
  ⇒ x = 0 ∨ x = 1
  using mask-eq1-nochoice len32 by force

```

```

lemmas const-le-unat-word-32 = const-le-unat[where 'a=32, folded word-bits-def]

lemmas createNewCaps-guard-helper =
  createNewCaps-guard[where 'a=32, folded word-bits-def]

lemma word-log2-max-word32[simp]:
  word-log2 (w :: 32 word) < 32
  using word-log2-max[where w=w]
  by (simp add: word-size)

lemma mapping-two-power-16-64-inequality:
  assumes sz: sz ≤ 4 and len: unat (len :: word32) = 2 ^ sz
  shows unat (len * 8 - 1) ≤ 127
  using pow-sub-less[where 'a=32 and b=3, simplified]
proof -
  have len2: len = 2 ^ sz
    apply (rule word-unat.Rep-eqD, simp only: len)
    using sz
    apply simp
  done

  show ?thesis using two-power-increasing-less-1[where 'a=32 and n=sz + 3
and m=7]
  apply (simp add: word-le-nat-alt sz power-add len2 field-simps bintrunc-Suc-numeral)
  using le-trans take-bit-nat-less-eq-self by blast
qed

lemmas pre-helper2-32 = pre-helper2[where 'a=32, folded word-bits-def]

lemmas of-nat-shift-distinct-helper-machine =
  of-nat-shift-distinct-helper[where 'a=32, folded word-bits-def]

lemmas ptr-add-distinct-helper-32 =
  ptr-add-distinct-helper[where 'a=32, folded word-bits-def]

lemmas mask-out-eq-0-32 = mask-out-eq-0[where 'a=32, folded word-bits-def]

lemmas neg-mask-mask-unat-32 = neg-mask-mask-unat[where 'a=32, folded word-bits-def]

lemmas unat-less-iff-32 = unat-less-iff[where 'a=32, folded word-bits-def]

lemmas is-aligned-no-overflow3-32 = is-aligned-no-overflow3[where 'a=32, folded
word-bits-def]

lemmas unat-ucast-16-32 = unat-signed-ucast-less-ucast[where 'a=16 and 'b=32,
simplified]

```

```

lemma scast-mask-8:
  scast (mask 8 :: sword32) = (mask 8 :: word32)
  by (clarsimp simp: mask-eq)

lemmas ucast-le-8-32-equiv = ucast-le-up-down-iff[where 'a=8 and 'b=32, simplified]

lemma signed-unat-minus-one-32:
  unat (-1 :: 32 signed word) = 4294967295
  by (simp del: word-pow-0 diff-0 add: unat-sub-if' minus-one-word)

lemmas two-bits-cases-32 = two-bits-cases[where 'a=32, simplified]

lemmas word-ctz-not-minus-1-32 = word-ctz-not-minus-1[where 'a=32, simplified]

lemmas sint-ctz-32 = sint-ctz[where 'a=32, simplified]

lemmas scast-specific-plus32 =
  scast-of-nat-signed-to-unsigned-add[where 'a=32 and x=word-ctz x and y=0x20
for x,
```

$$\textit{simplified}$$

```

lemmas scast-specific-plus32-signed =
  scast-of-nat-unsigned-to-signed-add[where 'a=32 and x=word-ctz x and y=0x20
for x,
```

$$\textit{simplified}$$

```

lemma neg-0-unat: x ≠ 0 ⇒ 0 < unat x for x::machine-word
  by (simp add: unat-gt-0)

end

theory Word-Lemmas-64-Internal
imports Word-Lib.Word-Lib-Sumo Word-Lib.Word-64 Word-Lib.Machine-Word-64
begin

lemmas word-and-max-simps = word-and-max-simps word64-and-max-simp

lemmas unat-add-simple = iffD1[OF unat-add-lem[where 'a = 64, folded word-bits-def]]

lemma unat-length-4-helper:
  assumes unat (l::machine-word) = length args ∧ l < 4
  shows ∃ x xa xb xc xs. args = x#xa#xb#xc#xs
proof –
  from assms(2) have l ≥ 4 by auto
  hence unat l ≥ 4 by (simp add: const-le-unat)

```


with *assms* **have** *length args* ≥ 4 **by** *auto*
thus *?thesis* **by** (*auto simp: numeral-eq-Suc Suc-le-length-iff*)
qed

lemma *ucast-drop-big-mask*:
 $UCAST(64 \rightarrow 16) (x \ \&\& \ 0xFFFF) = UCAST(64 \rightarrow 16) x$
by *word-bitwise*

lemma *first-port-last-port-compare*:
 $UCAST(16 \rightarrow 32 \text{ signed}) (UCAST(64 \rightarrow 16) (xa \ \&\& \ 0xFFFF))$
 $<_s UCAST(16 \rightarrow 32 \text{ signed}) (UCAST(64 \rightarrow 16) (x \ \&\& \ 0xFFFF))$
 $= (UCAST(64 \rightarrow 16) xa < UCAST(64 \rightarrow 16) x)$
apply (*clarsimp simp: word-sless-alt ucast-drop-big-mask*)
apply (*subst sint-ucast-eq-uint, clarsimp simp: is-down*)
by (*simp add: word-less-alt*)

lemma *machine-word-and-3F-less-40*:
 $(w :: \text{machine-word}) \ \&\& \ 0x3F < 0x40$
by (*rule word-and-less', simp*)

lemmas *unat64-eq-of-nat = unat-eq-of-nat*[**where** *'a=64, folded word-bits-def*]

lemma *unat-mask-3-less-8*:
 $unat (p \ \&\& \ \text{mask } 3 :: \text{word64}) < 8$
apply (*rule unat-less-helper*)
apply (*rule order-le-less-trans, rule word-and-le1*)
apply (*simp add: mask-def*)
done

lemma *scast-specific-plus64*:
 $scast (of_nat (word_ctz x) + 0x20 :: 64 \text{ signed word}) = of_nat (word_ctz x) +$
 $(0x20 :: \text{machine-word})$
by (*metis of-nat-add of-nat-numeral scast-of-nat*)

lemma *scast-specific-plus64-signed*:
 $scast (of_nat (word_ctz x) + 0x20 :: \text{machine-word}) = of_nat (word_ctz x) + (0x20$
 $:: 64 \text{ signed word})$
by (*metis scast-scast-id(2) scast-specific-plus64*)

lemmas *mask-64-id[simp] = mask-len-id*[**where** *'a=64, folded word-bits-def*]
 $mask-len-id$ [**where** *'a=64, simplified*]

lemma *neg-0-unat*: $x \neq 0 \implies 0 < unat x$ **for** $x :: \text{machine-word}$
by (*simp add: unat-gt-0*)

end

11.2 Distinct Proposition

theory *Distinct-Prop*

imports

Word-Lib.Many-More

HOL-Library.Prefix-Order

begin

primrec

distinct-prop :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a list \Rightarrow bool)

where

distinct-prop P [] = True

| *distinct-prop* P (x # xs) = (($\forall y \in \text{set } xs. P x y$) \wedge *distinct-prop* P xs)

primrec

distinct-sets :: 'a set list \Rightarrow bool

where

distinct-sets [] = True

| *distinct-sets* (x#xs) = (x \cap \bigcup (set xs) = {}) \wedge *distinct-sets* xs)

lemma *distinct-prop-map*:

distinct-prop P (map f xs) = *distinct-prop* ($\lambda x y. P (f x) (f y)$) xs

by (*induct xs*) *auto*

lemma *distinct-prop-append*:

distinct-prop P (xs @ ys) =

(*distinct-prop* P xs \wedge *distinct-prop* P ys \wedge ($\forall x \in \text{set } xs. \forall y \in \text{set } ys. P x y$))

by (*induct xs arbitrary: ys*) (*auto simp: conj-comms ball-Un*)

lemma *distinct-prop-distinct*:

[*distinct xs*; $\bigwedge x y. [x \in \text{set } xs; y \in \text{set } xs; x \neq y] \Longrightarrow P x y$] \Longrightarrow *distinct-prop* P xs

by (*induct xs*) *auto*

lemma *distinct-prop-True* [*simp*]:

distinct-prop ($\lambda x y. \text{True}$) xs

by (*induct xs, auto*)

lemma *distinct-prefix*:

[*distinct xs*; *ys* \leq xs] \Longrightarrow *distinct ys*

apply (*induct xs arbitrary: ys; clarsimp*)

subgoal for a xs ys

apply (*cases ys; clarsimp*)

by (*fastforce simp: less-eq-list-def dest: set-mono-prefix*)

done

lemma *distinct-sets-prop*:

distinct-sets $xs = \text{distinct-prop } (\lambda x y. x \cap y = \{\}) xs$
by (*induct xs*) *auto*

lemma *distinct-take-strg*:
distinct xs \longrightarrow *distinct (take n xs)*
by *simp*

lemma *distinct-prop-prefixE*:
 $\llbracket \text{distinct-prop } P ys; \text{prefix } xs ys \rrbracket \Longrightarrow \text{distinct-prop } P xs$
apply (*induct xs arbitrary: ys; clarsimp*)
subgoal for *a xs ys*
apply (*cases ys; clarsimp*)
by (*fastforce dest: set-mono-prefix*)
done

lemma *distinct-sets-union-sub*:
 $\llbracket x \in A; \text{distinct-sets } [A,B] \rrbracket \Longrightarrow A \cup B - \{x\} = A - \{x\} \cup B$
by (*auto simp: distinct-sets-def*)

lemma *distinct-sets-append*:
distinct-sets (xs @ ys) \Longrightarrow *distinct-sets xs* \wedge *distinct-sets ys*
apply (*subst distinct-sets-prop*)
apply (*subst (asm) distinct-sets-prop*)
apply (*subst (asm) distinct-prop-append*)
apply *clarsimp*
done

lemma *distinct-sets-append1*:
distinct-sets (xs @ ys) \Longrightarrow *distinct-sets xs*
by (*drule distinct-sets-append, simp*)

lemma *distinct-sets-append2*:
distinct-sets (xs @ ys) \Longrightarrow *distinct-sets ys*
by (*drule distinct-sets-append, simp*)

lemma *distinct-sets-append-Cons*:
distinct-sets (xs @ a # ys) \Longrightarrow *distinct-sets (xs @ ys)*
apply (*subst distinct-sets-prop*)
apply (*subst (asm) distinct-sets-prop*)
apply (*subst distinct-prop-append*)
apply (*subst (asm) distinct-prop-append*)
apply *clarsimp*
done

lemma *distinct-sets-append-Cons-disjoint*:
distinct-sets (xs @ a # ys) \Longrightarrow $a \cap \bigcup (\text{set } xs) = \{\}$
apply (*subst (asm) distinct-sets-prop*)
apply (*subst (asm) distinct-prop-append*)
apply (*subst Int-commute*)

apply (*subst Union-disjoint*)
apply *clarsimp*
done

lemma *distinct-prop-take*:
 $\llbracket \text{distinct-prop } P \text{ } xs; i < \text{length } xs \rrbracket \implies \text{distinct-prop } P \text{ } (\text{take } i \text{ } xs)$
by (*metis take-is-prefix distinct-prop-prefixE*)

lemma *distinct-sets-take*:
 $\llbracket \text{distinct-sets } xs; i < \text{length } xs \rrbracket \implies \text{distinct-sets } (\text{take } i \text{ } xs)$
by (*simp add: distinct-sets-prop distinct-prop-take*)

lemma *distinct-prop-take-Suc*:
 $\llbracket \text{distinct-prop } P \text{ } xs; i < \text{length } xs \rrbracket \implies \text{distinct-prop } P \text{ } (\text{take } (\text{Suc } i) \text{ } xs)$
by (*metis distinct-prop-take not-less take-all*)

lemma *distinct-sets-take-Suc*:
 $\llbracket \text{distinct-sets } xs; i < \text{length } xs \rrbracket \implies \text{distinct-sets } (\text{take } (\text{Suc } i) \text{ } xs)$
by (*simp add: distinct-sets-prop distinct-prop-take-Suc*)

lemma *distinct-prop-rev*:
 $\text{distinct-prop } P \text{ } (\text{rev } xs) = \text{distinct-prop } (\lambda y \ x. P \ x \ y) \text{ } xs$
by (*induct xs*) (*auto simp: distinct-prop-append*)

lemma *distinct-sets-rev [simp]*:
 $\text{distinct-sets } (\text{rev } xs) = \text{distinct-sets } xs$
apply (*unfold distinct-sets-prop*)
apply (*subst distinct-prop-rev*)
apply (*subst Int-commute*)
apply *clarsimp*
done

lemma *distinct-sets-drop*:
 $\llbracket \text{distinct-sets } xs; i < \text{length } xs \rrbracket \implies \text{distinct-sets } (\text{drop } i \text{ } xs)$
apply (*cases i=0, simp*)
apply (*subst distinct-sets-rev [symmetric]*)
apply (*subst rev-drop*)
apply (*subst distinct-sets-take, simp-all*)
done

lemma *distinct-sets-drop-Suc*:
 $\llbracket \text{distinct-sets } xs; i < \text{length } xs \rrbracket \implies \text{distinct-sets } (\text{drop } (\text{Suc } i) \text{ } xs)$
apply (*subst distinct-sets-rev [symmetric]*)
apply (*subst rev-drop*)
apply (*subst distinct-sets-take, simp-all*)
done

lemma *distinct-sets-take-nth*:
 $\llbracket \text{distinct-sets } xs; i < \text{length } xs; x \in \text{set } (\text{take } i \text{ } xs) \rrbracket \implies x \cap xs ! i = \{\}$

```

apply (drule (1) distinct-sets-take-Suc)
apply (subst (asm) take-Suc-conv-app-nth, assumption)
apply (unfold distinct-sets-prop)
apply (subst (asm) distinct-prop-append)
apply clarsimp
done

```

lemma *distinct-sets-drop-nth*:

```

 $\llbracket \text{distinct-sets } xs; i < \text{length } xs; x \in \text{set } (\text{drop } (\text{Suc } i) \text{ } xs) \rrbracket \implies x \cap xs ! i = \{\}$ 
apply (drule (1) distinct-sets-drop)
apply (subst (asm) drop-Suc-nth, assumption)
apply fastforce
done

```

lemma *distinct-sets-append-distinct*:

```

 $\llbracket x \in \text{set } xs; y \in \text{set } ys; \text{distinct-sets } (xs @ ys) \rrbracket \implies x \cap y = \{\}$ 
unfolding distinct-sets-prop by (clarsimp simp: distinct-prop-append)

```

lemma *distinct-sets-update*:

```

 $\llbracket a \subseteq xs ! i; \text{distinct-sets } xs; i < \text{length } xs \rrbracket \implies \text{distinct-sets } (xs[i := a])$ 
apply (subst distinct-sets-prop)
apply (subst (asm) distinct-sets-prop)
apply (subst upd-conv-take-nth-drop, simp)
apply (subst distinct-prop-append)
apply (intro conjI)
  apply (erule (1) distinct-prop-take)
  apply (rule conjI |clarsimp)+
  apply (fold distinct-sets-prop)
  apply (drule (1) distinct-sets-drop)
  apply (subst (asm) drop-Suc-nth, assumption)
  apply fastforce
apply (drule (1) distinct-sets-drop)
apply (subst (asm) drop-Suc-nth, assumption)
apply clarsimp
apply clarsimp
apply (rule conjI)
  apply (drule (2) distinct-sets-take-nth)
  apply blast
apply clarsimp
apply (thin-tac  $P \subseteq Q$  for  $P Q$ )
apply (subst (asm) id-take-nth-drop, assumption)
apply (drule distinct-sets-append-Cons)
apply (erule (2) distinct-sets-append-distinct)
done

```

lemma *distinct-sets-map-update*:

```

 $\llbracket \text{distinct-sets } (\text{map } f \text{ } xs); i < \text{length } xs; f a \subseteq f(xs ! i) \rrbracket$ 
 $\implies \text{distinct-sets } (\text{map } f \text{ } (xs[i := a]))$ 
by (metis distinct-sets-update length-map map-update nth-map)

```

lemma *Union-list-update*:

$\llbracket i < \text{length } xs; \text{distinct-sets } (\text{map } f \text{ } xs) \rrbracket$
 $\implies (\bigcup_{x \in \text{set } (xs [i := a])}. f \ x) = (\bigcup_{x \in \text{set } xs}. f \ x) - f \ (xs ! i) \cup f \ a$
apply (*induct xs arbitrary: i; clarsimp*)
subgoal for $x \ xs \ i$
apply (*cases i; (clarsimp, fastforce)*)
done
done

lemma *fst-enumerate*:

$i < \text{length } xs \implies \text{fst } (\text{enumerate } n \ xs ! i) = i + n$
by (*metis add.commute fst-conv nth-enumerate-eq*)

lemma *snd-enumerate*:

$i < \text{length } xs \implies \text{snd } (\text{enumerate } n \ xs ! i) = xs ! i$
by (*metis nth-enumerate-eq snd-conv*)

lemma *enumerate-member*:

assumes $i < \text{length } xs$
shows $(n + i, xs ! i) \in \text{set } (\text{enumerate } n \ xs)$
proof –
have *pair-unpack*: $\bigwedge a \ b \ x. ((a, b) = x) = (a = \text{fst } x \wedge b = \text{snd } x)$ **by** *fastforce*
from *assms* **have** $(n + i, xs ! i) = \text{enumerate } n \ xs ! i$
by (*auto simp: fst-enumerate snd-enumerate pair-unpack*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *distinct-prop-nth*:

$\llbracket \text{distinct-prop } P \ ls; n < n'; n' < \text{length } ls \rrbracket \implies P \ (ls ! n) \ (ls ! n')$
apply (*induct ls arbitrary: n n'; simp*)
subgoal for $l \ ls \ n \ n'$
apply (*cases n'; simp*)
apply (*cases n; simp*)
done
done

lemma *distinct-prop-iff*:

$\llbracket \bigwedge x \ y. P \ x \ y \longleftrightarrow Q \ x \ y \rrbracket \implies \text{distinct-prop } P \ xs \longleftrightarrow \text{distinct-prop } Q \ xs$
by (*induction xs*) *auto*

lemma *distinct-prop-impl*:

$\llbracket \bigwedge x \ y. x \in \text{set } xs \implies y \in \text{set } xs \implies P \ x \ y \implies Q \ x \ y; \text{distinct-prop } P \ xs \rrbracket \implies$
distinct-prop $Q \ xs$
by (*induction xs*) *auto*

lemma *distinct-iff-distinct-prop-ne*: $\text{distinct } xs \longleftrightarrow \text{distinct-prop } (\neq) \ xs$

by (*induction xs*) *auto*

end

theory *WordSetup*

imports

Word-Lemmas-32-Internal

Word-Lemmas-64-Internal

Distinct-Prop

begin

lemma *distinct-prop-enum*:

$\llbracket \bigwedge x y. \llbracket x \leq stop; y \leq stop; x \neq y \rrbracket \implies P x y \rrbracket$

$\implies distinct-prop P [0 :: 'a :: len word .e. stop]$

apply (*simp add: upto-enum-def distinct-prop-map del: upt.simps*)

apply (*rule distinct-prop-distinct*)

apply *simp*

apply (*simp add: less-Suc-eq-le del: upt.simps*)

apply (*erule-tac x=of-nat x in meta-allE*)

apply (*erule-tac x=of-nat y in meta-allE*)

apply (*frule-tac y=x in unat-le*)

apply (*frule-tac y=y in unat-le*)

apply (*erule word-unat.Rep-cases*)+

apply (*simp add: toEnum-of-nat[OF unat-lt2p]*
word-le-nat-alt)

done

lemma *distinct-prop-enum-step*:

$\llbracket \bigwedge x y. \llbracket x \leq stop \ div \ step; y \leq stop \ div \ step; x \neq y \rrbracket \implies P (x * step) (y * step) \rrbracket$

$\implies distinct-prop P [0, step .e. stop]$

apply (*simp add: upto-enum-step-def distinct-prop-map*)

apply (*rule distinct-prop-enum*)

apply *simp*

done

lemmas *word-bits-def =*

Machine-Word-64-Basics.word-bits-def Machine-Word-32-Basics.word-bits-def

lemmas *word-size-def =*

Machine-Word-64-Basics.word-size-def Machine-Word-32-Basics.word-size-def

lemmas *word-bits-size =*

Machine-Word-64.word-bits-size Machine-Word-32.word-bits-size

lemmas *word-bits-len-of =*

```

Machine-Word-64.word-bits-len-of Machine-Word-32.word-bits-len-of

lemmas word-bits-conv =
  Machine-Word-64-Basics.word-bits-conv Machine-Word-32-Basics.word-bits-conv

hide-const (open) Machine-Word-32-Basics.word-bits
hide-const (open) Machine-Word-64-Basics.word-bits
hide-const (open) Machine-Word-32-Basics.word-size
hide-const (open) Machine-Word-64-Basics.word-size
hide-const (open) Machine-Word-32-Basics.word-size-bits
hide-const (open) Machine-Word-64-Basics.word-size-bits

end

theory Addr-Type-ARM
  imports
    Target-Architecture
    WordSetup
  begin

if-architecture-context (ARM)
  begin
type-synonym addr-bitsize = 32

definition addr-bitsize :: nat where addr-bitsize ≡ 32
definition addr-align :: nat where addr-align ≡ 2

abbreviation (input) array-outer-max-size-exponent ≡ 19::nat
abbreviation (input) array-outer-max-count-exponent ≡ 13::nat
abbreviation (input) array-inner-max-size-exponent ≡ 6::nat

abbreviation word-bits ≡ Machine-Word-32-Basics.word-bits
abbreviation word-size ≡ Machine-Word-32-Basics.word-size
abbreviation word-size-bits ≡ Machine-Word-32-Basics.word-size-bits
type-synonym char-c = 8 word
  end

end

theory Addr-Type-ARM64
  imports
    Target-Architecture
    WordSetup

```



```

begin

if-architecture-context (ARM64)
begin
type-synonym addr-bitsize = 64

definition addr-bitsize :: nat where addr-bitsize ≡ 64
definition addr-align :: nat where addr-align ≡ 3

abbreviation (input) array-outer-max-size-exponent ≡ 26::nat
abbreviation (input) array-outer-max-count-exponent ≡ 20::nat
abbreviation (input) array-inner-max-size-exponent ≡ 6::nat

abbreviation word-bits ≡ Machine-Word-64-Basics.word-bits
abbreviation word-size ≡ Machine-Word-64-Basics.word-size
abbreviation word-size-bits ≡ Machine-Word-64-Basics.word-size-bits
type-synonym char-c = 8 signed word
end

end

theory Addr-Type-ARM-HYP
imports
  Target-Architecture
  WordSetup
begin

if-architecture-context (ARM-HYP)
begin
type-synonym addr-bitsize = 32

definition addr-bitsize :: nat where addr-bitsize ≡ 32
definition addr-align :: nat where addr-align ≡ 2

abbreviation (input) array-outer-max-size-exponent ≡ 19::nat
abbreviation (input) array-outer-max-count-exponent ≡ 13::nat
abbreviation (input) array-inner-max-size-exponent ≡ 6::nat

abbreviation word-bits ≡ Machine-Word-32-Basics.word-bits
abbreviation word-size ≡ Machine-Word-32-Basics.word-size
abbreviation word-size-bits ≡ Machine-Word-32-Basics.word-size-bits
type-synonym char-c = 8 word
end

end

```

```

theory Addr-Type-RISCV64
  imports
    Target-Architecture
    WordSetup
begin

if-architecture-context (RISCV64)
begin
type-synonym addr-bitsize = 64

definition addr-bitsize :: nat where addr-bitsize ≡ 64
definition addr-align :: nat where addr-align ≡ 3

abbreviation (input) array-outer-max-size-exponent ≡ 26::nat
abbreviation (input) array-outer-max-count-exponent ≡ 20::nat
abbreviation (input) array-inner-max-size-exponent ≡ 6::nat

abbreviation word-bits ≡ Machine-Word-64-Basics.word-bits
abbreviation word-size ≡ Machine-Word-64-Basics.word-size
abbreviation word-size-bits ≡ Machine-Word-64-Basics.word-size-bits
type-synonym char-c = 8 word
end

end

```

```

theory Addr-Type-X64
  imports
    Target-Architecture
    WordSetup
begin

if-architecture-context (X64)
begin

type-synonym addr-bitsize = 64

definition addr-bitsize :: nat where addr-bitsize ≡ 64
definition addr-align :: nat where addr-align ≡ 3

abbreviation (input) array-outer-max-size-exponent ≡ 26::nat
abbreviation (input) array-outer-max-count-exponent ≡ 20::nat
abbreviation (input) array-inner-max-size-exponent ≡ 6::nat

abbreviation word-bits ≡ Machine-Word-64-Basics.word-bits

```

```

abbreviation word-size  $\equiv$  Machine-Word-64-Basics.word-size
abbreviation word-size-bits  $\equiv$  Machine-Word-64-Basics.word-size-bits
type-synonym char-c = 8 word
end

end

```

```

theory Addr-Type
  imports
    ARM/Addr-Type-ARM
    ARM64/Addr-Type-ARM64
    ARM-HYP/Addr-Type-ARM-HYP
    RISCV64/Addr-Type-RISCV64
    X64/Addr-Type-X64
  begin

  typ addr-bitsize
  term array-outer-max-size-exponent
  type-synonym addr = addr-bitsize word

  declare addr-align-def[simp]
  declare addr-bitsize-def[simp]

  definition addr-card :: nat where
    addr-card  $\equiv$  card (UNIV::addr set)

  lemma addr-card:
    addr-card =  $2^{\text{addr-bitsize}}$ 
  by (simp add: addr-card-def card-word)

  lemma len-of-addr-card:
     $2^{\text{len-of TYPE(addr-bitsize)}} = \text{addr-card}$ 
  by (simp add: addr-card)

  lemma of-nat-addr-card [simp]:
    of-nat addr-card = (0::addr)
  by (simp add: addr-card)

end

```

```

theory CTypesBase
imports
  Addr-Type
begin

```

11.3 Type setup

```
type-synonym byte = 8 word
```

```
type-synonym memory = addr ⇒ byte
```

```
type-synonym 'a mem-upd = addr ⇒ 'a ⇒ memory ⇒ memory
```

```
type-synonym 'a mem-read = addr ⇒ memory ⇒ 'a
```

```
class unit-class =
```

```
  assumes there-is-only-one: x = y
```

```
instantiation unit :: unit-class
```

```
begin
```

```
instance by (intro-classes, simp)
```

```
end
```

11.3.1 Pointers

```
datatype 'a ptr = Ptr addr
```

```
abbreviation
```

```
  NULL :: 'a ptr where
```

```
  NULL ≡ Ptr 0
```

```
ML <
```

```
  structure Ptr-Syntax =
```

```
  struct
```

```
    val show-ptr-types = Attrib.setup-config-bool @{binding show-ptr-types} (K true)
```

```
    fun ptr-tr' cnst ctxt typ ts = if Config.get ctxt show-ptr-types then
```

```
      case Term.strip-type typ of
```

```
        ([@{typ addr}], Type (@{type-name ptr}, [T])) =>
```

```
          list-comb
```

```
            (Syntax.const cnst $ Syntax-Phases.term-of-typ ctxt T
```

```
              , ts)
```

```
          | - => raise Match
```

```
    else raise Match
```

```
    fun ptr-coerce-tr' cnst ctxt typ ts = if Config.get ctxt show-ptr-types then
```

```
      case Term.strip-type typ of
```

```
        ([Type (@{type-name ptr}, [S]), Type (@{type-name ptr}, [T])) =>
```

```
          list-comb
```

```
            (Syntax.const cnst $ Syntax-Phases.term-of-typ ctxt S
```

```
              $ Syntax-Phases.term-of-typ ctxt T
```

```
              , ts)
```

```
          | - => raise Match
```

```
    else raise Match
```

```
  end
```

```
>
```

syntax

$-Ptr :: type \Rightarrow logic ((1PTR/(1'(-))))$

translations

$PTR('a) \Rightarrow CONST Ptr :: (addr \Rightarrow 'a ptr)$

typed-print-translation

$\langle [(@\{const-syntax Ptr\}, Ptr-Syntax.ptr-tr' @\{syntax-const -Ptr\})] \rangle$

primrec

$ptr-val :: 'a ptr \Rightarrow addr$

where

$ptr-val-def: ptr-val (Ptr a) = a$

primrec

$ptr-coerce :: 'a ptr \Rightarrow 'b ptr$ **where**

$ptr-coerce (Ptr a) = Ptr a$

syntax

$-Ptr-coerce :: type \Rightarrow type \Rightarrow logic ((1PTR'-COERCE/(1'(- \rightarrow -))))$

translations

$PTR-COERCE('a \rightarrow 'b) \Rightarrow CONST ptr-coerce :: ('a ptr \Rightarrow 'b ptr)$

typed-print-translation

$\langle [(@\{const-syntax ptr-coerce\}, Ptr-Syntax.ptr-coerce-tr' @\{syntax-const -Ptr-coerce\})] \rangle$

definition

$ptr-less :: 'a ptr \Rightarrow 'a ptr \Rightarrow bool$ (**infixl** $<_p$ 50) **where**

$p <_p q \equiv ptr-val p < ptr-val q$

definition

$ptr-le :: 'a ptr \Rightarrow 'a ptr \Rightarrow bool$ (**infixl** \leq_p 50) **where**

$p \leq_p q \equiv ptr-val p \leq ptr-val q$

instantiation $ptr :: (type) ord$

begin

definition

$ptr-less-def': p < q \equiv p <_p q$

definition

$ptr-le-def': p \leq q \equiv p \leq_p q$

instance ..

end

lemma $ptr-val-case: ptr-val p = (case p of Ptr v \Rightarrow v)$

by ($cases p$) $simp$

```

instantiation ptr :: (type) linorder
begin
instance
  by (intro-classes)
    (unfold ptr-le-def' ptr-le-def ptr-less-def' ptr-less-def ptr-val-case,
     auto split: ptr.splits)
end

```

11.3.2 Raw heap

A raw map from addresses to bytes

type-synonym heap-mem = addr ⇒ byte

For heap h , pointer p and nat n , *heap-list* h n p returns the list of bytes in the heap taken from addresses $\{p..+n\}$

primrec

heap-list :: heap-mem ⇒ nat ⇒ addr ⇒ byte list

where

heap-list-base: *heap-list* h 0 p = []

| *heap-list-rec*: *heap-list* h (Suc n) p = h p # *heap-list* h n ($p + 1$)

11.4 Intervals

For word a and nat b , $\{a..+b\}$ is the set of words x , with $unat (x - a) < b$.

definition

intvl :: 'a::len word × nat ⇒ 'a::len word set **where**

intvl x ≡ $\{z. \exists k. z = fst\ x + of_nat\ k \wedge k < snd\ x\}$

abbreviation

intvl-abbr :: 'a::len word ⇒ nat ⇒ 'a word set ($\{..+..\}$) **where**

$\{a..+b\}$ ≡ *intvl* (a, b)

11.5 dt-tuple: a reimplementaion of 3 item tuples

datatype

($'a, 'b, 'c$) *dt-tuple* = DTuple (*dt-fst*: 'a) (*dt-snd*: 'b) (*dt-trd*: 'c)

lemma *dt-prj-simps*[simp]:

dt-fst (DTuple a b c) = a

dt-snd (DTuple a b c) = b

dt-trd (DTuple a b c) = c

by (*auto*)

lemma *split-DTuple-All*:

$(\forall x. P\ x) = (\forall a\ b\ c. P\ (DTuple\ a\ b\ c))$

apply (*rule iffI; clarsimp*)

subgoal for x **by** (*cases* x , *simp*)

done

lemma *surjective-dt-tuple*:

$p = DTuple (dt-fst p) (dt-snd p) (dt-trd p)$
by (*cases p simp*)

lemma *split-DTuple-all[no-atp]*: $(\bigwedge x. PROP P x) \equiv (\bigwedge a b c. PROP P (DTuple a b c))$

proof

fix $a b c$

assume $\bigwedge x. PROP P x$

then show $PROP P (DTuple a b c)$.

next

fix x

assume $\bigwedge a b c. PROP P (DTuple a b c)$

from $\langle PROP P (DTuple (dt-fst x) (dt-snd x) (dt-trd x)) \rangle$ **show** $PROP P x$ **by**
simp

qed

type-synonym *normalisor* = *byte list* \Rightarrow *byte list*

11.6 Properties of pointers

lemma *Ptr-ptr-val [simp]*:

$Ptr (ptr-val p) = p$
by (*cases p simp*)

lemma *ptr-val-ptr-coerce [simp]*:

$ptr-val (ptr-coerce p) = ptr-val p$
by (*cases p simp*)

lemma *Ptr-ptr-coerce [simp]*:

$Ptr (ptr-val p) = ptr-coerce p$
by (*cases p simp*)

lemma *ptr-coerce-id [simp]*:

$ptr-coerce p = p$
by (*cases p simp*)

lemma *ptr-coerce-idem [simp]*:

$ptr-coerce (ptr-coerce p) = ptr-coerce p$
by (*cases p simp*)

lemma *ptr-val-inj [simp]*:

$(ptr-val p = ptr-val q) = (p = q)$
by (*cases p, cases q auto*)

lemma *ptr-coerce-NULL [simp]*:

$(ptr-coerce p = NULL) = (p = NULL)$

by (*cases p*) *simp*

lemma *NULL-ptr-val*:

$(p = \text{NULL}) = (\text{ptr-val } p = 0)$

by (*cases p*) *simp*

lemma *ptr-NULL-conv*: *ptr-coerce NULL = NULL*

by *simp*

instantiation *ptr* :: (*type*) *finite*

begin

instance

by (*intro-classes*)

(*auto intro!*: *finite-code finite-imageD* [**where** *f=ptr-val*] *injI*)

end

11.7 Properties of the raw heap

lemma *heap-list-length* [*simp*]:

$\text{length } (\text{heap-list } h \ n \ p) = n$

by (*induct n arbitrary: p*) *auto*

lemma *heap-list-split*:

shows $k \leq n \implies \text{heap-list } h \ n \ x = \text{heap-list } h \ k \ x \ @ \ \text{heap-list } h \ (n - k) \ (x + \text{of-nat } k)$

proof (*induct n arbitrary: k x*)

case 0 thus ?case **by** *simp*

next

case (Suc n) thus ?case

by (*cases k, auto simp: ac-simps*)

qed

lemma *heap-list-split2*:

$\text{heap-list } h \ (x + y) \ p = \text{heap-list } h \ x \ p \ @ \ \text{heap-list } h \ y \ (p + \text{of-nat } x)$

by (*subst heap-list-split* [**where** *k=x*], *auto*)

11.8 Properties of intervals

lemma *intvlI*:

$x < n \implies p + \text{of-nat } x \in \{p..+n\}$

by (*force simp: intvl-def*)

lemma *intvlD*:

$q \in \{p..+n\} \implies \exists k. q = p + \text{of-nat } k \wedge k < n$

by (*force simp: intvl-def*)

lemma *intvl-empty* [*simp*]:

$\{p..+0\} = \{\}$

by (fast dest: intvlD)

lemma *intvl-Suc*:

$q \in \{p..+Suc\ 0\} \implies p = q$
by (force dest: intvlD)

lemma *intvl-self*:

$0 < n \implies x \in \{x..+n\}$
by (force simp: intvl-def)

lemma *intvl-start-inter*:

$\llbracket 0 < m; 0 < n \rrbracket \implies \{p..+m\} \cap \{p..+n\} \neq \{\}$
by (force simp: disjoint-iff-not-equal dest: intvl-self)

lemma *intvl-overflow*:

assumes $2^{\wedge}len\text{-of}\ TYPE('a) \leq n$
shows $\{(p::'a::len\ word)..+n\} = UNIV$

proof -

have *witness*:

$\bigwedge x. x = p + of\text{-nat}\ (unat\ (x - p)) \wedge unat\ (x - p) < n$
using *assms* by *simp unat-arith*

show *?thesis unfolding intvl-def* by (auto intro!: *witness*)

qed

lemma *intvl-self-offset*:

fixes $p::'a::len\ word$

assumes $a: 2^{\wedge}len\text{-of}\ TYPE('a) - n < x$ and $b: x < 2^{\wedge}len\text{-of}\ TYPE('a)$ and
 $c: (p::'a::len\ word) \notin \{p + of\text{-nat}\ x..+n\}$

shows *False*

proof -

let $?j = 2^{\wedge}len\text{-of}\ TYPE('a) - x$

from b have $b': of\text{-nat}\ x + of\text{-nat}\ ?j = (0::'a::len\ word)$ using *of-nat-2p* by *auto*

moreover from a b have $?j < n$ by *arith*

with b b' c show *?thesis* by (force simp: *intvl-def*)

qed

lemma *intvl-mem-offset*:

$\llbracket q \in \{p..+unat\ x\}; q \notin \{p..+unat\ y\}; unat\ y \leq unat\ x \rrbracket \implies$
 $q \in \{p + y..+unat\ x - unat\ y\}$

apply (clarsimp simp: *intvl-def*)

subgoal for k

apply (rule *exI* [where $x=k - unat\ y$])

apply *auto*

done

done

lemma *intvl-plus-sub-offset*:

$x \in \{p + y..+q - \text{unat } y\} \implies x \in \{p..+q\}$
apply (*clarsimp simp: intvl-def*)
subgoal for *k*
apply (*rule exI [where x=k + unat y]*)
apply *auto*
done
done

lemma *intvl-plus-sub-Suc*:
 $x \in \{p + 1..+q - \text{Suc } 0\} \implies x \in \{p..+q\}$
by (*rule intvl-plus-sub-offset [where y=1], simp*)

lemma *intvl-neq-start*:
 $\llbracket (q::'a::\text{len word}) \in \{p..+n\}; p \neq q \rrbracket \implies q \in \{p + 1..+n - \text{Suc } 0\}$
apply (*clarsimp simp: intvl-def*)
by (*metis One-nat-def Suc-eq-plus1-left add.commute gr0-conv-Suc less-diff-conv of-nat-Suc of-nat-gt-0*)

lemmas *unatsimps'* =
word-arith-nat-defs word-unat.eq-norm len-of-addr-card mod-less

lemma *intvl-offset-nmem*:
 $\llbracket q \in \{(p::'a::\text{len word})..+\text{unat } x\}; y \leq 2^{\wedge} \text{len-of TYPE('a)} - \text{unat } x \rrbracket \implies$
 $q \notin \{p + x..+y\}$
apply (*clarsimp simp: intvl-def*)
apply (*simp only: unatsimps'*)
apply (*subst (asm) word-unat.Abs-inject*)
apply (*auto simp: unats-def*)
done

lemma *intvl-Suc-nmem'* [*simp*]:
 $n < 2^{\wedge} \text{len-of TYPE('a)} \implies (p::'a::\text{len word}) \notin \{p + 1..+n - \text{Suc } 0\}$
by (*clarsimp simp: intvl-def*)
(unat-arith, simp add: unatsimps' take-bit-nat-eq-self)

lemma *intvl-Suc-nmem''*:
 $n \leq 2^{\wedge} \text{len-of TYPE('a)} \implies (p::'a::\text{len word}) \notin \{p + 1..+n - \text{Suc } 0\}$
by (*simp add: intvl-offset-nmem intvl-self*)

lemma *intvl-start-le*:
 $x \leq y \implies \{p..+x\} \subseteq \{p..+y\}$
by (*force simp: intvl-def*)

lemma *intvl-sub-eq*:
assumes $x \leq y$
shows $\{p + x..+\text{unat } (y - x)\} = \{p..+\text{unat } y\} - \{p..+\text{unat } x\}$
proof –
have $\text{unat } y - \text{unat } x \leq 2^{\wedge} \text{len-of TYPE('a)} - \text{unat } x$
by (*insert unat-lt2p [of y], arith*)

moreover have $x \leq y$ **by** *fact*
moreover hence $\text{unat } (y - x) = \text{unat } y - \text{unat } x$
by (*simp add: word-le-nat-alt, unat-arith*)
ultimately show *?thesis*
by (*force dest: intvl-offset-nmem intvl-mem-offset elim: intvl-plus-sub-offset*
simp: word-le-nat-alt)

qed

lemma *intvl-disj-offset*:
 $\{x + a..+c\} \cap \{x + b..+d\} = \{\} = (\{a..+c\} \cap \{b..+d\} = \{\})$
by (*force simp: intvl-def*)

lemma *intvl-sub-offset*:
 $\text{unat } x + y \leq z \implies \{k + x..+y\} \subseteq \{k..+z\}$
apply(*clarsimp simp: intvl-def*)
subgoal for k
apply(*rule exI [where x=unat x + k]*)
apply *clarsimp*
done
done

lemma *disjnt-intvl-offsetp[simp]*:
 $\text{disjnt } \{a + x ..+ n\} \{a + y ..+ m\} \longleftrightarrow \text{disjnt } \{x ..+ n\} \{y ..+ m\}$
by (*simp add: disjnt-def intvl-disj-offset*)

lemma *intvl-eq-of-nat-Ico-add*: $\{\text{of-nat } n::'a::\text{len word}..+ m\} = \text{of-nat } \{n ..< n + m\}$
by (*force simp: image-iff intvl-def Bex-def nat-le-iff-add simp flip: of-nat-add*)

lemma *intvl-le*:
assumes $n2 + \text{off1} \leq n1$
shows $\{p + \text{of-nat } \text{off1} ..+ n2\} \subseteq \{p..+n1\}$
using *assms*
by (*auto simp add: intvl-def*)
(metis add.commute add-mono-thms-linordered-field(1) less-le-trans word-of-nat-plus)

lemma *intvl-disj-left*:
fixes $a b :: \text{addr}$
assumes $a-n: \text{unat } a + n \leq \text{unat } b$ **and** $b-m: \text{unat } b + m \leq \text{addr-card} + \text{unat } a$
shows $\{a ..+ n\} \cap \{b ..+ m\} = \{\}$
proof (*safe dest!: intvlD intro!: empty-iff[THEN iffD2]*)
fix $i j$ **assume** $i: i < n$ **and** $j: j < m$ **and** $\text{eq}: a + \text{of-nat } i = b + \text{of-nat } j$
have $a-i\text{-eq}: \text{unat } (a + \text{of-nat } i) = \text{unat } a + \text{of-nat } i$
using $a-n i$
by (*metis (mono-tags, lifting) Abs-fnat-hom-add add-left-mono le-unat-voi*
less-or-eq-imp-le of-nat-id word-unat.Rep-inverse)
from $a-n b-m$ **have** $n-m: n + m \leq \text{addr-card}$

```

  by simp
show False
proof cases
  assume unat b + j < addr-card
  then have unat (b + of-nat j) = unat b + of-nat j
  by (metis len-of-addr-card of-nat-add of-nat-id unat-of-nat-eq word-unat.Rep-inverse)
  with eq a-i-eq have unat a + i = unat b + j
  unfolding word-unat-eq-iff by simp
  with a-n i show False by simp
next
  assume ¬ unat b + j < addr-card
  with j n-m have unat (b + of-nat j) + addr-card = unat b + j
  unfolding addr-card-def card-word unat-arith-simps(5)
  by (simp add: unat-of-nat-eq)
  then have unat a + i + addr-card = unat b + j
  using a-i-eq eq unfolding word-unat-eq-iff by simp
  with b-m j show False by simp
qed
qed
end

```

```

theory CTypesDefs
imports
  CTypesBase
begin

```

11.9 C types setup

```

type-synonym field-name = string
type-synonym qualified-field-name = field-name list

```

```

type-synonym typ-name = string

```

A *typ-desc* wraps a *typ-struct* with a typ name. A *typ-struct* is either a Scalar, with size, alignment and either a field description (for *typ-info*) or a 'normalisor' (for *typ-uinfo*), or an Aggregate, with a list of *typ-desc*, field name, and field descriptor (for the complete sub-structure) or unit (for *typ-uinfo*). The field description for aggregates is an extension of the original work of H. Tuch. It is used to make the construction of a new structure from nested structures / arrays more efficient. Properties like commutation of fields can be expressed and proven for the toplevel fields only, without having to re-examine the nested leaves of the tree.

datatype

$(\text{'a}, \text{'b}) \text{ typ-desc} = \text{TypDesc nat } (\text{'a}, \text{'b}) \text{ typ-struct typ-name}$
and $(\text{'a}, \text{'b}) \text{ typ-struct} = \text{TypScalar nat nat 'a} \mid$
 $\text{TypAggregate } ((\text{'a}, \text{'b}) \text{ typ-desc, field-name, 'b}) \text{ dt-tuple list}$

datatype-compatible dt-tuple

datatype-compatible $\text{typ-desc typ-struct}$

print-theorems

lemma typ-desc-induct :

$\llbracket \bigwedge \text{nat typ-struct list. } P2 \text{ typ-struct} \implies P1 (\text{TypDesc nat typ-struct list}); \bigwedge \text{nat1}$
 $\text{nat2 a. } P2 (\text{TypScalar nat1 nat2 a});$
 $\bigwedge \text{list. } P3 \text{ list} \implies P2 (\text{TypAggregate list}); P3 \llbracket; \bigwedge \text{dt-tuple list. } \llbracket P4 \text{ dt-tuple};$
 $P3 \text{ list} \rrbracket \implies P3 (\text{dt-tuple} \# \text{ list});$
 $\bigwedge \text{typ-desc list b. } P1 \text{ typ-desc} \implies P4 (\text{DTuple typ-desc list b}) \rrbracket$
 $\implies P1 \text{ typ-desc}$
by $(\text{rule compat-typ-desc.induct})$

lemma typ-struct-induct :

$\llbracket \bigwedge \text{nat typ-struct list. } P2 \text{ typ-struct} \implies P1 (\text{TypDesc nat typ-struct list}); \bigwedge \text{nat1}$
 $\text{nat2 a. } P2 (\text{TypScalar nat1 nat2 a});$
 $\bigwedge \text{list. } P3 \text{ list} \implies P2 (\text{TypAggregate list}); P3 \llbracket; \bigwedge \text{dt-tuple list. } \llbracket P4 \text{ dt-tuple};$
 $P3 \text{ list} \rrbracket \implies P3 (\text{dt-tuple} \# \text{ list});$
 $\bigwedge \text{typ-desc list b. } P1 \text{ typ-desc} \implies P4 (\text{DTuple typ-desc list b}) \rrbracket$
 $\implies P2 \text{ typ-struct}$
by $(\text{rule compat-typ-struct.induct})$

lemma typ-list-induct :

$\llbracket \bigwedge \text{nat typ-struct list. } P2 \text{ typ-struct} \implies P1 (\text{TypDesc nat typ-struct list}); \bigwedge \text{nat1}$
 $\text{nat2 a. } P2 (\text{TypScalar nat1 nat2 a});$
 $\bigwedge \text{list. } P3 \text{ list} \implies P2 (\text{TypAggregate list}); P3 \llbracket; \bigwedge \text{dt-tuple list. } \llbracket P4 \text{ dt-tuple};$
 $P3 \text{ list} \rrbracket \implies P3 (\text{dt-tuple} \# \text{ list});$
 $\bigwedge \text{typ-desc list b. } P1 \text{ typ-desc} \implies P4 (\text{DTuple typ-desc list b}) \rrbracket$
 $\implies P3 \text{ list}$
by $(\text{rule compat-typ-desc-char-list-dt-tuple-list.induct})$

lemma $\text{typ-dt-tuple-induct}$:

$\llbracket \bigwedge \text{nat typ-struct list. } P2 \text{ typ-struct} \implies P1 (\text{TypDesc nat typ-struct list}); \bigwedge \text{nat1}$
 $\text{nat2 a. } P2 (\text{TypScalar nat1 nat2 a});$
 $\bigwedge \text{list. } P3 \text{ list} \implies P2 (\text{TypAggregate list}); P3 \llbracket; \bigwedge \text{dt-tuple list. } \llbracket P4 \text{ dt-tuple};$
 $P3 \text{ list} \rrbracket \implies P3 (\text{dt-tuple} \# \text{ list});$
 $\bigwedge \text{typ-desc list b. } P1 \text{ typ-desc} \implies P4 (\text{DTuple typ-desc list b}) \rrbracket$
 $\implies P4 \text{ dt-tuple}$
by $(\text{rule compat-typ-desc-char-list-dt-tuple.induct})$

— Declare as default induct rule with old case names

lemmas $\text{typ-desc-typ-struct-inducts}$ $[\text{case-names}$

$\text{TypDesc TypScalar TypAggregate Nil-typ-desc Cons-typ-desc DTuple-typ-desc,}$
 $\text{induct type}] =$

typ-desc-induct typ-struct-induct typ-list-induct typ-dt-tuple-induct

— Make sure list induct rule is tried first

declare *list.induct* [*induct type*]

type-synonym (*'a*, *'b*) *typ-tuple* = ((*'a*, *'b*) *typ-desc.field-name*, *'b*) *dt-tuple*

type-synonym *typ-uinfo* = (*normalisor*, *unit*) *typ-desc*

type-synonym *typ-uinfo-struct* = (*normalisor*, *unit*) *typ-struct*

type-synonym *typ-uinfo-tuple* = (*normalisor*, *unit*) *typ-tuple*

record *'a field-desc* =

field-access :: *'a* \Rightarrow *byte list* \Rightarrow *byte list*

field-update :: *byte list* \Rightarrow *'a* \Rightarrow *'a*

field-sz :: *nat*

type-synonym (*'a*, *'b*) *typ-info* = (*'a field-desc*, *'b*) *typ-desc*

type-synonym (*'a*, *'b*) *typ-info-struct* = (*'a field-desc*, *'b*) *typ-struct*

type-synonym (*'a*, *'b*) *typ-info-tuple* = (*'a field-desc*, *'b*) *typ-tuple*

type-synonym *'a xtyp-tuple* = (*'a*, *'a*) *typ-tuple*

type-synonym *'a xtyp-info* = (*'a field-desc*, *'a field-desc*) *typ-desc*

type-synonym *'a xtyp-info-struct* = (*'a field-desc*, *'a field-desc*) *typ-struct*

type-synonym *'a xtyp-info-tuple* = *'a field-desc xtyp-tuple*

definition *fu-commutes* :: (*'b* \Rightarrow *'a* \Rightarrow *'a*) \Rightarrow (*'c* \Rightarrow *'a* \Rightarrow *'a*) \Rightarrow *bool* **where**

fu-commutes f g $\equiv \forall v bs bs'. f bs (g bs' v) = g bs' (f bs v)$

size-td returns the sum of the sizes of all Scalar fields comprising a *typ-desc* i.e. the overall size of the type

primrec

size-td :: (*'a*, *'b*) *typ-desc* \Rightarrow *nat* **and**

size-td-struct :: (*'a*, *'b*) *typ-struct* \Rightarrow *nat* **and**

size-td-list :: (*'a*, *'b*) *typ-tuple list* \Rightarrow *nat* **and**

size-td-tuple :: (*'a*, *'b*) *typ-tuple* \Rightarrow *nat*

where

tz0: *size-td* (*TypDesc* *algn st nm*) = *size-td-struct st*

| *tz1*: *size-td-struct* (*TypScalar n algn d*) = *n*

| *tz2*: *size-td-struct* (*TypAggregate xs*) = *size-td-list xs*

| *tz3*: *size-td-list* [] = 0

| *tz4*: *size-td-list* (*x#xs*) = *size-td-tuple x* + *size-td-list xs*

| *tz5*: *size-td-tuple* (*DTuple t n d*) = *size-td t*

access-ti overlays the byte-wise representation of an object on a given byte list, given the *typ-info* (i.e. the layout)

primrec

access-ti :: ('a, 'b) *typ-info* ⇒ ('a ⇒ *byte list* ⇒ *byte list*) **and**
access-ti-struct :: ('a, 'b) *typ-info-struct* ⇒
('a ⇒ *byte list* ⇒ *byte list*) **and**
access-ti-list :: ('a, 'b) *typ-info-tuple list* ⇒
('a ⇒ *byte list* ⇒ *byte list*) **and**
access-ti-tuple :: ('a, 'b) *typ-info-tuple* ⇒ ('a ⇒ *byte list* ⇒ *byte list*)

where

fa0: *access-ti* (*TypDesc* *algn st nm*) = *access-ti-struct st*

| *fa1*: *access-ti-struct* (*TypScalar n algn d*) = *field-access d*

| *fa2*: *access-ti-struct* (*TypAggregate xs*) = *access-ti-list xs*

| *fa3*: *access-ti-list* [] = (λ*v bs.* [])

| *fa4*: *access-ti-list* (*x#xs*) =
(λ*v bs.* *access-ti-tuple x v* (*take* (*size-td-tuple x*) *bs*) @
access-ti-list xs v (*drop* (*size-td-tuple x*) *bs*))

| *fa5*: *access-ti-tuple* (*DTuple t nm d*) = *access-ti t*

access-ti₀ overlays the representation of an object on a list of zero bytes

definition *access-ti₀* :: ('a, 'b) *typ-info* ⇒ ('a ⇒ *byte list*) **where**

access-ti₀ t ≡ λ*v.* *access-ti t v* (*replicate* (*size-td t*) 0)

update-ti updates an object, given a list of bytes (the representation of the new value), and the *typ-info*

primrec

update-ti :: ('a, 'b) *typ-info* ⇒ (*byte list* ⇒ 'a ⇒ 'a) **and**

update-ti-struct :: ('a, 'b) *typ-info-struct* ⇒ (*byte list* ⇒ 'a ⇒ 'a) **and**

update-ti-list :: ('a, 'b) *typ-info-tuple list* ⇒ (*byte list* ⇒ 'a ⇒ 'a) **and**

update-ti-tuple :: ('a, 'b) *typ-info-tuple* ⇒ (*byte list* ⇒ 'a ⇒ 'a)

where

fu0: *update-ti* (*TypDesc* *algn st nm*) = *update-ti-struct st*

| *fu1*: *update-ti-struct* (*TypScalar n algn d*) = *field-update d*

| *fu2*: *update-ti-struct* (*TypAggregate xs*) = *update-ti-list xs*

| *fu3*: *update-ti-list* [] = (λ*bs.* *id*)

| *fu4*: *update-ti-list* (*x#xs*) = (λ*bs v.*
update-ti-tuple x (*take* (*size-td-tuple x*) *bs*)
(*update-ti-list xs* (*drop* (*size-td-tuple x*) *bs*) *v*))

| *fu5*: *update-ti-tuple* (*DTuple t nm d*) = *update-ti t*

update-ti-t updates an object only if the length of the supplied representation equals the object size

definition *update-ti-t* :: ('a, 'b) *typ-info* ⇒ (*byte list* ⇒ 'a ⇒ 'a) **where**

update-ti-t t ≡ λ*bs.* *if length bs = size-td t then*

update-ti t bs else id

definition *update-ti-struct-t* :: ('a, 'b) *typ-info-struct* \Rightarrow (byte list \Rightarrow 'a \Rightarrow 'a)
where

update-ti-struct-t $t \equiv \lambda bs.$ if length *bs* = *size-td-struct* *t* then
update-ti-struct *t* *bs* else *id*

definition *update-ti-list-t* :: ('a, 'b) *typ-info-tuple list* \Rightarrow (byte list \Rightarrow 'a \Rightarrow 'a)
where

update-ti-list-t $t \equiv \lambda bs.$ if length *bs* = *size-td-list* *t* then
update-ti-list *t* *bs* else *id*

definition *update-ti-tuple-t* :: ('a, 'b) *typ-info-tuple* \Rightarrow (byte list \Rightarrow 'a \Rightarrow 'a) **where**
update-ti-tuple-t $t \equiv \lambda bs.$ if length *bs* = *size-td-tuple* *t* then
update-ti-tuple *t* *bs* else *id*

lemma *update-ti-t-struct-t* [*simp*]: *update-ti-t* (*TypDesc* *algn* *st* *nm*) = *update-ti-struct-t* *st*

apply (*rule* *ext*)
apply (*simp* *add*: *update-ti-t-def* *update-ti-struct-t-def*)
done

lemma *update-ti-update-ti-t*:

length *bs* = *size-td* *s* \implies *update-ti* *s* *bs* *v* = *update-ti-t* *s* *bs* *v*
unfolding *update-ti-t-def* **by** *simp*

field-desc generates the access/update pair for a field, given the field's *type-desc*

definition *field-desc* :: ('a, 'b) *typ-info* \Rightarrow 'a *field-desc* **where**
field-desc $t \equiv (\mid$ *field-access* = *access-ti* *t*,
field-update = *update-ti-t* *t*, *field-sz* = *size-td* *t* $\mid)$

declare *field-desc-def* [*simp* *add*]

definition *field-desc-struct* :: ('a, 'b) *typ-info-struct* \Rightarrow 'a *field-desc* **where**
field-desc-struct $t \equiv (\mid$ *field-access* = *access-ti-struct* *t*,
field-update = *update-ti-struct-t* *t*, *field-sz* = *size-td-struct* *t* $\mid)$

declare *field-desc-struct-def* [*simp* *add*]

definition *field-desc-list* :: ('a, 'b) *typ-info-tuple list* \Rightarrow 'a *field-desc*
where

field-desc-list $t \equiv (\mid$ *field-access* = *access-ti-list* *t*,
field-update = *update-ti-list-t* *t*, *field-sz* = *size-td-list* *t* $\mid)$

declare *field-desc-list-def* [*simp* *add*]

definition *field-desc-tuple* :: ('a, 'b) *typ-info-tuple* \Rightarrow 'a *field-desc*
where

field-desc-tuple $t \equiv (\mid$ *field-access* = *access-ti-tuple* *t*,
field-update = *update-ti-tuple-t* *t*, *field-sz* = *size-td-tuple* *t* $\mid)$

declare *field-desc-tuple-def* [*simp add*]

primrec

map-td :: (nat \Rightarrow nat \Rightarrow 'a \Rightarrow 'b) \Rightarrow ('c \Rightarrow 'd) \Rightarrow ('a, 'c) *typ-desc* \Rightarrow ('b, 'd) *typ-desc* **and**

map-td-struct :: (nat \Rightarrow nat \Rightarrow 'a \Rightarrow 'b) \Rightarrow ('c \Rightarrow 'd) \Rightarrow ('a, 'c) *typ-struct* \Rightarrow ('b, 'd) *typ-struct* **and**

map-td-list :: (nat \Rightarrow nat \Rightarrow 'a \Rightarrow 'b) \Rightarrow ('c \Rightarrow 'd) \Rightarrow ('a, 'c) *typ-tuple list* \Rightarrow ('b, 'd) *typ-tuple list* **and**

map-td-tuple :: (nat \Rightarrow nat \Rightarrow 'a \Rightarrow 'b) \Rightarrow ('c \Rightarrow 'd) \Rightarrow ('a, 'c) *typ-tuple* \Rightarrow ('b, 'd) *typ-tuple*

where

mat0: *map-td* *f g* (*TypDesc* *algn st nm*) = *TypDesc* *algn* (*map-td-struct* *f g st nm*)

| *mat1*: *map-td-struct* *f g* (*TypScalar* *n algn d*) = *TypScalar* *n algn* (*f n algn d*)

| *mat2*: *map-td-struct* *f g* (*TypAggregate* *xs*) = *TypAggregate* (*map-td-list* *f g xs*)

| *mat3*: *map-td-list* *f g* [] = []

| *mat4*: *map-td-list* *f g* (*x#xs*) = *map-td-tuple* *f g x* # *map-td-list* *f g xs*

| *mat5*: *map-td-tuple* *f g* (*DTuple* *t n d*) = *DTuple* (*map-td* *f g t*) *n* (*g d*)

definition *field-norm* :: nat \Rightarrow nat \Rightarrow ('a, 'b) *field-desc-scheme* \Rightarrow (byte list \Rightarrow byte list)

where

field-norm \equiv λn *algn d bs*.

if *length* *bs* = *n* then

field-access *d* (*field-update* *d bs undefined*) (*replicate* *n 0*) else

[]

definition *export-uinfo* :: ('a, 'b) *typ-info* \Rightarrow *typ-uinfo* **where**

export-uinfo *t* \equiv *map-td* *field-norm* (λ -. ()) *t*

primrec

wf-desc :: ('a, 'b) *typ-desc* \Rightarrow bool **and**

wf-desc-struct :: ('a, 'b) *typ-struct* \Rightarrow bool **and**

wf-desc-list :: ('a, 'b) *typ-tuple list* \Rightarrow bool **and**

wf-desc-tuple :: ('a, 'b) *typ-tuple* \Rightarrow bool

where

wfd0: *wf-desc* (*TypDesc* *algn ts n*) = *wf-desc-struct* *ts*

| *wfd1*: *wf-desc-struct* (*TypScalar* *n algn d*) = True

| *wfd2*: *wf-desc-struct* (*TypAggregate* *ts*) = *wf-desc-list* *ts*

| *wfd3*: *wf-desc-list* [] = True

| *wfd4*: $wf\text{-desc-list } (x\#xs) = (wf\text{-desc-tuple } x \wedge \neg dt\text{-snd } x \in dt\text{-snd } \text{' set } xs \wedge wf\text{-desc-list } xs)$

| *wfd5*: $wf\text{-desc-tuple } (DTuple } x } n } d) = wf\text{-desc } x$

primrec

$wf\text{-size-desc} :: ('a, 'b) \text{ typ-desc} \Rightarrow \text{bool}$ **and**
 $wf\text{-size-desc-struct} :: ('a, 'b) \text{ typ-struct} \Rightarrow \text{bool}$ **and**
 $wf\text{-size-desc-list} :: ('a, 'b) \text{ typ-tuple list} \Rightarrow \text{bool}$ **and**
 $wf\text{-size-desc-tuple} :: ('a, 'b) \text{ typ-tuple} \Rightarrow \text{bool}$

where

wfsd0: $wf\text{-size-desc } (TypDesc } algn } ts } n) = wf\text{-size-desc-struct } ts$

| *wfsd1*: $wf\text{-size-desc-struct } (TypScalar } n } algn } d) = (0 < n)$

| *wfsd2*: $wf\text{-size-desc-struct } (TypAggregate } ts) = (ts \neq [] \wedge wf\text{-size-desc-list } ts)$

| *wfsd3*: $wf\text{-size-desc-list } [] = \text{True}$

| *wfsd4*: $wf\text{-size-desc-list } (x\#xs) = (wf\text{-size-desc-tuple } x \wedge wf\text{-size-desc-list } xs)$

| *wfsd5*: $wf\text{-size-desc-tuple } (DTuple } x } n } d) = wf\text{-size-desc } x$

definition

$typ\text{-struct} :: ('a, 'b) \text{ typ-desc} \Rightarrow ('a, 'b) \text{ typ-struct}$

where

$typ\text{-struct } t = (\text{case } t \text{ of } TypDesc } algn } st } sz \Rightarrow st)$

lemma *typ-struct* [*simp*]:

$typ\text{-struct } (TypDesc } algn } st } sz) = st$

by (*simp add: typ-struct-def*)

primrec

$typ\text{-name} :: ('a, 'b) \text{ typ-desc} \Rightarrow \text{typ-name}$

where

$typ\text{-name } (TypDesc } algn } st } nm) = nm$

primrec

$norm\text{-tu} :: \text{typ-uinfo} \Rightarrow \text{normalisor}$ **and**

$norm\text{-tu-struct} :: \text{typ-uinfo-struct} \Rightarrow \text{normalisor}$ **and**

$norm\text{-tu-list} :: \text{typ-uinfo-tuple list} \Rightarrow \text{normalisor}$ **and**

$norm\text{-tu-tuple} :: \text{typ-uinfo-tuple} \Rightarrow \text{normalisor}$

where

tn0: $norm\text{-tu } (TypDesc } algn } st } nm) = norm\text{-tu-struct } st$

| *tn1*: $norm\text{-tu-struct } (TypScalar } n } aln } f) = f$

```

| tn2: norm-tu-struct (TypAggregate xs) = norm-tu-list xs

| tn3: norm-tu-list [] = (λbs. [])
| tn4: norm-tu-list (x#xs) = (λbs.
    norm-tu-tuple x (take (size-td-tuple x) bs) @
    norm-tu-list xs (drop (size-td-tuple x) bs))

| tn5: norm-tu-tuple (DTuple t n d) = norm-tu t

class c-type-name =
  fixes
    typ-name-itself :: 'a itself ⇒ typ-name

class c-type = c-type-name +
  fixes
    typ-info-t :: 'a itself ⇒ 'a xtyp-info
  assumes typ-name-itself-typ-name: typ-name-itself T = typ-name (typ-info-t T)

instance c-type ⊆ type ..

definition (in c-type) typ-uinfo-t :: 'a itself ⇒ typ-uinfo where
  typ-uinfo-t t ≡ export-uinfo (typ-info-t TYPE('a))

definition (in c-type) to-bytes :: 'a ⇒ byte list ⇒ byte list where
  to-bytes v ≡ access-ti (typ-info-t TYPE('a)) v

definition (in c-type) from-bytes :: byte list ⇒ 'a where
  from-bytes bs ≡
    field-update (field-desc (typ-info-t TYPE('a))) bs undefined

type-synonym ('a, 'b) flr = (('a, 'b) typ-desc × nat) option

primrec
  field-lookup :: ('a, 'b) typ-desc ⇒ qualified-field-name ⇒ nat ⇒ ('a, 'b) flr and
  field-lookup-struct :: ('a, 'b) typ-struct ⇒ qualified-field-name ⇒ nat ⇒
    ('a, 'b) flr and
  field-lookup-list :: ('a, 'b) typ-tuple list ⇒ qualified-field-name ⇒ nat ⇒
    ('a, 'b) flr and
  field-lookup-tuple :: ('a, 'b) typ-tuple ⇒ qualified-field-name ⇒ nat ⇒ ('a, 'b) flr
where
  fl0: field-lookup (TypDesc algn st nm) f m =
    (if f=[] then Some (TypDesc algn st nm,m) else field-lookup-struct st f m)

| fl1: field-lookup-struct (TypScalar n algn d) f m = None
| fl2: field-lookup-struct (TypAggregate xs) f m = field-lookup-list xs f m

```

```

| fl3: field-lookup-list [] f m = None
| fl4: field-lookup-list (x#xs) f m = (
  case field-lookup-tuple x f m of
    None => field-lookup-list xs f (m + size-td (dt-fst x)) |
    Some y => Some y)
| fl5: field-lookup-tuple (DTuple t nm d) f m =
  (if nm=hd f & f ≠ [] then field-lookup t (tl f) m else None)

```

lemma *field-lookup-wf-desc-pres*:

```

fixes t::('a, 'b) typ-desc
and st::('a, 'b) typ-struct
and ts::('a, 'b) typ-tuple list
and x::('a, 'b) typ-tuple

```

shows

```

wf-desc t => field-lookup t f n = Some (s, m) => wf-desc s
wf-desc-struct st => field-lookup-struct st f n = Some (s, m) => wf-desc s
wf-desc-list ts => field-lookup-list ts f n = Some (s, m) => wf-desc s
wf-desc-tuple x => field-lookup-tuple x f n = Some (s, m) => wf-desc s
by (induct t and st and ts and x arbitrary: n s m f and n s m f and n s m f
and n s m f)
  (auto split: if-split-asm option.splits)

```

definition *map-td-flr* :: (nat => nat => 'a => 'b) => ('c => 'd) =>
 (('a, 'c) typ-desc × nat) option => ('b, 'd) flr

where

```

map-td-flr f g ≡ case-option None (λ(s,n). Some (map-td f g s,n))

```

definition

import-flr :: (nat => nat => 'b => 'a) => ('d => 'c) => ('a, 'c) flr => (('b, 'd)
 typ-desc × nat) option => bool

where

```

import-flr f g s k ≡ case-option (k=None)
  (λ(s,m). case-option False (λ(t,n). n=m & map-td f g t=s) k )
  s

```

definition

field-offset-untyped :: ('a, 'b) typ-desc => qualified-field-name => nat

where

```

field-offset-untyped t n ≡ snd (the (field-lookup t n 0))

```

definition (in *c-type*)

field-offset :: 'a itself => qualified-field-name => nat

where

```

field-offset t n ≡ field-offset-untyped (typ-uinfo-t TYPE('a)) n

```

definition (in *c-type*)

field-ti :: 'a itself ⇒ qualified-field-name → 'a xtyp-info

where

field-ti t n ≡ case-option None (Some ∘ fst)
(*field-lookup* (*typ-info-t* TYPE('a)) n 0)

definition (in *c-type*)

field-size :: 'a itself ⇒ qualified-field-name ⇒ nat

where

field-size t n ≡ *size-td* (*the* (*field-ti* t n))

definition (in *c-type*)

field-lvalue :: 'a ptr ⇒ qualified-field-name ⇒ addr (&'(->-'))

where

&(p→f) ≡ *ptr-val* (p::'a ptr) + *of-nat* (*field-offset* TYPE('a) f)

definition (in *c-type*)

size-of :: 'a itself ⇒ nat **where**

size-of t ≡ *size-td* (*typ-info-t* TYPE('a))

lemma (in *c-type*) *size-of-fold*: *size-td* (*typ-info-t* TYPE('a)) = *size-of* TYPE('a)

by (*simp* *add*: *size-of-def*)

definition (in *c-type*)

norm-bytes :: 'a itself ⇒ normalisor **where**

norm-bytes t ≡ *norm-tu* (*export-uinfo* (*typ-info-t* t))

definition (in *c-type*) *to-bytes-p* :: 'a ⇒ byte list **where**

to-bytes-p v ≡ *to-bytes* v (*replicate* (*size-of* TYPE('a)) 0)

definition (in *c-type*) *zero* :: 'a **where**

zero ≡ *from-bytes* (*replicate* (*size-of* TYPE('a)) 0)

hide-const (open) *zero* — mandatory qualifier: *c-type-class.zero*

syntax

-zero :: type ⇒ logic ((1ZERO/(1'(-))))

translations

ZERO('a) => CONST *c-type-class.zero* :: ('a)

typed-print-translation ‹

let

val *show-zero-types* = *Attrib.setup-config-bool* @{*binding* *show-zero-types*} (*K* *true*);

fun *zero-tr'* *ctxt* *typ* - =

if *Config.get* *ctxt* *show-zero-types* *then*

Syntax.const @{*syntax-const -zero*} \$ *Syntax-Phases.term-of-ty* *ctxt* *typ*

```

    else
      raise Match;

in [(@{const-syntax c-type-class.zero}, zero-tr')]
end;

primrec
  align-td-wo-align :: ('a, 'b) typ-desc  $\Rightarrow$  nat and
  align-td-wo-align-struct :: ('a, 'b) typ-struct  $\Rightarrow$  nat and
  align-td-wo-align-list :: ('a, 'b) typ-tuple list  $\Rightarrow$  nat and
  align-td-wo-align-tuple :: ('a, 'b) typ-tuple  $\Rightarrow$  nat
where
  al0: align-td-wo-align (TypDesc algn st nm) = align-td-wo-align-struct st

| al1: align-td-wo-align-struct (TypScalar n algn d) = algn
| al2: align-td-wo-align-struct (TypAggregate xs) = align-td-wo-align-list xs

| al3: align-td-wo-align-list [] = 0
| al4: align-td-wo-align-list (x#xs) = max (align-td-wo-align-tuple x) (align-td-wo-align-list xs)

| al5: align-td-wo-align-tuple (DTuple t n d) = align-td-wo-align t

primrec
  align-td :: ('a, 'b) typ-desc  $\Rightarrow$  nat and
  align-td-struct :: ('a, 'b) typ-struct  $\Rightarrow$  nat and
  align-td-list :: ('a, 'b) typ-tuple list  $\Rightarrow$  nat and
  align-td-tuple :: ('a, 'b) typ-tuple  $\Rightarrow$  nat
where
  al0: align-td (TypDesc algn st nm) = algn

| al1: align-td-struct (TypScalar n algn d) = algn
| al2: align-td-struct (TypAggregate xs) = align-td-list xs

| al3: align-td-list [] = 0
| al4: align-td-list (x#xs) = max (align-td-tuple x) (align-td-list xs)

| al5: align-td-tuple (DTuple t n d) = align-td t

primrec
  wf-align :: ('a, 'b) typ-desc  $\Rightarrow$  bool and
  wf-align-struct :: ('a, 'b) typ-struct  $\Rightarrow$  bool and
  wf-align-list :: ('a, 'b) typ-tuple list  $\Rightarrow$  bool and
  wf-align-tuple :: ('a, 'b) typ-tuple  $\Rightarrow$  bool
where
  wfal0: wf-align (TypDesc algn ts n) = (align-td-wo-align-struct ts  $\leq$  algn  $\wedge$ 
    align-td-struct ts  $\leq$  algn  $\wedge$ 
    wf-align-struct ts)

```

| *wfal1*: *wf-align-struct* (*TypScalar* *n* *algn* *d*) = *True*
| *wfal2*: *wf-align-struct* (*TypAggregate* *ts*) = (*wf-align-list* *ts*)

| *wfal3*: *wf-align-list* [] = *True*
| *wfal4*: *wf-align-list* (*x#xs*) =
 (*wf-align-tuple* *x* \wedge *wf-align-list* *xs*)

| *wfal5*: *wf-align-tuple* (*DTuple* *x* *n* *d*) = *wf-align* *x*

definition (in *c-type*) *align-of* :: 'a itself \Rightarrow nat **where**
align-of *t* \equiv $2^{\wedge}(\text{align-td } (\text{typ-info-t } \text{TYPE}('a)))$

lemma *align-td-wo-align-le-align-td*:

fixes *t*::('a,'b) *typ-info* **and**
st::('a,'b) *typ-info-struct* **and**
fs::('a,'b) *typ-info-tuple list* **and**
f::('a,'b) *typ-info-tuple*

shows

wf-align *t* \implies *align-td-wo-align* *t* \leq *align-td* *t*
wf-align-struct *st* \implies *align-td-wo-align-struct* *st* \leq *align-td-struct* *st*
wf-align-list *fs* \implies *align-td-wo-align-list* *fs* \leq *align-td-list* *fs*
wf-align-tuple *f* \implies *align-td-wo-align-tuple* *f* \leq *align-td-tuple* *f*
apply (*induct* *t* **and** *st* **and** *fs* **and** *f* *rule*: *typ-desc-typ-struct-inducts*)
apply *auto*

done

lemma (in *c-type*) *align-td-wo-align-le-align-of*:

assumes *wf*: *wf-align* (*typ-info-t* *TYPE*('a'))
shows $2^{\wedge}(\text{align-td-wo-align } (\text{typ-info-t } \text{TYPE}('a))) \leq \text{align-of } (\text{TYPE}('a))$
using *align-td-wo-align-le-align-td* (1) [OF *wf*]
by (*simp* *add*: *align-of-def*)

definition (in *c-type*)

ptr-add :: 'a *ptr* \Rightarrow int \Rightarrow 'a *ptr* (**infixl** $+_p$ 65)

where

ptr-add (*a* :: 'a *ptr*) *w* \equiv
Ptr (*ptr-val* *a* + *of-int* *w* * *of-nat* (*size-of* (*TYPE*('a'))))

lemma (in *c-type*) *ptr-add-def'*:

ptr-add (*Ptr* *p* :: ('a) *ptr*) *n*
= (*Ptr* (*p* + *of-int* *n* * *of-nat* (*size-of* *TYPE*('a'))))
by (*cases* *p*, *auto* *simp*: *ptr-add-def* *scast-id*)

definition (in *c-type*)

ptr-sub :: 'a *ptr* \Rightarrow 'a *ptr* \Rightarrow *addr-bitsize* *signed word* (**infixl** $-_p$ 65)

where

$ptr\text{-}sub (a :: 'a\ ptr) p \equiv$
 $ucast (ptr\text{-}val\ a - ptr\text{-}val\ p) \text{ div of-nat } (size\text{-}of\ (TYPE('a)))$

definition (in *c-type*) $ptr\text{-}aligned :: 'a\ ptr \Rightarrow bool$ **where**
 $ptr\text{-}aligned\ p \equiv align\text{-}of\ TYPE('a)\ dvd\ unat\ (ptr\text{-}val\ (p :: 'a\ ptr))$

type-synonym $'a\ ptr\text{-}guard = 'a\ ptr \Rightarrow bool$

definition (in *c-type*) $c\text{-}null\text{-}guard :: 'a\ ptr\text{-}guard$ **where**
 $c\text{-}null\text{-}guard \equiv \lambda p. 0 \notin \{ptr\text{-}val\ p..+size\text{-}of\ TYPE('a)\}$

definition (in *c-type*) $c\text{-}guard :: 'a\ ptr\text{-}guard$ **where**
 $c\text{-}guard \equiv \lambda p. ptr\text{-}aligned\ p \wedge c\text{-}null\text{-}guard\ p$

primrec

$td\text{-}set :: ('a, 'b)\ typ\text{-}desc \Rightarrow nat \Rightarrow (('a, 'b)\ typ\text{-}desc \times nat)\ set$ **and**
 $td\text{-}set\text{-}struct :: ('a, 'b)\ typ\text{-}struct \Rightarrow nat \Rightarrow (('a, 'b)\ typ\text{-}desc \times nat)\ set$ **and**
 $td\text{-}set\text{-}list :: ('a, 'b)\ typ\text{-}tuple\ list \Rightarrow nat \Rightarrow (('a, 'b)\ typ\text{-}desc \times nat)\ set$ **and**
 $td\text{-}set\text{-}tuple :: ('a, 'b)\ typ\text{-}tuple \Rightarrow nat \Rightarrow (('a, 'b)\ typ\text{-}desc \times nat)\ set$

where

$ts0: td\text{-}set\ (TypDesc\ algn\ st\ nm)\ m = \{(TypDesc\ algn\ st\ nm, m)\} \cup td\text{-}set\text{-}struct\ st\ m$

| $ts1: td\text{-}set\text{-}struct\ (TypScalar\ n\ algn\ d)\ m = \{\}$
| $ts2: td\text{-}set\text{-}struct\ (TypAggregate\ xs)\ m = td\text{-}set\text{-}list\ xs\ m$

| $ts3: td\text{-}set\text{-}list\ []\ m = \{\}$
| $ts4: td\text{-}set\text{-}list\ (x\#\ xs)\ m = td\text{-}set\text{-}tuple\ x\ m \cup td\text{-}set\text{-}list\ xs\ (m + size\text{-}td\ (dt\text{-}fst\ x))$

| $ts5: td\text{-}set\text{-}tuple\ (DTuple\ t\ nm\ d)\ m = td\text{-}set\ t\ m$

instantiation $typ\text{-}desc :: (type, type)\ ord$
begin

definition

$typ\text{-}tag\text{-}le\text{-}def: s \leq (t :: ('a, 'b)\ typ\text{-}desc) \equiv (\exists n. (s, n) \in td\text{-}set\ t\ 0)$

definition

$typ\text{-}tag\text{-}lt\text{-}def: s < (t :: ('a, 'b)\ typ\text{-}desc) \equiv s \leq t \wedge s \neq t$

instance ..

end

definition

$fd\text{-}cons\text{-}double\text{-}update :: ('a, 'x)\ field\text{-}desc\text{-}scheme \Rightarrow bool$

where

$fd\text{-}cons\text{-}double\text{-}update\ d \equiv$

$(\forall v \text{ bs } \text{bs}'. \text{length } \text{bs} = \text{length } \text{bs}' \longrightarrow \text{field-update } d \text{ bs } (\text{field-update } d \text{ bs}' v) = \text{field-update } d \text{ bs } v)$

definition

$\text{fd-cons-update-access} :: ('a, 'x) \text{field-desc-scheme} \Rightarrow \text{nat} \Rightarrow \text{bool}$

where

$\text{fd-cons-update-access } d \ n \equiv$
 $(\forall v \text{ bs}. \text{length } \text{bs} = n \longrightarrow \text{field-update } d (\text{field-access } d \ v \ \text{bs}) \ v = v)$

definition

$\text{norm-desc} :: ('a, 'x) \text{field-desc-scheme} \Rightarrow \text{nat} \Rightarrow (\text{byte list} \Rightarrow \text{byte list})$

where

$\text{norm-desc } d \ n \equiv \lambda \text{bs}. \text{field-access } d (\text{field-update } d \ \text{bs} \ \text{undefined}) (\text{replicate } n \ 0)$

definition

$\text{fd-cons-length} :: ('a, 'x) \text{field-desc-scheme} \Rightarrow \text{nat} \Rightarrow \text{bool}$

where

$\text{fd-cons-length } d \ n \equiv \forall v \ \text{bs}. \text{length } \text{bs} = n \longrightarrow \text{length } (\text{field-access } d \ v \ \text{bs}) = n$

definition

$\text{fd-cons-access-update} :: ('a, 'x) \text{field-desc-scheme} \Rightarrow \text{nat} \Rightarrow \text{bool}$

where

$\text{fd-cons-access-update } d \ n \equiv \forall \text{bs } \text{bs}' \ v \ v'. \text{length } \text{bs} = n \longrightarrow$
 $\text{length } \text{bs}' = n \longrightarrow$
 $\text{field-access } d (\text{field-update } d \ \text{bs } v) \ \text{bs}' = \text{field-access } d (\text{field-update } d \ \text{bs } v') \ \text{bs}'$

definition

$\text{fd-cons-update-normalise} :: ('a, 'x) \text{field-desc-scheme} \Rightarrow \text{nat} \Rightarrow \text{bool}$

where

$\text{fd-cons-update-normalise } d \ n \equiv$
 $(\forall v \ \text{bs}. \text{length } \text{bs} = n \longrightarrow \text{field-update } d (\text{norm-desc } d \ n \ \text{bs}) \ v = \text{field-update } d \ \text{bs } v)$

definition

$\text{fd-cons-desc} :: ('a, 'x) \text{field-desc-scheme} \Rightarrow \text{nat} \Rightarrow \text{bool}$

where

$\text{fd-cons-desc } d \ n \equiv \text{fd-cons-double-update } d \ \wedge$
 $\text{fd-cons-update-access } d \ n \ \wedge$
 $\text{fd-cons-access-update } d \ n \ \wedge$
 $\text{fd-cons-length } d \ n$

definition

$\text{fd-cons} :: ('a, 'b) \text{typ-info} \Rightarrow \text{bool}$

where

$\text{fd-cons } t \equiv \text{fd-cons-desc } (\text{field-desc } t) (\text{size-td } t)$

definition

$fd-cons-struct :: ('a, 'b) typ-info-struct \Rightarrow bool$
where
 $fd-cons-struct\ t \equiv fd-cons-desc\ (field-desc-struct\ t)\ (size-td-struct\ t)$

definition
 $fd-cons-list :: ('a, 'b) typ-info-tuple\ list \Rightarrow bool$
where

$fd-cons-list\ t \equiv fd-cons-desc\ (field-desc-list\ t)\ (size-td-list\ t)$

definition
 $fd-cons-tuple :: ('a, 'b) typ-info-tuple \Rightarrow bool$
where

$fd-cons-tuple\ t \equiv fd-cons-desc\ (field-desc-tuple\ t)\ (size-td-tuple\ t)$

definition
 $fa-fu-ind :: 'a\ field-desc \Rightarrow 'a\ field-desc \Rightarrow nat \Rightarrow nat \Rightarrow bool$
where

$fa-fu-ind\ d\ d'\ n\ n' \equiv \forall v\ bs\ bs'.\ length\ bs = n \longrightarrow length\ bs' = n' \longrightarrow$
 $field-access\ d\ (field-update\ d'\ bs\ v)\ bs' = field-access\ d\ v\ bs'$

definition
 $wf-fdp :: (('a, 'b) typ-info \times qualified-field-name)\ set \Rightarrow bool$
where

$wf-fdp\ t \equiv \forall x\ m.\ (x, m) \in t \longrightarrow (fd-cons\ x \wedge (\forall y\ n.\ (y, n) \in t \wedge \neg m \leq n \wedge \neg$
 $n \leq m$
 $\longrightarrow fu-commutes\ (field-update\ (field-desc\ x))\ (field-update\ (field-desc\ y)) \wedge$
 $fa-fu-ind\ (field-desc\ x)\ (field-desc\ y)\ (size-td\ y)\ (size-td\ x)))$

lemma $wf-fdp-list$:
 $wf-fdp\ (xs \cup ys) \Longrightarrow wf-fdp\ xs \wedge wf-fdp\ ys$
by (*auto simp: wf-fdp-def*)

primrec
 $wf-fd :: ('a, 'b) typ-info \Rightarrow bool$ **and**
 $wf-fd-struct :: ('a, 'b) typ-info-struct \Rightarrow bool$ **and**
 $wf-fd-list :: ('a, 'b) typ-info-tuple\ list \Rightarrow bool$ **and**
 $wf-fd-tuple :: ('a, 'b) typ-info-tuple \Rightarrow bool$

where
 $wffd0: wf-fd\ (TypDesc\ algn\ ts\ n) = (wf-fd-struct\ ts)$

| $wffd1: wf-fd-struct\ (TypScalar\ n\ algn\ d) = fd-cons-struct\ (TypScalar\ n\ algn$
 $d::('a, 'b) typ-info-struct)$

| $wffd2: wf-fd-struct\ (TypAggregate\ ts) = wf-fd-list\ ts$

| $wffd3: wf-fd-list\ [] = True$

| $wffd4: wf-fd-list\ (x\#\ xs) = (wf-fd-tuple\ x \wedge wf-fd-list\ xs \wedge$
 $fu-commutes\ (update-ti-tuple-t\ x)\ (update-ti-list-t\ xs) \wedge$

$fa\text{-}fu\text{-}ind\ (field\text{-}desc\text{-}tuple\ x)\ (field\text{-}desc\text{-}list\ xs)\ (size\text{-}td\text{-}list\ xs)\ (size\text{-}td\text{-}tuple\ x)\wedge$
 $fa\text{-}fu\text{-}ind\ (field\text{-}desc\text{-}list\ xs)\ (field\text{-}desc\text{-}tuple\ x)\ (size\text{-}td\text{-}tuple\ x)\ (size\text{-}td\text{-}list\ xs))$
 $| wffd5: wf\text{-}fd\text{-}tuple\ (DTuple\ x\ n\ d) = wf\text{-}fd\ x$

definition

$tf\text{-}set :: ('a, 'b)\ typ\text{-}info \Rightarrow (('a, 'b)\ typ\text{-}info \times qualified\text{-}field\text{-}name)\ set$
where
 $tf\text{-}set\ td \equiv \{(s,f) \mid s\ f.\ \exists n.\ field\text{-}lookup\ td\ f\ 0 = Some\ (s,n)\}$

definition

$tf\text{-}set\text{-}struct :: ('a, 'b)\ typ\text{-}info\text{-}struct \Rightarrow (('a, 'b)\ typ\text{-}info \times qualified\text{-}field\text{-}name)\ set$
where
 $tf\text{-}set\text{-}struct\ td \equiv \{(s,f) \mid s\ f.\ \exists n.\ field\text{-}lookup\text{-}struct\ td\ f\ 0 = Some\ (s,n)\}$

definition

$tf\text{-}set\text{-}list :: ('a, 'b)\ typ\text{-}info\text{-}tuple\ list \Rightarrow (('a, 'b)\ typ\text{-}info \times qualified\text{-}field\text{-}name)\ set$
where
 $tf\text{-}set\text{-}list\ td \equiv \{(s,f) \mid s\ f.\ \exists n.\ field\text{-}lookup\text{-}list\ td\ f\ 0 = Some\ (s,n)\}$

definition

$tf\text{-}set\text{-}tuple :: ('a, 'b)\ typ\text{-}info\text{-}tuple \Rightarrow (('a, 'b)\ typ\text{-}info \times qualified\text{-}field\text{-}name)\ set$
where
 $tf\text{-}set\text{-}tuple\ td \equiv \{(s,f) \mid s\ f.\ \exists n.\ field\text{-}lookup\text{-}tuple\ td\ f\ 0 = Some\ (s,n)\}$

record 'a leaf-desc =

$lf\text{-}fd :: 'a\ field\text{-}desc$
 $lf\text{-}sz :: nat$
 $lf\text{-}fn :: qualified\text{-}field\text{-}name$

primrec

$lf\text{-}set :: ('a, 'b)\ typ\text{-}info \Rightarrow qualified\text{-}field\text{-}name \Rightarrow 'a\ leaf\text{-}desc\ set$ **and**
 $lf\text{-}set\text{-}struct :: ('a, 'b)\ typ\text{-}info\text{-}struct \Rightarrow qualified\text{-}field\text{-}name \Rightarrow 'a\ leaf\text{-}desc\ set$
and
 $lf\text{-}set\text{-}list :: ('a, 'b)\ typ\text{-}info\text{-}tuple\ list \Rightarrow qualified\text{-}field\text{-}name \Rightarrow 'a\ leaf\text{-}desc\ set$
and
 $lf\text{-}set\text{-}tuple :: ('a, 'b)\ typ\text{-}info\text{-}tuple \Rightarrow qualified\text{-}field\text{-}name \Rightarrow 'a\ leaf\text{-}desc\ set$
where
 $fds0: lf\text{-}set\ (TypDesc\ algn\ st\ nm)\ fn = lf\text{-}set\text{-}struct\ st\ fn$

$| fds1: lf\text{-}set\text{-}struct\ (TypScalar\ n\ algn\ d)\ fn = \{(\lfloor lf\text{-}fd = d, lf\text{-}sz = n, lf\text{-}fn = fn \rfloor)\}$

| *fds2*: *lf-set-struct* (*TypAggregate xs*) *fn* = *lf-set-list xs fn*

| *fds3*: *lf-set-list* [] *fn* = {}

| *fds4*: *lf-set-list* (*x#xs*) *fn* = *lf-set-tuple x fn* \cup *lf-set-list xs fn*

| *fds5*: *lf-set-tuple* (*DTuple t n d*) *fn* = *lf-set t (fn@[n])*

definition

wf-lf :: 'a *leaf-desc set* \Rightarrow *bool*

where

wf-lf D $\equiv \forall x. x \in D \longrightarrow$ (*fd-cons-desc* (*lf-fd x*) (*lf-sz x*) \wedge ($\forall y. y \in D \longrightarrow$ *lf-fn y* \neq *lf-fn x*
 \longrightarrow *fu-commutes* (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd y*)) \wedge
fa-fu-ind (*lf-fd x*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x*)))

definition

ti-ind :: 'a *leaf-desc set* \Rightarrow 'a *leaf-desc set* \Rightarrow *bool*

where

ti-ind X Y $\equiv \forall x y. x \in X \wedge y \in Y \longrightarrow$ (
fu-commutes (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd y*)) \wedge
fa-fu-ind (*lf-fd x*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x*) \wedge
fa-fu-ind (*lf-fd y*) (*lf-fd x*) (*lf-sz x*) (*lf-sz y*))

definition

t2d :: (('a, 'b) *typ-info* \times *qualified-field-name*) \Rightarrow 'a *leaf-desc*

where

t2d x \equiv (\lfloor *lf-fd* = *field-desc* (*fst x*), *lf-sz* = *size-td* (*fst x*), *lf-fn* = *snd x*)

definition

fd-consistent :: ('a, 'b) *typ-info* \Rightarrow *bool*

where

fd-consistent t $\equiv \forall f s n. \text{field-lookup } t f 0 = \text{Some } (s,n)$
 \longrightarrow *fd-cons s*

class *wf-type* = *c-type* +

assumes *wf-desc* [*simp*]: *wf-desc* (*typ-info-t TYPE('a)*)
assumes *wf-size-desc* [*simp*]: *wf-size-desc* (*typ-info-t TYPE('a)*)
assumes *wf-lf* [*simp*]: *wf-lf* (*lf-set* (*typ-info-t TYPE('a)*) [])

definition

super-update-bs :: *byte list* \Rightarrow *byte list* \Rightarrow *nat* \Rightarrow *byte list*

where

super-update-bs v bs n \equiv *take n bs* @ *v* @
drop (n + length v) bs

definition

disj-fn :: *qualified-field-name* \Rightarrow *qualified-field-name* \Rightarrow *bool*
where
disj-fn *s t* $\equiv \neg s \leq t \wedge \neg t \leq s$

definition

fs-path :: *qualified-field-name list* \Rightarrow *qualified-field-name set*
where
fs-path *xs* $\equiv \{x. \exists k. k \in \text{set } xs \wedge x \leq k\} \cup \{x. \exists k. k \in \text{set } xs \wedge k \leq x\}$

definition

field-names :: ('a, 'b) *typ-desc* \Rightarrow *qualified-field-name set*
where
field-names *t* $\equiv \{f. \text{field-lookup } t f 0 \neq \text{None}\}$

definition

align-field :: ('a, 'b) *typ-desc* \Rightarrow *bool*
where
align-field *ti* $\equiv \forall f s n. \text{field-lookup } ti f 0 = \text{Some } (s,n) \longrightarrow$
 $2^{\wedge}(\text{align-td } s) \text{ dvd } n$

class *mem-type-sans-size* = *wf-type* +

assumes *upd*:

length *bs* = *size-of* *TYPE*('a) \longrightarrow
update-ti-t (*typ-info-t* *TYPE*('a)) *bs v*
= *update-ti-t* (*typ-info-t* *TYPE*('a)) *bs w*

assumes *align-size-of*: *align-of* (*TYPE*('a)) *dvd size-of* *TYPE*('a)

assumes *align-field*: *align-field* (*typ-info-t* *TYPE*('a))

assumes *wf-align*: *wf-align* (*typ-info-t* *TYPE*('a))

class *mem-type* = *mem-type-sans-size* +

assumes *max-size*: *size-of* (*TYPE*('a)) < *addr-card*

primrec

aggregate :: ('a, 'b) *typ-desc* \Rightarrow *bool* **and**
aggregate-struct :: ('a, 'b) *typ-struct* \Rightarrow *bool*

where

aggregate (*TypDesc* *algn st tn*) = *aggregate-struct st*

| *aggregate-struct* (*TypScalar* *n algn d*) = *False*

| *aggregate-struct* (*TypAggregate* *ts*) = *True*

class *simple-mem-type* = *mem-type* +

assumes *simple-tag*: \neg *aggregate* (*typ-info-t* *TYPE*('a))

definition

field-of :: *addr* \Rightarrow ('a, 'b) *typ-desc* \Rightarrow ('a, 'b) *typ-desc* \Rightarrow *bool*
where

$field-of\ q\ s\ t \equiv (s, unat\ q) \in td-set\ t\ 0$

definition (in *c-type*)

$field-of-t :: 'a\ ptr \Rightarrow 'b::c-type\ ptr \Rightarrow bool$

where

$field-of-t\ p\ q \equiv field-of\ (ptr-val\ p - ptr-val\ q)\ (typ-uinfo-t\ TYPE('a))$
 $(typ-uinfo-t\ TYPE('b))$

definition (in *c-type*)

$h-val :: heap-mem \Rightarrow 'a\ ptr \Rightarrow 'a$

where

$h-val\ h \equiv \lambda p. from-bytes\ (heap-list\ h\ (size-of\ TYPE\ ('a))$
 $(ptr-val\ (p::'a\ ptr)))$

primrec

$heap-update-list :: addr \Rightarrow byte\ list \Rightarrow heap-mem \Rightarrow heap-mem$

where

$heap-update-list-base: heap-update-list\ p\ []\ h = h$

| $heap-update-list-rec:$

$heap-update-list\ p\ (x\#\!xs)\ h = heap-update-list\ (p + 1)\ xs\ (h(p:=\ x))$

type-synonym $'a\ typ-heap-g = 'a\ ptr \Rightarrow 'a$

definition (in *c-type*)

$lift :: heap-mem \Rightarrow 'a\ typ-heap-g$

where

$lift\ h \equiv h-val\ h$

definition (in *c-type*)

$heap-update :: 'a\ ptr \Rightarrow 'a \Rightarrow heap-mem \Rightarrow heap-mem$

where

$heap-update\ p\ v\ h \equiv heap-update-list\ (ptr-val\ p)\ (to-bytes\ v\ (heap-list\ h\ (size-of$
 $TYPE('a))\ (ptr-val\ p)))\ h$

definition (in *c-type*)

$heap-update-padding :: 'a\ ptr \Rightarrow 'a \Rightarrow byte\ list \Rightarrow heap-mem \Rightarrow heap-mem$ **where**
 $heap-update-padding\ p\ v\ bs\ h =$

$heap-update-list\ (ptr-val\ p)\ (to-bytes\ v\ bs)\ h$

lemma (in *c-type*) *heap-update-heap-update-padding-conv:*

$heap-update\ p\ v\ h = heap-update-padding\ p\ v\ (heap-list\ h\ (size-of\ TYPE('a))$
 $(ptr-val\ p))\ h$

by (*simp* *add: heap-update-def heap-update-padding-def*)

definition (in *c-type*) *heap-upd* **where**

$heap-upd\ p\ f\ s = heap-update\ p\ (f\ (h-val\ s\ p))\ s$

definition *heap-upd-list* :: $nat \Rightarrow addr \Rightarrow (byte\ list \Rightarrow byte\ list) \Rightarrow heap-mem \Rightarrow$

```

heap-mem where
  heap-upd-list n p f h = heap-update-list p (f (heap-list h n p)) h

fun
  fold-td' :: (typ-name ⇒ ('a × field-name) list ⇒ 'a) × ('a, 'b) typ-desc ⇒ 'a
where
  fot0: fold-td' (f, TypDesc algn st nm) = (case st of
    TypScalar n algn d ⇒ d |
    TypAggregate ts ⇒ f nm (map (λx. case x of DTuple t n d ⇒ (fold-td'
(f,t),n)) ts))

definition
  fold-td :: (typ-name ⇒ ('a × field-name) list ⇒ 'a) ⇒ ('a, 'b) typ-desc ⇒ 'a
where
  fold-td ≡ λf t. fold-td' (f,t)

declare fold-td-def [simp]

definition
  fold-td-struct :: typ-name ⇒ (typ-name ⇒ ('a × field-name) list ⇒ 'a) ⇒ ('a,
'b) typ-struct ⇒ 'a
where
  fold-td-struct tn f st ≡ (case st of
    TypScalar n algn d ⇒ d |
    TypAggregate ts ⇒ f tn (map (λx. case x of DTuple t n d ⇒ (fold-td'
(f,t),n)) ts))

declare fold-td-struct-def [simp]

definition
  fold-td-list :: typ-name ⇒ (typ-name ⇒ ('a × field-name) list ⇒ 'a) ⇒ ('a, 'b)
typ-tuple list ⇒ 'a
where
  fold-td-list tn f ts ≡ f tn (map (λx. case x of DTuple t n d ⇒ (fold-td' (f,t),n))
ts)

declare fold-td-list-def [simp]

definition
  fold-td-tuple :: (typ-name ⇒ ('a × field-name) list ⇒ 'a) ⇒ ('a, 'b) typ-tuple ⇒
'a
where
  fold-td-tuple f x ≡ (case x of DTuple t n d ⇒ fold-td' (f,t))

declare fold-td-tuple-def [simp]

fun
  map-td' :: ((nat ⇒ nat ⇒ 'a ⇒ 'b) × ('c ⇒ 'd)) × ('a, 'c) typ-desc ⇒ ('b, 'd)

```

typ-desc

where

$map\text{-}td' ((f,g), TypDesc\ alg\ n\ st\ nm) = (TypDesc\ alg\ n\ (case\ st\ of$
 $TypScalar\ n\ alg\ d \Rightarrow TypScalar\ n\ alg\ (f\ n\ alg\ d) |$
 $TypAggregate\ ts \Rightarrow TypAggregate\ (map\ (\lambda x. case\ x\ of\ DTuple\ t\ n\ d \Rightarrow$
 $DTuple\ (map\text{-}td' ((f,g),t))\ n\ (g\ d))\ ts))\ nm)$

definition

$tnSum :: typ\text{-}name \Rightarrow (nat \times field\text{-}name)\ list \Rightarrow nat$

where

$tnSum \equiv \lambda tn\ ts. foldr\ ((+) \circ fst)\ ts\ 0$

definition

$tnMax :: typ\text{-}name \Rightarrow (nat \times field\text{-}name)\ list \Rightarrow nat$

where

$tnMax \equiv \lambda tn\ ts. foldr\ (\lambda x\ y. max\ (fst\ x)\ y)\ ts\ 0$

definition

$wfd :: typ\text{-}name \Rightarrow (bool \times field\text{-}name)\ list \Rightarrow bool$

where

$wfd \equiv \lambda tn\ ts. distinct\ (map\ snd\ ts) \wedge foldr\ (\wedge)\ (map\ fst\ ts)\ True$

definition

$wfsd :: typ\text{-}name \Rightarrow (bool \times field\text{-}name)\ list \Rightarrow bool$

where

$wfsd \equiv \lambda tn\ ts. ts \neq [] \wedge foldr\ (\wedge)\ (map\ fst\ ts)\ True$

definition $component\text{-}desc\text{-}tuple\ t \equiv case\ t\ of\ (DTuple\ t\ n\ d) \Rightarrow d$

lemma $component\text{-}desc\text{-}tuple\text{-}simp\ [simp]: component\text{-}desc\text{-}tuple\ (DTuple\ t\ n\ d) = d$

by ($simp\ add: component\text{-}desc\text{-}tuple\text{-}def$)

primrec $split\text{-}list :: nat\ list \Rightarrow 'a\ list \Rightarrow 'a\ list\ list$ **where**

$split\text{-}list\ []\ bs = [bs]$

| $split\text{-}list\ (n\#\ ns)\ bs = take\ n\ bs \# split\text{-}list\ ns\ (drop\ n\ bs)$

lemma $concat\text{-}split\ [simp]: concat\ (split\text{-}list\ ns\ bs) = bs$

by ($induct\ ns\ arbitrary: bs$) $auto$

primrec $split\text{-}map' :: ('b \Rightarrow nat) \Rightarrow ('b \Rightarrow 'a\ list \Rightarrow 'c) \Rightarrow 'b\ list \Rightarrow 'a\ list \Rightarrow 'c\ list$ **where**

$split\text{-}map'\ n\ f\ []\ bs = []$

| $split\text{-}map'\ n\ f\ (x\#\ xs)\ bs = f\ x\ (take\ (n\ x)\ bs) \# split\text{-}map'\ n\ f\ xs\ (drop\ (n\ x)\ bs)$

lemma $length\text{-}split\text{-}map' [simp]: length\ (split\text{-}map'\ n\ f\ xs\ bs) = length\ xs$

by (*induct xs arbitrary: bs*) *auto*

lemma *split-map' n f xs bs = map ($\lambda(f',bs'). f' bs'$) (zip (map f xs) (split-list (map n xs) bs))*

by (*induct xs arbitrary: bs*) *auto*

primrec *split-map:: ('b \Rightarrow 'a list \Rightarrow 'c list) \Rightarrow 'b list \Rightarrow 'a list \Rightarrow 'c list list where*
split-map f [] bs = []
| split-map f (x#xs) bs = f x bs # split-map f xs (drop (length (f x bs)) bs)

lemma *length-split-map [simp]: length (split-map f xs bs) = length xs*

by (*induct xs arbitrary: bs*) *auto*

primrec *split-fold:: ('b \Rightarrow 'a list \Rightarrow 'd \Rightarrow ('d \times 'a list)) \Rightarrow 'b list \Rightarrow 'a list \Rightarrow 'd \Rightarrow ('d \times 'a list) where*
split-fold f [] bs s = (s, bs)
| split-fold f (x#xs) bs s = (let (s', bs') = f x bs s in split-fold f xs bs' s')

definition *apply-field-update:: 'a field-desc \Rightarrow byte list \Rightarrow 'a \Rightarrow ('a \times byte list) where*

apply-field-update d bs s \equiv (field-update d bs s, drop (field-sz d) bs)

definition *apply-field-updates :: 'a field-desc list \Rightarrow byte list \Rightarrow 'a \Rightarrow ('a \times byte list) where*

apply-field-updates \equiv split-fold apply-field-update

lemma *apply-field-updates-Nil [simp]: apply-field-updates [] bs s = (s, bs)*

by (*simp add: apply-field-updates-def*)

lemma *apply-field-updates-Cons [simp]: apply-field-updates (d#ds) bs s =*

apply-field-updates ds (drop (field-sz d) bs) (field-update d bs s)

by (*simp add: apply-field-updates-def apply-field-update-def*)

definition *component-access-tuple v t bs \equiv case t of (DTuple t n d) \Rightarrow field-access d v bs*

lemma *component-access-tuple-simp [simp]: component-access-tuple v (DTuple t n d) = field-access d v*

by (*auto simp add: component-access-tuple-def*)

definition *component-access-struct st v bs =*

(case st of TypScalar - - d \Rightarrow field-access d v bs

| TypAggregate fs \Rightarrow concat (split-map (component-access-tuple v) fs bs))

lemma *component-access-struct-scalar [simp]:*

component-access-struct (TypScalar sz align d) = field-access d

```

apply (rule ext)
apply (rule ext)
apply (simp add: component-access-struct-def)
done

```

lemma *component-access-struct-aggregate* [simp]:
component-access-struct (TypAggregate fs) v bs = concat (split-map (component-access-tuple v) fs bs)
by (simp add: component-access-struct-def)

lemma *component-access-struct-aggregate-nil* [simp]:
component-access-struct (TypAggregate []) = (λv bs. [])
apply (rule ext)
apply (rule ext)
apply simp
done

definition *component-update-tuple* t bs ≡ *case t of (DTuple t n d) ⇒ field-update d bs*

lemma *component-update-tuple-simp* [simp]: *component-update-tuple (DTuple t n d) = field-update d*
by (auto simp add: component-update-tuple-def)

definition *component-update-struct* st bs v =
(case st of TypScalar - - d ⇒ field-update d bs v
| TypAggregate fs ⇒ fst (apply-field-updates (map dt-trd fs) bs v))

lemma *component-update-struct-scalar* [simp]:
component-update-struct (TypScalar sz align d) = field-update d
apply (rule ext)
apply (rule ext)
apply (simp add: component-update-struct-def)
done

lemma *component-update-struct-aggregate* [simp]:
component-update-struct (TypAggregate fs) bs v = fst (apply-field-updates (map dt-trd fs) bs v)
by (simp add: component-update-struct-def)

lemma *component-update-struct-aggregate-nil* [simp]:
component-update-struct (TypAggregate []) = (λbs. id)
apply (rule ext)
apply (rule ext)
apply simp
done

definition *component-sz-struct* *st* =
 (case *st* of *TypScalar* - - *d* ⇒ *field-sz* *d*
 | *TypAggregate* *fs* ⇒ *foldl* (+) 0 (map (*field-sz* o *dt-trd*) *fs*))

lemma *component-sz-struct-scalar* [*simp*]:
component-sz-struct (*TypScalar* *sz align d*) = *field-sz* *d*
 by (*simp* add: *component-sz-struct-def*)

lemma *component-sz-struct-aggregate* [*simp*]:
component-sz-struct (*TypAggregate* *fs*) = *foldl* (+) 0 (map (*field-sz* o *dt-trd*) *fs*)
 by (*simp* add: *component-sz-struct-def*)

definition *component-desc-struct* *st* =
 (|*field-access* = *component-access-struct* *st*,
 |*field-update* = *component-update-struct* *st*,
 |*field-sz* = *component-sz-struct* *st*)

lemma *component-desc-struct-simps* [*simp*]:
field-access (*component-desc-struct* *st*) = *component-access-struct* *st*
field-update (*component-desc-struct* *st*) = *component-update-struct* *st*
field-sz (*component-desc-struct* *st*) = *component-sz-struct* *st*
component-desc-struct (*TypScalar* *sz align d*) = *d*
 by (*auto simp* add: *component-desc-struct-def*)

definition *component-desc* *t* ≡ case *t* of *TypDesc* *algn st n* ⇒ *component-desc-struct* *st*

lemma *component-desc-simps* [*simp*]: *component-desc* (*TypDesc* *algn st n*) = *component-desc-struct* *st*
 by (*simp* add: *component-desc-def*)

definition (in *c-type*) *xto-bytes* :: 'a ⇒ byte list ⇒ byte list **where**
xto-bytes ≡ *field-access* (*component-desc* (*typ-info-t* *TYPE*('a)))

definition (in *c-type*) *xfrom-bytes* :: byte list ⇒ 'a **where**
xfrom-bytes *bs* ≡ *field-update* (*component-desc* (*typ-info-t* *TYPE*('a))) *bs* *undefined*

primrec
oplevel-field-descs :: 'a *xtyp-info* ⇒ 'a *field-desc* list **and**
oplevel-field-descs-struct :: 'a *xtyp-info-struct* ⇒ 'a *field-desc* list **and**
oplevel-field-descs-list :: 'a *xtyp-info-tuple* list ⇒ 'a *field-desc* list **and**
oplevel-field-descs-tuple :: 'a *xtyp-info-tuple* ⇒ 'a *field-desc* list

where

tfd0: *toplevel-field-descs* (*TypDesc* *algn* *st* *nm*) = *toplevel-field-descs-struct* *st*

| *tfd1*: *toplevel-field-descs-struct* (*TypScalar* *n* *algn* *d*) = [*d*]

| *tfd2*: *toplevel-field-descs-struct* (*TypAggregate* *xs*) = *toplevel-field-descs-list* *xs*

| *tfd3*: *toplevel-field-descs-list* [] = []

| *tfd4*: *toplevel-field-descs-list* (*x#xs*) = *toplevel-field-descs-tuple* *x* @ *toplevel-field-descs-list* *xs*

| *tfd5*: *toplevel-field-descs-tuple* (*DTuple* *t* *nm* *d*) = [*d*]

locale *padding-base* =

fixes *acc*::'a ⇒ *byte list* ⇒ *byte list*

fixes *upd*:: *byte list* ⇒ 'a ⇒ 'a

fixes *sz*::*nat*

begin

definition *eq-padding*::*byte list* ⇒ *byte list* ⇒ *bool* **where**

eq-padding *bs* *bs'* ≡ *length* *bs* = *sz* ∧ *length* *bs'* = *sz* ∧ (∀ *v*. *acc* *v* *bs* = *acc* *v* *bs'*)

definition *eq-upto-padding*::*byte list* ⇒ *byte list* ⇒ *bool* **where**

eq-upto-padding *bs* *bs'* ≡ *length* *bs* = *sz* ∧ *length* *bs'* = *sz* ∧ (∀ *v*. *upd* *bs* *v* = *upd* *bs'* *v*)

definition *is-padding-byte*::*nat* ⇒ *bool* **where**

is-padding-byte *i* ≡ *i* < *sz* ∧ (∀ *v* *bs*. *length* *bs* = *sz* →
(*acc* *v* *bs* ! *i* = *bs* ! *i*) ∧
(∀ *b*. *upd* *bs* *v* = *upd* (*bs*[*i*:=*b*]) *v*))

definition *is-value-byte*::*nat* ⇒ *bool* **where**

is-value-byte *i* ≡ *i* < *sz* ∧ (∀ *v* *bs*. *length* *bs* = *sz* →
(∀ *bs'*. *length* *bs'* = *sz* → (*acc* (*upd* *bs* *v*) *bs'* ! *i*) = *bs* ! *i*) ∧
(∀ *b*. *acc* *v* *bs* = *acc* *v* (*bs*[*i*:=*b*])))

lemma *is-padding-byteI*:

assumes *i* < *sz*

assumes ∧ *v* *bs*. *i* < *sz* ⇒ *length* *bs* = *sz* ⇒ *acc* *v* *bs* ! *i* = *bs* ! *i*

assumes ∧ *v* *bs* *b*. *i* < *sz* ⇒ *length* *bs* = *sz* ⇒ *upd* *bs* *v* = *upd* (*bs*[*i*:=*b*]) *v*

shows *is-padding-byte* *i*

using *assms*

by (*simp* *add*: *is-padding-byte-def*)

lemma *is-value-byteI*:

assumes *i* < *sz*

assumes ∧ *v* *bs* *bs'*. *i* < *sz* ⇒ *length* *bs* = *sz* ⇒ *length* *bs'* = *sz* ⇒

(*acc* (*upd* *bs* *v*) *bs'* ! *i*) = *bs* ! *i*

assumes ∧ *v* *bs* *b*. *i* < *sz* ⇒ *length* *bs* = *sz* ⇒ *acc* *v* *bs* = *acc* *v* (*bs*[*i*:=*b*])

shows *is-value-byte* i
using *assms*
by (*simp add: is-value-byte-def*)

lemma *is-padding-byte-acc-id*:
 $is-padding-byte\ i \implies length\ bs = sz \implies (acc\ v\ bs\ !\ i) = bs\ !\ i$
by (*simp add: is-padding-byte-def*)

lemma *is-value-byte-acc-upd-cancel*:
 $is-value-byte\ i \implies length\ bs = sz \implies length\ bs' = sz \implies$
 $(acc\ (upd\ bs\ v)\ bs'\ !\ i) = bs\ !\ i$
by (*simp add: is-value-byte-def*)

lemma *is-padding-byte-eq-upto-padding*: $is-padding-byte\ i \implies length\ bs = sz \implies$
 $eq-upto-padding\ bs\ (bs[i:=b])$
by (*simp add: eq-upto-padding-def is-padding-byte-def*)

lemma *is-value-byte-eq-padding*: $is-value-byte\ i \implies length\ bs = sz \implies eq-padding$
 $bs\ (bs[i:=b])$
by (*simp add: eq-padding-def is-value-byte-def*)

lemma *is-padding-byte-acc-neq*:
 $is-padding-byte\ i \implies b \neq bs!i \implies length\ bs = sz \implies acc\ v\ bs \neq acc\ v\ (bs[i:=b])$
by (*metis is-padding-byte-def length-list-update nth-list-update-eq*)

lemma *is-padding-byte-acc-eq*:
 $is-padding-byte\ i \implies length\ bs = sz \implies$
 $acc\ v\ bs\ !\ i = bs\ !\ i$
by (*auto simp add: is-padding-byte-def*)

lemma *is-padding-byte-upd-eq*:
 $is-padding-byte\ i \implies length\ bs = sz \implies upd\ bs\ v = upd\ (bs[i:=b])\ v$
by (*auto simp add: is-padding-byte-def*)

lemma *is-padding-byte-not-eq-padding*: $is-padding-byte\ i \implies b \neq bs!i \implies length$
 $bs = sz \implies \neg eq-padding\ bs\ (bs[i:=b])$
by (*simp add: padding-base.eq-padding-def padding-base.is-padding-byte-acc-neq*)

lemma *is-value-byte-upd-neq*:
 $is-value-byte\ i \implies b \neq bs!i \implies length\ bs = sz \implies upd\ bs\ v \neq upd\ (bs[i:=b])\ v$
by (*metis length-list-update nth-list-update-eq is-value-byte-def*)

lemma *is-value-byte-update-depends*:
assumes *is-value*: $is-value-byte\ i$
assumes *lbs*: $length\ bs = sz$
shows $\exists b. upd\ (bs[i := b])\ v \neq upd\ bs\ v$
proof –
obtain b **where** $b \neq bs!i$

by (*metis len8 len-gt-0 less-le word-power-less-1*)
from *is-value-byte-upd-neq* [*OF is-value this lbs, of v*] **show** *?thesis* **by** *metis*
qed

lemma *is-value-byte-acc-eq*:
is-value-byte i \implies *length bs = sz* \implies *acc v bs = acc v (bs[i:=b])*
by (*auto simp add: is-value-byte-def*)

lemma *is-value-byte-not-eq-upto-padding*:
is-value-byte i \implies *b \neq bs!i* \implies *length bs = sz* \implies \neg *eq-upto-padding bs (bs[i:=b])*
by (*simp add: padding-base.eq-upto-padding-def padding-base.is-value-byte-upd-neq*)

lemma *eq-paddingI*[*intro?*]:
assumes *length bs = sz*
assumes *length bs' = sz*
assumes $\bigwedge v. acc\ v\ bs = acc\ v\ bs'$
shows *eq-padding bs bs'*
using *assms* **by** (*auto simp add: eq-padding-def*)

lemma *eq-upto-paddingI*[*intro?*]:
assumes *length bs = sz*
assumes *length bs' = sz*
assumes $\bigwedge v. upd\ bs\ v = upd\ bs'\ v$
shows *eq-upto-padding bs bs'*
using *assms* **by** (*auto simp add: eq-upto-padding-def*)

lemma *eq-padding-length1*: *eq-padding bs bs'* \implies *length bs = sz*
by (*simp add: eq-padding-def*)

lemma *eq-padding-length2*: *eq-padding bs bs'* \implies *length bs' = sz*
by (*simp add: eq-padding-def*)

lemma *eq-upto-padding-length1*: *eq-upto-padding bs bs'* \implies *length bs = sz*
by (*simp add: eq-upto-padding-def*)

lemma *eq-upto-padding-length2*: *eq-upto-padding bs bs'* \implies *length bs' = sz*
by (*simp add: eq-upto-padding-def*)

lemma *eq-padding-length-eq*: *eq-padding bs bs'* \implies *length bs = length bs'*
by (*simp add: eq-padding-def*)

lemma *eq-upto-padding-length-eq*: *eq-upto-padding bs bs'* \implies *length bs = length bs'*
by (*simp add: eq-upto-padding-def*)

lemma *eq-padding-acc*: *eq-padding bs bs'* \implies *acc v bs = acc v bs'*
by (*simp add: eq-padding-def*)

lemma *eq-upto-padding-upd*: $eq\text{-}upto\text{-}padding\ bs\ bs' \implies upd\ bs\ v = upd\ bs'\ v$
by (*simp add: eq-upto-padding-def*)

lemma *eq-padding-refl[simp]*: $length\ bs = sz \implies eq\text{-}padding\ bs\ bs$
by (*simp add: eq-padding-def*)

lemma *eq-upto-padding-refl[simp]*: $length\ bs = sz \implies eq\text{-}upto\text{-}padding\ bs\ bs$
by (*simp add: eq-upto-padding-def*)

lemma *eq-padding-sym*: $eq\text{-}padding\ bs\ bs' \longleftrightarrow eq\text{-}padding\ bs'\ bs$
by (*auto simp add: eq-padding-def*)

lemma *eq-padding-symp*: *symp eq-padding*
by (*simp add: symp-def eq-padding-sym*)

lemma *eq-upto-padding-sym*: $eq\text{-}upto\text{-}padding\ bs\ bs' \longleftrightarrow eq\text{-}upto\text{-}padding\ bs'\ bs$
by (*auto simp add: eq-upto-padding-def*)

lemma *eq-upto-padding-symp*: *symp eq-upto-padding*
by (*simp add: symp-def eq-upto-padding-sym*)

lemma *eq-padding-trans*: $eq\text{-}padding\ bs1\ bs2 \implies eq\text{-}padding\ bs2\ bs3 \implies eq\text{-}padding\ bs1\ bs3$
by (*auto simp add: eq-padding-def*)

lemma *eq-padding-transp*: *transp eq-padding*
unfolding *transp-def*
by (*auto intro: eq-padding-trans*)

lemma *eq-upto-padding-trans*: $eq\text{-}upto\text{-}padding\ bs1\ bs2 \implies eq\text{-}upto\text{-}padding\ bs2\ bs3 \implies eq\text{-}upto\text{-}padding\ bs1\ bs3$
by (*auto simp add: eq-upto-padding-def*)

lemma *eq-upto-padding-transp*: *transp eq-upto-padding*
unfolding *transp-def*
by (*auto intro: eq-upto-padding-trans*)

lemma *eq-padding-to-padding-byte-eq*: $eq\text{-}padding\ bs\ bs' \implies length\ bs = sz \wedge length\ bs' = sz \wedge (\forall i. is\text{-}padding\text{-}byte\ i \longrightarrow bs\ !\ i = bs'\ !\ i)$
by (*metis is-padding-byte-acc-id padding-base.eq-padding-def*)

lemma *eq-upto-padding-to-value-byte-eq*: $eq\text{-}upto\text{-}padding\ bs\ bs' \implies length\ bs = sz \wedge length\ bs' = sz \wedge (\forall i. is\text{-}value\text{-}byte\ i \longrightarrow bs\ !\ i = bs'\ !\ i)$
by (*metis (no-types, opaque-lifting) is-value-byte-acc-upd-cancel local.eq-upto-padding-def*)

end

locale *complement-padding* = *padding-base* +
assumes *complement*: $i < sz \implies$
is-padding-byte $i \neq$ *is-value-byte* i
begin

lemma *eq-padding-eq-upto-padding-complement*:

assumes *lbs*: $length\ bs = sz$
assumes *i-bound*: $i < length\ bs$
assumes *neq*: $b \neq bs!i$
shows *eq-padding* $bs\ (bs[i := b]) \neq$ *eq-upto-padding* $bs\ (bs[i := b])$
by (*metis* (*no-types*, *opaque-lifting*) *complement* *i-bound* *lbs* *length-list-update* *neq*
padding-base.eq-paddingI
padding-base.eq-upto-padding-def *padding-base.is-padding-byte-not-eq-padding*
padding-base.is-padding-byte-upd-eq *padding-base.is-value-byte-acc-eq* *padding-base.is-value-byte-not-eq-upto-p*)

lemma *eq-padding-padding-byte-id*:

assumes *eq-padding*: *eq-padding* $bs\ bs'$
assumes *is-padding*: *is-padding-byte* i
shows $bs!i = bs'!i$
by (*metis* *eq-padding* *is-padding* *is-padding-byte-acc-id* *padding-base.eq-padding-def*)

lemma *eq-upto-padding-value-byte-id*:

assumes *eq-padding*: *eq-upto-padding* $bs\ bs'$
assumes *is-value*: *is-value-byte* i
shows $bs!i = bs'!i$
by (*metis* (*no-types*, *opaque-lifting*) *eq-padding* *eq-upto-padding-length1*
eq-upto-padding-length2 *eq-upto-padding-upd* *is-value* *is-value-byte-acc-upd-cancel*)

lemma *eq-upto-padding-neq-is-padding-byte*:

assumes *eq-upto-padding*: *eq-upto-padding* $bs\ bs'$
assumes *i-bound*: $i < length\ bs$
assumes *neq*: $bs!i \neq bs'!i$
shows *is-padding-byte* i
by (*metis* *eq-upto-padding* *eq-upto-padding-value-byte-id* *i-bound* *neq*
complement-padding.complement *complement-padding-axioms* *padding-base.eq-upto-padding-length2*
padding-base.eq-upto-padding-length-eq)

lemma *eq-padding-neq-is-value-byte*:

assumes *eq-upto-padding*: *eq-padding* $bs\ bs'$
assumes *i-bound*: $i < length\ bs$
assumes *neq*: $bs!i \neq bs'!i$
shows *is-value-byte* i
by (*metis* *complement* *eq-padding-padding-byte-id* *eq-upto-padding* *i-bound*
neq *padding-base.eq-padding-length2* *padding-base.eq-padding-length-eq*)


```

lemma padding-eq-complement:
  assumes eq-padding: eq-padding bs bs'
  assumes eq-upto-padding: eq-upto-padding bs bs'
  shows bs = bs'
  by (meson eq-padding eq-padding-padding-byte-id eq-upto-padding eq-upto-padding-length-eq
    eq-upto-padding-neq-is-padding-byte nth-equalityI)

```

end

```

locale padding-lense = complement-padding +
  assumes double-update: upd bs (upd bs' s) = upd bs s
  assumes access-update: acc (upd bs s) bs' = acc (upd bs s') bs'
  assumes update-access: upd (acc s bs) s = s
  assumes access-size: acc s (take sz bs) = acc s bs
  assumes access-result-size: length (acc s bs) = sz
  assumes update-size: upd (take sz bs) = upd bs
begin

```

```

lemma update-size-ext: upd (take sz bs) s = upd bs s
  by (simp add: update-size)

```

```

lemma update-access-append: upd ((acc s bs)@bs') s = s (is ?lhs = ?rhs)

```

```

proof –
  have ?lhs = upd (take sz ((acc s bs)@bs')) s
    by (rule update-size-ext [symmetric])
  also
  from access-result-size
  have take sz ((acc s bs)@bs') = acc s bs
    by auto
  also note update-access
  finally show ?thesis .

```

qed

```

lemma field-access-eq-padding1: length bs = sz  $\implies$  eq-padding (acc v bs) bs
  apply (rule eq-paddingI)
  apply (simp add: access-result-size)
  apply simp
  by (smt (verit) complement is-padding-byte-acc-id is-value-byte-acc-upd-cancel
    nth-equalityI access-result-size update-access padding-lense-axioms)

```

```

lemma field-access-eq-padding2: length bs = sz  $\implies$  eq-padding (acc v2 bs) (acc v2
  bs)
  apply (rule eq-paddingI)
  apply (simp-all add: access-result-size)
  done

```

lemma *field-access-eq-upto-padding*: $\text{length } bs = sz \implies \text{length } bs = sz \implies$
eq-upto-padding (*acc v bs*) (*acc v bs'*)
apply (*rule eq-upto-paddingI*)
apply (*simp add: access-result-size*)
apply (*simp add: access-result-size*)
by (*metis access-update double-update update-access*)

lemma *padding-byte-to-eq-padding-eq*:
eq-padding *bs bs'*
if
length *bs* = *sz*
length *bs'* = *sz*
 $(\forall i. \text{is-padding-byte } i \longrightarrow bs ! i = bs' ! i)$
using *that(1,2)*
apply (*rule eq-paddingI*)
apply (*rule nth-equalityI*)
unfolding *access-result-size*
apply (*rule refl*)
by (*metis access-result-size complement is-padding-byte-acc-id is-value-byte-acc-upd-cancel*
that
update-access)

lemma *eq-padding-is-padding-byte-conv*:
eq-padding *bs bs'* $\longleftrightarrow \text{length } bs = sz \wedge \text{length } bs' = sz \wedge (\forall i. \text{is-padding-byte } i$
 $\longrightarrow bs ! i = bs' ! i)$
using *eq-padding-to-padding-byte-eq padding-byte-to-eq-padding-eq* **by** *blast*

lemma *value-byte-to-eq-upto-padding-eq*:
assumes *lbs*: *length* *bs* = *sz*
assumes *lbs'*: *length* *bs'* = *sz*
assumes *is-value*: $\forall i. \text{is-value-byte } i \longrightarrow bs ! i = bs' ! i$
shows *eq-upto-padding* *bs bs'*
proof (*rule eq-upto-paddingI [OF lbs lbs']*)
fix *v*
have *leq*: *length* (*acc* (*upd* *bs' v*) *bs*) = *sz*
by (*simp add: access-result-size*)
then have *acc* (*upd* *bs' v*) *bs* = *bs*
by (*smt (verit) eq-padding-neq-is-value-byte field-access-eq-padding1*
is-value is-value-byte-acc-upd-cancel lbs lbs' nth-equalityI)
then
show *upd* *bs v* = *upd* *bs' v*
by (*metis double-update update-access*)
qed

lemma *eq-upto-padding-is-value-byte-conv*:
eq-upto-padding *bs bs'* $\longleftrightarrow \text{length } bs = sz \wedge \text{length } bs' = sz \wedge (\forall i. \text{is-value-byte } i$
 $\longrightarrow bs ! i = bs' ! i)$
using *eq-upto-padding-to-value-byte-eq value-byte-to-eq-upto-padding-eq* **by** *blast*

end

locale *wf-field-desc* = *padding-lense* (*field-access* *d*) (*field-update* *d*) (*field-sz* *d*) **for**
d::'a *field-desc*

locale *field-desc-independent* =

fixes

acc::'a \Rightarrow *byte list* \Rightarrow *byte list* **and**

upd::*byte list* \Rightarrow 'a \Rightarrow 'a **and**

D:: 'a *field-desc* *set*

assumes *fu-commutes*: $d \in D \Longrightarrow \text{fu-commutes } \text{upd } (\text{field-update } d)$

assumes *acc-upd-old*: $d \in D \Longrightarrow \text{acc } (\text{field-update } d \text{ } bs \text{ } v) \text{ } bs' = \text{acc } v \text{ } bs'$

assumes *acc-upd-new*: $d \in D \Longrightarrow \text{field-access } d \text{ } (\text{upd } bs' \text{ } v) \text{ } bs = \text{field-access } d \text{ } v$
bs

primrec

field-descs :: 'a *xtyp-info* \Rightarrow 'a *field-desc* *list* **and**

field-descs-struct :: 'a *xtyp-info-struct* \Rightarrow 'a *field-desc* *list* **and**

field-descs-list :: 'a *xtyp-info-tuple* *list* \Rightarrow 'a *field-desc* *list* **and**

field-descs-tuple :: 'a *xtyp-info-tuple* \Rightarrow 'a *field-desc* *list*

where

fd0: *field-descs* (*TypDesc* *align* *st* *nm*) = *field-descs-struct* *st*

| *fd1*: *field-descs-struct* (*TypScalar* *n* *align* *d*) = [*d*]

| *fd2*: *field-descs-struct* (*TypAggregate* *xs*) = *field-descs-list* *xs*

| *fd3*: *field-descs-list* [] = []

| *fd4*: *field-descs-list* (*x*#*xs*) = *field-descs-tuple* *x* @ *field-descs-list* *xs*

| *fd5*: *field-descs-tuple* (*DTuple* *t* *nm* *d*) = [*d*] @ *field-descs* *t*

primrec

wf-component-descs :: 'a *xtyp-info* \Rightarrow *bool* **and**

wf-component-descs-struct :: 'a *xtyp-info-struct* \Rightarrow *bool* **and**

wf-component-descs-list :: 'a *xtyp-info-tuple* *list* \Rightarrow *bool* **and**

wf-component-descs-tuple :: 'a *xtyp-info-tuple* \Rightarrow *bool*

where

wfc0: *wf-component-descs* (*TypDesc* *align* *st* *nm*) = *wf-component-descs-struct* *st*

| *wfc1*: *wf-component-descs-struct* (*TypScalar* *n* *align* *d*) = (*n* = *field-sz* *d*)

| *wfc2*: *wf-component-descs-struct* (*TypAggregate* *xs*) = (*wf-component-descs-list*
xs)

| *wfc3*: *wf-component-descs-list* [] = *True*

| *wfc4*: *wf-component-descs-list* (*x*#*xs*) = (*wf-component-descs-tuple* *x* \wedge *wf-component-descs-list*
xs)

| *wfc5*: *wf-component-descs-tuple* (*DTuple* *t* *nm* *d*) =
(*d* = *component-desc* *t* \wedge *wf-component-descs* *t*)

```

primrec field-descs-independent:: 'a field-desc list  $\Rightarrow$  bool
  where
    field-descs-independent [] = True |
    field-descs-independent (d#ds) =
      (field-desc-independent (field-access d) (field-update d) (set ds)  $\wedge$ 
       field-descs-independent ds)

primrec
  component-descs-independent :: 'a xtyp-info  $\Rightarrow$  bool and
  component-descs-independent-struct :: 'a xtyp-info-struct  $\Rightarrow$  bool and
  component-descs-independent-list :: 'a xtyp-info-tuple list  $\Rightarrow$  bool and
  component-descs-independent-tuple :: 'a xtyp-info-tuple  $\Rightarrow$  bool
where
  cdi0: component-descs-independent (TypDesc align st nm) = component-descs-independent-struct
  st

  | cdi1: component-descs-independent-struct (TypScalar n align d) = True
  | cdi2: component-descs-independent-struct (TypAggregate xs) = (component-descs-independent-list
  xs)

  | cdi3: component-descs-independent-list [] = True
  | cdi4: component-descs-independent-list (f#fs) = (field-descs-independent (toplevel-field-descs-list
  (f#fs))  $\wedge$ 
           component-descs-independent-tuple f  $\wedge$  component-descs-independent-list
  fs)

  | cdi5: component-descs-independent-tuple (DTuple ft fn d) = (component-descs-independent
  ft)

locale wf-field-descs =
  fixes D::'a field-desc set
  assumes wf-desc[intro]:  $d \in D \Longrightarrow$  wf-field-desc d

lemma wf-field-descs-union [simp]: wf-field-descs (D  $\cup$  E) = (wf-field-descs D  $\wedge$ 
wf-field-descs E)
  by (auto simp add: wf-field-descs-def)

lemma wf-field-descs-empty[simp]: wf-field-descs {}
  by (simp add: wf-field-descs-def)

lemma wf-field-descs-insert [simp]: wf-field-descs (insert d D) = (wf-field-desc d
 $\wedge$  wf-field-descs D)
  by (auto simp add: wf-field-descs-def)

definition padding-desc:: nat  $\Rightarrow$  'a field-desc where
  padding-desc n = ( $\lambda$ field-access =  $\lambda$ v bs. if  $n \leq$  length bs then take n bs else
  replicate n 0, field-update =  $\lambda$ bs. id, field-sz = n)

```

definition *is-padding-desc* $d = (\exists n. d = \text{padding-desc } n)$

definition *padding-tag* $n = \text{TypDesc } 0 (\text{TypScalar } n 0 (\text{padding-desc } n)) \text{ '!pad-tyr'}$

definition *is-padding-tag* $t = (\exists n. t = \text{padding-tag } n)$

definition *padding-field-name* $f = (\exists xs. f = \text{CHR } \text{'!'} \# xs)$

lemma *padding-field-name-empty*[simp]: $\text{padding-field-name } [] = \text{False}$
by (*simp add: padding-field-name-def*)

lemma *padding-field-name-cons*[simp]: $\text{padding-field-name } (f\#fs) = (f = \text{CHR } \text{'!'})$
by (*simp add: padding-field-name-def*)

definition *qualified-padding-field-name* $fs = (\exists ffs'. fs = f\#fs' \wedge \text{padding-field-name } (\text{last } ffs'))$

lemma *qualified-pading-field-name-empty*[simp]: $\text{qualified-padding-field-name } [] = \text{False}$
by (*simp add: qualified-padding-field-name-def*)

lemma *qualified-pading-field-name-single*[simp]: $\text{qualified-padding-field-name } [f] = \text{padding-field-name } f$
by (*simp add: qualified-padding-field-name-def*)

lemma *qualified-pading-field-name-cons*[simp]: $\text{qualified-padding-field-name } (f\#fs) = \text{padding-field-name } (\text{last } (f\#fs))$
by (*simp add: qualified-padding-field-name-def*)

primrec

wf-padding :: ('a, 'b) *typ-info* \Rightarrow *bool* **and**
wf-padding-struct :: ('a, 'b) *typ-info-struct* \Rightarrow *bool* **and**
wf-padding-list :: ('a, 'b) *typ-info-tuple list* \Rightarrow *bool* **and**
wf-padding-tuple :: ('a, 'b) *typ-info-tuple* \Rightarrow *bool*

where

wf-padding (*TypDesc* *algn st nm*) = *wf-padding-struct st*

| *wf-padding-struct* (*TypScalar* *m algn d*) = *True*

| *wf-padding-struct* (*TypAggregate xs*) = *wf-padding-list xs*

| *wf-padding-list* [] = *True*

| *wf-padding-list* (*x#xs*) = (*wf-padding-tuple x* \wedge *wf-padding-list xs*)

| *wf-padding-tuple* (*DTuple s f d*) = ((*padding-field-name f* \longrightarrow *is-padding-tag s*) \wedge *wf-padding s*)

```

class xmem-type = mem-type +
  assumes wf-component-descs: wf-component-descs (typ-info-t TYPE('a))
  assumes component-descs-independent: component-descs-independent (typ-info-t
TYPE('a))
  assumes wf-field-descs: wf-field-descs (set (field-descs (typ-info-t TYPE('a))))
  assumes wf-padding: wf-padding (typ-info-t TYPE('a))

```

```

locale fields-contained =
  fixes
    acc::'a  $\Rightarrow$  byte list  $\Rightarrow$  byte list and
    upd::byte list  $\Rightarrow$  'a  $\Rightarrow$  'a and
    D::'a field-desc set
  assumes access-contained:  $d \in D \Longrightarrow acc\ s = acc\ s' \Longrightarrow field-access\ d\ s =$ 
field-access\ d\ s'
  assumes update-contained:  $d \in D \Longrightarrow \exists bs'. \forall s'. acc\ s' = acc\ s \longrightarrow field-update$ 
d\ bs\ s' = upd\ bs'\ s'

```

```

primrec
  contained-field-descs :: 'a xtyp-info  $\Rightarrow$  bool and
  contained-field-descs-struct :: 'a xtyp-info-struct  $\Rightarrow$  bool and
  contained-field-descs-list :: 'a xtyp-info-tuple list  $\Rightarrow$  bool and
  contained-field-descs-tuple :: 'a xtyp-info-tuple  $\Rightarrow$  bool
where
  wfd0: contained-field-descs (TypDesc algn st nm) = contained-field-descs-struct
st
  | wfd1: contained-field-descs-struct (TypScalar n algn d) = True
  | wfd2: contained-field-descs-struct (TypAggregate xs) = contained-field-descs-list
xs
  | wfd3: contained-field-descs-list [] = True
  | wfd4: contained-field-descs-list (x#xs) = (contained-field-descs-tuple x  $\wedge$  con-
tained-field-descs-list xs)
  | wfd5: contained-field-descs-tuple (DTuple t nm d) =
    (contained-field-descs t  $\wedge$ 
     fields-contained (field-access d) (field-update d) (set (toplevel-field-descs t)))

```

```

class xmem-contained-type = xmem-type +
  assumes contained-field-descs: contained-field-descs (typ-info-t (TYPE('a)))

```

In order to construct a type of class *mem-type* we usually construct extended type-info for class *xmem-contained-type*. The extended type-info is better behaved with respect to composition of sub-structures / arrays.

end

theory *CTypes*
imports
 CTypesDefs HOL-Eisbach.Eisbach-Tools
begin

11.10 *super-update-bs*

lemma *length-super-update-bs* [*simp*]:
 $n + \text{length } v \leq \text{length } bs \implies \text{length } (\text{super-update-bs } v \text{ } bs \ n) = \text{length } bs$
 by (*clarsimp simp: super-update-bs-def*)

lemma *drop-super-update-bs*:
 $\llbracket k \leq n; n \leq \text{length } bs \rrbracket \implies \text{drop } k \ (\text{super-update-bs } v \text{ } bs \ n) = \text{super-update-bs } v \ (\text{drop } k \ bs) \ (n - k)$
 by (*simp add: super-update-bs-def take-drop*)

lemma *drop-super-update-bs2*:
 $\llbracket n \leq \text{length } bs; n + \text{length } v \leq k \rrbracket \implies$
 $\text{drop } k \ (\text{super-update-bs } v \text{ } bs \ n) = \text{drop } k \ bs$
 by (*clarsimp simp: super-update-bs-def min-def split: if-split-asm*)

lemma *take-super-update-bs*:
 $\llbracket k \leq n; n \leq \text{length } bs \rrbracket \implies \text{take } k \ (\text{super-update-bs } v \text{ } bs \ n) = \text{take } k \ bs$
 by (*clarsimp simp: super-update-bs-def min-def split: if-split-asm*)

lemma *take-super-update-bs2*:
 $\llbracket n \leq \text{length } bs; n + \text{length } v \leq k \rrbracket \implies$
 $\text{take } k \ (\text{super-update-bs } v \text{ } bs \ n) = \text{super-update-bs } v \ (\text{take } k \ bs) \ n$
 apply (*clarsimp simp: super-update-bs-def min-def split: if-split-asm*)
 apply (*cases n=k; simp add: drop-take*)
 done

lemma *take-drop-super-update-bs*:
 $\text{length } v = n \implies m \leq \text{length } bs \implies \text{take } n \ (\text{drop } m \ (\text{super-update-bs } v \text{ } bs \ m)) =$
 v
 by (*simp add: super-update-bs-def*)

lemma *super-update-bs-take-drop*:
 $n + m \leq \text{length } bs \implies \text{super-update-bs } (\text{take } m \ (\text{drop } n \ bs)) \ bs \ n = bs$
 by (*simp add: super-update-bs-def*) (*metis append.assoc append-take-drop-id take-add*)

lemma *super-update-bs-same-length*: $\text{length } bs = \text{length } xbs \implies \text{super-update-bs } bs \ xbs \ 0 = bs$
 by (*simp add: super-update-bs-def*)

lemma *super-update-bs-append-drop-first*:

$length\ xbs = m \implies n + length\ bs \leq m \implies drop\ m\ (super-update-bs\ bs\ (xbs\ @\ ybs)\ n) = ybs$

by (*simp add: super-update-bs-def*)

lemma *super-update-bs-append-take-first*:

$length\ xbs = m \implies n + length\ bs \leq m \implies take\ m\ (super-update-bs\ bs\ (xbs\ @\ ybs)\ n) = (super-update-bs\ bs\ xbs\ n)$

by (*simp add: super-update-bs-def*)

lemma *super-update-bs-append-drop-second*:

$length\ xbs = m \implies m \leq n \implies$

$drop\ m\ (super-update-bs\ bs\ (xbs\ @\ ybs)\ n) = (super-update-bs\ bs\ ybs\ (n - m))$

by (*simp add: super-update-bs-def*)

lemma *super-update-bs-append-take-second*:

$length\ xbs = m \implies m \leq n \implies$

$take\ m\ (super-update-bs\ bs\ (xbs\ @\ ybs)\ n) = xbs$

by (*simp add: super-update-bs-def*)

lemma *super-update-bs-length*: $length\ bs + n \leq length\ xbs \implies length\ (super-update-bs\ bs\ xbs\ n) = length\ xbs$

by (*simp add: super-update-bs-def*)

lemma *super-update-bs-append2*: $length\ xbs \leq n \implies$

$super-update-bs\ bs\ (xbs\ @\ ybs)\ n = xbs\ @\ super-update-bs\ bs\ ybs\ (n - length\ xbs)$

by (*simp add: super-update-bs-def*)

lemma *super-update-bs-append1*: $n + length\ bs \leq length\ xbs \implies$

$super-update-bs\ bs\ (xbs\ @\ ybs)\ n = super-update-bs\ bs\ xbs\ n\ @\ ybs$

by (*simp add: super-update-bs-def*)

11.11 Rest

lemma *fu-commutes*:

$fu-commutes\ f\ g \implies f\ bs\ (g\ bs'\ v) = g\ bs'\ (f\ bs\ v)$

by (*simp add: fu-commutes-def*)

lemma *size-td-list-append* [*simp*]:

$size-td-list\ (xs@ys) = size-td-list\ xs + size-td-list\ ys$

by (*induct xs*) (*auto*)

lemma *access-ti-append*:

$\bigwedge bs. length\ bs = size-td-list\ (xs@ys) \implies$

$access-ti-list\ (xs@ys)\ t\ bs =$

$access-ti-list\ xs\ t\ (take\ (size-td-list\ xs)\ bs)\ @$

$access-ti-list\ ys\ t\ (drop\ (size-td-list\ xs)\ bs)$

proof (*induct xs*)
case *Nil* **show** ?*case* **by** *simp*
next
case (*Cons x xs*) **thus** ?*case* **by** (*simp add: min-def ac-simps drop-take*)
qed

lemma *update-ti-append* [*simp*]:
 $\bigwedge bs. \text{update-ti-list } (xs@ys) \text{ } bs \text{ } v =$
 $\text{update-ti-list } xs \text{ } (\text{take } (\text{size-td-list } xs) \text{ } bs)$
 $(\text{update-ti-list } ys \text{ } (\text{drop } (\text{size-td-list } xs) \text{ } bs) \text{ } v)$

proof (*induct xs*)
case *Nil* **show** ?*case* **by** *simp*
next
case (*Cons x xs*) **thus** ?*case* **by** (*simp add: drop-take ac-simps min-def*)
qed

lemma *update-ti-struct-t-typscalar* [*simp*]:
 $\text{update-ti-struct-t } (\text{TypScalar } n \text{ } \text{algn } d) =$
 $(\lambda bs. \text{if length } bs = n \text{ then field-update } d \text{ } bs \text{ else id})$
by (*rule ext, simp add: update-ti-struct-t-def*)

lemma *update-ti-list-t-empty* [*simp*]:
 $\text{update-ti-list-t } [] = (\lambda x. \text{id})$
by (*rule ext, simp add: update-ti-list-t-def*)

lemma *update-ti-list-t-cons* [*simp*]:
 $\text{update-ti-list-t } (x\#xs) = (\lambda bs \text{ } v.$
 $\text{if length } bs = \text{size-td-tuple } x + \text{size-td-list } xs \text{ then}$
 $\text{update-ti-tuple-t } x \text{ } (\text{take } (\text{size-td-tuple } x) \text{ } bs)$
 $(\text{update-ti-list-t } xs \text{ } (\text{drop } (\text{size-td-tuple } x) \text{ } bs) \text{ } v) \text{ else}$
 $v)$
by (*force simp: update-ti-list-t-def update-ti-tuple-t-def min-def*)

lemma *update-ti-append-s* [*simp*]:
 $\bigwedge bs. \text{update-ti-list-t } (xs@ys) \text{ } bs \text{ } v = ($
 $\text{if length } bs = \text{size-td-list } xs + \text{size-td-list } ys \text{ then}$
 $\text{update-ti-list-t } xs \text{ } (\text{take } (\text{size-td-list } xs) \text{ } bs)$
 $(\text{update-ti-list-t } ys \text{ } (\text{drop } (\text{size-td-list } xs) \text{ } bs) \text{ } v) \text{ else}$
 $v)$

proof (*induct xs*)
case *Nil* **show** ?*case* **by** (*simp add: update-ti-list-t-def*)
next
case (*Cons x xs*) **thus** ?*case* **by** (*simp add: min-def drop-take ac-simps*)
qed

lemma *update-ti-tuple-t-dtuple* [*simp*]:
 $\text{update-ti-tuple-t } (DTuple \text{ } t \text{ } f \text{ } d) = \text{update-ti-t } t$
by (*rule ext, simp add: update-ti-tuple-t-def update-ti-t-def*)

lemma *field-desc-empty* [simp]:
 $\text{field-desc } (\text{TypDesc } \text{algn } (\text{TypAggregate } [])) \text{ nm} =$
 $(\mid \text{field-access} = \lambda x \text{ bs. } [],$
 $\text{field-update} = \lambda x. \text{id}, \text{field-sz} = 0 \mid)$
by (force simp: update-ti-t-def)

lemma *export-uinfo-typdesc-simp* [simp]:
 $\text{export-uinfo } (\text{TypDesc } \text{algn } \text{st } \text{nm}) = \text{map-td field-norm } (\lambda-. ()) (\text{TypDesc } \text{algn } \text{st}$
 $\text{nm})$
by (simp add: export-uinfo-def)

lemma *map-td-list-append* [simp]:
 $\text{map-td-list } f \ g \ (xs@ys) = \text{map-td-list } f \ g \ xs \ @ \ \text{map-td-list } f \ g \ ys$
by (induct xs) auto

lemma *map-td-id*:
 $\text{map-td } (\lambda n \ \text{algn. id}) \ \text{id} \ t = (t::('a, 'b) \text{typ-desc})$
 $\text{map-td-struct } (\lambda n \ \text{algn. id}) \ \text{id} \ st = (st::('a, 'b) \text{typ-struct})$
 $\text{map-td-list } (\lambda n \ \text{algn. id}) \ \text{id} \ ts = (ts::('a, 'b) \text{typ-tuple list})$
 $\text{map-td-tuple } (\lambda n \ \text{algn. id}) \ \text{id} \ x = (x::('a, 'b) \text{typ-tuple})$
by (induction t **and** st **and** ts **and** x) simp-all

lemma *dt-snd-map-td-list*:
 $\text{dt-snd } ' \ \text{set } (\text{map-td-list } f \ g \ ts) = \text{dt-snd } ' \ \text{set } ts$
proof (induct ts)
case (Cons x xs) **thus** ?case **by** (cases x) auto
qed simp

lemma *wf-desc-map*:
shows $\text{wf-desc } (\text{map-td } f \ g \ t) = \text{wf-desc } t$ **and**
 $\text{wf-desc-struct } (\text{map-td-struct } f \ g \ st) = \text{wf-desc-struct } st$ **and**
 $\text{wf-desc-list } (\text{map-td-list } f \ g \ ts) = \text{wf-desc-list } ts$ **and**
 $\text{wf-desc-tuple } (\text{map-td-tuple } f \ g \ x) = \text{wf-desc-tuple } x$
proof (induct t **and** st **and** ts **and** x)
case (Cons-typ-desc x xs) **thus** ?case
by (cases x, auto simp: dt-snd-map-td-list)
qed auto

lemma *wf-desc-list-append* [simp]:
 $\text{wf-desc-list } (xs@ys) =$
 $(\text{wf-desc-list } xs \ \wedge \ \text{wf-desc-list } ys \ \wedge \ \text{dt-snd } ' \ \text{set } xs \ \cap \ \text{dt-snd } ' \ \text{set } ys = \{\})$
by (induct xs) auto

lemma *wf-size-desc-list-append* [simp]:
 $\text{wf-size-desc-list } (xs@ys) = (\text{wf-size-desc-list } xs \ \wedge \ \text{wf-size-desc-list } ys)$
by (induct xs) auto

lemma *norm-tu-list-append* [simp]:
norm-tu-list (xs@ys) bs =
norm-tu-list xs (take (size-td-list xs) bs) @ norm-tu-list ys (drop (size-td-list xs)
bs)
by (*induct xs arbitrary: bs, auto simp: min-def ac-simps drop-take*)

lemma *wf-size-desc-gt*:
shows *wf-size-desc (t::('a, 'b) typ-desc) $\implies 0 < \text{size-td } t$ and*
wf-size-desc-struct st $\implies 0 < \text{size-td-struct (st::('a, 'b) typ-struct)$ and
 $\llbracket ts \neq [] \rrbracket; \text{wf-size-desc-list } ts \rrbracket \implies 0 < \text{size-td-list (ts::('a, 'b) typ-tuple list)}$
and
wf-size-desc-tuple x $\implies 0 < \text{size-td-tuple (x::('a, 'b) typ-tuple)$
by (*induct t and st and ts and x rule: typ-desc-tyt-struct-inducts, auto*)

lemma *field-lookup-empty* [simp]:
field-lookup t [] n = Some (t,n)
by (*cases t*) *clarsimp*

lemma *field-lookup-tuple-empty* [simp]:
field-lookup-tuple x [] n = None
by (*cases x*) *clarsimp*

lemma *field-lookup-list-empty* [simp]:
field-lookup-list ts [] n = None
by (*induct ts arbitrary: n*) *auto*

lemma *field-lookup-struct-empty* [simp]:
field-lookup-struct st [] n = None
by (*cases st*) *auto*

lemma *field-lookup-list-append*:
field-lookup-list (xs@ys) f n = (case field-lookup-list xs f n of
None \implies field-lookup-list ys f (n + size-td-list xs)
| Some y \implies Some y)

proof (*induct xs arbitrary: n*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons x xs*) **thus** *?case*
by (*cases x*) (*auto simp: ac-simps split: option.splits*)
qed

lemma *field-lookup-list-None*:
f \notin dt-snd ' set ts \implies field-lookup-list ts (f#fs) m = None

proof (*induct ts arbitrary: f fs m*)
case (*Cons x -*) **thus** *?case* **by** (*cases x*) *auto*
qed *simp*

lemma *field-lookup-list-Some*:
f \in dt-snd ' set ts \implies field-lookup-list ts [f] m \neq None

proof (*induct ts arbitrary: f m*)
case (*Cons x -*) **thus** *?case by (cases x) auto*
qed simp

lemma *field-lookup-offset-le:*

shows $\bigwedge s m n f. \text{field-lookup } t f m = \text{Some } ((s::('a,'b) \text{ typ-desc}),n) \implies m \leq n$
and

$\bigwedge s m n f. \text{field-lookup-struct } st f m = \text{Some } ((s::('a,'b) \text{ typ-desc}),n) \implies m \leq n$ **and**

$\bigwedge s m n f. \text{field-lookup-list } ts f m = \text{Some } ((s::('a, 'b) \text{ typ-desc}),n) \implies m \leq n$ **and**

$\bigwedge s m n f. \text{field-lookup-tuple } x f m = \text{Some } ((s::('a, 'b) \text{ typ-desc}),n) \implies m \leq n$

proof (*induct t and st and ts and x*)

case (*Cons-typ-desc x xs*) **thus** *?case by (fastforce split: option.splits)*
qed (*auto split: if-split-asm*)

lemma *field-lookup-offset':*

shows $\bigwedge f m m' n t'. (\text{field-lookup } t f m = \text{Some } ((t'::('a,'b) \text{ typ-desc}),m + n)) =$
 $(\text{field-lookup } t f m' = \text{Some } (t',m' + n))$ **and**

$\bigwedge f m m' n t'. (\text{field-lookup-struct } st f m = \text{Some } ((t'::('a,'b) \text{ typ-desc}),m + n)) =$
 $(\text{field-lookup-struct } st f m' = \text{Some } (t',m' + n))$ **and**

$\bigwedge f m m' n t'. (\text{field-lookup-list } ts f m = \text{Some } ((t'::('a,'b) \text{ typ-desc}),m + n)) =$
 $(\text{field-lookup-list } ts f m' = \text{Some } (t',m' + n))$ **and**

$\bigwedge f m m' n t'. (\text{field-lookup-tuple } x f m = \text{Some } ((t'::('a,'b) \text{ typ-desc}),m + n)) =$
 $(\text{field-lookup-tuple } x f m' = \text{Some } (t',m' + n))$

proof (*induct t and st and ts and x*)

case (*Cons-typ-desc x xs*)
show *?case*
proof

assume *ls: field-lookup-list (x # xs) f m = Some (t', m + n)*

show *field-lookup-list (x # xs) f m' = Some (t', m' + n) (is ?X)*
proof cases

assume *ps: field-lookup-tuple x f m = None*
moreover from this ls have $\exists k. n = \text{size-td } (dt\text{-fst } x) + k$
by (*clarsimp dest!: field-lookup-offset-le, arith*)

moreover have *field-lookup-tuple x f m' = None*
apply (*rule ccontr*)
using *ps*
apply *clarsimp*

subgoal premises prems for a b
proof -

obtain *k where b = m' + k*
using *prems field-lookup-offset-le*
by (*metis less-eqE*)

then show *?thesis*

```

    using prems
    by (clarsimp simp: Cons-typ-desc [where m'=m])
qed
done
ultimately show ?X using ls
  by (clarsimp simp: add.assoc [symmetric])
    (subst (asm) Cons-typ-desc [where m'=m' + size-td (dt-fst x)], fast)
next
assume nps: field-lookup-tuple x f m ≠ None
moreover from this have field-lookup-tuple x f m' ≠ None
  apply (clarsimp)
  subgoal premises prems for a b
  proof -
    obtain k where b = m + k
    using prems field-lookup-offset-le
    by (metis less-eqE)
    then show ?thesis
    using prems
    apply clarsimp
    apply (subst (asm) Cons-typ-desc [where m'=m'])
    apply fast
    done
  qed
done
ultimately show ?X using ls
  by (clarsimp, subst (asm) Cons-typ-desc [where m'=m'], simp)
qed
next
assume ls: field-lookup-list (x # xs) f m' = Some (t', m' + n)
show field-lookup-list (x # xs) f m = Some (t', m + n) (is ?X)
proof cases
  assume ps: field-lookup-tuple x f m' = None
  moreover from this ls have ∃k. n = size-td (dt-fst x) + k
    by (clarsimp dest!: field-lookup-offset-le, arith)
  moreover have field-lookup-tuple x f m = None
    apply (rule ccontr)
    using ps
    apply clarsimp
    subgoal premises prems for a b
    proof -
      obtain k where b = m + k
      using prems field-lookup-offset-le
      by (metis less-eqE)
      then show ?thesis using prems
      by (clarsimp simp: Cons-typ-desc [where m'=m'])
    qed
  done
ultimately show ?X using ls
  by (clarsimp simp: add.assoc [symmetric])

```

```

      (subst (asm) Cons-typ-desc [where m'=m + size-td (dt-fst x)], fast)
next
assume nps: field-lookup-tuple x f m' ≠ None
moreover from this have field-lookup-tuple x f m ≠ None
  apply clarsimp
  subgoal premises prems for a b
  proof -
    obtain k where b = m' + k
    using prems field-lookup-offset-le
    by (metis less-eqE)
  then show ?thesis using prems
  apply clarsimp
  apply (subst (asm) Cons-typ-desc [where m'=m])
  apply fast
  done
qed
done
ultimately show ?X using ls
  by (clarsimp, subst (asm) Cons-typ-desc [where m'=m], simp)
qed
qed
qed auto

```

lemma *field-lookup-offset*:
 $(\text{field-lookup } t \ f \ m = \text{Some } (t', m + n)) = (\text{field-lookup } t \ f \ 0 = \text{Some } (t', n))$
 by (simp add: field-lookup-offset' [where m'=0])

lemma *field-lookup-offset2*:
 $\text{field-lookup } t \ f \ m = \text{Some } (t', n) \implies \text{field-lookup } t \ f \ 0 = \text{Some } (t', n - m)$
 by (simp add: field-lookup-offset-le
 flip: field-lookup-offset [where m=m])

lemma *field-lookup-list-offset*:
 $(\text{field-lookup-list } ts \ f \ m = \text{Some } (t', m + n)) = (\text{field-lookup-list } ts \ f \ 0 = \text{Some } (t', n))$
 by (simp add: field-lookup-offset' [where m'=0])

lemma *field-lookup-list-offset2*:
 $\text{field-lookup-list } ts \ f \ m = \text{Some } (t', n) \implies \text{field-lookup-list } ts \ f \ 0 = \text{Some } (t', n - m)$
 by (simp add: field-lookup-offset-le
 flip: field-lookup-list-offset [where m=m])

lemma *field-lookup-list-offset3*:
 $\text{field-lookup-list } ts \ f \ 0 = \text{Some } (t', n) \implies \text{field-lookup-list } ts \ f \ m = \text{Some } (t', m + n)$
 by (simp add: field-lookup-list-offset)

lemma *field-lookup-list-offsetD*:

[[*field-lookup-list* *ts* *f* 0 = *Some* (*s*,*k*);
field-lookup-list *ts* *f* *m* = *Some* (*t*,*n*)]] $\implies s=t \wedge n=m+k$

subgoal premises *prems*

proof –

have $\exists k. n = m + k$ **using** *prems* *field-lookup-offset-le* **by** (*metis less-eqE*)

then show *?thesis* **using** *prems*

by (*clarsimp simp: field-lookup-list-offset*)

qed

done

lemma *field-lookup-offset-None*:

(*field-lookup* *t* *f* *m* = *None*) = (*field-lookup* *t* *f* 0 = *None*)

by (*auto simp: field-lookup-offset2 field-lookup-offset* [**where** *m=m,symmetric*]
intro: ccontr)

lemma *field-lookup-list-offset-None*:

(*field-lookup-list* *ts* *f* *m* = *None*) = (*field-lookup-list* *ts* *f* 0 = *None*)

by (*auto simp: field-lookup-list-offset2 field-lookup-list-offset* [**where** *m=m,symmetric*]
intro: ccontr)

lemma *map-td-size* [*simp*]:

shows *size-td* (*map-td* *f* *g* *t*) = *size-td* *t* **and**

size-td-struct (*map-td-struct* *f* *g* *st*) = *size-td-struct* *st* **and**

size-td-list (*map-td-list* *f* *g* *ts*) = *size-td-list* *ts* **and**

size-td-tuple (*map-td-tuple* *f* *g* *x*) = *size-td-tuple* *x*

by (*induct* *t* **and** *st* **and** *ts* **and** *x*, *auto*)

lemma *td-set-map-td1*:

(*s*, *n*) \in *td-set* *t* *k* \implies (*map-td* *f* *g* *s*, *n*) \in *td-set* (*map-td* *f* *g* *t*) *k* **and**

(*s*, *n*) \in *td-set-struct* *st* *k* \implies (*map-td* *f* *g* *s*, *n*) \in *td-set-struct* (*map-td-struct* *f* *g* *st*) *k* **and**

(*s*, *n*) \in *td-set-list* *ts* *k* \implies (*map-td* *f* *g* *s*, *n*) \in *td-set-list* (*map-td-list* *f* *g* *ts*) *k* **and**

(*s*, *n*) \in *td-set-tuple* *td* *k* \implies (*map-td* *f* *g* *s*, *n*) \in *td-set-tuple* (*map-td-tuple* *f* *g* *td*) *k*

apply (*induct* *t* **and** *st* **and** *ts* **and** *td* *arbitrary: n k* **and** *n k* **and** *n k* **and** *n k*)

by (*auto, metis dt-tuple.collapse map-td-size(4) tz5*)

lemma *size-td-tuple-dt-fst*:

size-td-tuple *p* = *size-td* (*dt-fst* *p*)

by (*cases* *p*, *simp*)

lemma *td-set-map-td2*:

(*s'*, *n*) \in *td-set* (*map-td* *f* *g* *t*) *k* \implies $\exists s. (s, n) \in$ *td-set* *t* *k* $\wedge s' =$ *map-td* *f* *g* *s* **and**

(*s'*, *n*) \in *td-set-struct* (*map-td-struct* *f* *g* *st*) *k* \implies $\exists s. (s, n) \in$ *td-set-struct* *st* *k* $\wedge s' =$ *map-td* *f* *g* *s* **and**

(*s'*, *n*) \in *td-set-list* (*map-td-list* *f* *g* *ts*) *k* \implies $\exists s. (s, n) \in$ *td-set-list* *ts* *k* $\wedge s' =$ *map-td* *f* *g* *s* **and**

$(s', n) \in \text{td-set-tuple } (\text{map-td-tuple } f g \text{ td}) k \implies \exists s. (s, n) \in \text{td-set-tuple } \text{td } k \wedge s' = \text{map-td } f g s$
proof (*induct t and st and ts and td arbitrary: n k and n k and n k and n k*)
)

```

  case (TypDesc nat typ-struct list)
  then show ?case by auto
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply (clarsimp, safe)
    apply blast
    by (metis map-td-size(4) size-td-tuple-dt-fst)
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by auto
qed

```

lemma *td-set-offset1*:

$(s, n) \in \text{td-set } t k \implies (s, n + l) \in \text{td-set } t (k + l)$ **and**
 $(s, n) \in \text{td-set-struct } st k \implies (s, n + l) \in \text{td-set-struct } st (k + l)$ **and**
 $(s, n) \in \text{td-set-list } xs k \implies (s, n + l) \in \text{td-set-list } xs (k + l)$ **and**
 $(s, n) \in \text{td-set-tuple } \text{td } k \implies (s, n + l) \in \text{td-set-tuple } \text{td } (k + l)$
apply (*induct t and st and xs and td arbitrary: s n k l and s n k l and s n k l and s n k l*)
by (*auto, smt (verit, best) add.commute add.left-commute*)

lemma *td-set-offset2*:

$(s, n + l) \in \text{td-set } t (k + l) \implies (s, n) \in \text{td-set } t k$ **and**
 $(s, n + l) \in \text{td-set-struct } st (k + l) \implies (s, n) \in \text{td-set-struct } st k$ **and**
 $(s, n + l) \in \text{td-set-list } xs (k + l) \implies (s, n) \in \text{td-set-list } xs k$ **and**
 $(s, n + l) \in \text{td-set-tuple } \text{td } (k + l) \implies (s, n) \in \text{td-set-tuple } \text{td } k$
apply (*induct t and st and xs and td arbitrary: s n k l and s n k l and s n k l and s n k l*)
by (*auto, smt (verit, best) add.commute add.left-commute*)

lemma *td-set-offset-conv*: $(s, n) \in \text{td-set } t k \longleftrightarrow (s, n + l) \in \text{td-set } t (k + l)$
using *td-set-offset1 td-set-offset2* **by** *blast*

lemma *td-set-offset-0-conv*: $(s, n + k) \in \text{td-set } t k \longleftrightarrow (s, n) \in \text{td-set } t 0$
using *td-set-offset-conv* [**where** *k=0 and n=n and l=k and s=s and t=t*]

by *simp*

lemma *td-set-offset-le'*:

$\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set } t m \longrightarrow m \leq n$
 $\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set-struct } st m \longrightarrow m \leq n$
 $\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set-list } ts m \longrightarrow m \leq n$
 $\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set-tuple } x m \longrightarrow m \leq n$
by (*induct t and st and ts and x*) *fastforce+*

lemma *td-set-offset-0-conv'*: $(s, n) \in \text{td-set } t k \longleftrightarrow (\exists n'. (s, n') \in \text{td-set } t 0 \wedge n = n' + k)$

by (*metis add.commute le-Suc-ex td-set-offset-0-conv td-set-offset-le'(1)*)

lemma *td-set-list-set-td-set1*: $(s, n) \in \text{td-set-list } ts k \implies$

$(\exists t n'. t \in \text{set } ts \wedge (s, n') \in \text{td-set } (dt\text{-fst } t) 0)$

apply (*induct ts arbitrary: n k*)

apply *simp*

apply *simp*

by (*metis dt-tuple.collapse td-set-offset-0-conv' ts5*)

lemma *export-uinfo-size* [*simp*]:

$\text{size-td } (\text{export-uinfo } t) = \text{size-td } (t::('a,'b) \text{ typ-info})$

by (*simp add: export-uinfo-def*)

lemma (*in c-type*) *typ-uinfo-size* [*simp*]:

$\text{size-td } (\text{typ-uinfo-t } \text{TYPE}('a)) = \text{size-td } (\text{typ-info-t } \text{TYPE}('a))$

by (*simp add: typ-uinfo-t-def export-uinfo-def*)

lemma *wf-size-desc-map*:

shows $\text{wf-size-desc } (\text{map-td } f g t) = \text{wf-size-desc } t$ and

$\text{wf-size-desc-struct } (\text{map-td-struct } f g st) = \text{wf-size-desc-struct } st$ and

$\text{wf-size-desc-list } (\text{map-td-list } f g ts) = \text{wf-size-desc-list } ts$ and

$\text{wf-size-desc-tuple } (\text{map-td-tuple } f g x) = \text{wf-size-desc-tuple } x$

proof (*induction t and st and ts and x*)

case (*TypAggregate list*)

then show ?case by (*cases list*) *auto*

qed *auto*

lemma *map-td-flr-Some* [*simp*]:

$\text{map-td-flr } f g (\text{Some } (t,n)) = \text{Some } (\text{map-td } f g t,n)$

by (*clarsimp simp: map-td-flr-def*)

lemma *map-td-flr-None* [*simp*]:

$\text{map-td-flr } f g \text{None} = \text{None}$

by (*clarsimp simp: map-td-flr-def*)

lemma *field-lookup-map*:

shows $\bigwedge f m s. \text{field-lookup } t f m = s \implies$

$field_lookup (map_td\ fupd\ g\ t)\ f\ m = map_td_flr\ fupd\ g\ s$ **and**
 $\bigwedge f\ m\ s.\ field_lookup_struct\ st\ f\ m = s \implies$
 $field_lookup_struct (map_td_struct\ fupd\ g\ st)\ f\ m = map_td_flr\ fupd\ g\ s$
and
 $\bigwedge f\ m\ s.\ field_lookup_list\ ts\ f\ m = s \implies$
 $field_lookup_list (map_td_list\ fupd\ g\ ts)\ f\ m = map_td_flr\ fupd\ g\ s$ **and**
 $\bigwedge f\ m\ s.\ field_lookup_tuple\ x\ f\ m = s \implies$
 $field_lookup_tuple (map_td_tuple\ fupd\ g\ x)\ f\ m = map_td_flr\ fupd\ g\ s$
proof (*induct t and st and ts and x*)
case (*Cons-typ-desc x xs thus ?case*)
by (*clarsimp, cases x, auto simp: map-td-flr-def split: option.splits*)
qed *auto*

lemma *field-lookup-export-uinfo-Some:*
 $field_lookup (t::('a,'b)\ typ_info)\ f\ m = Some\ (s,n) \implies$
 $field_lookup (export_uinfo\ t)\ f\ m = Some\ (export_uinfo\ s,n)$
by (*simp add: export-uinfo-def field-lookup-map*)

lemma *field-lookup-struct-export-Some:*
 $field_lookup_struct (st::('a,'b)\ typ_struct)\ f\ m = Some\ (s,n) \implies$
 $field_lookup_struct (map_td_struct\ fupd\ g\ st)\ f\ m = Some\ (map_td\ fupd\ g\ s,n)$
by (*simp add: field-lookup-map*)

lemma *field-lookup-struct-export-None:*
 $field_lookup_struct (st::('a,'b)\ typ_struct)\ f\ m = None \implies$
 $field_lookup_struct (map_td_struct\ fupd\ g\ st)\ f\ m = None$
by (*simp add: field-lookup-map*)

lemma *field-lookup-list-export-Some:*
 $field_lookup_list (ts::('a,'b)\ typ_tuple\ list)\ f\ m = Some\ (s,n) \implies$
 $field_lookup_list (map_td_list\ fupd\ g\ ts)\ f\ m = Some\ (map_td\ fupd\ g\ s,n)$
by (*simp add: field-lookup-map*)

lemma *field-lookup-list-export-None:*
 $field_lookup_list (ts::('a,'b)\ typ_tuple\ list)\ f\ m = None \implies$
 $field_lookup_list (map_td_list\ fupd\ g\ ts)\ f\ m = None$
by (*simp add: field-lookup-map*)

lemma *field-lookup-tuple-export-Some:*
 $field_lookup_tuple (x::('a,'b)\ typ_tuple)\ f\ m = Some\ (s,n) \implies$
 $field_lookup_tuple (map_td_tuple\ fupd\ g\ x)\ f\ m = Some\ (map_td\ fupd\ g\ s,n)$
by (*simp add: field-lookup-map*)

lemma *field-lookup-tuple-export-None:*
 $field_lookup_tuple (x::('a,'b)\ typ_tuple)\ f\ m = None \implies$
 $field_lookup_tuple (map_td_tuple\ fupd\ g\ x)\ f\ m = None$
by (*simp add: field-lookup-map*)

lemma *import-flr-Some [simp]:*

```

import-flr f g (Some (map-td f g t,n)) (Some (t,n))
by (clarsimp simp: import-flr-def)

lemma import-flr-None [simp]:
import-flr f g None None
by (clarsimp simp: import-flr-def)

lemma field-lookup-export-uinfo-rev'':
 $\bigwedge f m s.$  field-lookup (map-td fupd g t) f m = s  $\implies$ 
import-flr fupd g s ((field-lookup t f m)::('a,'b) flr)
 $\bigwedge f m s.$  field-lookup-struct (map-td-struct fupd g st) f m = s  $\implies$ 
import-flr fupd g s ((field-lookup-struct st f m)::('a,'b) flr)
 $\bigwedge f m s.$  field-lookup-list (map-td-list fupd g ts) f m = s  $\implies$ 
import-flr fupd g s ((field-lookup-list ts f m)::('a,'b) flr)
 $\bigwedge f m s.$  field-lookup-tuple (map-td-tuple fupd g x) f m = s  $\implies$ 
import-flr fupd g s ((field-lookup-tuple x f m)::('a,'b) flr)
proof (induct t and st and ts and x)
case (TypDesc nat typ-struct list)
then show ?case by (clarsimp simp: import-flr-def export-uinfo-def)
next
case (TypScalar nat1 nat2 a)
then show ?case by auto
next
case (TypAggregate list)
then show ?case by auto
next
case Nil-typ-desc
then show ?case by auto
next
case (Cons-typ-desc dt-tuple list)
then show ?case
apply clarsimp
apply (clarsimp split: option.splits)
apply (cases f, clarsimp+)
apply (rule conjI, clarsimp)
apply (cases dt-tuple, simp)
apply clarsimp
apply (drule field-lookup-tuple-export-Some [where fupd=fupd and g=g])
apply simp
apply (rule conjI; clarsimp)
apply (drule field-lookup-tuple-export-None [where fupd=fupd and g=g])
apply simp
apply metis
done
next
case (DTuple-typ-desc typ-desc list b)
then show ?case by (auto)
qed

```

lemma *field-lookup-export-uinfo-rev'*:

($\forall f m s.$ *field-lookup* (*map-td fupd g t*) $f m = s \longrightarrow$
import-flr fupd g s ((*field-lookup t f m*)::('a,'b) *flr*)) \wedge
($\forall f m s.$ *field-lookup-struct* (*map-td-struct fupd g st*) $f m = s \longrightarrow$
import-flr fupd g s ((*field-lookup-struct st f m*)::('a,'b) *flr*)) \wedge
($\forall f m s.$ *field-lookup-list* (*map-td-list fupd g ts*) $f m = s \longrightarrow$
import-flr fupd g s ((*field-lookup-list ts f m*)::('a,'b) *flr*)) \wedge
($\forall f m s.$ *field-lookup-tuple* (*map-td-tuple fupd g x*) $f m = s \longrightarrow$
import-flr fupd g s ((*field-lookup-tuple x f m*)::('a,'b) *flr*))
by (*auto simp: field-lookup-export-uinfo-rev''*)

lemma *field-lookup-export-uinfo-Some-rev*:

field-lookup (*export-uinfo* ($t::('a,'b)$ *typ-info*)) $f m = \text{Some } (s,n) \implies$
 $\exists k.$ *field-lookup t f m* = *Some* (k,n) \wedge *export-uinfo k* = s
apply(*insert field-lookup-export-uinfo-rev'* [*of field-norm* ($\lambda.$ ()) *t undefined un-*
defined undefined])
apply *clarsimp*
apply(*drule spec* [**where** $x=f$])
apply(*drule spec* [**where** $x=m$])
apply(*clarsimp simp: import-flr-def export-uinfo-def split: option.splits*)
done

lemma (**in** *c-type*) *field-lookup-uinfo-Some-rev*:

field-lookup (*typ-uinfo-t* (*TYPE*('a))) $f m = \text{Some } (s,n) \implies$
 $\exists k.$ *field-lookup* (*typ-info-t* (*TYPE*('a))) $f m = \text{Some } (k,n) \wedge$ *export-uinfo k*
 $= s$
apply (*simp add: typ-uinfo-t-def*)
apply (*rule field-lookup-export-uinfo-Some-rev*)
apply *assumption*
done

lemma (**in** *c-type*) *field-lookup-offset-untyped-eq* [*simp*]:

field-lookup (*typ-info-t* *TYPE*('a)) $f 0 = \text{Some } (s,n) \implies$
field-offset-untyped (*typ-uinfo-t* *TYPE*('a)) $f = n$
apply(*simp add: field-offset-untyped-def typ-uinfo-t-def*)
apply(*drule field-lookup-export-uinfo-Some*)
apply(*simp add: typ-uinfo-t-def export-uinfo-def*)
done

lemma (**in** *c-type*) *field-lookup-offset-eq* [*simp*]:

field-lookup (*typ-info-t* *TYPE*('a)) $f 0 = \text{Some } (s,n) \implies$
field-offset *TYPE*('a) $f = n$
by(*simp add: field-offset-def*)

lemma *field-offset-self* [*simp*]:

field-offset t [] = 0

by (simp add: field-offset-def field-offset-untyped-def)

lemma (in *c-type*) *field-ti-self* [simp]:
field-ti TYPE('a) [] = Some (typ-info-t TYPE('a))
 by (simp add: field-ti-def)

lemma (in *c-type*) *field-size-self* [simp]:
field-size TYPE('a) [] = size-td (typ-info-t TYPE('a))
 by (simp add: field-size-def)

lemma *field-lookup-prefix-None''*:
 (∀ f g m. *field-lookup* (t::('a,'b) typ-desc) f m = None → *field-lookup* t (f@g) m = None)
 (∀ f g m. *field-lookup-struct* (st::('a,'b) typ-struct) f m = None → f ≠ [] → *field-lookup-struct* st (f@g) m = None)
 (∀ f g m. *field-lookup-list* (ts::('a,'b) typ-tuple list) f m = None → f ≠ [] → *field-lookup-list* ts (f@g) m = None)
 (∀ f g m. *field-lookup-tuple* (x::('a,'b) typ-tuple) f m = None → f ≠ [] → *field-lookup-tuple* x (f@g) m = None)
 by (induct t and st and ts and x)
 (clarsimp split: option.splits)+

lemma *field-lookup-prefix-None'*:
 (∀ f g m. *field-lookup* (t::('a,'b) typ-desc) f m = None → *field-lookup* t (f@g) m = None) ∧
 (∀ f g m. *field-lookup-struct* (st::('a,'b) typ-struct) f m = None → f ≠ [] → *field-lookup-struct* st (f@g) m = None) ∧
 (∀ f g m. *field-lookup-list* (ts::('a,'b) typ-tuple list) f m = None → f ≠ [] → *field-lookup-list* ts (f@g) m = None) ∧
 (∀ f g m. *field-lookup-tuple* (x::('a,'b) typ-tuple) f m = None → f ≠ [] → *field-lookup-tuple* x (f@g) m = None)
 by (auto simp: field-lookup-prefix-None'')

lemma *field-lookup-prefix-Some''*:
 (∀ f g t' m n. *field-lookup* t f m = Some ((t':('a,'b) typ-desc),n) → wf-desc t →
 field-lookup t (f@g) m = *field-lookup* t' g n)
 (∀ f g t' m n. *field-lookup-struct* st f m = Some ((t':('a,'b) typ-desc),n) → wf-desc-struct st →
 field-lookup-struct st (f@g) m = *field-lookup* t' g n)
 (∀ f g t' m n. *field-lookup-list* ts f m = Some ((t':('a,'b) typ-desc),n) → wf-desc-list ts →
 field-lookup-list ts (f@g) m = *field-lookup* t' g n)
 (∀ f g t' m n. *field-lookup-tuple* x f m = Some ((t':('a,'b) typ-desc),n) → wf-desc-tuple x →
 field-lookup-tuple x (f@g) m = *field-lookup* t' g n)

proof(induct t and st and ts and x)
 case (TypDesc nat typ-struct list)

```

then show ?case by auto
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply(clarsimp split: option.splits)
    apply(cases dt-tuple)
    subgoal for f
    apply(cases f, clarsimp)
    by (clarsimp simp: field-lookup-list-None split: if-split-asm)
    done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by auto
qed

```

lemma *field-lookup-prefix-Some'*:

```

(∀ f g t' m n. field-lookup t f m = Some ((t'::('a, 'b) typ-desc), n) → wf-desc t
→
  field-lookup t (f@g) m = field-lookup t' g n) ∧
(∀ f g t' m n. field-lookup-struct st f m = Some ((t'::('a, 'b) typ-desc), n) →
wf-desc-struct st →
  field-lookup-struct st (f@g) m = field-lookup t' g n) ∧
(∀ f g t' m n. field-lookup-list ts f m = Some ((t'::('a, 'b) typ-desc), n) →
wf-desc-list ts →
  field-lookup-list ts (f@g) m = field-lookup t' g n) ∧
(∀ f g t' m n. field-lookup-tuple x f m = Some ((t'::('a, 'b) typ-desc), n) →
wf-desc-tuple x →
  field-lookup-tuple x (f@g) m = field-lookup t' g n)
by (auto simp: field-lookup-prefix-Some'')

```

lemma *field-lookup-append-eq*:

```

wf-desc t ⇒
  field-lookup t (f @ g) n =
  Option.bind (field-lookup t f n) (λ(t', m). field-lookup t' g m)
by (cases field-lookup t f n)
  (auto simp add: field-lookup-prefix-None''(1)[rule-format]
  field-lookup-prefix-Some''(1)[rule-format])

```

lemma *field-lookup-offset-shift*:

NO-MATCH $0\ m \implies \text{field-lookup } t\ f\ 0 = \text{Some } (t', n) \implies \text{field-lookup } t\ f\ m = \text{Some } (t', m + n)$

by (*simp* *add*: *field-lookup-offset*)

lemma *field-lookup-offset-shift'*:

$\text{field-lookup } t\ f\ m = \text{Some } (s, k) \implies \text{field-lookup } t\ f\ n = \text{Some } (s, k + n - m)$

by (*metis* *CTypes.field-lookup-offset2* *Nat.add-diff-assoc2* *add.commute* *field-lookup-offset* *field-lookup-offset-le(1)*)

lemma *field-lookup-append*:

assumes *t*: *wf-desc* *t*

and *f*: $\text{field-lookup } t\ f\ n = \text{Some } (u, m)$ **and** *g*: $\text{field-lookup } u\ g\ l = \text{Some } (v, k)$

shows $\text{field-lookup } t\ (f\ @\ g)\ n = \text{Some } (v, k + m - l)$

using *field-lookup-prefix-Some''(1)*[*rule-format*, *OF* *f* *t*, *of* *g*]

g[*THEN* *field-lookup-offset-shift'*, *of* *m*]

by *simp*

lemma *field-lvalue-empty-simp* [*simp*]:

Ptr $\&(p \rightarrow []) = p$

by (*simp* *add*: *field-lvalue-def* *field-offset-def* *field-offset-untyped-def*)

lemma *map-td-align* [*simp*]:

$\text{align-td-wo-align } (\text{map-td } f\ g\ t) = \text{align-td-wo-align } (t::('a,'b)\ \text{typ-desc})$

$\text{align-td-wo-align-struct } (\text{map-td-struct } f\ g\ st) = \text{align-td-wo-align-struct } (st::('a,'b)\ \text{typ-struct})$

$\text{align-td-wo-align-list } (\text{map-td-list } f\ g\ ts) = \text{align-td-wo-align-list } (ts::('a,'b)\ \text{typ-tuple list})$

$\text{align-td-wo-align-tuple } (\text{map-td-tuple } f\ g\ x) = \text{align-td-wo-align-tuple } (x::('a,'b)\ \text{typ-tuple})$

by (*induct* *t* **and** *st* **and** *ts* **and** *x*) *auto*

lemma *map-td-align'* [*simp*]:

$\text{align-td } (\text{map-td } f\ g\ t) = \text{align-td } (t::('a,'b)\ \text{typ-desc})$

$\text{align-td-struct } (\text{map-td-struct } f\ g\ st) = \text{align-td-struct } (st::('a,'b)\ \text{typ-struct})$

$\text{align-td-list } (\text{map-td-list } f\ g\ ts) = \text{align-td-list } (ts::('a,'b)\ \text{typ-tuple list})$

$\text{align-td-tuple } (\text{map-td-tuple } f\ g\ x) = \text{align-td-tuple } (x::('a,'b)\ \text{typ-tuple})$

by (*induct* *t* **and** *st* **and** *ts* **and** *x*) *auto*

lemma *typ-uinfo-align* [*simp*]:

$\text{align-td-wo-align } (\text{export-uinfo } t) = \text{align-td-wo-align } (t::('a,'b)\ \text{typ-info})$

by (*simp* *add*: *export-uinfo-def*)

lemma *ptr-aligned-Ptr-0* [*simp*]:

ptr-aligned *NULL*

by (*simp* *add*: *ptr-aligned-def*)

lemma *td-set-self* [*simp*]:

$(t, m) \in \text{td-set } t\ m$

by (*cases* *t*) *simp*

lemma *td-set-wf-size-desc* [rule-format]:
 $(\forall s m n. \text{wf-size-desc } t \longrightarrow ((s::('a,'b) \text{ typ-desc}),m) \in \text{td-set } t n \longrightarrow \text{wf-size-desc } s)$
 $(\forall s m n. \text{wf-size-desc-struct } st \longrightarrow ((s::('a,'b) \text{ typ-desc}),m) \in \text{td-set-struct } st n \longrightarrow \text{wf-size-desc } s)$
 $(\forall s m n. \text{wf-size-desc-list } ts \longrightarrow ((s::('a,'b) \text{ typ-desc}),m) \in \text{td-set-list } ts n \longrightarrow \text{wf-size-desc } s)$
 $(\forall s m n. \text{wf-size-desc-tuple } x \longrightarrow ((s::('a,'b) \text{ typ-desc}),m) \in \text{td-set-tuple } x n \longrightarrow \text{wf-size-desc } s)$
by (*induct t and st and ts and x*) *force+*

lemma *td-set-size-lte'*:
 $(\forall s k m. ((s::('a,'b) \text{ typ-desc}),k) \in \text{td-set } t m \longrightarrow \text{size } s = \text{size } t \wedge s=t \wedge k=m \vee \text{size } s < \text{size } t)$
 $(\forall s k m. ((s::('a,'b) \text{ typ-desc}),k) \in \text{td-set-struct } st m \longrightarrow \text{size } s < \text{size } st)$
 $(\forall s k m. ((s::('a,'b) \text{ typ-desc}),k) \in \text{td-set-list } xs m \longrightarrow \text{size } s < \text{size-list } (\text{size-dt-tuple size } (\lambda-. 0) (\lambda-. 0)) xs)$
 $(\forall s k m. ((s::('a,'b) \text{ typ-desc}),k) \in \text{td-set-tuple } x m \longrightarrow \text{size } s < \text{size-dt-tuple size } (\lambda-. 0) (\lambda-. 0) x)$
by (*induct t and st and xs and x*) *force+*

lemma *td-set-size-lte*:
 $(s,k) \in \text{td-set } t m \implies \text{size } s = \text{size } t \wedge s=t \wedge k=m \vee \text{size } s < \text{size } t$
by (*simp add: td-set-size-lte'*)

lemma *td-set-struct-size-lte*:
 $(s,k) \in \text{td-set-struct } st m \implies \text{size } s < \text{size } st$
by (*simp add: td-set-size-lte'*)

lemma *td-set-list-size-lte*:
 $(s,k) \in \text{td-set-list } ts m \implies \text{size } s < \text{size-list } (\text{size-dt-tuple size } (\lambda-. 0) (\lambda-. 0)) ts$
by (*simp add: td-set-size-lte'*)

lemma *td-aggregate-not-in-td-set-list* [simp]:
 $\neg (\text{TypDesc } \text{align } (\text{TypAggregate } xs) \text{tn},k) \in \text{td-set-list } xs m$
by (*fastforce dest: td-set-list-size-lte simp: size-char-def*)

lemma *sub-size-td*:
 $(s::('a,'b) \text{ typ-desc}) \leq t \implies \text{size } s \leq \text{size } t$
by (*fastforce dest: td-set-size-lte simp: typ-tag-le-def*)

lemma *sub-tag-antisym*:
 $\llbracket (s::('a,'b) \text{ typ-desc}) \leq t; t \leq s \rrbracket \implies s=t$
apply(*frule sub-size-td*)
apply(*frule sub-size-td [where t=s]*)
apply(*clarsimp simp: typ-tag-le-def*)
apply(*drule td-set-size-lte*)

apply *clarsimp*
done

lemma *sub-tag-refl*:
 $(s::('a,'b) \text{ typ-desc}) \leq s$
unfolding *typ-tag-le-def* **by** (*cases s, fastforce*)

lemma *sub-tag-sub'*:
 $\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set } t m \longrightarrow \text{td-set } s n \subseteq \text{td-set } t m$
 $\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set-struct } ts m \longrightarrow \text{td-set } s n \subseteq \text{td-set-struct } ts m$
 $\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set-list } xs m \longrightarrow \text{td-set } s n \subseteq \text{td-set-list } xs m$
 $\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set-tuple } x m \longrightarrow \text{td-set } s n \subseteq \text{td-set-tuple } x m$
by (*induct t and ts and xs and x*) *fastforce+*

lemma *sub-tag-sub*:
 $(s,n) \in \text{td-set } t m \implies \text{td-set } s n \subseteq \text{td-set } t m$
by (*simp add: sub-tag-sub'*)

lemma *td-set-fst*:
 $\forall m n. \text{fst } ' \text{td-set } (s::('a, 'b) \text{ typ-desc}) m = \text{fst } ' \text{td-set } s n$
 $\forall m n. \text{fst } ' \text{td-set-struct } (st::('a,'b) \text{ typ-struct}) m = \text{fst } ' \text{td-set-struct } st n$
 $\forall m n. \text{fst } ' \text{td-set-list } (xs::('a, 'b) \text{ typ-tuple list}) m = \text{fst } ' \text{td-set-list } xs n$
 $\forall m n. \text{fst } ' \text{td-set-tuple } (x::('a, 'b) \text{ typ-tuple}) m = \text{fst } ' \text{td-set-tuple } x n$
by (*induct s and st and xs and x*) (*all <clarsimp>, fast, fast*)

lemma *sub-tag-trans*:
 $\llbracket (s::('a,'b) \text{ typ-desc}) \leq t; t \leq u \rrbracket \implies s \leq u$
apply(*clarsimp simp: typ-tag-le-def*)
by (*meson sub-tag-sub'(1) subset-iff td-set-offset-0-conv*)

instantiation *typ-desc* :: (*type, type*) *order*

begin

instance

apply *intro-classes*
apply(*fastforce simp: typ-tag-lt-def typ-tag-le-def dest: td-set-size-lte*)
apply(*rule sub-tag-refl*)
apply(*erule (1) sub-tag-trans*)
apply(*erule (1) sub-tag-antisym*)
done

end

lemma *td-set-offset-size''*:
 $\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set } t m \longrightarrow \text{size-td } s + (n - m) \leq \text{size-td } t$
 $\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set-struct } st m \longrightarrow \text{size-td } s + (n - m) \leq$

size-td-struct st
 $\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set-list } ts \ m \longrightarrow \text{size-td } s + (n - m) \leq \text{size-td-list } ts$
size-td-list ts
 $\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set-tuple } x \ m \longrightarrow \text{size-td } s + (n - m) \leq \text{size-td-tuple } x$
proof (*induct t and st and ts and x*)
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply (*cases dt-tuple*)
apply *fastforce*
done
qed *auto*

lemma *td-set-offset-size'*:
 $(\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set } t \ m \longrightarrow \text{size-td } s + (n - m) \leq \text{size-td } t) \wedge$
 $(\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set-struct } st \ m \longrightarrow \text{size-td } s + (n - m) \leq \text{size-td-struct } st) \wedge$
 $(\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set-list } ts \ m \longrightarrow \text{size-td } s + (n - m) \leq \text{size-td-list } ts) \wedge$
 $(\forall s m n. ((s::('a,'b) \text{ typ-desc}),n) \in \text{td-set-tuple } x \ m \longrightarrow \text{size-td } s + (n - m) \leq \text{size-td-tuple } x)$
by (*auto simp: td-set-offset-size''*)

lemma *td-set-offset-size*:
 $(s,n) \in \text{td-set } t \ 0 \Longrightarrow \text{size-td } s + n \leq \text{size-td } t$
using *td-set-offset-size'* [*of t undefined undefined undefined*] **by** *fastforce*

lemma *td-set-struct-offset-size*:
 $(s,n) \in \text{td-set-struct } st \ m \Longrightarrow \text{size-td } s + (n - m) \leq \text{size-td-struct } st$
using *td-set-offset-size'* [*of undefined st undefined undefined*] **by** *clarsimp*

lemma *td-set-list-offset-size*:
 $(s,n) \in \text{td-set-list } ts \ 0 \Longrightarrow \text{size-td } s + n \leq \text{size-td-list } ts$
using *td-set-offset-size'* [*of undefined undefined ts undefined*] **by** *fastforce*

lemma *td-set-offset-size-m*:
 $(s,n) \in \text{td-set } t \ m \Longrightarrow \text{size-td } s + (n - m) \leq \text{size-td } t$
using *insert td-set-offset-size'* [*of t undefined undefined undefined*] **by** *clarsimp*

lemma *td-set-list-offset-size-m*:
 $(s,n) \in \text{td-set-list } t \ m \Longrightarrow \text{size-td } s + (n - m) \leq \text{size-td-list } t$
using *insert td-set-offset-size'* [*of undefined undefined t undefined*] **by** *clarsimp*

lemma *td-set-tuple-offset-size-m*:
 $(s,n) \in \text{td-set-tuple } t \ m \Longrightarrow \text{size-td } s + (n - m) \leq \text{size-td-tuple } t$

using *td-set-offset-size'* [of undefined undefined undefined *t*]
by *clarsimp*

lemma *td-set-list-offset-le*:
 $(s,n) \in \text{td-set-list } ts \ m \implies m \leq n$
by (*simp add: td-set-offset-le'*)

lemma *td-set-tuple-offset-le*:
 $(s,n) \in \text{td-set-tuple } ts \ m \implies m \leq n$
by (*simp add: td-set-offset-le'*)

lemma *field-of-self* [*simp*]:
field-of 0 *t t*
by (*simp add: field-of-def*)

lemma *td-set-export-uinfo'*:
 $\forall f \ m \ n \ s. ((s::('a,'b) \text{typ-info}),n) \in \text{td-set } t \ m \longrightarrow$
 $(\text{export-uinfo } s,n) \in \text{td-set } (\text{export-uinfo } t) \ m$
 $\forall f \ m \ n \ s. ((s::('a,'b) \text{typ-info}),n) \in \text{td-set-struct } st \ m \longrightarrow$
 $(\text{export-uinfo } s,n) \in \text{td-set-struct } (\text{map-td-struct field-norm } (\lambda-. ()) \ st) \ m$
 $\forall f \ m \ n \ s. ((s::('a,'b) \text{typ-info}),n) \in \text{td-set-list } ts \ m \longrightarrow$
 $(\text{export-uinfo } s,n) \in \text{td-set-list } (\text{map-td-list field-norm } (\lambda-. ()) \ ts) \ m$
 $\forall f \ m \ n \ s. ((s::('a,'b) \text{typ-info}),n) \in \text{td-set-tuple } x \ m \longrightarrow$
 $(\text{export-uinfo } s,n) \in \text{td-set-tuple } (\text{map-td-tuple field-norm } (\lambda-. ()) \ x) \ m$
proof (*induct t and st and ts and x*)
case (*Cons-typ-desc dt-tuple list*)
then show ?*case* **by**(*cases dt-tuple*) (*simp add: export-uinfo-def*)
qed (*auto simp add: export-uinfo-def*)

lemma *td-set-export-uinfo*:
 $(\forall f \ m \ n \ s. ((s::('a,'b) \text{typ-info}),n) \in \text{td-set } t \ m \longrightarrow$
 $(\text{export-uinfo } s,n) \in \text{td-set } (\text{export-uinfo } t) \ m) \wedge$
 $(\forall f \ m \ n \ s. ((s::('a,'b) \text{typ-info}),n) \in \text{td-set-struct } st \ m \longrightarrow$
 $(\text{export-uinfo } s,n) \in \text{td-set-struct } (\text{map-td-struct field-norm } (\lambda-. ()) \ st) \ m) \wedge$
 $(\forall f \ m \ n \ s. ((s::('a,'b) \text{typ-info}),n) \in \text{td-set-list } ts \ m \longrightarrow$
 $(\text{export-uinfo } s,n) \in \text{td-set-list } (\text{map-td-list field-norm } (\lambda-. ()) \ ts) \ m) \wedge$
 $(\forall f \ m \ n \ s. ((s::('a,'b) \text{typ-info}),n) \in \text{td-set-tuple } x \ m \longrightarrow$
 $(\text{export-uinfo } s,n) \in \text{td-set-tuple } (\text{map-td-tuple field-norm } (\lambda-. ()) \ x) \ m)$
by (*auto simp: td-set-export-uinfo'*)

lemma *td-set-export-uinfoD*:
 $(s,n) \in \text{td-set } t \ m \implies (\text{export-uinfo } s,n) \in \text{td-set } (\text{export-uinfo } t) \ m$
using *td-set-export-uinfo* [of *t* undefined undefined undefined] **by** *clarsimp*

lemma *td-set-field-lookup''*:
 $\forall s \ m \ n. \text{wf-desc } t \longrightarrow (((s::('a,'b) \text{typ-desc}),m + n) \in \text{td-set } t \ m \longrightarrow$

```

    (∃ f. field-lookup t f m = Some (s,m+n))
  ∀ s m n. wf-desc-struct st → (((s::('a,'b) typ-desc),m + n) ∈ td-set-struct st m
→
  (∃ f. field-lookup-struct st f m = Some (s,m+n)))
  ∀ s m n. wf-desc-list ts → (((s::('a,'b) typ-desc),m + n) ∈ td-set-list ts m →
  (∃ f. field-lookup-list ts f m = Some (s,m+n)))
  ∀ s m n. wf-desc-tuple x → (((s::('a,'b) typ-desc),m + n) ∈ td-set-tuple x m →
  (∃ f. field-lookup-tuple x f m = Some (s,m+n)))
proof (induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case
  apply clarsimp
  subgoal for s m n
  apply(cases s, clarsimp)
  apply (rule conjI)
  apply clarsimp
  apply(rule exI [where x=[]])
  apply clarsimp+
  apply((erule allE)+, erule (1) impE)
  apply clarsimp
  subgoal for - - - f
  apply(cases f, clarsimp+)
  subgoal for x xs
  apply(rule exI [where x=x#xs])
  apply clarsimp
  done
  done
  done
  done
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
  apply clarsimp
  apply(rule conjI, clarsimp)
  apply((erule allE)+, erule (1) impE)
  apply(clarsimp)

  subgoal for s m n f
  apply (cases f, clarsimp+)
  subgoal for x xs

```

```

    apply(rule exI [where x=x#xs])
    apply(clarsimp)
  done
done
apply clarsimp
subgoal premises prems for s m n
  using prems(1-3,6)
  prems(5)[rule-format, of s m + size-td (dtfst dt-tuple) n - size-td (dtfst
dt-tuple)]
  apply -
  apply(frul td-set-list-offset-le)
  apply clarsimp
  subgoal for f
    apply(rule exI [where x=f])
    apply(clarsimp split: option.splits)
    apply(cases dt-tuple, clarsimp split: if-split-asm)
    apply(cases f, clarsimp+)
    apply(simp add: field-lookup-list-None)
  done
done
done
next
case (DTuple-tyt-desc tyt-desc list b)
then show ?case
  apply clarsimp
  apply((erule allE)+, erule (1) impE)
  apply clarsimp
  subgoal for s m n f
    apply(rule exI [where x=list#f])
    apply clarsimp
  done
done
qed

```

lemma *td-set-field-lookup'*:

$$\begin{aligned}
& (\forall s m n. \text{wf-desc } t \longrightarrow (((s::('a,'b) \text{typ-desc}), m + n) \in \text{td-set } t m \longrightarrow \\
& \quad (\exists f. \text{field-lookup } t f m = \text{Some } (s, m+n)))) \wedge \\
& (\forall s m n. \text{wf-desc-struct } st \longrightarrow (((s::('a,'b) \text{typ-desc}), m + n) \in \text{td-set-struct } st \\
& m \longrightarrow \\
& \quad (\exists f. \text{field-lookup-struct } st f m = \text{Some } (s, m+n)))) \wedge \\
& (\forall s m n. \text{wf-desc-list } ts \longrightarrow (((s::('a,'b) \text{typ-desc}), m + n) \in \text{td-set-list } ts m \longrightarrow \\
& \quad (\exists f. \text{field-lookup-list } ts f m = \text{Some } (s, m+n)))) \wedge \\
& (\forall s m n. \text{wf-desc-tuple } x \longrightarrow (((s::('a,'b) \text{typ-desc}), m + n) \in \text{td-set-tuple } x m \\
& \longrightarrow \\
& \quad (\exists f. \text{field-lookup-tuple } x f m = \text{Some } (s, m+n)))) \\
& \text{by (auto simp: td-set-field-lookup')}
\end{aligned}$$

lemma *td-set-field-lookup-rev''*:

```

  ∀ s m n. (∃ f. field-lookup t f m = Some (s,m+n)) →
    ((s::('a,'b) typ-desc),m + n) ∈ td-set t m
  ∀ s m n. (∃ f. field-lookup-struct st f m = Some (s,m+n)) →
    ((s::('a,'b) typ-desc),m + n) ∈ td-set-struct st m
  ∀ s m n. (∃ f. field-lookup-list ts f m = Some (s,m+n)) →
    ((s::('a,'b) typ-desc),m + n) ∈ td-set-list ts m
  ∀ s m n. (∃ f. field-lookup-tuple x f m = Some (s,m+n)) →
    ((s::('a,'b) typ-desc),m + n) ∈ td-set-tuple x m
proof(induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case
  apply clarsimp
  subgoal for s m n f
  apply(cases f, clarsimp)
  apply clarsimp
  apply((erule allE)+, erule impE, fast)
  apply fast
  done
  done
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
  apply clarsimp
  apply(clarsimp split: option.splits)
  subgoal premises prems for s m n f
  using prems (1, 4-5)
  prems(3)[rule-format, where s = s and m = m + size-td (dt-fst dt-tuple)]
and n = n - size-td (dt-fst dt-tuple)]
  apply -

  apply(frule field-lookup-offset-le)
  apply clarsimp
  apply fast
  done
  subgoal by auto
  done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
  apply clarsimp

```

subgoal for $s m n f$
apply(*cases f, clarsimp+*)
apply((*erule allE*)+, *erule impE, fast*)
apply *assumption*
done
done
qed

lemma *td-set-field-lookup-rev'*:
 $(\forall s m n. (\exists f. \text{field-lookup } t f m = \text{Some } (s, m+n)) \longrightarrow$
 $((s::('a, 'b) \text{typ-desc}), m + n) \in \text{td-set } t m) \wedge$
 $(\forall s m n. (\exists f. \text{field-lookup-struct } st f m = \text{Some } (s, m+n)) \longrightarrow$
 $((s::('a, 'b) \text{typ-desc}), m + n) \in \text{td-set-struct } st m) \wedge$
 $(\forall s m n. (\exists f. \text{field-lookup-list } ts f m = \text{Some } (s, m+n)) \longrightarrow$
 $((s::('a, 'b) \text{typ-desc}), m + n) \in \text{td-set-list } ts m) \wedge$
 $(\forall s m n. (\exists f. \text{field-lookup-tuple } x f m = \text{Some } (s, m+n)) \longrightarrow$
 $((s::('a, 'b) \text{typ-desc}), m + n) \in \text{td-set-tuple } x m)$
by (*auto simp: td-set-field-lookup-rev'*)

lemma *td-set-field-lookup*:
 $wf\text{-desc } t \Longrightarrow k \in \text{td-set } t 0 = (\exists f. \text{field-lookup } t f 0 = \text{Some } k)$
using *td-set-field-lookup'* [*of t undefined undefined undefined*]
td-set-field-lookup-rev' [*of t undefined undefined undefined*]
apply *clarsimp*
apply(*cases k, clarsimp*)
by (*metis add-0*)

lemma *td-set-field-lookupD*:
 $\text{field-lookup } t f m = \text{Some } k \Longrightarrow k \in \text{td-set } t m$
using *td-set-field-lookup-rev'* [*of t undefined undefined undefined*]
apply(*cases k, clarsimp*)
by (*metis field-lookup-offset-le(1) le-iff-add*)

lemma *td-set-struct-field-lookup-structD*:
 $\text{field-lookup-struct } st f m = \text{Some } k \Longrightarrow k \in \text{td-set-struct } st m$
using *td-set-field-lookup-rev'* [*of undefined st undefined undefined*]
apply(*cases k, clarsimp*)
by (*metis field-lookup-offset-le(2) le-iff-add*)

lemma *field-lookup-struct-td-simp* [*simp*]:
 $\text{field-lookup-struct } ts f m \neq \text{Some } (\text{TypDesc } \text{algn } ts \text{ nm}, m)$
by (*fastforce dest: td-set-struct-field-lookup-structD td-set-struct-size-lte*)

lemma *td-set-list-field-lookup-listD*:
 $\text{field-lookup-list } xs f m = \text{Some } k \Longrightarrow k \in \text{td-set-list } xs m$
using *td-set-field-lookup-rev'* [*of undefined undefined xs undefined*]
apply(*cases k, clarsimp*)
by (*metis field-lookup-offset-le(3) le-Suc-ex*)

lemma *td-set-tuple-field-lookup-tupleD*:

field-lookup-tuple x f m = Some k $\implies k \in \text{td-set-tuple } x \ m$
using *td-set-field-lookup-rev'* [of undefined undefined undefined *x*]
apply(*cases k, clarsimp*)
by (*metis field-lookup-offset-le(4) nat-le-iff-add*)

lemma *field-lookup-offset-size''*:

field-lookup t f n = Some (u, m) $\implies \text{size-td } u + m \leq \text{size-td } t + n$
by (*metis Nat.add-diff-assoc field-lookup-offset-le(1)*
le-diff-conv td-set-field-lookupD td-set-offset-size')

lemma *field-lookup-offset-size'*:

assumes *n*: *field-lookup t f 0 = Some (t',n)* **shows** *size-td t' + n* \leq *size-td t*
using *field-lookup-offset-size''[OF n]* **by** *simp*

lemma *intvl-subset-of-field-lookup*:

field-lookup t f 0 = Some (u, n) \implies
 $\{a + \text{of-nat } n \ .. + \text{size-td } u\} \subseteq \{a \ .. + \text{size-td } t\}$
by (*simp add: field-lookup-offset-size' intvl-le*)

lemma *field-lookup-wf-size-desc-gt*:

$\llbracket \text{field-lookup } t \ f \ n = \text{Some } (a,b); \text{wf-size-desc } t \rrbracket \implies 0 < \text{size-td } a$
by (*fastforce simp: td-set-wf-size-desc wf-size-desc-gt dest!: td-set-field-lookupD*)

lemma *field-lookup-inject''*:

$\forall f \ g \ m \ s. \text{wf-size-desc } t \longrightarrow \text{field-lookup } (t::('a,'b) \text{ typ-desc}) \ f \ m = \text{Some } s \wedge$
field-lookup t g m = Some s $\longrightarrow f=g$
 $\forall f \ g \ m \ s. \text{wf-size-desc-struct } st \longrightarrow \text{field-lookup-struct } (st::('a,'b) \text{ typ-struct}) \ f \ m$
 $= \text{Some } s \wedge \text{field-lookup-struct } st \ g \ m = \text{Some } s \longrightarrow f=g$
 $\forall f \ g \ m \ s. \text{wf-size-desc-list } ts \longrightarrow \text{field-lookup-list } (ts::('a,'b) \text{ typ-tuple list}) \ f \ m =$
 $\text{Some } s \wedge \text{field-lookup-list } ts \ g \ m = \text{Some } s \longrightarrow f=g$
 $\forall f \ g \ m \ s. \text{wf-size-desc-tuple } x \longrightarrow \text{field-lookup-tuple } (x::('a,'b) \text{ typ-tuple}) \ f \ m =$
 $\text{Some } s \wedge \text{field-lookup-tuple } x \ g \ m = \text{Some } s \longrightarrow f=g$

proof(*induct t and st and ts and x*)

case (*TypDesc nat typ-struct list*)
then show *?case by clarsimp fast*

next

case (*TypScalar nat1 nat2 a*)
then show *?case by auto*

next

case (*TypAggregate list*)
then show *?case by auto*

next

case *Nil-typ-desc*
then show *?case by auto*

next

case (*Cons-typ-desc dt-tuple list*)
then show *?case*


```

apply clarsimp
subgoal for f g m a b
  apply(clarsimp split: option.splits)
    apply fast
  subgoal
    apply(frule td-set-tuple-field-lookup-tupleD)
    apply(drule field-lookup-offset-le)
    apply(drule td-set-tuple-offset-size-m)
  subgoal premises prems
    using prems(3-8)
    apply(cases dt-tuple, simp)
  subgoal premises prems
  proof -
    have  $0 < \text{size-td } a$ 
      using prems
    apply -
    apply(clarsimp split: if-split-asm; drule (2) field-lookup-wf-size-desc-gt)
    done
  then show ?thesis
    using prems
    by simp
  qed
  done
done
subgoal
  apply(frule td-set-tuple-field-lookup-tupleD)
  apply(drule field-lookup-offset-le)
  apply(drule td-set-tuple-offset-size-m)
  subgoal premises prems
    using prems(3-8)
    apply(cases dt-tuple, simp)
  subgoal premises prems
  proof -
    have  $0 < \text{size-td } a$ 
      using prems
    apply -
    apply(clarsimp split: if-split-asm; drule (2) field-lookup-wf-size-desc-gt)
    done
  then show ?thesis
    using prems
    by simp
  qed
  done
done
subgoal by best
done
done
next
  case (DTuple-typ-desc typ-desc list b)

```

```

then show ?case
  apply clarsimp
  subgoal for f g m a b
    apply (drule spec [where x=tl f])
    apply (drule spec [where x=tl g])
    apply (cases f; simp)
    apply (cases g; simp)
    apply fastforce
  done
done
qed

```

```

lemma field-lookup-inject':
  ( $\forall f g m s. wf\text{-size}\text{-desc } t \longrightarrow field\text{-lookup } (t::('a,'b) \text{ typ-desc}) f m = Some s \wedge field\text{-lookup } t g m = Some s \longrightarrow f=g$ )  $\wedge$ 
  ( $\forall f g m s. wf\text{-size}\text{-desc}\text{-struct } st \longrightarrow field\text{-lookup}\text{-struct } (st::('a,'b) \text{ typ-struct}) f m = Some s \wedge field\text{-lookup}\text{-struct } st g m = Some s \longrightarrow f=g$ )  $\wedge$ 
  ( $\forall f g m s. wf\text{-size}\text{-desc}\text{-list } ts \longrightarrow field\text{-lookup}\text{-list } (ts::('a,'b) \text{ typ-tuple list}) f m = Some s \wedge field\text{-lookup}\text{-list } ts g m = Some s \longrightarrow f=g$ )  $\wedge$ 
  ( $\forall f g m s. wf\text{-size}\text{-desc}\text{-tuple } x \longrightarrow field\text{-lookup}\text{-tuple } (x::('a,'b) \text{ typ-tuple}) f m = Some s \wedge field\text{-lookup}\text{-tuple } x g m = Some s \longrightarrow f=g$ )
  by (auto simp: field-lookup-inject'')

```

```

lemma field-lookup-inject:
  [ $field\text{-lookup } (t::('a,'b) \text{ typ-desc}) f m = Some s;$ 
   $field\text{-lookup } t g m = Some s; wf\text{-size}\text{-desc } t$ ]  $\implies f=g$ 
  using field-lookup-inject' [of t undefined undefined undefined]
  apply (cases s)
  apply clarsimp
  apply (drule spec [where x=f])
  apply (drule spec [where x=g])
  apply fast
  done

```

```

lemma fd-cons-update-normalise:
  [ $fd\text{-cons}\text{-update}\text{-access } d n; fd\text{-cons}\text{-access}\text{-update } d n;$ 
   $fd\text{-cons}\text{-double}\text{-update } d; fd\text{-cons}\text{-length } d n$ ]  $\implies$ 
   $fd\text{-cons}\text{-update}\text{-normalise } d n$ 
  apply (clarsimp simp: fd-cons-update-access-def fd-cons-update-normalise-def norm-desc-def)
  subgoal for v bs
    apply (drule spec [where x=field-update d bs v])
    apply (drule spec [where x=replicate (length bs) 0])
    apply clarsimp
    apply (clarsimp simp: fd-cons-access-update-def)
    apply (drule spec [where x=bs])
    apply clarsimp
    apply (drule spec [where x=replicate (length bs) 0])
    apply clarsimp
  done

```

```

apply(drule spec [where  $x=v$ ])
apply(drule spec [where  $x=undefined$ ])
apply clarsimp
apply(clarsimp simp: fd-cons-double-update-def)
apply(drule spec [where  $x=v$ ])
apply(drule spec [where  $x=field-access\ d\ (field-update\ d\ bs\ undefined)$ 
                    (replicate (length bs) 0)])
apply(drule spec [where  $x=bs$ ])
apply clarsimp
apply(erule impE)
apply(clarsimp simp: fd-cons-length-def)
apply clarsimp
done
done

```

lemma *update-ti-t-update-ti-struct-t* [*simp*]:
update-ti-t (*TypDesc* *algn st tn*) = *update-ti-struct-t st*
by (*auto simp: update-ti-t-def update-ti-struct-t-def*)

lemma *fd-cons-fd-cons-struct* [*simp*]:
fd-cons (*TypDesc* *algn st tn*) = *fd-cons-struct st*
by (*clarsimp simp: fd-cons-def fd-cons-struct-def*)

lemma *update-ti-struct-t-update-ti-list-t* [*simp*]:
update-ti-struct-t (*TypAggregate* *ts*) = *update-ti-list-t ts*
by (*auto simp: update-ti-struct-t-def update-ti-list-t-def*)

lemma *fd-cons-struct-fd-cons-list* [*simp*]:
fd-cons-struct (*TypAggregate* *ts*) = *fd-cons-list ts*
by (*clarsimp simp: fd-cons-struct-def fd-cons-list-def*)

lemma *fd-cons-list-empty* [*simp*]:
fd-cons-list []
by (*clarsimp simp: fd-cons-list-def fd-cons-double-update-def*
fd-cons-update-access-def fd-cons-access-update-def fd-cons-length-def
fd-cons-update-normalise-def update-ti-list-t-def fd-cons-desc-def)

lemma *fd-cons-double-update-list-append*:
[[*fd-cons-double-update* (*field-desc-list* *xs*);
 fd-cons-double-update (*field-desc-list* *ys*);
 fu-commutes (*field-update* (*field-desc-list* *xs*)) (*field-update* (*field-desc-list* *ys*))
]] \implies
fd-cons-double-update (*field-desc-list* (*xs@ys*))
by (*auto simp: fd-cons-double-update-def fu-commutes-def*)

lemma *fd-cons-update-access-list-append*:
[[*fd-cons-update-access* (*field-desc-list* *xs*) (*size-td-list* *xs*);
 fd-cons-update-access (*field-desc-list* *ys*) (*size-td-list* *ys*);
 fd-cons-length (*field-desc-list* *xs*) (*size-td-list* *xs*);
]]

$fd-cons-length (field-desc-list ys) (size-td-list ys) \parallel \implies$
 $fd-cons-update-access (field-desc-list (xs@ys)) (size-td-list (xs@ys))$
by (*auto simp: fd-cons-update-access-def fd-cons-length-def access-ti-append*)

lemma *min-ll*:
 $min (x + y) x = (x::nat)$
by *simp*

lemma *fd-cons-access-update-list-append*:
 $\parallel fd-cons-access-update (field-desc-list xs) (size-td-list xs);$
 $fd-cons-access-update (field-desc-list ys) (size-td-list ys);$
 $fu-commutes (field-update (field-desc-list xs)) (field-update (field-desc-list ys))$
 $\parallel \implies$
 $fd-cons-access-update (field-desc-list (xs@ys)) (size-td-list (xs@ys))$
apply(*clarsimp simp: fd-cons-access-update-def*)
subgoal for $bs\ bs'\ v\ v'$
apply(*drule spec [where x=take (size-td-list xs) bs]*)
apply *clarsimp*
apply(*simp add: access-ti-append*)
apply(*drule spec [where x=drop (size-td-list xs) bs]*)
apply *clarsimp*
apply(*drule spec [where x=take (size-td-list xs) bs']*)
apply(*simp add: min-ll*)
apply(*drule spec [where x=update-ti-list-t ys (drop (size-td-list xs) bs) v]*)
apply(*drule spec [where x=update-ti-list-t ys (drop (size-td-list xs) bs) v']*)
apply *simp*
apply(*frule fu-commutes [where bs=take (size-td-list xs) bs and*
 $bs'=drop (size-td-list xs) bs$ **and** $v=v$ *]*)
apply *clarsimp*
apply(*frule fu-commutes [where bs=take (size-td-list xs) bs and*
 $bs'=drop (size-td-list xs) bs$ **and** $v=v'$ *]*)
apply *clarsimp*
done
done

lemma *fd-cons-length-list-append*:
 $\parallel fd-cons-length (field-desc-list xs) (size-td-list xs);$
 $fd-cons-length (field-desc-list ys) (size-td-list ys) \parallel \implies$
 $fd-cons-length (field-desc-list (xs@ys)) (size-td-list (xs@ys))$
by (*auto simp: fd-cons-length-def access-ti-append*)

lemma *wf-fdp-insert*:
 $wf-fdp (insert x xs) \implies wf-fdp \{x\} \wedge wf-fdp xs$
by (*auto simp: wf-fdp-def*)

lemma *wf-fdp-fd-cons*:
 $\parallel wf-fdp X; (t,m) \in X \parallel \implies fd-cons t$
by (*auto simp: wf-fdp-def*)

lemma *wf-fdp-fu-commutes*:

$\llbracket \text{wf-fdp } X; (s,m) \in X; (t,n) \in X; \neg m \leq n; \neg n \leq m \rrbracket \implies$
 fu-commutes (*field-update* (*field-desc* *s*)) (*field-update* (*field-desc* *t*))
by (*auto simp: wf-fdp-def*)

lemma *wf-fdp-fa-fu-ind*:

$\llbracket \text{wf-fdp } X; (s,m) \in X; (t,n) \in X; \neg m \leq n; \neg n \leq m \rrbracket \implies$
 fa-fu-ind (*field-desc* *s*) (*field-desc* *t*) (*size-td* *t*) (*size-td* *s*)
by (*auto simp: wf-fdp-def*)

lemma *wf-fdp-mono*:

$\llbracket \text{wf-fdp } Y; X \subseteq Y \rrbracket \implies \text{wf-fdp } X$
by (*fastforce simp: wf-fdp-def*)

lemma *tf0 [simp]*:

tf-set (*TypDesc* *algn* *st* *nm*) = $\{(TypDesc \text{ algn } st \text{ nm}, [])\} \cup \text{tf-set-struct } st$
by (*auto simp: tf-set-def tf-set-struct-def*)

lemma *tf1 [simp]*: *tf-set-struct* (*TypScalar* *m* *algn* *d*) = $\{\}$

by (*clarsimp simp: tf-set-struct-def*)

lemma *tf2 [simp]*: *tf-set-struct* (*TypAggregate* *xs*) = *tf-set-list* *xs*

by (*auto simp: tf-set-struct-def tf-set-list-def*)

lemma *tf3 [simp]*: *tf-set-list* $[] = \{\}$

by (*simp add: tf-set-list-def*)

lemma *tf4*: *tf-set-list* (*x#xs*) = *tf-set-tuple* *x* $\cup \{t. t \in \text{tf-set-list } xs \wedge \text{snd } t \notin \text{snd } \text{tf-set-tuple } x\}$

apply(*clarsimp simp: tf-set-list-def tf-set-tuple-def*)

apply(*cases* *x*)

apply *clarsimp*

subgoal for *x1 a b*

apply(*rule equalityI; clarsimp*)

subgoal for *a' b'*

apply(*cases* *b'*; *clarsimp*)

apply(*erule disjE; clarsimp?*)

apply(*fastforce dest: field-lookup-list-offset2 split: option.splits*)[1]

done

apply (*rule conjI; clarsimp*)

subgoal for - *ys n*

apply(*cases* *ys*; *clarsimp split: option.splits*)

apply(*rule conjI; clarsimp simp: image-def*)

apply(*drule field-lookup-list-offset3*[**where** *m=size-td* *x1*])

apply *fast*

apply(*drule field-lookup-list-offset3*[**where** *m=size-td* *x1*])

apply *fast*

done

done
done

lemma *tf4D*:
 $t \in \text{tf-set-list } (x\#xs) \implies t \in (\text{tf-set-tuple } x \cup \text{tf-set-list } xs)$
by (*clarsimp simp: tf4*)

lemma *tf4'* [*simp*]: $\text{wf-desc-list } (x\#xs) \implies$
 $\text{tf-set-list } (x\#xs) = \text{tf-set-tuple } x \cup \text{tf-set-list } xs$
apply(*simp add: tf4*)
apply(*rule equalityI; clarsimp*)
subgoal for - - *ys*
apply(*clarsimp simp: tf-set-tuple-def tf-set-list-def*)
apply(*cases x, simp*)
apply(*clarsimp split: if-split-asm*)
 apply(*cases ys; simp*)
 subgoal for - - - - *list*
 apply(*drule field-lookup-list-None [where fs=list and m=0]*)
 apply *fastforce*
 done
 done
done
done

lemma *tf5* [*simp*]: $\text{tf-set-tuple } (DTuple t m d) = \{(a,m\#b) \mid a b. (a,b) \in \text{tf-set } t\}$
apply(*clarsimp simp: tf-set-tuple-def tf-set-def*)
apply(*rule equalityI; clarsimp*)
subgoal for - *xs n*
apply(*cases xs; clarsimp*)
 done
done

lemma *tf-set-self* [*simp*]:
 $(t, []) \in \text{tf-set } t$
by (*clarsimp simp: tf-set-def*)

lemma *tf-set-list-mem*:
 $\text{wf-desc-list } ts \implies DTuple t n d \in \text{set } ts \implies (t,[n]) \in \text{tf-set-list } ts$
by (*induct ts*) *auto*

lemma *tf-set-list-append*:
 $\text{wf-desc-list } (xs@ys) \implies \text{tf-set-list } (xs@ys) = \text{tf-set-list } xs \cup \text{tf-set-list } ys$
apply(*induct xs; clarsimp*)
apply(*subst tf4'; fastforce*)
done

lemma *lf-set-list-append* [*simp*]:
 $\text{lf-set-list } (xs@ys) \text{ fn} = \text{lf-set-list } xs \text{ fn} \cup \text{lf-set-list } ys \text{ fn}$
by (*induct xs*) *auto*

lemma *ti-ind-sym*:

ti-ind $X Y \implies ti-ind Y X$

by (*auto simp: ti-ind-def fu-commutes-def*)

lemma *ti-ind-sym2*:

ti-ind $X Y = ti-ind Y X$

by (*blast dest: ti-ind-sym*)

lemma *ti-ind-list* [*simp*]:

ti-ind $(X \cup Y) Z = (ti-ind X Z \wedge ti-ind Y Z)$

unfolding *ti-ind-def* **by** *auto*

lemma *ti-empty* [*simp*]:

ti-ind $\{\} X$

by (*simp add: ti-ind-def*)

lemma *wf-lf-list*:

lf-fn $'X \cap lf-fn 'Y = \{\} \implies$

$wf-lf (X \cup Y) = (wf-lf X \wedge wf-lf Y \wedge ti-ind X Y)$

unfolding *wf-lf-def ti-ind-def field-desc-def fu-commutes-def*

apply (*rule iffI; clarsimp*)

subgoal for $x y$

apply (*frule spec [where x=x]*)

apply (*drule spec [where x=y]*)

apply *clarsimp*

apply (*drule spec [where x=y]*)

apply (*drule spec [where x=x]*)

apply *clarsimp*

apply *fastforce*

done

subgoal by *fast*

done

lemma *wf-lf-listD*:

wf-lf $(X \cup Y) \implies wf-lf X \wedge wf-lf Y$

unfolding *wf-lf-def ti-ind-def field-desc-def fu-commutes-def* **by** *clarsimp*

lemma *ti-ind-fn*:

fixes $t::('a,'b) typ-info$ **and**

$st::('a,'b) typ-info-struct$ **and**

$ts::('a,'b) typ-info-tuple list$ **and**

$x::('a,'b) typ-info-tuple$

shows

$\forall fn. ti-ind (lf-set t fn) Y = ti-ind (lf-set t []) Y$

$\forall fn. ti-ind (lf-set-struct st fn) Y = ti-ind (lf-set-struct st []) Y$

$\forall fn. ti-ind (lf-set-list ts fn) Y = ti-ind (lf-set-list ts []) Y$

$\forall fn. ti-ind (lf-set-tuple x fn) Y = ti-ind (lf-set-tuple x []) Y$

apply (*induct t and st and ts and x, all <clarsimp>*)

apply (*fastforce simp: ti-ind-def*)
apply *auto*
done

lemma *ti-ind-ld-td'*:

fixes $t::('a,'b)$ *typ-info* **and**
 $st::('a,'b)$ *typ-info-struct* **and**
 $ts::('a,'b)$ *typ-info-tuple list* **and**
 $x::('a,'b)$ *typ-info-tuple*

shows

$ti-ind (lf-set t []) Y \longrightarrow ti-ind \{t2d (t,[])\} Y$
 $ti-ind (lf-set-struct st []) Y \longrightarrow ti-ind \{\lfloor lf-fd = field-desc-struct st, lf-sz = size-td-struct st, lf-fn = [] \rfloor\} Y$
 $ti-ind (lf-set-list ts []) Y \longrightarrow ti-ind \{\lfloor lf-fd = field-desc-list ts, lf-sz = size-td-list ts, lf-fn = [] \rfloor\} Y$
 $ti-ind (lf-set-tuple x []) Y \longrightarrow ti-ind \{\lfloor lf-fd = field-desc-tuple x, lf-sz = size-td-tuple x, lf-fn = [] \rfloor\} Y$

apply(*induct t and st and ts and x, all <clarsimp simp: t2d-def>*)

apply(*(clarsimp simp: ti-ind-def fu-commutes-def fa-fu-ind-def)+*)[3]

apply(*subst (asm) ti-ind-fn*)

apply *simp*

done

lemma *ti-ind-ld-td-struct*:

$ti-ind (lf-set-struct st fn) Y \Longrightarrow$
 $ti-ind \{\lfloor lf-fd = field-desc-struct st, lf-sz = size-td-struct st, lf-fn = [] \rfloor\} Y$
by (*subst (asm) ti-ind-fn (simp only: ti-ind-ld-td')*)

lemma *ti-ind-ld-td-list*:

$ti-ind (lf-set-list ts fn) Y \Longrightarrow$
 $ti-ind \{\lfloor lf-fd = field-desc-list ts, lf-sz = size-td-list ts, lf-fn = [] \rfloor\} Y$
by (*subst (asm) ti-ind-fn (simp only: ti-ind-ld-td')*)

lemma *ti-ind-ld-td-tuple*:

$ti-ind (lf-set-tuple x fn) Y \Longrightarrow$
 $ti-ind \{\lfloor lf-fd = field-desc-tuple x, lf-sz = size-td-tuple x, lf-fn = [] \rfloor\} Y$
by (*subst (asm) ti-ind-fn (simp only: ti-ind-ld-td')*)

lemma *ti-ind-ld'*:

fixes $t::('a,'b)$ *typ-info* **and**
 $st::('a,'b)$ *typ-info-struct* **and**
 $ts::('a,'b)$ *typ-info-tuple list* **and**
 $x::('a,'b)$ *typ-info-tuple*

shows

$ti-ind (lf-set t []) Y \longrightarrow ti-ind (t2d '(tf-set t)) Y$
 $ti-ind (lf-set-struct st []) Y \longrightarrow ti-ind (t2d '(tf-set-struct st)) Y$
 $ti-ind (lf-set-list ts []) Y \longrightarrow ti-ind (t2d '(tf-set-list ts)) Y$
 $ti-ind (lf-set-tuple x []) Y \longrightarrow ti-ind (t2d '(tf-set-tuple x)) Y$
proof(*induct t and st and ts and x*)


```

case (TypDesc nat typ-struct list)
then show ?case
  apply clarsimp
  apply(frule ti-ind-ld-td-struct)
  apply(subst insert-def)
  apply(subst ti-ind-list)
  apply(clarsimp simp: ti-ind-def t2d-def)
done
next
case (TypScalar nat1 nat2 a)
then show ?case by auto
next
case (TypAggregate list)
then show ?case by auto
next
case Nil-typ-desc
then show ?case by auto
next
case (Cons-typ-desc dt-tuple list)
then show ?case apply clarsimp
  apply(clarsimp simp: ti-ind-def t2d-def)
  apply(drule tf4D)
  apply clarsimp
  by (smt (verit, best) field-desc.select-convs(2) fst-eqD image-eqI
      leaf-desc.select-convs(1) leaf-desc.select-convs(2))
next
case (DTuple-typ-desc typ-desc list b)
then show ?case
  apply clarsimp
  apply(subst (asm) (2) ti-ind-fn)
  apply(simp add: t2d-def)
  apply(clarsimp simp: ti-ind-def)
  by (metis (mono-tags, lifting) field-desc.select-convs(2) fst-conv image-eqI
      leaf-desc.select-convs(1) leaf-desc.select-convs(2))
qed

```

lemma *ti-ind-ld*:

$ti-ind (lf-set t fn) Y \implies ti-ind (t2d \text{ ' } (tf-set t)) Y$
by (subst (asm) ti-ind-fn) (simp only: ti-ind-ld')

lemma *ti-ind-ld-struct*:

$ti-ind (lf-set-struct t fn) Y \implies ti-ind (t2d \text{ ' } (tf-set-struct t)) Y$
by (subst (asm) ti-ind-fn) (simp only: ti-ind-ld')

lemma *ti-ind-ld-list*:

$ti-ind (lf-set-list t fn) Y \implies ti-ind (t2d \text{ ' } (tf-set-list t)) Y$
by (subst (asm) ti-ind-fn) (simp only: ti-ind-ld')

lemma *ti-ind-ld-tuple*:

$ti-ind (lf-set-tuple t fn) Y \implies ti-ind (t2d (tf-set-tuple t)) Y$
by (*subst (asm) ti-ind-fn (simp only: ti-ind-ld')*)

lemma *lf-set-fn'*:

fixes $t::('a,'b) typ-info$ **and**

$st::('a,'b) typ-info-struct$ **and**

$ts::('a,'b) typ-info-tuple list$ **and**

$x::('a,'b) typ-info-tuple$

shows

$\forall s fn. s \in lf-set t fn \longrightarrow fn \leq lf-fn s$

$\forall s fn. s \in lf-set-struct st fn \longrightarrow fn \leq lf-fn s$

$\forall s fn. s \in lf-set-list ts fn \longrightarrow fn \leq lf-fn s$

$\forall s fn. s \in lf-set-tuple x fn \longrightarrow fn \leq lf-fn s$

proof (*induct t and st and ts and x*)

case (*DTuple-typ-desc typ-desc list b*)

then show *?case*

apply *clarsimp*

subgoal for *s fn*

apply (*drule spec [where x=s]*)

apply (*drule spec [where x=fn @ [list]]*)

apply (*clarsimp simp: prefix-def less-eq-list-def*)

done

done

qed *auto*

lemma *lf-set-fn*:

$s \in lf-set (t::('a,'b) typ-info) fn \implies fn \leq lf-fn s$

by (*clarsimp simp: lf-set-fn'*)

lemma *ln-fn-disj*:

$dt-snd x \notin dt-snd (set xs) \implies lf-fn (lf-set-tuple x fn) \cap lf-fn (lf-set-list xs fn) = \{\}$

apply (*induct xs arbitrary: x; clarsimp*)

apply (*rule set-eqI, clarsimp*)

apply (*erule disjE*)

apply (*clarsimp dest!: lf-set-fn simp: split-DTuple-all prefix-def less-eq-list-def*)

apply *fastforce*

done

lemma *wf-lf-fn*:

fixes $t::('a,'b) typ-info$ **and**

$st::('a,'b) typ-info-struct$ **and**

$ts::('a,'b) typ-info-tuple list$ **and**

$x::('a,'b) typ-info-tuple$

shows

$\forall fn. wf-desc t \longrightarrow wf-lf (lf-set t fn) = wf-lf (lf-set t [])$

$\forall fn. wf-desc-struct st \longrightarrow wf-lf (lf-set-struct st fn) = wf-lf (lf-set-struct st [])$

$\forall fn. wf-desc-list ts \longrightarrow wf-lf (lf-set-list ts fn) = wf-lf (lf-set-list ts [])$

$\forall fn. wf-desc-tuple x \longrightarrow wf-lf (lf-set-tuple x fn) = wf-lf (lf-set-tuple x [])$

proof (*induct t and st and ts and x*)

```

case (TypDesc nat typ-struct list)
then show ?case
  apply clarify
  subgoal for fn
    apply(drule spec [where x=fn])
    apply clarsimp
  done
done
next
case (TypScalar nat1 nat2 a)
then show ?case
  apply clarsimp
  apply(clarsimp simp: wf-lf-def)
done
next
case (TypAggregate list)
then show ?case
  apply clarify
  subgoal for fn
    apply(drule spec [where x=fn])
    apply clarsimp
  done
done
next
case Nil-typ-desc
then show ?case by clarsimp
next
case (Cons-typ-desc dt-tuple list)
then show ?case
  apply clarify
  subgoal for fn
    apply(drule spec [where x=fn])+
    apply clarsimp
    apply(subst wf-lf-list)
    apply(erule ln-fn-disj)
    apply(subst wf-lf-list)
    apply(erule ln-fn-disj)
    apply clarsimp
    apply(subst ti-ind-fn)
    apply(subst ti-ind-sym2)
    apply(subst ti-ind-fn)
    apply(subst ti-ind-sym2)
    apply(clarsimp)
  done
done
next
case (DTuple-typ-desc typ-desc list b)
then show ?case
  apply clarify

```

```

subgoal for fn
  apply(frule spec [where  $x=[list]$ ])
  apply(drule spec [where  $x=fn@[list]$ ])
  apply clarsimp
  done
done
qed

lemma wf-lf-fd-cons':
  fixes  $t::('a,'b)$  typ-info and
   $st::('a,'b)$  typ-info-struct and
   $ts::('a,'b)$  typ-info-tuple list and
   $x::('a,'b)$  typ-info-tuple
  shows
   $\forall m. wf\text{-}lf\ (lf\text{-}set\ t\ []) \longrightarrow wf\text{-}desc\ t \longrightarrow fd\text{-}cons\ t$ 
   $\forall m. wf\text{-}lf\ (lf\text{-}set\text{-}struct\ st\ []) \longrightarrow wf\text{-}desc\text{-}struct\ st \longrightarrow fd\text{-}cons\text{-}struct\ st$ 
   $\forall m. wf\text{-}lf\ (lf\text{-}set\text{-}list\ ts\ []) \longrightarrow wf\text{-}desc\text{-}list\ ts \longrightarrow fd\text{-}cons\text{-}list\ ts$ 
   $\forall m. wf\text{-}lf\ (lf\text{-}set\text{-}tuple\ x\ []) \longrightarrow wf\text{-}desc\text{-}tuple\ x \longrightarrow fd\text{-}cons\text{-}tuple\ x$ 
proof(induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case by clarsimp
next
  case (TypScalar nat1 nat2 a)
  then show ?case
  apply(clarsimp simp: wf-lf-def fd-cons-struct-def fd-cons-def)
  apply(clarsimp simp: fd-cons-desc-def fd-cons-double-update-def fd-cons-update-access-def
    fd-cons-access-update-def fd-cons-length-def)
  done
next
  case (TypAggregate list)
  then show ?case by clarsimp
next
  case Nil-tyt-desc
  then show ?case by clarsimp
next
  case (Cons-tyt-desc dt-tuple list)
  then show ?case
  apply clarsimp
  apply(subst (asm) wf-lf-list)
  apply(erule ln-fn-disj)
  apply clarsimp
  apply(drule ti-ind-ld-td-tuple)
  apply(drule ti-ind-sym)
  apply(drule ti-ind-ld-td-list)
  apply(drule ti-ind-sym)
  apply(clarsimp simp: ti-ind-def)
  apply(clarsimp simp: fd-cons-list-def fd-cons-desc-def)
  apply(rule conjI)
  apply(clarsimp simp: fd-cons-tuple-def fd-cons-double-update-def fd-cons-desc-def)

```

```

  apply(simp add: fu-commutes-def)
  apply(rule conjI)
  apply(clarsimp simp: fd-cons-tuple-def fd-cons-update-access-def fd-cons-desc-def)
  apply(clarsimp simp: fd-cons-length-def)
  apply(rule conjI)
  apply(clarsimp simp: fd-cons-tuple-def fd-cons-desc-def)
  apply(clarsimp simp: fd-cons-access-update-def)
  subgoal for bs bs' v v'
  apply(simp add: fu-commutes-def)
  apply(clarsimp simp: fa-fu-ind-def)
  subgoal premises prems
  using prems (1-14, 16-18)
  prems (15)[rule-format, where bs'= take (size-td-tuple dt-tuple) bs' and
    bs= take (size-td-tuple dt-tuple) bs and v=v and v'=v' ]
  apply -
  apply(simp add: min-ll)
  done
done
  apply(clarsimp simp: fd-cons-length-def fd-cons-tuple-def fd-cons-desc-def)
done
next
case (DTuple-typ-desc typ-desc list b)
then show ?case
  apply clarsimp
  apply(subst (asm) (2) wf-lf-fn, assumption)
  apply(clarsimp simp: fd-cons-def fd-cons-tuple-def export-uinfo-def)
done
qed

```

lemma *wf-lf-fd-cons*:

$\llbracket \text{wf-lf } (lf\text{-set } t \text{ fn}); \text{wf-desc } t \rrbracket \implies \text{fd-cons } t$
by (subst (asm) wf-lf-fn; simp only: wf-lf-fd-cons')

lemma *wf-lf-fd-cons-struct*:

$\llbracket \text{wf-lf } (lf\text{-set-struct } t \text{ fn}); \text{wf-desc-struct } t \rrbracket \implies \text{fd-cons-struct } t$
by (subst (asm) wf-lf-fn; simp only: wf-lf-fd-cons')

lemma *wf-lf-fd-cons-list*:

$\llbracket \text{wf-lf } (lf\text{-set-list } t \text{ fn}); \text{wf-desc-list } t \rrbracket \implies \text{fd-cons-list } t$
by (subst (asm) wf-lf-fn; simp only: wf-lf-fd-cons')

lemma *wf-lf-fd-cons-tuple*:

$\llbracket \text{wf-lf } (lf\text{-set-tuple } t \text{ fn}); \text{wf-desc-tuple } t \rrbracket \implies \text{fd-cons-tuple } t$
by (subst (asm) wf-lf-fn; simp only: wf-lf-fd-cons')

lemma *wf-lf-fdp'*:

$\forall m. \text{wf-lf } (lf\text{-set } (t::('a,'b) \text{ typ-info}) []) \longrightarrow \text{wf-desc } t \longrightarrow \text{wf-fdp } (lf\text{-set } t)$
 $\forall m. \text{wf-lf } (lf\text{-set-struct } (st::('a,'b) \text{ typ-info-struct}) []) \longrightarrow \text{wf-desc-struct } st \longrightarrow$

```

wf-fdp (tf-set-struct st)
  ∀ m. wf-lf (lf-set-list (ts::('a,'b) typ-info-tuple list) []) → wf-desc-list ts → wf-fdp
(tf-set-list ts)
  ∀ m. wf-lf (lf-set-tuple (x::('a,'b) typ-info-tuple) []) → wf-desc-tuple x → wf-fdp
(tf-set-tuple x)
proof (induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case
    apply (clarsimp simp: wf-fdp-def)
    apply (fastforce elim: wf-lf-fd-cons-struct)
    done
next
  case (TypScalar nat1 nat2 a)
  then show ?case by (clarsimp simp: wf-fdp-def)
next
  case (TypAggregate list)
  then show ?case by (clarsimp simp: wf-fdp-def)
next
  case Nil-typ-desc
  then show ?case by (clarsimp simp: wf-fdp-def)
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply (clarsimp simp: wf-fdp-def)
    apply (subst (asm) wf-lf-list)
    apply (erule ln-fn-disj)
    apply clarsimp
    apply (drule ti-ind-ld-tuple)
    apply (drule ti-ind-sym)
    apply (drule ti-ind-ld-list)
    apply (drule ti-ind-sym)
    apply (rule conjI, clarsimp)
    apply (erule disjE, fast)
    apply (clarsimp simp: ti-ind-def t2d-def)
  subgoal for x m y n
    apply (drule spec [where x=t2d (x,m)])
    apply (drule spec [where x=t2d (y,n)])
    apply (clarsimp simp: t2d-def image-def)
    apply (erule impE)
    apply (rule conjI)
    apply (rule bexI [where x=(x,m)])
    apply clarsimp
    apply assumption
    apply (rule bexI [where x=(y,n)])
    apply clarsimp
    apply assumption
    apply clarsimp
  done

```

```

apply clarsimp
subgoal for  $x\ m\ y\ n$ 
  apply(erule disjE)
  apply(clarsimp simp: ti-ind-def t2d-def)
  apply(drule spec [where x=t2d (y,n)])
  apply(drule spec [where x=t2d (x,m)])
  apply(clarsimp simp: t2d-def image-def)
  apply(erule impE)
  apply(standard)
  apply(rule bexI [where x=(y,n)])
  apply clarsimp
  apply assumption
  apply(rule bexI [where x=(x,m)])
  apply clarsimp
  apply assumption
  apply clarsimp
  apply(clarsimp simp: fu-commutes-def)
  apply fast
  done
done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
    apply (clarsimp simp: wf-fdp-def)
    apply(subst (asm) (2) wf-lf-fn, assumption)
    apply(clarsimp simp: wf-fdp-def)
    apply fast
    done
qed

lemma wf-lf-fdp:
  [ wf-lf (lf-set t []); wf-desc t ]  $\implies$  wf-fdp (tf-set t)
  by (simp only: wf-lf-fdp')

lemma wf-fd-field-lookup [rule-format]:
   $\forall f\ m\ n\ s.$  wf-fd (t::('a,'b) typ-info)  $\longrightarrow$  field-lookup t f m = Some (s,n)  $\longrightarrow$  wf-fd
 $s$ 
   $\forall f\ m\ n\ s.$  wf-fd-struct (st::('a,'b) typ-info-struct)  $\longrightarrow$  field-lookup-struct st f m =
Some (s,n)  $\longrightarrow$  wf-fd s
   $\forall f\ m\ n\ s.$  wf-fd-list (ts::('a,'b) typ-info-tuple list)  $\longrightarrow$  field-lookup-list ts f m =
Some (s,n)  $\longrightarrow$  wf-fd s
   $\forall f\ m\ n\ s.$  wf-fd-tuple (x::('a,'b) typ-info-tuple)  $\longrightarrow$  field-lookup-tuple x f m =
Some (s,n)  $\longrightarrow$  wf-fd s
  by (induct t and st and ts and x (clarsimp split: option.splits)+)

lemma wf-fd-field-lookupD:
  [ field-lookup t f m = Some (s,n); wf-fd t ]  $\implies$  wf-fd s
  by (rule wf-fd-field-lookup)

```

lemma *wf-fd-tf-set*:

$\llbracket \text{wf-fd } t; ((s::('a,'b) \text{ typ-info}),m) \in \text{tf-set } t \rrbracket \implies \text{wf-fd } s$

by (*fastforce simp: tf-set-def wf-fd-field-lookupD*)

lemma *tf-set-field-lookupD*:

$\text{field-lookup } t f m = \text{Some } (s,n) \implies (s,f) \in \text{tf-set } t$

unfolding *tf-set-def*

by (*clarsimp simp flip: field-lookup-offset[where m=m] dest!: field-lookup-offset-le*)
arith

lemma *fu-commutes-ts*:

$(\bigwedge t. t \in \text{dt-fst } ' \text{ set } ts \implies \text{fu-commutes } d (\text{update-ti-t } t)) \implies$

$\text{fu-commutes } d (\text{update-ti-list-t } ts)$

by (*induct ts; clarsimp simp: fu-commutes-def (clarsimp simp: split-DTuple-all)*)

lemma *fa-fu-ind-ts*:

$(\bigwedge t. t \in \text{dt-fst } ' \text{ set } ts \implies \text{fa-fu-ind } d (\text{field-desc } t) (\text{size-td } t) n) \implies$

$\text{fa-fu-ind } d (\!| \text{field-access} = \text{access-ti-list } ts,$

$\text{field-update} = \text{update-ti-list-t } ts, \text{field-sz} = \text{size-td-list } ts)$

$(\text{size-td-list } ts) n$

by (*induct ts; clarsimp simp: fa-fu-ind-def (clarsimp simp: split-DTuple-all)*)

lemma *fa-fu-ind-ts2*:

$(\bigwedge t. t \in \text{dt-fst } ' \text{ set } ts \implies \text{fa-fu-ind } (\text{field-desc } t) d n (\text{size-td } t)) \implies$

$\text{fa-fu-ind } (\!| \text{field-access} = \text{access-ti-list } ts,$

$\text{field-update} = \text{update-ti-list-t } ts, \text{field-sz} = \text{size-td-list } ts)$

$n (\text{size-td-list } ts)$

by (*induct ts; clarsimp simp: fa-fu-ind-def (clarsimp simp: split-DTuple-all)*)

lemma *wf-fdp-fd [rule-format]*:

$\forall m. \text{wf-fdp } (\text{tf-set } t) \longrightarrow \text{wf-desc } t \longrightarrow \text{wf-fd } (t::('a,'b) \text{ typ-info})$

$\forall m. (\text{case } st \text{ of } \text{TypScalar } sz \text{ algn } d \Rightarrow \text{fd-cons-struct } ((\text{TypScalar } sz \text{ algn } d)::('a,'b) \text{ typ-info-struct})$

$\quad | - \Rightarrow \text{wf-fdp } (\text{tf-set-struct } st)) \longrightarrow \text{wf-desc-struct } st \longrightarrow \text{wf-fd-struct}$

$(st::('a,'b) \text{ typ-info-struct})$

$\forall m. \text{wf-fdp } (\text{tf-set-list } ts) \longrightarrow \text{wf-desc-list } ts \longrightarrow \text{wf-fd-list } (ts::('a,'b) \text{ typ-info-tuple}$

$\text{list})$

proof (*induct t and st and ts and x*)

case (*TypDesc nat typ-struct list*)

then show *?case*

apply (*clarsimp split: typ-struct.split-asm*)

apply (*clarsimp simp: wf-fdp-def fd-cons-def fd-cons-struct-def*)

apply (*fastforce dest: wf-fdp-mono*)

done

next

case (*TypScalar nat1 nat2 a*)

then show *?case by auto*

next


```

    case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple xs)
  then show ?case
    apply clarsimp
    apply (rule conjI, fastforce dest: wf-fdp-mono)
    apply (rule conjI)
    subgoal
      using wf-fdp-mono
      by blast
    subgoal
      apply (clarsimp simp: wf-fdp-def)
      apply (cases dt-tuple, clarsimp)
      subgoal for a b x3

        apply (frule spec [where x=a])
        apply (drule spec [where x=[b]])
        apply clarsimp
        apply (rule conjI)
        apply (rule fu-commutes-ts)
        apply clarsimp
        subgoal for x
          apply (drule spec [where x=dt-fst x], erule impE, rule exI [where x=[dt-snd
x]])
            apply (metis Prefix-Order.Cons-prefix-Cons dt-tuple.exhaust-sel rev-image-eqI
tf-set-list-mem)
              apply simp
              done
            apply (rule conjI)
            apply (rule fa-fu-ind-ts)
            apply clarsimp
            subgoal for x

              apply (drule spec [where x=dt-fst x], erule impE, rule exI [where x=[dt-snd
x]])
                apply (metis Prefix-Order.Cons-prefix-Cons dt-tuple.exhaust-sel rev-image-eqI
tf-set-list-mem)
                  apply simp
                  done

              apply (rule fa-fu-ind-ts2)
              subgoal for t
                apply (drule spec [where x=t])
                apply clarsimp
                subgoal for x

```

```

    apply(cases x, clarsimp)
    subgoal for ba aa
      apply(drule spec [where x=[ba]])
      apply clarsimp
      apply(simp add: tf-set-list-mem)
      apply clarsimp
    by (metis Prefix-Order.Cons-prefix-Cons dt-tuple.sel(2) imageI tf-set-self)

    done
  done
done
done
done
next
case (DTuple-typ-desc typ-desc list b)
then show ?case
  apply(clarsimp simp: wf-fdp-def)
  subgoal for x m
    apply(drule spec [where x=x])
    apply(drule spec [where x=list#m])
    apply clarsimp
  subgoal for y n
    apply(drule spec [where x=y])
    apply auto
  done
done
done
qed

lemma wf-fdp-fdD:
  [[ wf-fdp (tf-set t); wf-desc t ] ]  $\implies$  wf-fd (t::('a,'b) typ-info)
  by (rule wf-fdp-fd)

lemma wf-fdp-fd-listD:
  [[ wf-fdp (tf-set-list t); wf-desc-list t ] ]  $\implies$  wf-fd-list t
  by (rule wf-fdp-fd)

lemma fd-consistentD:
  [[ field-lookup t f 0 = Some (s,n); fd-consistent t ] ]
   $\implies$  fd-cons s
  by (fastforce simp: fd-consistent-def)

lemma wf-fd-cons-access-update' [rule-format]:
  wf-fd (t::('a,'b) typ-info)  $\longrightarrow$  fd-cons-access-update (field-desc t) (size-td t)
  wf-fd-struct (st::('a,'b) typ-info-struct)  $\longrightarrow$  fd-cons-access-update (field-desc-struct
st) (size-td-struct st)
  wf-fd-list (ts::('a,'b) typ-info-tuple list)  $\longrightarrow$  fd-cons-access-update (field-desc-list
ts) (size-td-list ts)
  wf-fd-tuple (x::('a,'b) typ-info-tuple)  $\longrightarrow$  fd-cons-access-update (field-desc-tuple x)

```

```

(size-td-tuple x)
proof(induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case by (clarsimp split: typ-struct.splits)
next
  case (TypScalar nat1 nat2 a)
  then show ?case
    apply (clarsimp split: typ-struct.splits)
    apply(clarsimp simp: fd-cons-access-update-def fd-cons-struct-def fd-cons-desc-def)
    done
next
  case (TypAggregate list)
  then show ?case by (clarsimp split: typ-struct.splits)
next
  case Nil-tyt-desc
  then show ?case apply (clarsimp split: typ-struct.splits)
    apply(clarsimp simp: fd-cons-access-update-def)
    done
next
  case (Cons-tyt-desc dt-tuple list)
  then show ?case
    apply (clarsimp split: typ-struct.splits)
    apply(clarsimp simp: fd-cons-access-update-def)
    apply(simp add: fu-commutes-def)
    apply(clarsimp simp: fa-fu-ind-def)
    by (metis (no-types, lifting) add-diff-cancel-left' length-drop length-take min-ll)
next
  case (DTuple-tyt-desc typ-desc list b)
  then show ?case by (clarsimp split: typ-struct.splits)
qed

```

lemma *wf-fd-cons-access-updateD*:
 $wf\text{-}fd\ t \implies fd\text{-}cons\text{-}access\text{-}update\ (field\text{-}desc\ t)\ (size\text{-}td\ t)$
by (*rule wf-fd-cons-access-update'*)

lemma *wf-fd-cons-access-update-structD*:
 $wf\text{-}fd\text{-}struct\ t \implies fd\text{-}cons\text{-}access\text{-}update\ (field\text{-}desc\text{-}struct\ t)\ (size\text{-}td\text{-}struct\ t)$
by (*rule wf-fd-cons-access-update'*)

lemma *wf-fd-cons-access-update-listD*:
 $wf\text{-}fd\text{-}list\ t \implies fd\text{-}cons\text{-}access\text{-}update\ (field\text{-}desc\text{-}list\ t)\ (size\text{-}td\text{-}list\ t)$
by (*rule wf-fd-cons-access-update'*)

lemma *wf-fd-cons-access-update-tupleD*:
 $wf\text{-}fd\text{-}tuple\ t \implies fd\text{-}cons\text{-}access\text{-}update\ (field\text{-}desc\text{-}tuple\ t)\ (size\text{-}td\text{-}tuple\ t)$
by (*rule wf-fd-cons-access-update'*)

lemma *wf-fd-norm-tu*:
 $\forall bs. wf\text{-}fd\ t \implies length\ bs = size\text{-}td\ t \implies norm\text{-}tu\ (export\text{-}uinfo\ (t::('a,'b)))$

```

typ-info) bs = (access-ti t (update-ti-t t bs undefined) (replicate (size-td t) 0))
  ∀ bs. wf-fd-struct st → length bs = size-td-struct st → norm-tu-struct (map-td-struct
field-norm (λ-. ()) (st::('a,'b) typ-info-struct)) bs = (access-ti-struct st (update-ti-struct-t
st bs undefined) (replicate (size-td-struct st) 0))
  ∀ bs. wf-fd-list ts → length bs = size-td-list ts → norm-tu-list (map-td-list
field-norm (λ-. ()) (ts::('a,'b) typ-info-tuple list)) bs = (access-ti-list ts (update-ti-list-t
ts bs undefined) (replicate (size-td-list ts) 0))
  ∀ bs. wf-fd-tuple x → length bs = size-td-tuple x → norm-tu-tuple (map-td-tuple
field-norm (λ-. ()) (x::('a,'b) typ-info-tuple)) bs = (access-ti-tuple x (update-ti-tuple-t
x bs undefined) (replicate (size-td-tuple x) 0))
  apply(induct t and st and ts and x, all ⟨clarsimp simp: export-uinfo-def⟩)
  apply(simp add: field-norm-def)
  apply(simp add: fu-commutes-def)
  apply(simp add: fa-fu-ind-def)
  apply(drule wf-fd-cons-access-update-listD)
  apply(clarsimp simp: fd-cons-access-update-def export-uinfo-def min-def)
  done

```

lemma *wf-fd-norm-tuD*:

```

[[ wf-fd t; length bs = size-td t ]] ⇒ norm-tu (export-uinfo t) bs =
  (access-ti0 t (update-ti-t t bs undefined))
using wf-fd-norm-tu(1) [of t] by (clarsimp simp: access-ti0-def)

```

lemma *wf-fd-norm-tu-structD*:

```

[[ wf-fd-struct t; length bs = size-td-struct t ]] ⇒ norm-tu-struct (map-td-struct
field-norm (λ-. ()) t) bs =
  (access-ti-struct t (update-ti-struct-t t bs undefined) (replicate (size-td-struct
t) 0))
using wf-fd-norm-tu(2) [of t] by clarsimp

```

lemma *wf-fd-norm-tu-listD*:

```

[[ wf-fd-list t; length bs = size-td-list t ]] ⇒ norm-tu-list (map-td-list field-norm
(λ-. ()) t) bs =
  (access-ti-list t (update-ti-list-t t bs undefined) (replicate (size-td-list t) 0))
using wf-fd-norm-tu(3) [of t] by clarsimp

```

lemma *wf-fd-norm-tu-tupleD*:

```

[[ wf-fd-tuple t; length bs = size-td-tuple t ]] ⇒ norm-tu-tuple (map-td-tuple
field-norm (λ-. ()) t) bs =
  (access-ti-tuple t (update-ti-tuple-t t bs undefined) (replicate (size-td-tuple t)
0))
using wf-fd-norm-tu(4) [of t] by clarsimp

```

lemma *wf-fd-cons* [*rule-format*]:

```

wf-fd t → fd-cons (t::('a,'b) typ-info)
wf-fd-struct st → fd-cons-struct (st::('a,'b) typ-info-struct)
wf-fd-list ts → fd-cons-list (ts::('a,'b) typ-info-tuple list)
wf-fd-tuple x → fd-cons-tuple (x::('a,'b) typ-info-tuple)
proof (induct t and st and ts and x)

```

```

    case (TypDesc nat typ-struct list)
    then show ?case by auto
next
    case (TypScalar nat1 nat2 a)
    then show ?case by auto
next
    case (TypAggregate list)
    then show ?case by auto
next
    case Nil-typ-desc
    then show ?case by auto
next
    case (Cons-typ-desc dt-tuple list)
    then show ?case
      apply clarsimp
      apply (clarsimp simp: fd-cons-list-def fd-cons-desc-def)
      apply (rule conjI)
      apply (clarsimp simp: fd-cons-tuple-def fd-cons-double-update-def fd-cons-desc-def)
      apply (simp add: fu-commutes-def)
      apply (rule conjI)
      apply (clarsimp simp: fd-cons-tuple-def fd-cons-update-access-def fd-cons-desc-def)
      apply (clarsimp simp: fd-cons-length-def)
      apply (rule conjI)
      apply (clarsimp simp: fd-cons-tuple-def fd-cons-desc-def)
      apply (clarsimp simp: fd-cons-access-update-def fu-commutes-def fa-fu-ind-def)
    )
      apply (smt (verit) diff-add-inverse length-drop length-take min-ll)
      apply (clarsimp simp: fd-cons-length-def fd-cons-tuple-def fd-cons-desc-def)
      done
next
    case (DTuple-typ-desc typ-desc list b)
    then show ?case by (clarsimp simp: fd-cons-def fd-cons-tuple-def export-uinfo-def)
qed

```

lemma *wf-fd-consD*:
wf-fd t \implies *fd-cons t*
by (rule *wf-fd-cons*)

lemma *wf-fd-cons-structD*:
wf-fd-struct t \implies *fd-cons-struct t*
by (rule *wf-fd-cons*)

lemma *wf-fd-cons-listD*:
wf-fd-list t \implies *fd-cons-list t*
by (rule *wf-fd-cons*)

lemma *wf-fd-cons-tupleD*:
wf-fd-tuple t \implies *fd-cons-tuple t*
by (rule *wf-fd-cons*)

lemma *fd-cons-list-append*:
 $\llbracket \text{wf-fd-list } xs; \text{wf-fd-list } ys; \text{fu-commutes} \text{ (field-update (field-desc-list } xs)) \text{ (field-update (field-desc-list } ys)) \rrbracket \implies$
 $\text{fd-cons-list } (xs@ys)$
apply(*frule wf-fd-cons-listD*)
apply(*frule wf-fd-cons-listD [where t=ys]*)
apply(*unfold fd-cons-list-def fd-cons-desc-def*)
apply(*fastforce intro: fd-cons-double-update-list-append fd-cons-update-access-list-append*
fd-cons-access-update-list-append fd-cons-length-list-append)
done

lemma (**in** *wf-type*) *wf-fd [simp]*:
 $\text{wf-fd } (\text{typ-info-t } \text{TYPE}('a))$
by (*fastforce intro: wf-fdp-fdD wf-lf-fdp wf-lf*)

lemma (**in** *wf-type*) *fd-cons [simp]*:
 $\text{fd-consistent } (\text{typ-info-t } \text{TYPE}('a))$
unfolding *fd-consistent-def* **by** (*fastforce intro: wf-fd-consD wf-fd-field-lookupD*)

lemma (**in** *wf-type*) *field-lvalue-append [simp]*:
 $\llbracket \text{field-ti } \text{TYPE}('a) \text{ f} = \text{Some } t;$
 $\text{export-uinfo } t = \text{typ-uinfo-t } \text{TYPE}('b::\text{c-type});$
 $\text{field-ti } \text{TYPE}('b) \text{ g} = \text{Some } k \rrbracket \implies$
 $\&(((\text{Ptr } \&((p::'a \text{ ptr}) \rightarrow f))::'b \text{ ptr}) \rightarrow g) = \&(p \rightarrow f @ g)$
apply(*clarsimp simp: field-lvalue-def c-type-class.field-ti-def field-ti-def field-offset-def*
field-offset-untyped-def typ-uinfo-t-def
split: option.splits)
apply(*subst field-lookup-prefix-Some'*)
apply(*fastforce dest: field-lookup-export-uinfo-Some*)
apply(*simp add: export-uinfo-def wf-desc-map*)
apply(*drule field-lookup-export-uinfo-Some*)
apply(*simp add: export-uinfo-def*)
apply(*drule field-lookup-export-uinfo-Some*)
apply(*simp add: export-uinfo-def*)
apply (*simp add: c-type-class.field-lvalue-def c-type-class.field-offset-def*
c-type-class.typ-uinfo-t-def export-uinfo-def field-lookup-offset-shift' field-offset-untyped-def)
done

lemma (**in** *wf-type*) *field-lvalue-cons-unfold'*:
— rhs contains additional type variable 'b, hence simplifier won't apply this rule
 $\llbracket \text{field-ti } \text{TYPE}('a) [f] = \text{Some } t;$
 $\text{export-uinfo } t = \text{typ-uinfo-t } \text{TYPE}('b::\text{c-type});$
 $\text{field-ti } \text{TYPE}('b) \text{ g} = \text{Some } k \rrbracket \implies$
 $\&(p \rightarrow f \# g) = \&(((\text{Ptr } \&((p::'a \text{ ptr}) \rightarrow [f]))::'b \text{ ptr}) \rightarrow g)$

using *field-lvalue-append* [where $f=[f]$ and $g=g$]
by *simp*

lemma (in *mem-type*) *field-lvalue-cons-unfold*:

[[*field-ti* *TYPE*('b) $g = \text{Some } k$;
export-uinfo $t = \text{export-uinfo } (\text{typ-info-t } \text{TYPE}('b::\text{c-type}))$;
field-ti *TYPE*('a) $[f] = \text{Some } t$] \implies
 $\&(p \rightarrow f \# g) \equiv \&(((\text{Ptr } \&((p::'a \text{ ptr}) \rightarrow [f]))::'b \text{ ptr}) \rightarrow g)$
using *field-lvalue-cons-unfold'* [*simplified c-type-class.typ-uinfo-t-def*]
apply –
apply (*rule eq-reflection*)
apply *blast*
done

lemma *field-access-update-take-drop* [*rule-format*]:

$\forall f s m n bs bs' v. \text{field-lookup } t f m = \text{Some } (s, m+n) \longrightarrow$
 $\text{length } bs = \text{size-td } t \longrightarrow \text{length } bs' = \text{size-td } s \longrightarrow \text{wf-fd } t \longrightarrow$
 $\text{field-access } (\text{field-desc } s) (\text{field-update } (\text{field-desc } t) bs v) bs'$
 $= \text{field-access } (\text{field-desc } s) (\text{field-update } (\text{field-desc } s)$
 $\quad (\text{take } (\text{size-td } (s::('a, 'b) \text{ typ-info})) (\text{drop } n bs)) \text{undefined}) bs'$
 $\forall f s m n bs bs' v. \text{field-lookup-struct } st f m = \text{Some } (s, m+n) \longrightarrow$
 $\text{length } bs = \text{size-td-struct } st \longrightarrow \text{length } bs' = \text{size-td } s \longrightarrow \text{wf-fd-struct } st \longrightarrow$
 $\text{field-access } (\text{field-desc } s) (\text{field-update } (\text{field-desc-struct } st) bs v) bs'$
 $= \text{field-access } (\text{field-desc } s) (\text{field-update } (\text{field-desc } s)$
 $\quad (\text{take } (\text{size-td } (s::('a, 'b) \text{ typ-info})) (\text{drop } n bs)) \text{undefined}) bs'$
 $\forall f s m n bs bs' v. \text{field-lookup-list } ts f m = \text{Some } (s, m+n) \longrightarrow$
 $\text{length } bs = \text{size-td-list } ts \longrightarrow \text{length } bs' = \text{size-td } s \longrightarrow \text{wf-fd-list } ts \longrightarrow$
 $\text{field-access } (\text{field-desc } s) (\text{field-update } (\text{field-desc-list } ts) bs v) bs'$
 $= \text{field-access } (\text{field-desc } s) (\text{field-update } (\text{field-desc } s)$
 $\quad (\text{take } (\text{size-td } (s::('a, 'b) \text{ typ-info})) (\text{drop } n bs)) \text{undefined}) bs'$
 $\forall f s m n bs bs' v. \text{field-lookup-tuple } x f m = \text{Some } (s, m+n) \longrightarrow$
 $\text{length } bs = \text{size-td-tuple } x \longrightarrow \text{length } bs' = \text{size-td } s \longrightarrow \text{wf-fd-tuple } x \longrightarrow$
 $\text{field-access } (\text{field-desc } s) (\text{field-update } (\text{field-desc-tuple } x) bs v) bs'$
 $= \text{field-access } (\text{field-desc } s) (\text{field-update } (\text{field-desc } s)$
 $\quad (\text{take } (\text{size-td } (s::('a, 'b) \text{ typ-info})) (\text{drop } n bs)) \text{undefined}) bs'$

proof (*induct t and st and ts and x*)

case (*TypDesc nat typ-struct list*)
then show *?case*
apply *clarsimp*
apply (*fastforce dest!*: *wf-fd-cons-structD*
simp: *fd-cons-struct-def fd-cons-desc-def fd-cons-access-update-def*)
done

next

case (*TypScalar nat1 nat2 a*)
then show *?case by auto*

next

case (*TypAggregate list*)
then show *?case by auto*

next

```

case Nil-typ-desc
then show ?case by auto
next
case (Cons-typ-desc dt-tuple list)
then show ?case
  apply clarsimp
  subgoal for f s m n bs bs' v
    apply (clarsimp simp: fd-cons-desc-def split: option.splits)
    apply (cases f; clarsimp)
    subgoal premises prems for a lista
      using prems (1-7, 10-12)
      prems(9) [rule-format, where f= a#lista and s=s and m= m + size-td
(dt-fst dt-tuple) and n=n - size-td (dt-fst dt-tuple)]
      apply clarsimp
      apply (frule field-lookup-offset-le)
      apply simp
      apply (cases dt-tuple, clarsimp simp: fu-commutes-def)
      done
    apply (cases f; clarsimp)
    subgoal for a lista
      apply (drule spec [where x=a#lista])
      apply (drule spec [where x=s])
      apply (drule spec [where x=m])
      apply (drule spec [where x=n])
      apply clarsimp
      apply (frule field-lookup-offset-le)
      apply simp
      apply (cases dt-tuple, clarsimp)
      apply (clarsimp split: if-split-asm)
      by (fastforce dest: td-set-field-lookupD td-set-offset-size-m simp: ac-simps
drop-take min-def)
    done
  done
next
case (DTuple-typ-desc typ-desc list b)
then show ?case by auto
qed

```

lemma *field-access-update-take-dropD*:

```

[[ field-lookup t f m = Some (s,m+n); length bs = size-td t;
length bs' = size-td s; wf-fd t ]] ==>
field-access (field-desc s) (field-update (field-desc t) bs v) bs'
= field-access (field-desc s) (field-update (field-desc s)
(take (size-td (s::('a,'b) typ-info)) (drop n bs)) undefined) bs'
by (rule field-access-update-take-drop)

```

lemma (in *wf-type*) *fi-fa-consistentD*:

```

[[ field-lookup (typ-info-t TYPE('a)) f 0 = Some (d,n);

```



```

    length bs = size-of TYPE('a) ] ==>
    field-access (field-desc d) (from-bytes bs) (replicate (size-td d) 0) =
    norm-tu (export-uinfo d) (take (size-td d) (drop n bs))
apply(clarsimp simp: field-offset-def from-bytes-def size-of-def)
apply(frule field-lookup-export-uinfo-Some)
apply(subst wf-fd-norm-tuD)
    apply(fastforce intro: wf-fd-field-lookupD)
    apply(clarsimp simp: min-def split: if-split-asm)
    apply(drule td-set-field-lookupD)
    apply(drule td-set-offset-size)
    apply simp
apply(frule field-access-update-take-dropD[where v=undefined and m=0 and
    bs'=replicate (size-td d) 0,simplified]);
simp?)
apply(simp add: min-def access-ti0-def)
done

lemma fi-fu-consistent [rule-format]:
  ∀ f m n s bs v w. field-lookup t f m = Some (s,n + m) → wf-fd t →
    length bs = size-td t → length v = size-td (s::('a,'b) typ-info) →
    field-update (field-desc t) (super-update-bs v bs n) w =
    field-update (field-desc s) v (field-update (field-desc t) bs w)
  ∀ f m n s bs v w. field-lookup-struct st f m = Some (s,n + m) → wf-fd-struct st
  →
    length bs = size-td-struct st → length v = size-td (s::('a,'b) typ-info) →
    field-update (field-desc-struct st) (super-update-bs v bs n) w =
    field-update (field-desc s) v (field-update (field-desc-struct st) bs w)
  ∀ f m n s bs v w. field-lookup-list ts f m = Some (s,n + m) → wf-fd-list ts →
    length bs = size-td-list ts → length v = size-td (s::('a,'b) typ-info) →
    field-update (field-desc-list ts) (super-update-bs v bs n) w =
    field-update (field-desc s) v (field-update (field-desc-list ts) bs w)
  ∀ f m n s bs v w. field-lookup-tuple x f m = Some (s,n + m) → wf-fd-tuple x →
    length bs = size-td-tuple x → length v = size-td (s::('a,'b) typ-info) →
    field-update (field-desc-tuple x) (super-update-bs v bs n) w =
    field-update (field-desc s) v (field-update (field-desc-tuple x) bs w)
proof(induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case
    apply clarsimp
    apply(drule wf-fd-cons-structD)
    apply(clarsimp simp: fd-cons-struct-def fd-cons-double-update-def super-update-bs-def
    fd-cons-desc-def)
  done
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto

```

```

next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply clarsimp
    subgoal for f m n s bs v w
      apply(cases f; clarsimp)
      apply(clarsimp simp: fd-cons-desc-def split: option.splits)
      apply(clarsimp simp: fu-commutes-def)
      subgoal premises prems for a lista
        using prems (1-8,11-12)
          prems(10)[rule-format, where f=a#lista and s= s and m=m + size-td
(dt-fst dt-tuple)
            and n= n - size-td (dt-fst dt-tuple)]
          apply -
          apply(frule field-lookup-offset-le)
          apply simp
          apply(drule td-set-list-field-lookup-listD)
          apply(drule td-set-list-offset-size-m)
          apply (cases dt-tuple)
          apply(clarsimp simp: drop-super-update-bs take-super-update-bs)
          done
      subgoal for a lista
        apply(simp add: fu-commutes-def)
        by (smt (verit, best) add.commute append-take-drop-id diff-add-inverse2
length-append length-super-update-bs length-take min-ll
super-update-bs-append-drop-first super-update-bs-append-take-first
td-set-offset-size' td-set-tuple-field-lookup-tupleD)
      done
    done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by auto
qed

```

lemma *fi-fu-consistentD*:

```

[[ field-lookup t f 0 = Some (s,n); wf-fd t; length bs = size-td t;
length v = size-td s ]] ==>
  field-update (field-desc t) (super-update-bs v bs n) w =
    field-update (field-desc s) v (field-update (field-desc t) bs w)
using fi-fu-consistent(1) [of t f 0] by clarsimp

```

lemma (in *wf-type*) *norm*:

```

assumes lbs: length bs = size-of TYPE('a)
shows from-bytes (norm-bytes TYPE('a) bs) = ((from-bytes bs)::'a)
proof -

```

```

have wf-fd (typ-info-t TYPE('a))
  by simp
with lbs show ?thesis
  apply -
  apply (drule wf-fd-consD)
  apply (simp add: from-bytes-def norm-bytes-def)
  apply (clarsimp simp: fd-cons-def fd-cons-desc-def)
  apply (drule (3) fd-cons-update-normalise)
  apply (fastforce simp: fd-cons-update-normalise-def size-of-def wf-fd-norm-tuD
norm-desc-def
  access-ti0-def)
  done
qed

```

```

lemma (in wf-type) len:
  length bs = size-of TYPE('a)  $\implies$ 
  length (to-bytes (x::'a) bs) = size-of TYPE('a)
apply (simp add: size-of-def to-bytes-def)
by (metis fd-cons-def fd-cons-desc-def fd-cons-length-def
  field-desc.simps(1) field-desc-def local.wf-fd wf-fd-consD)

```

```

lemma (in wf-type) sz-nzero:
  0 < size-of (TYPE('a))
unfolding size-of-def
by (simp add: wf-size-desc-gt)

```

```

lemma not-disj-fn-empty1 [simp]:
   $\neg$  disj-fn [] s
by (simp add: disj-fn-def)

```

```

lemma disj-fn-comm: disj-fn a b  $\longleftrightarrow$  disj-fn b a
by (auto simp add: disj-fn-def)

```

```

lemma disj-fn-append-right: disj-fn x y  $\implies$  disj-fn x (y @ z)
by (auto simp: disj-fn-def less-eq-list-def prefix-def append-eq-append-conv2)

```

```

lemma disj-fn-cons-consI[simp]: (x = y  $\longrightarrow$  disj-fn a b)  $\implies$  disj-fn (x # a) (y #
b)
by (auto simp: disj-fn-def)

```

```

lemma fd-path-cons [simp]:
  f  $\notin$  fs-path (x#xs) = (disj-fn f x  $\wedge$  f  $\notin$  fs-path xs)
by (auto simp: fs-path-def disj-fn-def)

```

```

lemma fu-commutes-lookup-disjD:
  [ field-lookup t f m = Some (d,n); field-lookup t f' m' = Some (d',n');
  disj-fn f f'; wf-fdp (tf-set t) ]  $\implies$ 
  fu-commutes (field-update (field-desc (d::('a,'b) typ-info)))

```

(*field-update (field-desc d')*)

by (*auto simp: disj-fn-def wf-fdp-def dest!: tf-set-field-lookupD*)

lemma *field-lookup-fa-fu-lhs:*

$\forall f m n s d k. \text{field-lookup } t f m = \text{Some } (s,n) \longrightarrow \text{fa-fu-ind } (\text{field-desc } t) d k$
(size-td t)

$\longrightarrow \text{wf-fd } t \longrightarrow \text{fa-fu-ind } (\text{field-desc } (s::('a,'b) \text{ typ-info})) d k$ *(size-td s)*

$\forall f m n s d k. \text{field-lookup-struct } st f m = \text{Some } (s,n) \longrightarrow \text{fa-fu-ind } (\text{field-desc-struct } st) d k$
(size-td-struct st)

$\longrightarrow \text{wf-fd-struct } st \longrightarrow \text{fa-fu-ind } (\text{field-desc } (s::('a,'b) \text{ typ-info})) d k$ *(size-td s)*

$\forall f m n s d k. \text{field-lookup-list } ts f m = \text{Some } (s,n) \longrightarrow \text{fa-fu-ind } (\text{field-desc-list } ts) d k$
(size-td-list ts)

$\longrightarrow \text{wf-fd-list } ts \longrightarrow \text{fa-fu-ind } (\text{field-desc } (s::('a,'b) \text{ typ-info})) d k$ *(size-td s)*

$\forall f m n s d k. \text{field-lookup-tuple } x f m = \text{Some } (s,n) \longrightarrow \text{fa-fu-ind } (\text{field-desc-tuple } x) d k$
(size-td-tuple x)

$\longrightarrow \text{wf-fd-tuple } x \longrightarrow \text{fa-fu-ind } (\text{field-desc } (s::('a,'b) \text{ typ-info})) d k$ *(size-td s)*

proof (*induct t and st and ts and x*)

case (*TypDesc nat typ-struct list*)

then show *?case by (clarsimp simp: fa-fu-ind-def)*

next

case (*TypScalar nat1 nat2 a*)

then show *?case by (clarsimp simp: fa-fu-ind-def)*

next

case (*TypAggregate list*)

then show *?case by (clarsimp simp: fa-fu-ind-def)*

next

case *Nil-typ-desc*

then show *?case by (clarsimp simp: fa-fu-ind-def)*

next

case (*Cons-typ-desc dt-tuple list*)

then show *?case*

apply (*clarsimp simp: fa-fu-ind-def*)

subgoal for *f m n s d v bs bs'*

apply(*clarsimp split: option.splits*)

subgoal premises *prems*

using *prems (1-8, 10)*

apply *-*

apply (*rule*

prems (9)[rule-format, where f=f and s = s and m = m + size-td

(dt-fst dt-tuple)

and *n=n and d=d and k=length bs, OF prems(11), simplified]*)

subgoal for *v ys ys'*

apply(*drule spec [where x=v]*)

apply(*drule spec [where x=ys]*)

apply *clarsimp*

apply(*drule spec [where x=replicate (size-td-tuple dt-tuple) 0 @ ys']*)

apply *clarsimp*

apply(*drule wf-fd-cons-tupleD*)

```

    apply(clarsimp simp: fd-cons-tuple-def fd-cons-length-def fd-cons-desc-def)
    done
  subgoal by clarsimp
  subgoal by fast
  done
subgoal
  apply(drule spec [where x=f])
  apply(drule spec [where x=m])
  apply(drule spec [where x=n])
  apply(drule spec [where x=s])
  apply clarsimp
  apply(drule spec [where x=d])
  apply(drule spec [where x=length bs])
  apply(erule impE)
  apply clarsimp
subgoal for v ys ys'
  apply(drule spec [where x=v])
  apply(drule spec [where x=ys])
  apply clarsimp
  apply(drule spec [where x=ys'@replicate (size-td-list list) 0])
  apply clarsimp
  apply(drule wf-fd-cons-tupleD)
  apply(clarsimp simp: fd-cons-tuple-def fd-cons-length-def fd-cons-desc-def)
  done
  apply clarsimp
done
done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by (clarsimp simp: fa-fu-ind-def)
qed

```

lemma *field-lookup-fa-fu-lhs-listD*:

```

[[ field-lookup-list ts f m = Some (s,n); fa-fu-ind (field-desc-list ts) d k (size-td-list
ts);
wf-fd-list ts ]] ==> fa-fu-ind (field-desc (s::('a,'b) typ-info)) d k (size-td s)
using field-lookup-fa-fu-lhs(3) [of ts] by blast

```

lemma *field-lookup-fa-fu-lhs-tupleD*:

```

[[ field-lookup-tuple x f m = Some (s,n); fa-fu-ind (field-desc-tuple x) d k (size-td-tuple
x);
wf-fd-tuple x ]] ==> fa-fu-ind (field-desc (s::('a,'b) typ-info)) d k (size-td s)
using field-lookup-fa-fu-lhs(4) [of x] by blast

```

lemma *field-lookup-fa-fu-rhs*:

```

∀ f m n s d k . field-lookup t f m = Some (s,n) → fa-fu-ind d (field-desc t)
(size-td t) k
→ wf-fd t → fa-fu-ind d (field-desc (s::('a,'b) typ-info)) (size-td s) k

```

```

  ∀ f m n s d k. field-lookup-struct st f m = Some (s,n) → fa-fu-ind d (field-desc-struct
st) (size-td-struct st) k
    → wf-fd-struct st → fa-fu-ind d (field-desc (s::('a,'b) typ-info)) (size-td s) k
  ∀ f m n s d k. field-lookup-list ts f m = Some (s,n) → fa-fu-ind d (field-desc-list
ts) (size-td-list ts) k
    → wf-fd-list ts → fa-fu-ind d (field-desc (s::('a,'b) typ-info)) (size-td s) k
  ∀ f m n s d k. field-lookup-tuple x f m = Some (s,n) → fa-fu-ind d (field-desc-tuple
x) (size-td-tuple x) k
    → wf-fd-tuple x → fa-fu-ind d (field-desc (s::('a,'b) typ-info)) (size-td s) k
proof (induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case by (clarsimp simp: fa-fu-ind-def)
next
  case (TypScalar nat1 nat2 a)
  then show ?case by (clarsimp simp: fa-fu-ind-def)
next
  case (TypAggregate list)
  then show ?case by (clarsimp simp: fa-fu-ind-def)
next
  case Nil-typ-desc
  then show ?case by (clarsimp simp: fa-fu-ind-def)
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply (clarsimp simp: fa-fu-ind-def)
    subgoal for f m n s d v bs bs'
      apply(clarsimp split: option.splits)
      subgoal premises prems
        thm prems
        using prems (1-8, 10)
        apply -
        apply (rule
          prems (9)[rule-format, where f=f and s = s and m = m + size-td
(dt-fst dt-tuple)
          and n=n and d=d and k=length bs', OF prems(11), simplified])
      subgoal for v ys ys'
        apply(drule spec [where x=v])
        apply(drule spec [where x=access-ti-tuple dt-tuple v (replicate (size-td-tuple
dt-tuple) 0)@ys])
        apply clarsimp

        apply(drule wf-fd-cons-tupleD)
        apply(clarsimp simp: fd-cons-tuple-def fd-cons-length-def fd-cons-desc-def)
        apply(drule spec [where x=ys'])
        apply(clarsimp simp: fd-cons-tuple-def fd-cons-length-def fd-cons-update-access-def
fd-cons-desc-def)
        apply(clarsimp simp: fu-commutes-def)
        done
      subgoal by clarsimp

```

```

subgoal
  apply(drule spec [where x=f])
  apply(drule spec [where x=m])
  apply(drule spec [where x=n])
  apply(drule spec [where x=s])
  apply clarsimp
done
done
subgoal
  apply(drule spec [where x=f])
  apply(drule spec [where x=m])
  apply(drule spec [where x=n])
  apply(drule spec [where x=s])
  apply clarsimp
  apply(drule spec [where x=d])
  apply(drule spec [where x=length bs'])
  apply(erule impE)
  apply clarsimp
subgoal for v ys ys'
  apply(drule spec [where x=v])
  apply(drule spec [where x=ys@access-ti-list list v (replicate (size-td-list
list) 0)])
  apply(drule wf-fd-cons-tupleD)
  apply(drule wf-fd-cons-listD)
  apply(clarsimp simp: fd-cons-tuple-def fd-cons-list-def fd-cons-length-def
fd-cons-update-access-def fd-cons-desc-def)
  done
  apply fastforce
done
done
done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by (clarsimp simp: fa-fu-ind-def)
qed

lemma field-lookup-fa-fu-rhs-listD:
  [| field-lookup-list ts f m = Some (s,n);
   fa-fu-ind d (field-desc-list ts) (size-td-list ts) k; wf-fd-list ts |] ==>
  fa-fu-ind d (field-desc (s::('a,'b) typ-info)) (size-td s) k
  using field-lookup-fa-fu-rhs(3) [of ts] by blast

lemma field-lookup-fa-fu-rhs-tupleD:
  [| field-lookup-tuple x f m = Some (s,n);
   fa-fu-ind d (field-desc-tuple x) (size-td-tuple x) k; wf-fd-tuple x |] ==>
  fa-fu-ind d (field-desc (s::('a,'b) typ-info)) (size-td s) k
  using field-lookup-fa-fu-rhs(4) [of x] by blast

```

lemma *fa-fu-lookup-ind-list-tuple*:

shows

[[*field-lookup-tuple* $x f m = \text{Some } (d',n)$; *wf-fd-tuple* x ;
field-lookup-list $ts f' m' = \text{Some } (d',n')$; *wf-fd-list* ts ;
fa-fu-ind (*field-desc-list* ts) (*field-desc-tuple* x) (*size-td-tuple* x) (*size-td-list* ts)

]]

\implies *fa-fu-ind* (*field-desc* d) (*field-desc* d') (*size-td* d') (*size-td* d)

apply(*drule* (2) *field-lookup-fa-fu-lhs-listD*)

apply(*drule* (3) *field-lookup-fa-fu-rhs-tupleD*)

done

lemma *fa-fu-lookup-ind-tuple-list*:

[[*field-lookup-tuple* $x f m = \text{Some } (d,n)$; *wf-fd-tuple* x ;
field-lookup-list $ts f' m' = \text{Some } (d',n')$; *wf-fd-list* ts ;
fa-fu-ind (*field-desc-tuple* x) (*field-desc-list* ts) (*size-td-list* ts) (*size-td-tuple* x)

]]

\implies *fa-fu-ind* (*field-desc* d) (*field-desc* d') (*size-td* d') (*size-td* d)

apply(*drule* (2) *field-lookup-fa-fu-lhs-tupleD*)

apply(*drule* (3) *field-lookup-fa-fu-rhs-listD*)

done

lemma *fa-fu-lookup-disj*:

$\forall f m d n f' m' d' n'. \text{field-lookup } t f m = \text{Some } (d,n) \longrightarrow$

$\text{field-lookup } t f' m' = \text{Some } (d',n') \longrightarrow \text{disj-fn } f f' \longrightarrow$

$\text{wf-fd } t \longrightarrow \text{fa-fu-ind } (\text{field-desc } (d::('a,'b) \text{ typ-info})) (\text{field-desc } d') (\text{size-td } d')$
 $(\text{size-td } d)$

$\forall f m d n f' m' d' n'. \text{field-lookup-struct } st f m = \text{Some } (d,n) \longrightarrow$

$\text{field-lookup-struct } st f' m' = \text{Some } (d',n') \longrightarrow \text{disj-fn } f f' \longrightarrow$

$\text{wf-fd-struct } st \longrightarrow \text{fa-fu-ind } (\text{field-desc } (d::('a,'b) \text{ typ-info})) (\text{field-desc } d')$
 $(\text{size-td } d') (\text{size-td } d)$

$\forall f m d n f' m' d' n'. \text{field-lookup-list } ts f m = \text{Some } (d,n) \longrightarrow$

$\text{field-lookup-list } ts f' m' = \text{Some } (d',n') \longrightarrow \text{disj-fn } f f' \longrightarrow$

$\text{wf-fd-list } ts \longrightarrow \text{fa-fu-ind } (\text{field-desc } (d::('a,'b) \text{ typ-info})) (\text{field-desc } d') (\text{size-td } d')$
 $(\text{size-td } d)$

$\forall f m d n f' m' d' n'. \text{field-lookup-tuple } x f m = \text{Some } (d,n) \longrightarrow$

$\text{field-lookup-tuple } x f' m' = \text{Some } (d',n') \longrightarrow \text{disj-fn } f f' \longrightarrow$

$\text{wf-fd-tuple } x \longrightarrow \text{fa-fu-ind } (\text{field-desc } (d::('a,'b) \text{ typ-info})) (\text{field-desc } d')$
 $(\text{size-td } d') (\text{size-td } d)$

proof (*induct* t **and** st **and** ts **and** x)

case (*TypDesc* $\text{nat typ-struct list}$)

then show *?case by* (*clarsimp simp: disj-fn-def*)

next

case (*TypScalar* nat1 nat2 a)

then show *?case by* (*clarsimp simp: disj-fn-def*)

next

case (*TypAggregate* list)

then show *?case by* (*clarsimp simp: disj-fn-def*)

next

case *Nil-typ-desc*


```

then show ?case by (clarsimp simp: disj-fn-def)
next
case (Cons-typ-desc dt-tuple list)
then show ?case
  apply (clarsimp simp: disj-fn-def)
  apply (clarsimp simp: split: option.splits)
  subgoal by blast
  subgoal
    by (drule (3) fa-fu-lookup-ind-list-tuple; simp)
  subgoal by (drule (3) fa-fu-lookup-ind-tuple-list; simp)
  subgoal for f m d n f' m' d' n'
    apply (drule spec [where x=f ])
    apply (drule spec [where x=m])
    apply (drule spec [where x=d])
    apply clarsimp
    by blast
  done
next
case (DTuple-typ-desc typ-desc list b)
then show ?case
  apply (clarsimp simp: disj-fn-def)
  subgoal for f m d n f' m' d' n'
    apply (drule spec [where x=tl f])
    apply (drule spec [where x=m])
    apply (drule spec [where x=d])
    apply clarsimp
    apply (drule spec [where x=tl f'])
    apply (drule spec [where x=m'])
    apply (drule spec [where x=d'])
    apply clarsimp
    apply (cases f; clarsimp)
    apply (cases f'; clarsimp)
  done
done
qed

lemma fa-fu-lookup-disjD:
  [[ field-lookup t f m = Some (d,n); field-lookup t f' m' = Some (d',n');
    disj-fn f f'; wf-fd t ]]  $\implies$ 
  fa-fu-ind (field-desc (d::('a,'b) typ-info)) (field-desc d') (size-td d') (size-td d)
using fa-fu-lookup-disj(1) [of t] by fastforce

lemma field-access-update-disj:
  [[ field-lookup t f m = Some (d,n); field-lookup t f' m' = Some (d',n');
    disj-fn f f'; length bs = size-td d'; length bs' = size-td d; wf-fd t ]]  $\implies$ 
  access-ti d (update-ti-t d' bs v) bs' = access-ti d v bs'
by (fastforce dest: fa-fu-lookup-disjD simp: fa-fu-ind-def)

lemma td-set-list-intul-sub:

```

```

(d,n) ∈ td-set-list t m ⇒ {of-nat n..+size-td d} ⊆ {of-nat m..+size-td-list t}
apply(frule td-set-list-offset-le)
apply(drule td-set-list-offset-size-m)
apply clarsimp
apply(drule intvlD, clarsimp)
apply(clarsimp simp: intvl-def)
subgoal for k
  apply(rule exI [where x=k + n - m])
  apply simp
  done
done

```

lemma *td-set-tuple-intvl-sub*:

```

(d,n) ∈ td-set-tuple t m ⇒ {of-nat n..+size-td d} ⊆ {of-nat m..+size-td-tuple
t}
apply(frule td-set-tuple-offset-le)
apply(drule td-set-tuple-offset-size-m)
apply clarsimp
apply(drule intvlD, clarsimp)
apply(clarsimp simp: intvl-def)
subgoal for k
  apply(rule exI [where x=k + n - m])
  apply simp
  done
done

```

lemma *intvl-inter-le*:

```

assumes inter: a + of-nat k = c + of-nat ka and lt-d: ka < d and lt-ka: k ≤
ka
shows a ∈ {c..+d}
proof -
  from lt-ka inter have a = c + of-nat (ka - k) by (simp add: field-simps)
  moreover from lt-d have ka - k < d by simp
  ultimately show ?thesis by (force simp: intvl-def)
qed

```

lemma *intvl-inter*:

```

assumes nondisj: {a..+b} ∩ {c..+d} ≠ {}
shows a ∈ {c..+d} ∨ c ∈ {a..+b}
proof -
  from nondisj obtain k ka where a + of-nat k = c + of-nat ka
  and k < b and ka < d by (force simp: intvl-def)
  thus ?thesis by (force intro: intvl-inter-le)
qed

```

lemma *init-intvl-disj*:

```

k + z < addr-card ⇒ {(p::addr)+of-nat k..+z} ∩ {p..+k} = {}
apply(cases k ≠ 0; simp)
apply(rule ccontr)

```

```

apply(drule intvl-inter)
apply(erule disjE)
  apply(drule intvlD, clarsimp)
apply(metis add-lessD1 len-of-addr-card less-trans mod-less order-less-irrefl unat-of-nat)
apply(drule intvlD, clarsimp)
apply(subst (asm) Abs-fnat-homs)
apply(subst (asm) Word.of-nat-0)
apply(subst (asm) len-of-addr-card)
apply clarsimp
subgoal for k' q
  apply(cases q; simp)
done
done

```

lemma *final-intvl-disj*:

```


$$\llbracket k + z \leq n; n < \text{addr-card} \rrbracket \implies$$


$$\{(p::\text{addr})+\text{of-nat } k..+z\} \cap \{p+(\text{of-nat } k + \text{of-nat } z)..+n - (k+z)\} = \{\}$$

apply(cases z  $\neq$  0; simp)
apply(rule ccontr)
apply(drule intvl-inter)
apply(erule disjE)
  apply(drule intvlD, clarsimp)
  apply(subst (asm) Abs-fnat-homs)
  apply(subst (asm) Word.of-nat-0)
  apply(subst (asm) len-of-addr-card)
  apply clarsimp
subgoal for k' q
  apply(cases q; simp)
done
apply(drule intvlD, clarsimp)
apply(subst (asm) word-unat.norm-eq-iff [symmetric])
apply(subst (asm) len-of-addr-card)
apply simp
done

```

lemma *fa-fu-lookup-disj-inter*:

```


$$\forall f m d n f' d' n'. \text{field-lookup } t f m = \text{Some } (d,n) \longrightarrow$$


$$\text{field-lookup } t f' m = \text{Some } (d',n') \longrightarrow \text{disj-fn } f f' \longrightarrow$$


$$\text{wf-fd } t \longrightarrow \text{size-td } t < \text{addr-card} \longrightarrow$$


$$\{(of-nat n)::\text{addr}..+\text{size-td } (d::('a,'b) \text{ typ-info})\} \cap \{(of-nat n')..+\text{size-td } d'\} =$$


$$\{\}$$


$$\forall f m d n f' m d' n'. \text{field-lookup-struct } st f m = \text{Some } (d,n) \longrightarrow$$


$$\text{field-lookup-struct } st f' m = \text{Some } (d',n') \longrightarrow \text{disj-fn } f f' \longrightarrow$$


$$\text{wf-fd-struct } st \longrightarrow \text{size-td-struct } st < \text{addr-card} \longrightarrow$$


$$\{(of-nat n)::\text{addr}..+\text{size-td } (d::('a,'b) \text{ typ-info})\} \cap \{(of-nat n')..+\text{size-td } d'\} =$$


$$\{\}$$


$$\forall f m d n f' m d' n'. \text{field-lookup-list } ts f m = \text{Some } (d,n) \longrightarrow$$


$$\text{field-lookup-list } ts f' m = \text{Some } (d',n') \longrightarrow \text{disj-fn } f f' \longrightarrow$$


$$\text{wf-fd-list } ts \longrightarrow \text{size-td-list } ts < \text{addr-card} \longrightarrow$$


```

```

      {(of-nat n)::addr..+size-td (d::('a,'b) typ-info)} ∩ {of-nat n'..+size-td d'} =
    {}
    ∀ f m d n f' m d' n'. field-lookup-tuple x f m = Some (d,n) →
      field-lookup-tuple x f' m = Some (d',n') → disj-fn f f' →
      wf-fd-tuple x → size-td-tuple x < addr-card →
      {(of-nat n)::addr..+size-td (d::('a,'b) typ-info)} ∩ {of-nat n'..+size-td d'} =
    {}
proof(induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case by (clarsimp simp: disj-fn-def)
next
  case (TypScalar nat1 nat2 a)
  then show ?case by (clarsimp simp: disj-fn-def)
next
  case (TypAggregate list)
  then show ?case by (clarsimp simp: disj-fn-def)
next
  case Nil-typ-desc
  then show ?case by (clarsimp simp: disj-fn-def)
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply (clarsimp simp: disj-fn-def)
    apply(rule set-eqI)
    apply(clarsimp split: option.splits)
      apply fastforce
      apply(drule td-set-list-field-lookup-listD)
      apply(drule td-set-list-intvl-sub)
      apply(drule td-set-tuple-field-lookup-tupleD)
      apply(drule td-set-tuple-intvl-sub)
      apply (cases dt-tuple)
      apply(fastforce dest: init-intvl-disj)
      apply(drule td-set-list-field-lookup-listD)
      apply(drule td-set-list-intvl-sub)
      apply(drule td-set-tuple-field-lookup-tupleD)
      apply(drule td-set-tuple-intvl-sub)
      apply (cases dt-tuple)
      apply(fastforce dest: init-intvl-disj)
    apply fastforce
    done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
    apply (clarsimp simp: disj-fn-def)
    apply(rule set-eqI, clarsimp)
    subgoal for f d n f' m d' n' x
      apply(cases f; clarsimp)
      apply(cases f'; clarsimp)
      apply(fastforce simp: disj-fn-def)

```

done
done
qed

lemma *fa-fu-lookup-disj-interD*:

$\llbracket \text{field-lookup } t \text{ } f \text{ } m = \text{Some } (d, n); \text{field-lookup } t \text{ } f' \text{ } m = \text{Some } (d', n');$
 $\text{disj-fn } f \text{ } f'; \text{wf-fd } t; \text{size-td } t < \text{addr-card} \rrbracket \implies$
 $\{(of\text{-nat } n)::\text{addr}..+\text{size-td } (d::('a, 'b) \text{typ-info})\} \cap \{(of\text{-nat } n')::\text{size-td } d'\} =$
 $\{\}$
using *fa-fu-lookup-disj-inter(1)* [of *t*] **by** *clarsimp*

lemma *fa-fu-lookup-disj-inter-listD*:

$\llbracket \text{field-lookup-list } ts \text{ } f \text{ } m = \text{Some } (d, n);$
 $\text{field-lookup-list } ts \text{ } f' \text{ } m = \text{Some } (d', n'); \text{disj-fn } f \text{ } f';$
 $\text{wf-fd-list } ts; \text{size-td-list } ts < \text{addr-card} \rrbracket \implies$
 $\{(of\text{-nat } n)::\text{addr}..+\text{size-td } (d::('a, 'b) \text{typ-info})\} \cap$
 $\{(of\text{-nat } n')::\text{size-td } d'\} = \{\}$
using *fa-fu-lookup-disj-inter(3)* [of *ts*] **by** *clarsimp*

lemma (in *mem-type-sans-size*) *upd-rf*:

$\text{length } bs = \text{size-of } \text{TYPE}('a) \implies$
 $\text{update-ti-t } (\text{typ-info-t } \text{TYPE}('a)) \text{ } bs \text{ } v$
 $= \text{update-ti-t } (\text{typ-info-t } \text{TYPE}('a)) \text{ } bs \text{ } w$
by (*simp add: upd*)

lemma (in *mem-type-sans-size*) *inv*:

$\text{length } bs = \text{size-of } \text{TYPE}('a) \implies$
 $\text{from-bytes } (\text{to-bytes } (x::'a) \text{ } bs) = x$

unfolding *from-bytes-def to-bytes-def*

by (*metis fd-cons-def fd-cons-desc-def fd-cons-update-access-def field-desc.simps(1)*)

field-desc.simps(2) field-desc-def local.len local.size-of-fold
local.to-bytes-def local.upd local.wf-fd wf-fd-consD)

lemma (in *mem-type*) *align*:

$\text{align-of } (\text{TYPE}('a)) \text{ } dvd \text{ } \text{addr-card}$

proof –

have $*$: $\text{align-of } \text{TYPE}('a) < \text{addr-card}$

by (*smt (verit, best) int-dvd-int-iff less-imp-of-nat-less local.align-size-of*
local.max-size local.sz-nzero of-nat-0-less-iff of-nat-less-imp-less zdvd-not-zless)

from $*$ **show** *?thesis*

apply –

apply(*subst (asm) align-of-def*)

apply(*clarsimp simp: dvd-def align-of-def*)

apply(*rule exI [where $x = 2^{\wedge}(\text{len-of } \text{TYPE}(\text{addr-bitsize}) - \text{align-td } (\text{typ-info-t } \text{TYPE}('a)))$]*)

apply *clarsimp*

subgoal premises *prems*

proof –

```

from prems
have align-td (typ-info-t TYPE('a)) < addr-bitsize
  apply –
  apply (rule power-less-imp-less-exp [where a=2]; simp add: addr-card)
  done
then show ?thesis
  apply (simp add: addr-card flip: power-add)
  done
qed
done
qed

lemma (in mem-type) to-bytes-inj:
  to-bytes (v::'a) = to-bytes (v'::'a)  $\implies$  v=v'
  apply (drule fun-cong [where x=replicate (size-of TYPE('a)) 0])
  apply (drule arg-cong [where f=from-bytes::byte list  $\Rightarrow$  'a])
  apply (simp add: inv)
  done

lemmas unat-simps = unat-simps' max-size

lemmas (in mem-type) mem-type-simps [simp] = inv len sz-nzero max-size align
lemmas mem-type-simps [simp] = inv len sz-nzero max-size align

lemma (in mem-type) ptr-aligned-plus:
  assumes aligned: ptr-aligned (p::'a ptr)
  shows ptr-aligned (p +p i)
proof –
  have int (align-of TYPE('a)) dvd (i * int (size-of TYPE('a)))
  by (simp add: align-size-of)
  with aligned show ?thesis
  apply (cases p, simp add: ptr-aligned-def ptr-add-def scast-id)
  apply (simp only: unat-simps len-signed)
  apply (metis align align-size-of dvd-add dvd-mod dvd-mult2 mult commute)
  done
qed

lemma (in mem-type) mem-type-self [simp]:
  ptr-val (p::'a ptr)  $\in$  {ptr-val p..+size-of TYPE('a)}
  by (rule intvl-self, rule sz-nzero)

lemma (in mem-type) intvl-Suc-nmem [simp]:
  (p::addr)  $\notin$  {p + 1..+size-of TYPE('a) - Suc 0}
  by (rule intvl-Suc-nmem', subst len-of-addr-card, rule max-size)

lemma (in mem-type) wf-size-desc-typ-uinfo-t-simp [simp]:
  wf-size-desc (typ-uinfo-t TYPE('a))

```

by (simp add: typ-uinfo-t-def export-uinfo-def wf-size-desc-map)

lemma aggregate-map [simp]:
aggregate (map-td f g t) = aggregate t
apply(cases t)
subgoal for x1 st n
 apply(cases st; simp)
 done
done

lemma (in simple-mem-type) simple-tag-not-aggregate2 [simp]:
typ-uinfo-t TYPE('a) ≠ TypDesc algn (TypAggregate ts) tn
by (metis aggregate.simps aggregate-map aggregate-struct.simps(2) export-uinfo-def
 local.simple-tag local.typ-uinfo-t-def)

lemma (in simple-mem-type) simple-tag-not-aggregate3 [simp]:
typ-uinfo-t TYPE('a) ≠ TypDesc algn (TypAggregate ts) tn
by (metis aggregate.simps aggregate-struct.simps(2) simple-tag)

lemma (in mem-type) field-of-t-mem:
field-of-t (p::'a ptr) (q::'b::mem-type ptr) ⇒
 ptr-val p ∈ {ptr-val q..+size-of TYPE('b)}
apply(clarsimp simp: field-of-t-def field-of-def intvl-def)
apply(rule exI [where x=unat (ptr-val p - ptr-val q)])
apply simp
apply(drule td-set-offset-size)
apply(clarsimp simp: size-of-def)
by (metis (mono-tags, lifting) add-leD2 add-le-same-cancel2 c-type-class.size-of-def
 leD local.size-of-def local.sz-nzero nat-less-le)

lemma map-td-map:
map-td f p (map-td g q t) = map-td (λn algn. f n algn o g n algn) (p o q) t
map-td-struct f p (map-td-struct g q st) = map-td-struct (λn algn. f n algn o g n
algn) (p o q) st
map-td-list f p (map-td-list g q ts) = map-td-list (λn algn. f n algn o g n algn) (p
o q) ts
map-td-tuple f p (map-td-tuple g q x) = map-td-tuple (λn algn. f n algn o g n
algn) (p o q) x
by (induct t and st and ts and x) auto

lemma field-of-t-simple:
field-of-t p (x::'a::simple-mem-type ptr) ⇒ ptr-val p = ptr-val x
apply(clarsimp simp: field-of-t-def)
apply(cases typ-uinfo-t TYPE('a))
subgoal for x1 st n

 apply(cases st; clarsimp)

```

apply(clarsimp simp: field-of-def unat-eq-zero)
done
done

```

lemma *fold-td'-unfold*:

```

fold-td' t =
  (let (f,s) = t in
    case s of TypDesc algn' st nm =>
      (case st of
        TypScalar n algn d => d
        | TypAggregate ts => f nm (map (λx. case x of DTuple t n d => (fold-td'
(f,t),n)) ts)))
    apply (cases t, simp)
    subgoal for a b
    apply (cases b, simp)
    done
done

```

lemma *fold-td-alt-def'*:

```

fold-td f t = (case t of
  TypDesc algn' st nm =>
    (case st of
      TypScalar n algn d => d
      | TypAggregate ts => f nm (map (λx. (fold-td f (dt-fst x),dt-snd
x)) ts)))
  apply(cases t)
  apply(clarsimp split: typ-desc.split typ-struct.splits dt-tuple.splits)
  by (metis (no-types, lifting) dt-tuple.case dt-tuple.collapse)

```

lemma *fold-td-alt-def*:

```

fold-td f t ≡ (case t of
  TypDesc algn' st nm =>
    (case st of
      TypScalar n algn d => d
      | TypAggregate ts => f nm (map (λx. (fold-td f (dt-fst x),dt-snd
x)) ts)))
  by (fastforce simp: fold-td-alt-def' simp del: fold-td-def)

```

lemma *map-td'-map'*:

```

map-td f g t = (map-td' ((f,g),t))
TypDesc algn (map-td-struct f g st) (typ-name t) = (map-td' ((f,g),TypDesc algn
st (typ-name t)))
TypDesc algn (TypAggregate (map-td-list f g ts)) (typ-name t) = map-td' ((f,g),TypDesc
algn (TypAggregate ts) (typ-name t))
map-td-tuple f g x = DTuple (map-td' ((f,g),dt-fst x)) (dt-snd x) (g (dt-trd x))
  by (induct t and st and ts and x (auto simp: split-DTuple-all))

```

lemma *map-td'-map*:


```

map-td f g t = (case t of TypDesc algn st nm => TypDesc algn (case st of
  TypScalar n algn d => TypScalar n algn (f n algn d) |
  TypAggregate ts => TypAggregate (map (λx. DTuple (map-td f g (dt-fst x))
(dt-snd x) (g (dt-trd x))) ts)) nm)
apply(subst map-td'-map')
apply(subst map-td'-map')
apply(cases t, simp add: typ-struct.splits)
apply(auto simp: split-DTuple-all)
done

```

lemma *map-td-alt-def*:

```

map-td f g t ≡ (case t of TypDesc algn st nm => TypDesc algn (case st of
  TypScalar n algn d => TypScalar n algn (f n algn d) |
  TypAggregate ts => TypAggregate (map (λx. DTuple (map-td f g (dt-fst x))
(dt-snd x) (g (dt-trd x))) ts)) nm)
by (simp add: map-td'-map)

```

lemma *size-td-fm'*:

```

size-td (t::('a,'b) typ-desc) = fold-td tnSum (map-td (λn x d. n) g t)
size-td-struct (st::('a,'b) typ-struct) = fold-td-struct (typ-name t) tnSum (map-td-struct
(λn x d. n) g st)
size-td-list (ts::('a,'b) typ-tuple list) = fold-td-list (typ-name t) tnSum (map-td-list
(λn x d. n) g ts)
size-td-tuple (x::('a,'b) typ-tuple) = fold-td-tuple tnSum (map-td-tuple (λn x d.
n) g x)
by (induct t and st and ts and x) (auto simp: tnSum-def split: dt-tuple.splits)

```

lemma *size-td-fm*:

```

size-td (t::('a,'b) typ-desc) ≡ fold-td tnSum (map-td (λn algn d. n) id t)
using size-td-fm'(1) [of t id] by clarsimp

```

lemma *align-td-wo-align-fm'*:

```

align-td-wo-align (t::('a,'b) typ-desc) = fold-td tnMax (map-td (λn x d. x) g t)
align-td-wo-align-struct (st::('a,'b) typ-struct) = fold-td-struct (typ-name t) tn-
Max (map-td-struct (λn x d. x) g st)
align-td-wo-align-list (ts::('a,'b) typ-tuple list) = fold-td-list (typ-name t) tnMax
(map-td-list (λn x d. x) g ts)
align-td-wo-align-tuple (x::('a,'b) typ-tuple) = fold-td-tuple tnMax (map-td-tuple
(λn x d. x) g x)
by (induct t and st and ts and x) (auto simp: tnMax-def split: dt-tuple.splits)

```

lemma *align-td-wo-align-fm*:

```

align-td-wo-align (t::('a,'b) typ-desc) ≡ fold-td tnMax (map-td (λn algn d. algn)
id t)
using align-td-wo-align-fm'(1) [of t id] by clarsimp

```

thm *case-dt-tuple-def*

lemma *case-dt-tuple*:

snd ‹ *case-dt-tuple* ($\lambda t n d.$ *Pair* (*f t*) *n*) ‹ *X* = *dt-snd* ‹ *X*
by (*force simp: image-iff split-DTuple-all split: dt-tuple.splits*)

lemma *map-DTuple-dt-snd*:

map-td-tuple f g x = *DTuple a b c* \implies *b* = *dt-snd x*
by (*metis dt-tuple.inject map-td'-map'(4)*)

lemma *wf-desc-fm'*:

wf-desc (*t::('a,'b) typ-desc*) = *fold-td wfd* (*map-td* ($\lambda n x d.$ *True*) *g t*)
wf-desc-struct (*st::('a,'b) typ-struct*) = *fold-td-struct* (*typ-name t*) *wfd* (*map-td-struct*
($\lambda n x d.$ *True*) *g st*)
wf-desc-list (*ts::('a,'b) typ-tuple list*) = *fold-td-list* (*typ-name t*) *wfd* (*map-td-list*
($\lambda n x d.$ *True*) *g ts*)
wf-desc-tuple (*x::('a,'b) typ-tuple*) = *fold-td-tuple wfd* (*map-td-tuple* ($\lambda n x d.$ *True*)
g x)
supply *split-DTuple-all*[*simp*] *dt-tuple.splits*[*split*]
apply (*induct t and st and ts and x, all* ‹*clarsimp simp: wfd-def image-comp*[*symmetric*›)
apply (*rule iffI; clarsimp*)
apply (*metis* (*no-types*) *dt-prj-simps(2) dt-snd-map-td-list imageI*)
by (*metis* (*mono-tags*) *case-dt-tuple dt-prj-simps(2) dt-snd-map-td-list image-eqI*)

lemma *wf-desc-fm*:

wf-desc (*t::('a,'b) typ-desc*) \equiv *fold-td wfd* (*map-td* (λn *algn d.* *True*) *id t*)
using *wf-desc-fm'(1)* [*of t id*] **by** *auto*

lemma *update-tag-list-empty* [*simp*]:

(*map-td-list f g xs* = []) = (*xs* = [])
by (*cases xs, auto*)

lemma *wf-size-desc-fm'*:

wf-size-desc (*t::('a,'b) typ-desc*) = *fold-td wfsd* (*map-td* ($\lambda n x d.$ $0 < n$) *g t*)
wf-size-desc-struct (*st::('a,'b) typ-struct*) = *fold-td-struct* (*typ-name t*) *wfsd* (*map-td-struct*
($\lambda n x d.$ $0 < n$) *g st*)
ts \neq [] \implies *wf-size-desc-list* (*ts::('a,'b) typ-tuple list*) = *fold-td-list* (*typ-name t*)
wfsd (*map-td-list* ($\lambda n x d.$ $0 < n$) *g ts*)
wf-size-desc-tuple (*x::('a,'b) typ-tuple*) = *fold-td-tuple wfsd* (*map-td-tuple* ($\lambda n x$
d. $0 < n$) *g x*)
by (*induct t and st and ts and x*) (*auto simp: wfsd-def split: dt-tuple.splits*)

lemma *wf-size-desc-fm*:

wf-size-desc (*t::('a,'b) typ-desc*) \equiv *fold-td wfsd* (*map-td* (λn *algn d.* $0 < n$) *id t*)
using *wf-size-desc-fm'(1)* [*of t id*] **by** *auto*

typedef *stack-byte* = *UNIV::byte set*

by (*simp*)

definition *stack-byte-name* = "*stack-byte*"

instantiation *stack-byte* :: *c-type*

begin

definition *typ-name-itself* ($x::\text{stack-byte}$ *itself*) = *stack-byte-name*

definition

typ-info-stack-byte: *typ-info-t* ($x::\text{stack-byte}$ *itself*) \equiv
 TypDesc 0 (*TypScalar* 1 0
 ($\text{field-access} = \lambda v \text{ bs. } [\text{Rep-stack-byte } v],$
 $\text{field-update} = \lambda bs \ v. \text{ if length } bs \geq 1 \text{ then Abs-stack-byte (hd } bs) \text{ else}$
Abs-stack-byte (0::*byte*),
 $\text{field-sz} = 1$))
 stack-byte-name

instance

by (*intro-classes*) (*simp add: typ-name-itself-stack-byte-def typ-info-stack-byte*)
end

lemma *size-of-stack-byte* [*simp*]:

size-of *TYPE*(*stack-byte*) = 1
 size-td (*typ-info-t* *TYPE*(*stack-byte*)) = 1
 size-td (*typ-uinfo-t* *TYPE*(*stack-byte*)) = 1
 by (*simp-all add: size-of-def typ-info-stack-byte typ-uinfo-t-def*)

lemma *align-of-stack-byte* [*simp*]:

align-of *TYPE*(*stack-byte*) = 1
 align-td (*typ-info-t* *TYPE*(*stack-byte*)) = 0
 align-td (*typ-uinfo-t* *TYPE*(*stack-byte*)) = 0
 by (*simp-all add: align-of-def typ-info-stack-byte typ-uinfo-t-def*)

lemma *length-1-conv*: $\text{length } bs = \text{Suc } 0 \longleftrightarrow (\exists b. bs = [b])$

by (*cases* *bs*) *auto*

lemma *padding-lense-stack-byte*: *padding-lense* ($\lambda v \text{ bs. } [\text{Rep-stack-byte } v]$)

 ($\lambda bs \ v.$
 $\text{if } \text{Suc } 0 \leq \text{length } bs \text{ then Abs-stack-byte (hd } bs) \text{ else Abs-stack-byte } 0$)
 (*Suc* 0)

apply (*unfold-locales*)

apply (*simp add: padding-base.is-padding-byte-def padding-base.is-value-byte-def*
Abs-stack-byte-inverse)

subgoal

proof –

have $\text{length } [1::8 \text{ word}] = \text{Suc } 0$

by *simp*

then show *?thesis*

by (*metis* (*mono-tags*, *opaque-lifting*) *Abs-stack-byte-inject One-nat-def*
 Rep-stack-byte-inject UNIV-I hd-conv-nth length-append n-not-Suc-n
 plus-1-eq-Suc self-append-conv2 zero-neq-one)

qed

apply (*clarsimp*)

```

    apply clarsimp
    apply (clarsimp simp add: Rep-stack-byte-inverse)
    apply simp
    apply simp
    apply simp
done

instantiation stack-byte:: xmem-contained-type
begin
instance
  apply (intro-classes)
    apply (simp add: typ-info-stack-byte)
    apply (simp add: typ-info-stack-byte)
  apply (simp add: typ-info-stack-byte wf-lf-def fd-cons-desc-def fd-cons-double-update-def
    fd-cons-update-access-def fd-cons-access-update-def
    Rep-stack-byte-inverse fd-cons-length-def)
    apply (simp add: typ-info-stack-byte)
    apply simp
    apply (simp add: typ-info-stack-byte align-field-def)
    apply (simp add: typ-info-stack-byte)
    apply (simp add: addr-card)
    apply (simp add: typ-info-stack-byte)
    apply (simp add: typ-info-stack-byte)
    apply (simp add: typ-info-stack-byte wf-field-desc-def padding-lense-stack-byte)
    apply (simp add: typ-info-stack-byte)
    apply (simp add: typ-info-stack-byte)
  done

end

end

theory Vanilla32-Preliminaries
imports CTypes
begin

```

11.12 Words and Pointers

```

instantiation unit :: c-type
begin
definition [simp]: typ-name-itself (x::unit itself) = "unit"
definition typ-info-unit[simp]:
  typ-info-t (x::unit itself) ≡
  TypDesc 0 (TypScalar 1 0
    ((field-access = (λv bs. [0]), field-update = (λbs v. ()), field-sz = 1)))
  "unit"

```

```

      :: unit xtyp-info
instance by intro-classes (simp add: typ-name-itself-unit-def)
end

lemma typ-name-unit[simp]:
  typ-name (typ-info-t (TYPE(unit))) = "unit"
  typ-name (typ-uinfo-t TYPE(unit)) = "unit"
  by (simp-all add: typ-uinfo-t-def)

instantiation unit :: mem-type
begin
  definition
    to-bytes-unit :: unit  $\Rightarrow$  byte list where
    to-bytes-unit a  $\equiv$  [0]

  definition
    from-bytes-unit :: byte list  $\Rightarrow$  unit where
    from-bytes-unit bs  $\equiv$  ()

  definition
    size-of-unit :: unit itself  $\Rightarrow$  nat where
    size-of-unit x  $\equiv$  0

  definition
    align-of-unit :: unit itself  $\Rightarrow$  nat where
    align-of-unit x  $\equiv$  1

  instance
    apply intro-classes
    apply (auto simp: size-of-def align-of-def align-field-def addr-card
      wf-lf-def fd-cons-desc-def
      fd-cons-double-update-def fd-cons-update-access-def
      fd-cons-access-update-def fd-cons-length-def)

  done
end

definition
  bogus-log2lessthree (n::nat) ==
    if n = 128 then 4
    else if n = 64 then (3::nat)
    else if n = 32 then 2
    else if n = 16 then 1
    else if n = 8 then 0
    else undefined

definition
  len-exp (x::(a::len) itself)  $\equiv$  bogus-log2lessthree (len-of TYPE(a))

lemma lx8' [simp] : len-exp (x::8 itself) = 0
by (simp add: len-exp-def bogus-log2lessthree-def)

```

```

lemma lx16' [simp]: len-exp (x::16 itself) = 1
by (simp add: len-exp-def bogus-log2lessthree-def)

lemma lx32' [simp]: len-exp (x::32 itself) = 2
by (simp add: len-exp-def bogus-log2lessthree-def)

lemma lx64' [simp]: len-exp (x::64 itself) = 3
by (simp add: len-exp-def bogus-log2lessthree-def)

lemma lx-signed' [simp]: len-exp (x::('a::len) signed itself) = len-exp (TYPE('a))
by (simp add: len-exp-def)

class len8 = len +

  assumes len8-bytes: len-of TYPE('a::len) = 8 * (2len-exp TYPE('a))

  assumes len8-width: len-of TYPE('a::len) ≤ 128
begin

lemma len8-size:
  len-of TYPE('a) div 8 < addr-card
  apply(subgoal-tac len-of TYPE('a) ≤ 128)
  apply(simp add: addr-card)
  apply(rule len8-width)
  done

lemma len8-dv8:
  8 dvd len-of TYPE('a)
  by (simp add: len8-bytes)

lemma len8-pow:
  ∃ k. len-of TYPE('a) div 8 = 2k
  by (simp add: len8-bytes)

end

fun
  nat-to-bin-string :: nat ⇒ char list
  where
  ntbs: nat-to-bin-string n = (if (n = 0) then "0" else (if n mod 2 = 1 then CHR
  "1" else CHR "0")) # nat-to-bin-string (n div 2))

declare nat-to-bin-string.simps [simp del]

lemma nat-to-bin-string-simps:
  nat-to-bin-string 0 = "0"
  n > 0 ⇒ nat-to-bin-string n =
    (if n mod 2 = 1 then CHR "1" else CHR "0") # nat-to-bin-string (n div 2)

```

```

apply (induct n, auto simp: nat-to-bin-string.simps)
done

instance signed :: (len8) len8
apply intro-classes
apply (metis len8-bytes len-exp-def len-signed)
apply (metis len8-width len-signed)
done

end

theory Word-Mem-Encoding-ARM
imports ../Vanilla32-Preliminaries
begin

if-architecture-context (ARM)
begin

definition
  word-tag :: 'a::len8 word itself  $\Rightarrow$  'a word xtyp-info
where
  word-tag (w::'a::len8 word itself)  $\equiv$ 
    TypDesc (len-exp TYPE('a)) (TypScalar (len-of TYPE('a) div 8) (len-exp
    TYPE('a))
      (field-access =  $\lambda v$  bs. (rev o word-rsplit) v,
       field-update =  $\lambda bs$  v. (word-rcat (rev (take (len-of TYPE('a) div 8)
    bs))::'a::len8 word),
       field-sz = (len-of TYPE('a) div 8)))
    ("word" @ nat-to-bin-string (len-of TYPE('a)))
end

end

theory Word-Mem-Encoding-ARM64
imports Vanilla32-Preliminaries
begin

if-architecture-context (ARM64)
begin

definition
  word-tag :: 'a::len8 word itself  $\Rightarrow$  'a word xtyp-info
where
  word-tag (w::'a::len8 word itself)  $\equiv$ 
    TypDesc (len-exp TYPE('a)) (TypScalar (len-of TYPE('a) div 8)
    (len-exp TYPE('a)))

```

```

      (field-access = λv bs. (rev o word-rsplit) v,
       field-update = λbs v. (word-rcat (rev (take (len-of TYPE('a)
div 8) bs)):'a::len8 word),
       field-sz = (len-of TYPE('a) div 8))
    ("word" @ nat-to-bin-string (len-of TYPE('a)))
end
end

```

```

theory Word-Mem-Encoding-ARM-HYP
imports ../Vanilla32-Preliminaries
begin

```

```

if-architecture-context (ARM-HYP)
begin

```

definition

```

  word-tag :: 'a::len8 word itself ⇒ 'a word xtyp-info
where
  word-tag (w:'a::len8 word itself) ≡
    TypDesc (len-exp TYPE('a)) (TypScalar (len-of TYPE('a) div 8) (len-exp
TYPE('a)
      (field-access = λv bs. (rev o word-rsplit) v,
       field-update = λbs v. (word-rcat (rev (take (len-of TYPE('a) div 8)
bs)):'a::len8 word),
       field-sz = (len-of TYPE('a) div 8)))
    ("word" @ nat-to-bin-string (len-of TYPE('a)) )
end
end

```

```

theory Word-Mem-Encoding-RISCV64
imports ../Vanilla32-Preliminaries
begin

```

```

if-architecture-context (RISCV64)
begin

```

definition

```

  word-tag :: 'a::len8 word itself ⇒ 'a word xtyp-info
where
  word-tag (w:'a::len8 word itself) ≡
    TypDesc (len-exp TYPE('a)) (TypScalar (len-of TYPE('a) div 8)
(len-exp TYPE('a)
      (field-access = λv bs. (rev o word-rsplit) v,
       field-update = λbs v. (word-rcat (rev (take (len-of TYPE('a)

```



```


theory Word-Mem-Encoding-X64
  imports ../Vanilla32-Preliminaries
begin

if-architecture-context (X64)
begin

definition
  word-tag :: 'a::len8 word itself  $\Rightarrow$  'a word xtyp-info
where
  word-tag (w::'a::len8 word itself)  $\equiv$ 
    TypDesc (len-exp TYPE('a)) (TypScalar (len-of TYPE('a) div 8)
      (len-exp TYPE('a))
      (field-access =  $\lambda v$  bs. (rev o word-rsplit) v,
      field-update =  $\lambda bs$  v. (word-rcat (rev (take (len-of TYPE('a)
div 8) bs))::'a::len8 word),
      field-sz = (len-of TYPE('a) div 8)))
      ("word" @ nat-to-bin-string (len-of TYPE('a)))
end

end

theory Word-Mem-Encoding
  imports
    ARM / Word-Mem-Encoding-ARM
    ARM64 / Word-Mem-Encoding-ARM64
    ARM-HYP / Word-Mem-Encoding-ARM-HYP
    RISCV64 / Word-Mem-Encoding-RISCV64
    X64 / Word-Mem-Encoding-X64
begin

end

theory Vanilla32
imports Word-Mem-Encoding CTypes
begin


```

```

type-synonym exit-status = 32 word

instantiation word :: (len8) c-type
begin
definition typ-name-itself (w::'a::len8 word itself) =
  "word" @ nat-to-bin-string (len-of TYPE('a))
definition
  typ-info-word: typ-info-t (w::'a::len8 word itself) ≡ word-tag w
instance by (intro-classes)
  (simp add: typ-name-itself-word-def typ-info-word word-tag-def)
end

lemma align-of-word:
  align-of TYPE('a::len8 word) = len-of TYPE('a) div 8
  by (simp add: align-of-def typ-info-word word-tag-def len8-bytes)

instantiation word :: (len8) mem-type
begin

instance
proof intro-classes
  show  $\bigwedge bs\ v\ w. \text{length } bs = \text{size-of } TYPE('a\ \text{word}) \longrightarrow$ 
    update-ti-t (typ-info-t TYPE('a word)) bs v =
    update-ti-t (typ-info-t TYPE('a word)) bs w
  by (simp add: typ-info-word word-tag-def size-of-def update-ti-t-def)
next
  show wf-desc (typ-info-t TYPE('a word))
  by (simp add: typ-info-word word-tag-def)
next
  have 8 dvd len-of TYPE('a) by (rule len8-dv8)
  moreover have 0 < len-of TYPE('a) by simp
  ultimately have 0 < len-of TYPE('a) div 8 by - (erule dvdE, simp)
  thus wf-size-desc (typ-info-t TYPE('a word))
  by (simp add: typ-info-word word-tag-def len8-pow)
next
  have 8 dvd len-of TYPE('a) by (rule len8-dv8)
  thus wf-lf (lf-set (typ-info-t TYPE('a word)) [])
  by (auto simp: typ-info-word word-tag-def wf-lf-def
    fd-cons-def fd-cons-double-update-def fd-cons-update-access-def
    word-rcat-rsplit fd-cons-access-update-def fd-cons-length-def
    length-word-rsplit-exp-size' word-size fd-cons-update-normalise-def
    fd-cons-desc-def field-norm-def)
next
  show size-of TYPE('a word) < addr-card
  by (simp add: size-of-def typ-info-word word-tag-def)
  (rule len8-size)
next
  show align-of TYPE('a word) dvd size-of TYPE('a word)
  by (clarsimp simp: size-of-def align-of-word typ-info-word)

```

```

                                word-tag-def)
next
  show align-field (typ-info-t TYPE('a word))
    by (clarsimp simp: typ-info-word word-tag-def align-field-def)
next
  show wf-align (typ-info-t TYPE('a word))
    by (clarsimp simp: typ-info-word word-tag-def)
qed

end

instantiation word :: (len8) simple-mem-type
begin

instance
  apply intro-classes
  apply (clarsimp simp: typ-info-word word-tag-def typ-uinfo-t-def)
  done

end

class len-eq1 = len +
  assumes len-eq1: len-of TYPE('a::len) = 1
instance num1 :: len-eq1
by (intro-classes, simp)

class len-lg01 = len +
  assumes len-lg01: len-of TYPE('a::len) ∈ {1,2}
instance bit0 :: (len-eq1) len-lg01
by (intro-classes, simp add: len-eq1)
instance num1 :: len-lg01
by (intro-classes, simp)

class len-lg02 = len +
  assumes len-lg02: len-of TYPE('a::len) ∈ {1,2,4}
instance bit0 :: (len-lg01) len-lg02
apply intro-classes
apply (insert len-lg01[where 'a = 'a])
apply simp
done
instance num1 :: len-lg02
by (intro-classes, simp)

class len-lg03 = len +
  assumes len-lg03: len-of TYPE('a::len) ∈ {1,2,4,8}
instance bit0 :: (len-lg02) len-lg03
apply intro-classes
apply (insert len-lg02[where 'a = 'a])

```

```

apply simp
done
instance num1 :: len-lg03
by (intro-classes, simp)

class len-lg04 = len +
  assumes len-lg04: len-of TYPE('a::len) ∈ {1,2,4,8,16}
instance bit0 :: (len-lg03) len-lg04
apply intro-classes
apply (insert len-lg03[where 'a = 'a])
apply simp
done

instance num1 :: len-lg04
by (intro-classes, simp)
class len-lg15 = len +
  assumes len-lg15: len-of TYPE('a::len) ∈ {2,4,8,16,32}
instance bit0 :: (len-lg04) len-lg15
apply intro-classes
apply (insert len-lg04[where 'a = 'a])
apply simp
done

class len-lg26 = len +
  assumes len-lg26: len-of TYPE('a::len) ∈ {4,8,16,32,64}

instance bit0 :: (len-lg15) len-lg26
apply intro-classes
apply (insert len-lg15[where 'a = 'a])
apply simp
done

instance bit0 :: (len-lg26) len8
apply intro-classes
  apply simp
  apply (insert len-lg26[where 'a = 'a])
  apply (simp add: len-exp-def)
  apply (erule disjE)
  apply (simp add: bogus-log2lessthree-def len8-bytes len8-width)
  apply (erule disjE)
  apply (simp add: bogus-log2lessthree-def)
  apply (erule disjE)
  apply (simp add: bogus-log2lessthree-def)
  apply (erule disjE)
  apply (simp add: bogus-log2lessthree-def)
  apply (simp add: bogus-log2lessthree-def)
apply simp
apply auto
done

```

```

instance signed :: (len-eq1) len-eq1    using len-eq1 by (intro-classes, simp)
instance signed :: (len-lg01) len-lg01  using len-lg01 by (intro-classes, simp)
instance signed :: (len-lg02) len-lg02  using len-lg02 by (intro-classes, simp)
instance signed :: (len-lg03) len-lg03  using len-lg03 by (intro-classes, simp)
instance signed :: (len-lg04) len-lg04  using len-lg04 by (intro-classes, simp)
instance signed :: (len-lg15) len-lg15  using len-lg15 by (intro-classes, simp)
instance signed :: (len-lg26) len-lg26  using len-lg26 by (intro-classes, simp)

```

lemma

```

to-bytes (1*256*256*256 + 2*256*256 + 3*256 + (4::32 word)) bs = [4, 3,
2, 1]

```

```

by (simp add: to-bytes-def typ-info-word word-tag-def Let-def)

```

lemma size-td-words [simp]:

```

size-td (typ-info-t TYPE(8 word)) = 1
size-td (typ-info-t TYPE(16 word)) = 2
size-td (typ-info-t TYPE(32 word)) = 4
size-td (typ-info-t TYPE(64 word)) = 8
by (auto simp: typ-info-word word-tag-def)

```

lemma align-td-words [simp]:

```

align-td (typ-info-t TYPE(8 word)) = 0
align-td (typ-info-t TYPE(16 word)) = 1
align-td (typ-info-t TYPE(32 word)) = 2
align-td (typ-info-t TYPE(64 word)) = 3
by (auto simp: typ-info-word word-tag-def len8-bytes)

```

lemma size-of-words [simp]:

```

size-of TYPE(8 word) = 1
size-of TYPE(16 word) = 2
size-of TYPE(32 word) = 4
size-of TYPE(64 word) = 8
by (auto simp: size-of-def)

```

lemma align-of-words [simp]:

```

align-of TYPE(8 word) = 1
align-of TYPE(16 word) = 2
align-of TYPE(32 word) = 4
align-of TYPE(64 word) = 8
by (auto simp: align-of-word)

```

lemma size-td-words [simp]:

```

size-td (typ-info-t TYPE(8 sword)) = 1
size-td (typ-info-t TYPE(16 sword)) = 2
size-td (typ-info-t TYPE(32 sword)) = 4
size-td (typ-info-t TYPE(64 sword)) = 8
by (auto simp: typ-info-word word-tag-def)

```

```

lemma align-td-swords [simp]:
  align-td (typ-info-t TYPE(8 sword)) = 0
  align-td (typ-info-t TYPE(16 sword)) = 1
  align-td (typ-info-t TYPE(32 sword)) = 2
  align-td (typ-info-t TYPE(64 sword)) = 3
  by (auto simp: typ-info-word word-tag-def len8-bytes)

lemma size-of-swords [simp]:
  size-of TYPE(8 sword) = 1
  size-of TYPE(16 sword) = 2
  size-of TYPE(32 sword) = 4
  size-of TYPE(64 sword) = 8
  by (auto simp: size-of-def)

lemma align-of-swords [simp]:
  align-of TYPE(8 sword) = 1
  align-of TYPE(16 sword) = 2
  align-of TYPE(32 sword) = 4
  align-of TYPE(64 sword) = 8
  by (auto simp: align-of-word)

instantiation ptr :: (c-type-name) c-type
begin
definition typ-name-itself (p::'a::c-type-name ptr itself) = typ-name-itself TYPE('a)
  @ "+ptr"
definition
  typ-info-ptr:
  typ-info-t (p::'a::c-type-name ptr itself) ≡ TypDesc addr-align
    (TypScalar (addr-bitsize div 8) addr-align (|
      field-access = λp bs. rev (word-rsplit (ptr-val p)),
      field-update = λbs v. Ptr (word-rcat (rev (take (addr-bitsize div 8) bs))::addr),
      field-sz = (addr-bitsize div 8) |) )
    (typ-name-itself TYPE('a) @ "+ptr")

instance by (intro-classes) (simp add: typ-name-itself-ptr-def typ-info-ptr)
end

lemma align-of-ptr [simp]:
  align-of (p::'a::c-type ptr itself) = 2 ^ addr-align
  by (simp add: align-of-def typ-info-ptr)

instantiation ptr :: (c-type) mem-type
begin
instance
  apply (intro-classes)
  apply (auto simp: to-bytes-def from-bytes-def
    length-word-rsplit-exp-size' word-size
    word-rcat-rsplit size-of-def addr-card)

```

```

    typ-info-ptr fd-cons-double-update-def
    fd-cons-update-access-def fd-cons-access-update-def
    fd-cons-length-def fd-cons-update-normalise-def field-norm-def
    super-update-bs-def word-rsplit-rcat-size norm-bytes-def
    fd-consistent-def fd-cons-def wf-lf-def
    fd-cons-desc-def align-field-def update-ti-t-def)
done
end

lemma div-eq: 8 dvd n  $\implies$  (7 + n::nat) div 8 = n div 8
  by (simp add: div-plus-div-distrib-dvd-right)

lemma length-word-rsplit-len8: length ((word-rsplit::('a::len8 word)  $\Rightarrow$  8 word list)
w) = (LENGTH('a) div (8::nat))
proof -
  have 8 dvd len-of TYPE('a) by (rule len8-dv8)
  then show ?thesis
    by (simp add: length-word-rsplit-exp-size div-eq word-size)
qed

lemma all-value-byte-word: i < LENGTH('a::len8) div 8  $\implies$ 
padding-base.is-value-byte ( $\lambda v$  bs. rev (word-rsplit v))
( $\lambda bs$  v. word-rcat (rev (take (LENGTH('a) div 8) bs))::'a word)
(LENGTH('a) div 8) i
by (clarsimp simp add: padding-base.is-value-byte-def len8-bytes word-size word-rsplit-rcat-size)

lemma no-padding-byte-word: i < LENGTH('a::len8) div 8  $\implies$ 
 $\neg$  padding-base.is-padding-byte ( $\lambda v$  bs. rev (word-rsplit v))
( $\lambda bs$  v. word-rcat (rev (take (LENGTH('a) div 8) bs))::'a word) (LENGTH('a)
div 8)
  i
  apply (clarsimp simp add: padding-base.is-padding-byte-def len8-bytes word-size
)
  by (metis (mono-tags, opaque-lifting) Ex-list-of-length len8 len-gt-0 length-list-update
less-irrefl-nat nth-list-update unat-mono word-power-less-1)

instantiation word :: (len8) xmem-contained-type
begin
instance
  apply intro-classes
  apply (clarsimp simp add: typ-info-word word-tag-def)
  apply (clarsimp simp add: typ-info-word word-tag-def)
  apply (clarsimp simp add: typ-info-word word-tag-def)
  apply (unfold-locals; clarsimp simp add: padding-base.eq-padding-def padding-base.eq-upto-padding-def
length-word-rsplit-exp-size word-size word-rcat-rsplit div-eq length-word-rsplit-len8)
  apply (simp add: all-value-byte-word no-padding-byte-word)
  apply (simp add: wf-padding typ-info-word word-tag-def)

```

```

  apply (clarsimp simp add: typ-info-word word-tag-def)
done
end

```

```

instantiation ptr :: (c-type) simple-mem-type
begin
instance
  apply intro-classes
  apply (clarsimp simp: typ-info-ptr typ-uinfo-t-def)
done
end

```

```

lemma all-value-byte-ptr:  $\bigwedge i. i < (\text{addr-bitsize div } 8) \implies$ 
  padding-base.is-value-byte ( $\lambda p \text{ bs. rev (word-rsplit (ptr-val } p))$ )
  ( $\lambda \text{bs } v. PTR('a) (\text{word-rcat (rev (take (addr-bitsize div } 8) \text{bs})))$ )
  ( $\text{addr-bitsize div } 8$ ) i
  by (clarsimp simp add: padding-base.is-value-byte-def len8-bytes word-size word-rsplit-rcat-size)

```

```

lemma no-padding-byte-ptr:  $\bigwedge i. i < (\text{addr-bitsize div } 8) \implies$ 
   $\neg$  padding-base.is-padding-byte ( $\lambda p \text{ bs. rev (word-rsplit (ptr-val } p))$ )
  ( $\lambda \text{bs } v. PTR('a::c-type) (\text{word-rcat (rev (take (addr-bitsize div } 8) \text{bs})))$ )
  ( $\text{addr-bitsize div } 8$ ) i
  apply (clarsimp simp add: padding-base.is-padding-byte-def len8-bytes word-size
  )
  by (metis (mono-tags, opaque-lifting) Ex-list-of-length len8 len-gt-0 length-list-update
  less-irrefl-nat
  nth-list-update unat-mono word-power-less-1)

```

```

instantiation ptr :: (c-type) xmem-contained-type
begin
instance
  apply intro-classes
  apply (clarsimp simp: typ-info-ptr typ-uinfo-t-def)
  apply (clarsimp simp: typ-info-ptr typ-uinfo-t-def)
  apply (clarsimp simp: typ-info-ptr typ-uinfo-t-def)
  apply (unfold-locales; clarsimp simp add: padding-base.eq-padding-def padding-base.eq-upto-padding-def
  length-word-rsplit-exp-size word-size word-rcat-rsplit div-eq length-word-rsplit-len8)
  apply (simp add: no-padding-byte-ptr [simplified] all-value-byte-ptr [simplified])
  apply (clarsimp simp: typ-info-ptr typ-uinfo-t-def)
  apply (clarsimp simp add: typ-info-ptr typ-uinfo-t-def)
done
end

```

```

lemma size-td-ptr [simp]:
  size-td (typ-info-t TYPE('a::c-type ptr)) = addr-bitsize div 8
  by (simp add: typ-info-ptr)

```

```

lemma size-of-ptr [simp]:

```


$size-of\ TYPE('a::c-type\ ptr) = addr-bitsize\ div\ 8$
by (*simp* *add: size-of-def*)

lemma *align-td-ptr* [*simp*]: $align-td\ (typ-info-t\ TYPE('a::c-type\ ptr)) = addr-align$
by (*simp* *add: typ-info-ptr*)

lemma *ptr-add-word32-signed* [*simp*]:
fixes $a :: 32\ word\ ptr$
shows $ptr-val\ (a +_p\ x) = ptr-val\ a + 4 * of-int\ x$
by (*cases* a) (*simp* *add: CTypesDefs.ptr-add-def scast-id*)

lemma *ptr-add-word32* [*simp*]:
fixes $a :: 32\ word\ ptr$
shows $ptr-val\ (a +_p\ uint\ x) = ptr-val\ a + 4 * x$
by (*cases* a) (*simp* *add: ptr-add-def scast-id*)

lemma *ptr-add-word64-signed* [*simp*]:
fixes $a :: 64\ word\ ptr$
shows $ptr-val\ (a +_p\ x) = ptr-val\ a + 8 * of-int\ x$
by (*cases* a) (*simp* *add: CTypesDefs.ptr-add-def scast-id*)

lemma *ptr-add-word64* [*simp*]:
fixes $a :: 64\ word\ ptr$
shows $ptr-val\ (a +_p\ uint\ x) = ptr-val\ a + 8 * x$
by (*cases* a) (*simp* *add: ptr-add-def scast-id*)

lemma *ptr-add-0-id* [*simp*]: $x +_p\ 0 = x$
by (*simp* *add: CTypesDefs.ptr-add-def*)

lemma *from-bytes-ptr-to-bytes-ptr*:
 $from-bytes\ (to-bytes\ (v::addr-bitsize\ word)\ bs) = (Ptr\ v :: 'a::c-type\ ptr)$
by (*simp* *add: from-bytes-def to-bytes-def typ-info-ptr word-tag-def*
 $typ-info-word\ word-size$
 $length-word-rsplit-exp-size'\ word-rnat-rsplit\ update-ti-t-def$)

lemma *ptr-aligned-coerceI*:
 $ptr-aligned\ (ptr-coerce\ x::addr\ ptr) \implies ptr-aligned\ (x::'a::mem-type\ ptr\ ptr)$
by (*simp* *add: ptr-aligned-def*)

lemma *lift-ptr-ptr* [*simp*]:
 $\bigwedge p::'a::mem-type\ ptr\ ptr.$
 $lift\ h\ (ptr-coerce\ p) = ptr-val\ (lift\ h\ p)$
by (*simp* *add: lift-def h-val-def from-bytes-def*
 $typ-info-word\ word-tag-def\ typ-info-ptr\ update-ti-t-def$)

thm *typ-name.simps*

lemma *typ-name-itself-ptr*:
 $typ-name-itself\ (T::'a::c-type\ ptr\ itself) = typ-name-itself\ TYPE('a) @ "+ptr"$
by (*simp* *add: typ-info-ptr\ typ-name-itself-ptr-def*)

lemma *typ-name-itself-word*:

typ-name-itself (*T*::'a::len8 word *itself*) = *typ-name* (*typ-info-t* *T*)
by (*simp add: typ-info-word word-tag-def typ-name-itself-word-def*)

lemmas *Vanilla32-tags* [*simp*] =

typ-info-word
typ-info-ptr
typ-name-itself-ptr
word-tag-def
typ-name-itself-word

lemma *ptr-typ-name* [*simp*]:

typ-name (*typ-info-t* *TYPE*('a :: c-type) *ptr*) = *typ-name-itself* *TYPE*('a) @
"+ptr"
by *simp*

lemma *word-typ-name* [*simp*]:

typ-name (*typ-info-t* *TYPE*('a::len8 word)) = "word" @ *nat-to-bin-string* (*len-of*
TYPE('a))
by *simp*

lemma *typ-name-words* [*simp*]:

typ-name (*typ-uinfo-t* *TYPE*(8 word)) = "word00010"
typ-name (*typ-uinfo-t* *TYPE*(16 word)) = "word000010"
typ-name (*typ-uinfo-t* *TYPE*(32 word)) = "word0000010"
typ-name (*typ-uinfo-t* *TYPE*(64 word)) = "word00000010"
typ-name (*typ-info-t* *TYPE*(8 word)) = "word00010"
typ-name (*typ-info-t* *TYPE*(16 word)) = "word000010"
typ-name (*typ-info-t* *TYPE*(32 word)) = "word0000010"
typ-name (*typ-info-t* *TYPE*(64 word)) = "word00000010"
by (*auto simp: typ-uinfo-t-def nat-to-bin-string.simps export-uinfo-def*)

lemma *typ-name-words* [*simp*]:

typ-name (*typ-uinfo-t* *TYPE*(8 sword)) = "word00010"
typ-name (*typ-uinfo-t* *TYPE*(16 sword)) = "word000010"
typ-name (*typ-uinfo-t* *TYPE*(32 sword)) = "word0000010"
typ-name (*typ-uinfo-t* *TYPE*(64 sword)) = "word00000010"
typ-name (*typ-info-t* *TYPE*(8 sword)) = "word00010"
typ-name (*typ-info-t* *TYPE*(16 sword)) = "word000010"
typ-name (*typ-info-t* *TYPE*(32 sword)) = "word0000010"
typ-name (*typ-info-t* *TYPE*(64 sword)) = "word00000010"
by (*auto simp: typ-uinfo-t-def nat-to-bin-string.simps export-uinfo-def*)

lemma *ptr-arith*[*simp*]:

(*x* +_{*p*} *a* = *y* +_{*p*} *a*) = ((*x*::('a::c-type) *ptr*) = (*y*::'a *ptr*))
by (*clarsimp simp: CTypesDefs.ptr-add-def*)

lemma *ptr-arith'*[*simp*]:

$(ptr\text{-}coerce\ (x +_p\ a) = ptr\text{-}coerce\ (y +_p\ a)) = ((x :: ('a :: c\text{-}type)\ ptr) = (y :: 'a\ ptr))$
by (*clarsimp simp: CTypesDefs.ptr-add-def*)

lemma *typ-uinfo-t-signed-word-word-conv*: *typ-uinfo-t TYPE('a::len8) signed word*
 $= typ\text{-}uinfo\text{-}t\ TYPE('a\ word)$
by (*simp add: typ-uinfo-t-def field-norm-def*
len8-bytes word-rsplit-rcat-size word-size fun-eq-iff)

end

theory *Arrays*
imports
HOL-Library.Numeral-Type
begin

definition *has-size* :: *'a set* \Rightarrow *nat* \Rightarrow *bool* **where**
has-size s n = (*finite s* \wedge *card s* = *n*)

— If *'a* is not finite, there is no $n < CARD('a)$

definition *finite-index* :: *nat* \Rightarrow *'a::finite* **where**
finite-index = (*SOME f*. $\forall x. \exists !n. n < CARD('a) \wedge f\ n = x$)

lemma *card-image-inj*:
 $\llbracket finite\ S; \bigwedge x\ y. \llbracket x \in S; y \in S; f\ x = f\ y \rrbracket \Longrightarrow x = y \rrbracket \Longrightarrow card\ (f\ 'S) = card\ S$
by (*induct rule: finite-induct*) (*auto simp: card-insert-if*)

lemma *has-size-image-inj*:
 $\llbracket has\text{-}size\ S\ n; \bigwedge x\ y. x \in S \wedge y \in S \wedge f\ x = f\ y \Longrightarrow x = y \rrbracket \Longrightarrow has\text{-}size\ (f\ 'S)\ n$
by (*simp add: has-size-def card-image-inj*)

lemma *has-size-0[simp]*:
 $has\text{-}size\ S\ 0 = (S = \{\})$
by (*auto simp: has-size-def*)

lemma *has-size-suc*:
 $has\text{-}size\ S\ (Suc\ n) = (S \neq \{\}) \wedge (\forall a. a \in S \longrightarrow has\text{-}size\ (S - \{a\})\ n)$
unfolding *has-size-def*
by (*metis Diff-empty Suc-not-Zero bot-least card-Suc-Diff1 card-gt-0-iff finite-Diff-insert*
nat.inject neq0-conv subsetI subset-antisym)

lemma *has-index*:
 $\llbracket finite\ S; card\ S = n \rrbracket \Longrightarrow$

```

  (∃ f. (∀ m. m < n → f m ∈ S) ∧ (∀ x. x ∈ S → (∃! m. m < n ∧ f m = x)))
proof (induct n arbitrary: S)
  case 0 thus ?case by (auto simp: card-eq-0-iff)
next
  case (Suc n)
  then obtain b B where
    S: S = insert b B ∧ b ∉ B ∧ card B = n ∧ (n = 0 → B = {})
  by (auto simp: card-Suc-eq)
  with ⟨finite S⟩ Suc.hyps [of B]
  obtain f where IH: (∀ m < n. f m ∈ B) ∧ (∀ x. x ∈ B → (∃! m. m < n ∧ f m
= x)) by auto
  define f' where f' ≡ λ m. if m = card B then b else f m
  from Suc.prem1 S IH
  have (∀ m < Suc n. f' m ∈ S) ∧ (∀ x. x ∈ S → (∃! m. m < Suc n ∧ f' m = x))
  unfolding f'-def
  apply (clarsimp)
  apply (rule conjI, metis less-SucE)
  apply (metis less-SucE less-SucI nat-neq-iff)
  done
  thus ?case by blast
qed

```

```

lemma finite-index-works:
  ∃! n. n < CARD('n::finite) ∧ finite-index n = (i::'n)
proof -
  have ∃ f::nat ⇒ 'n. ∀ i. ∃! n. n < CARD('n) ∧ f n = i
  using has-index[where S = UNIV :: 'n set] by simp
  hence ∀ i. ∃! n. n < CARD('n::finite) ∧ finite-index n = (i::'n)
  unfolding finite-index-def by (rule someI-ex)
  thus ?thesis ..
qed

```

```

lemma finite-index-inj:
  [ i < CARD('a::finite); j < CARD('a) ] ⇒
  ((finite-index i :: 'a) = finite-index j) = (i = j)
  using finite-index-works[where i = finite-index j] by blast

```

```

lemma forall-finite-index:
  (∀ k::'a::finite. P k) = (∀ i. i < CARD('a) → P (finite-index i))
  by (metis finite-index-works)

```

11.13 Finite Cartesian Products

```

typedef ('a,'n::finite) array (-[-] [30,0] 31) = UNIV :: ('n => 'a) set
  by simp
setup-lifting type-definition-array

```

```

lift-definition index :: ('a,'n::finite) array ⇒ nat ⇒ 'a (-[-] [900,0] 901) is
  λ f i. f (finite-index i).

```

lemma *index-legacy-def*: $\text{index } x \ i = \text{Rep-array } x \ (\text{finite-index } i)$
by (*transfer*) *simp*

theorem *array-index-eq*:
 $((x :: 'a['n::\text{finite}]) = y) = (\forall i. i < \text{CARD}('n) \longrightarrow x.[i] = y.[i])$
by (*auto dest!*: *forall-finite-index* [*THEN iffD2*])
simp: *index-def* *Rep-array-inject* [*symmetric*])

lemmas *cart-eq* = *array-index-eq*

lemma *array-ext*:
fixes $x :: 'a['n::\text{finite}]$
shows $(\bigwedge i. i < \text{CARD}('n) \Longrightarrow x.[i] = y.[i]) \Longrightarrow x = y$
by (*simp add*: *array-index-eq*)

definition *FCP* :: $(\text{nat} \Rightarrow 'a) \Rightarrow 'a['b::\text{finite}]$ (**binder** *ARRAY* 10) **where**
 $\text{FCP} \equiv \lambda g. \text{SOME } a. \forall i. i < \text{CARD}('b) \longrightarrow a.[i] = g \ i$

definition *update* :: $'a['n::\text{finite}] \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a['n]$ **where**
 $\text{update } f \ i \ x \equiv \text{FCP} \ ((\text{index } f)(i := x))$

definition *fupdate* :: $\text{nat} \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a['b::\text{finite}] \Rightarrow 'a['b]$ **where**
 $\text{fupdate } i \ f \ x \equiv \text{update } x \ i \ (f \ (\text{index } x \ i))$

lemma *fcp-beta[simp]*: $i < \text{CARD}('n :: \text{finite}) \Longrightarrow (\text{FCP } g :: 'a['n]).[i] = g \ i$

proof –

have $\forall i < \text{CARD}('n). (\text{FCP } g :: 'a['n]).[i] = g \ i$

unfolding *FCP-def*

proof (*rule someI-ex*)

let $?g' = \lambda k :: 'n. g \ (\text{SOME } i. i < \text{CARD}('n) \wedge \text{finite-index } i = k)$

have $\forall i < \text{CARD}('n). (\text{Abs-array } ?g').[i] = g \ i$

by (*clarsimp simp*: *index-def* *Abs-array-inverse*)

(*metis* (*mono-tags*, *lifting*) *finite-index-inj* *someI-ex*)

thus $\exists x :: 'a['n]. \forall i < \text{CARD}('n). x.[i] = g \ i \ ..$

qed

thus $i < \text{CARD}('n :: \text{finite}) \Longrightarrow (\text{FCP } g :: 'a['n]).[i] = g \ i$ **by** *auto*

qed

lemma *fcp-unique*:
 $(\forall i. i < \text{CARD}('b::\text{finite}) \longrightarrow \text{index } f \ i = g \ i) =$
 $(\text{FCP } g = (f :: ('a, 'b) \text{ array}))$
by (*fastforce simp*: *cart-eq*)

lemma *fcp-eta[simp]*:
 $(\text{ARRAY } i. g.[i]) = g$
by (*simp add*: *cart-eq*)

lemma *index-update*[simp]:
 $n < \text{CARD}('b::\text{finite}) \implies \text{index} (\text{update} (f::'a['b]) n x) n = x$
by (*simp add: update-def*)

lemma *index-update2*[simp]:
 $\llbracket k < \text{CARD}('b::\text{finite}); n \neq k \rrbracket \implies \text{index} (\text{update} (f::'a['b]) n x) k = \text{index} f k$
by (*simp add: update-def*)

lemma *update-update*[simp]:
 $\text{update} (\text{update} f n x) n y = \text{update} f n y$
by (*simp add: cart-eq update-def*)

lemma *update-comm*[simp]:
 $n \neq m \implies \text{update} (\text{update} f m v) n v' = \text{update} (\text{update} f n v') m v$
by (*simp add: cart-eq update-def*)

lemma *update-index-same* [simp]:
 $\text{update} v n (\text{index} v n) = v$
by (*simp add: cart-eq update-def*)

function *foldli0* :: $(\text{nat} \Rightarrow 'a \Rightarrow 'acc \Rightarrow 'a \Rightarrow 'acc) \Rightarrow 'acc \Rightarrow \text{nat} \Rightarrow 'a['\text{index}::\text{finite}] \Rightarrow 'acc$ **where**
 $\text{foldli0} f a i \text{arr} = (\text{if } \text{CARD}('index) \leq i \text{ then } a \text{ else } \text{foldli0} f (f i a (\text{index} \text{arr } i)) (i+1) \text{arr})$
by *pat-completeness auto*

termination
by (*relation measure* $(\lambda(f,a,i,(\text{arr}::'b['c::\text{finite}])). \text{CARD}('c) - i)$) *auto*

definition *foldli* :: $(\text{nat} \Rightarrow 'a \Rightarrow 'acc \Rightarrow 'a \Rightarrow 'acc) \Rightarrow 'acc \Rightarrow ('a, 'i::\text{finite}) \text{array} \Rightarrow 'acc$ **where**
 $\text{foldli} f a \text{arr} = \text{foldli0} f a 0 \text{arr}$

definition *map-array* :: $('a \Rightarrow 'b) \Rightarrow 'a['n::\text{finite}] \Rightarrow 'b['n]$ **where**
 $\text{map-array} f a \equiv \text{ARRAY } i. f (a.[i])$

definition *map-array2* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a['n::\text{finite}] \Rightarrow 'b['n] \Rightarrow 'c['n]$ **where**
 $\text{map-array2} f a b \equiv \text{ARRAY } i. f (a.[i]) (b.[i])$

definition *zip-array* :: $'a['b::\text{finite}] \Rightarrow 'c['b] \Rightarrow ('a \times 'c)['b]$ **where**
 $\text{zip-array} \equiv \text{map-array2 } \text{Pair}$

definition *list-array* :: $('a, 'n::\text{finite}) \text{array} \Rightarrow 'a \text{ list}$ **where**
 $\text{list-array} a = \text{map} (\text{index } a) [0..<\text{CARD}('n)]$

lift-definition *set-array* :: $'a['n::\text{finite}] \Rightarrow 'a \text{ set}$ **is range** .

lemma *set-array-list*:

set-array a = set (list-array a)

by (*simp add: list-array-def index-def set-array.rep-eq image-def*)
(*metis atLeast0LessThan finite-index-works lessThan-iff*)

definition *rel-array* :: ('a ⇒ 'b ⇒ bool) ⇒ 'a['n::finite] ⇒ 'b['n] ⇒ bool **where**
rel-array f a b ≡ ∀ i < CARD('n). f (a.[i]) (b.[i])

lemma *map-array-index*:

fixes *a* :: 'a['n::finite]

shows *n* < CARD('n) ⇒ (map-array f a).[n] = f (a.[n])

by (*metis fcp-beta map-array-def*)

lemma *map-array2-index*:

fixes *a* :: 'a['n::finite]

shows *n* < CARD('n) ⇒ (map-array2 f a b).[n] = f (a.[n]) (b.[n])

by (*metis fcp-beta map-array2-def*)

lemma *zip-array-index*:

fixes *a* :: 'a['n::finite]

shows *n* < CARD('n) ⇒ (zip-array a b).[n] = (a.[n], b.[n])

by (*simp add: zip-array-def map-array2-index*)

lemma *map-array-id* [*simp*]:

map-array id = id

by (*auto simp: map-array-index array-ext*)

lemma *map-array-comp*:

map-array (g ∘ f) = map-array g ∘ map-array f

by (*auto simp: map-array-def intro!: array-ext*)

lemma *list-array-nth*:

fixes *a* :: 'a['n::finite]

shows *n* < CARD('n) ⇒ list-array a ! n = index a n

by (*simp add: list-array-def*)

lemma *list-array-length* [*simp*]:

length (list-array (a :: 'a['n::finite])) = CARD('n)

by (*simp add: list-array-def*)

lemma *in-set-array-index-conv*:

(*z* ∈ set-array (a :: 'a['n::finite])) = (∃ n < CARD('n). *z* = a.[n])

by (*metis in-set-conv-nth list-array-length list-array-nth nth-mem set-array-list*)

lemma *in-set-arrayE* [*elim!*]:

[[*z* ∈ set-array (a :: 'a['n::finite]); ∧ n . [[n < CARD('n); *z* = a.[n]]]] ⇒ *P*] ⇒

P
by (*metis in-set-array-index-conv*)

lemma *map-array-setI*:
 $(\bigwedge z. z \in \text{set-array } x \implies f z = g z) \implies \text{map-array } f x = \text{map-array } g x$
by (*rule array-ext*) (*auto simp: map-array-index in-set-array-index-conv*)

lemma *list-array-map-array*:
 $\text{list-array } (\text{map-array } f a) = \text{map } f (\text{list-array } a)$
by (*simp add: list-array-def map-array-index*)

lemma *list-array-FCP* [*simp*]:
 $\text{list-array } (\text{FCP } f :: 'a['n]) = \text{map } f [0..<\text{CARD}('n::\text{finite})]$
by (*simp add: list-array-def*)

lemma *set-array-FCP* [*simp*]:
 $\text{set-array } (\text{FCP } f :: 'a['n]) = f \{0..<\text{CARD}('n::\text{finite})\}$
by (*auto simp: set-array-list*)

lemma *map-array-set-img*:
 $\text{set-array} \circ \text{map-array } f = (\cdot) f \circ \text{set-array}$
by (*rule ext*) (*auto simp: map-array-def in-set-array-index-conv intro!: imageI*)

lemma *fcg-cong* [*cong*]:
 $(\bigwedge i. i < \text{CARD}('n::\text{finite}) \implies f i = g i) \implies \text{FCP } f = (\text{FCP } g :: 'a['n])$
by (*rule array-ext*) *simp*

bnf (*'a,'n::finite*) *array*
map: map-array
sets: set-array
bd: card-suc (BNF-Cardinal-Arithmetic.csum natLeq (card-of (UNIV :: 'n set)))
rel: rel-array

proof –
show *map-array id = id* **by** *simp*
next
fix *f :: 'a \Rightarrow 'b and g :: 'b \Rightarrow 'c*
show *map-array (g \circ f) = map-array g \circ map-array f*
by (*rule map-array-comp*)
next
fix *x :: 'a['n::finite] and f :: 'a \Rightarrow 'b and g*
assume $\bigwedge z. z \in \text{set-array } x \implies f z = g z$
thus *map-array f x = map-array g x*
by (*rule map-array-setI*)
next
fix *f :: 'a \Rightarrow 'b*
show *set-array \circ map-array f = (\cdot) f \circ set-array*
by (*rule map-array-set-img*)
next
show *card-order (card-suc (BNF-Cardinal-Arithmetic.csum natLeq (card-of UNIV)))*
by (*simp add: card-of-card-order-on card-order-card-suc card-order-csum natLeq-card-order*)
next
show *BNF-Cardinal-Arithmetic.cinfinite (card-suc (BNF-Cardinal-Arithmetic.csum*


```

natLeq (card-of UNIV)))
  by (simp add: Card-order-csum Cinfinit-card-suc card-of-card-order-on card-order-csum

      cinfinit-csum natLeq-Cinfinit natLeq-card-order)
next
  fix R :: 'a ⇒ 'b ⇒ bool and S :: 'b ⇒ 'c ⇒ bool
  show rel-array R OO rel-array S ≤ rel-array (R OO S)
    by (force simp: rel-array-def)
next
  fix R :: 'a ⇒ 'b ⇒ bool
  { fix a :: ('a × 'b)['n::finite] and i
    have [[set-array a ⊆ {(x, y). R x y}; i < CARD('n)] ⇒ R (fst (a.[i])) (snd
(a.[i]))
      by (meson Collect-case-prodD in-set-array-index-conv subset-iff)
    } note conv = this
  show rel-array R =
    (λx y. ∃z. set-array z ⊆ {(x, y). R x y} ∧ map-array fst z = x ∧ map-array
snd z = y)
    unfolding rel-array-def
    apply (intro ext iffI)
    subgoal for a b
      apply (rule exI [where x=zip-array a b])
      by (auto intro!: array-ext simp: conv map-array-index zip-array-index)
    subgoal
      by (auto intro!: array-ext simp: conv map-array-index zip-array-index)
    done
next
  show regularCard (card-suc (BNF-Cardinal-Arithmetic.csum natLeq (card-of UNIV)))
    by (simp add: Card-order-csum card-of-card-order-on card-order-csum cinf-
nite-csum
        natLeq-Cinfinit natLeq-card-order regularCard-card-suc)
next
  fix x :: 'a['n::finite]
  let ?U = UNIV :: 'n set
  have ordLeq3 (card-of (set-array x)) (card-of ?U) by transfer (rule card-of-image)
  also
  have ordLeq3 (card-of ?U) (BNF-Cardinal-Arithmetic.csum natLeq (card-of ?U))
    by (rule ordLeq-csum2) (rule card-of-Card-order)
  finally
  have ordLeq3 (card-of (set-array x)) (BNF-Cardinal-Arithmetic.csum natLeq
(card-of ?U)) .
  then show ordLess2 (card-of (set-array x))
    (card-suc (BNF-Cardinal-Arithmetic.csum natLeq (card-of ?U)))
    using ordLeq-ordLess-trans card-suc-greater card-order-csum natLeq-card-order
card-of-card-order-on
    by blast
qed

lemma arr-fupdate-same: i < CARD('n) ⇒

```

```

    index (fupdate i f (x::'a['n::finite])) i = f (index x i)
  by (simp add: fupdate-def)

lemma arr-fupdate-other: j < CARD('n)  $\implies$  i  $\neq$  j  $\implies$ 
  index (fupdate i f (x::'a['n::finite])) j = (index x j)
  by (simp add: fupdate-def)

lemmas arr-fupdate-simps = arr-fupdate-same arr-fupdate-other

lemma surj-finite-index: surj finite-index
  by (metis finite-index-works surj-def)

lemma finite-index-inj-on: inj-on (finite-index::nat $\implies$ 'a::finite) {i. i < CARD('a)}
  by (simp add: finite-index-inj inj-on-def)

lemma finite-index-bij-betw: bij-betw (finite-index::nat $\implies$ 'a::finite) {i. i < CARD('a)}
  UNIV
  using finite-index-inj-on
  by (metis (mono-tags, lifting) UNIV-I bij-betw-def finite-index-works imageI
    mem-Collect-eq subsetI subset-antisym)

end

theory Padding
imports Main
begin

lemma (0::nat) mod 2 = XXX
  apply (simp)
  oops

lemma (2 - (0::nat) mod 2) = XXX
  apply (simp)
  oops

lemma (2 - (0::nat) mod 2) mod 2 = XXX
  apply (simp)
  oops

definition padup :: nat  $\implies$  nat  $\implies$  nat where
  padup align n  $\equiv$  (align - n mod align) mod align

lemma padup-dvd:
  0 < b  $\implies$  (padup b n = 0) = (b dvd n)
  unfolding padup-def
  apply (subst dvd-eq-mod-eq-0)
  apply (subst mod-if [where m=b - n mod b])

```

```

apply clarsimp
apply(insert mod-less-divisor [of b n])
apply arith
done

```

```

lemma dvd-padup-add:
   $0 < x \implies x \text{ dvd } y + \text{padup } x \ y$ 
apply(clarsimp simp: padup-def)
apply(subst mod-if [where m=x - y mod x])
apply(clarsimp split: if-split-asm)
apply(rule conjI)
apply clarsimp
apply(subst ac-simps)
apply(subst diff-add-assoc)
apply(rule mod-less-eq-dividend)
apply(rule dvd-add)
apply simp
apply(subst minus-div-mult-eq-mod[symmetric])
apply(subst diff-diff-right)
apply(subst ac-simps)
apply(subst minus-mod-eq-mult-div[symmetric])
apply simp
apply simp
apply(auto simp: dvd-eq-mod-eq-0)
done

```

end

```

theory Lens
  imports Main Distinct-Prop
begin

```

11.14 Auxiliary Theorems

```

lemma distinct-prop-cons:
   $\text{list-all } (R \ x) \ xs \implies \text{distinct-prop } R \ xs \implies \text{distinct-prop } R \ (x \# \ xs)$ 
by (simp add: list-all-iff)

```

```

lemma distinct-prop-distrib-all:
   $\text{distinct-prop } (\lambda x \ y. \forall a. R \ a \ x \ y) \ xs \longleftrightarrow (\forall a. \text{distinct-prop } (R \ a) \ xs)$ 
by (induction xs) auto

```

```

lemma pairwise-set-of-distinct-prop:
   $(\bigwedge a \ b. R \ a \ b \longleftrightarrow R \ b \ a) \implies \text{distinct-prop } R \ xs \implies \text{pairwise } R \ (\text{set } xs)$ 
by (induction xs) (auto simp: pairwise-insert)

```

```

lemma distinct-prop-iff-nth:
   $\text{distinct-prop } R \ xs \longleftrightarrow (\forall i \ j. i < j \longrightarrow j < \text{length } xs \longrightarrow R \ (xs!i) \ (xs!j))$ 

```

proof (*induction xs*)
have *split*: $(\forall i. P\ i) \longleftrightarrow P\ 0 \wedge (\forall i. P\ (Suc\ i))$ **for** $P :: nat \Rightarrow bool$
by (*metis not0-implies-Suc*)

case (*Cons x xs*)
then have *distinct-prop* $R\ (x\ \#\ xs) \longleftrightarrow$
 $(\forall j. j < length\ xs \longrightarrow R\ x\ (xs\ !\ j)) \wedge$
 $(\forall i\ j. i < j \longrightarrow j < length\ xs \longrightarrow R\ (xs\ !\ i)\ (xs\ !\ j))$
by (*auto simp add: set-conv-nth*)
also have $\dots \longleftrightarrow (\forall i\ j. i < j \longrightarrow j < Suc\ (length\ xs) \longrightarrow R\ ((x\ \#\ xs)\ !\ i)\ ((x\ \#\ xs)\ !\ j))$
apply (*subst (4) split*)
apply (*subst (4) 6 split*)
apply *simp*
done
finally show *?case* **by** *simp*
qed *simp*

lemma *distinct-prop-mono*:
 $(\bigwedge x\ y. x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow P\ x\ y \Longrightarrow R\ x\ y) \Longrightarrow$
 $distinct-prop\ P\ xs \Longrightarrow distinct-prop\ R\ xs$
by (*induction xs*) *auto*

lemma *distinct-prop-nil*: $distinct-prop\ R\ []$
by *simp*

lemma *list-all-concat*: $list-all\ p\ (concat\ xs) = list-all\ (list-all\ p)\ xs$
by (*induction xs*) *auto*

lemma *list-all-conj*: $list-all\ P\ xs \Longrightarrow list-all\ Q\ xs \Longrightarrow list-all\ (\lambda x. P\ x \wedge Q\ x)\ xs$
using *Ball-set* **by** *blast*

lemma *list-all-zip-iff-list-all2*:
 $length\ as = length\ bs \Longrightarrow list-all\ R\ (zip\ as\ bs) \longleftrightarrow list-all2\ (\lambda a\ b. R\ (a,\ b))\ as\ bs$
by (*simp add: Ball-set-list-all list-all2-iff*)

lemma *disjnt-comm*: $disjnt\ A\ B \longleftrightarrow disjnt\ B\ A$
by (*metis disjnt-sym*)

lemma *disjnt-image*: $disjnt\ (f\ 'x)\ (f\ 'y) \Longrightarrow disjnt\ x\ y$
by (*auto simp: disjnt-def*)

lemma *disjnt-of-nat*:
 $s \subseteq \{0..<2^{LENGTH('a::len)}\} \Longrightarrow t \subseteq \{0..<2^{LENGTH('a)}\} \Longrightarrow$
 $disjnt\ ((of-nat :: nat \Rightarrow 'a\ word)\ 's)\ (of-nat\ 't) \longleftrightarrow disjnt\ s\ t$
proof (*rule disjnt-inj-on-iff[of - Pow {0..<2^{LENGTH('a)}}]*)
qed (*simp-all add: inj-on-word-of-nat*)

lemma *list-all-zip-zip-cons*:

$R\ a\ b\ c \implies$
 $list\text{-}all\ (\lambda(a, b, c). R\ a\ b\ c)\ (zip\ as\ (zip\ bs\ cs)) \implies$
 $list\text{-}all\ (\lambda(a, b, c). R\ a\ b\ c)\ (zip\ (a\#\!as)\ (zip\ (b\#\!bs)\ (c\#\!cs)))$
by *simp*

lemma *list-all-zip-zip-empty*:

$list\text{-}all\ (\lambda(a, b, c). R\ a\ b\ c)\ (zip\ []\ (zip\ []\ []))$
by *simp*

lemma *list-all-cons*: $P\ x \implies list\text{-}all\ P\ xs \implies list\text{-}all\ P\ (x\ \#\ xs)$

by *simp*

lemma *list-all-nil*: $list\text{-}all\ P\ []$

by *simp*

lemma *fold-functor*:

$F\ id = id \implies (\bigwedge a\ b. F\ (a\ \circ\ b) = F\ a\ \circ\ F\ b) \implies F\ (fold\ a\ xs) = fold\ (\lambda x. F\ (a\ x))\ xs$
by (*induction xs; simp*)

11.15 Legacy definition *fg-cons*

definition *fg-cons* :: $('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a \Rightarrow 'a) \Rightarrow bool$ **where**

$fg\text{-}cons\ acc\ upd \equiv$
 $(\forall bs\ v. acc\ (upd\ bs\ v) = bs) \wedge (\forall bs\ bs'\ v. upd\ bs\ (upd\ bs'\ v) = upd\ bs\ v) \wedge$
 $(\forall v. upd\ (acc\ v)\ v = v)$

11.16 Lense definition using an update function *lense*

type-synonym $'a\ upd = 'a \Rightarrow 'a$

named-theorems *update-compose*

locale *lense* =

fixes

$get :: 's \Rightarrow 'a$ **and**

$upd :: 'a\ upd \Rightarrow 's\ upd$

assumes *get-upd* [*simp*]: $get\ (upd\ f\ s) = f\ (get\ s)$

assumes *upd-same* : $f\ (get\ s) = get\ s \implies upd\ f\ s = s$

assumes *upd-compose*[*update-compose, simp*]: $upd\ f\ (upd\ g\ s) = upd\ (f\ \circ\ g)\ s$

begin

lemma *upd-comp*[*simp*]: $upd\ f\ \circ\ upd\ g = upd\ (f\ \circ\ g)$

by (*simp add: fun-eq-iff*)

lemma *upd-get* [*simp*]: $upd\ (\lambda_. get\ s)\ s = s$

by (*simp add: upd-same*)

```

lemma upd-id [simp]: upd ( $\lambda x. x$ ) s = s
  by (simp add: upd-same)
lemma upd-cong:  $f$  (get s) =  $f'$  (get s)  $\implies$  upd f s = upd f' s
  by (metis fun-upd-same get-upd upd-compose upd-same)

lemma upd-comp-fold: upd (fold f xs) = fold (upd  $\circ$  f) xs
  apply (induction xs)
  apply (simp-all add: upd-same del: comp-apply flip: upd-comp)
  apply (simp add: fun-eq-iff comp-def)
  done

lemma fg-cons: fg-cons get (upd  $\circ$  ( $\lambda x. x$ ))
  unfolding fg-cons-def o-def by (simp add: comp-def)

end

lemma lenseI:
  fixes get upd
  assumes 1: ( $\bigwedge s f. \text{get } (\text{upd } f s) = f (\text{get } s)$ ) ( $\bigwedge s. \text{upd } (\lambda x. \text{get } s) s = s$ )
    and 2[cong]: ( $\bigwedge s f g. f (\text{get } s) = g (\text{get } s) \implies \text{upd } g s = \text{upd } f s$ )
    and 3: ( $\bigwedge s f g. \text{upd } f (\text{upd } g s) = \text{upd } (f \circ g) s$ )
  shows lense get upd
  using 1 3
  by (simp add: lense-def)

lemma lenseI-equiv:
  ( $\bigwedge x. f (g x) = x \implies \bigwedge x. g (f x) = x \implies \text{lense } g (\lambda v x. f (v (g x)))$ )
  by (simp add: lense-def)

lemma lense-comp:
  assumes lense1: lense sel1 upd1
  assumes lense2: lense sel2 upd2
  shows lense (sel2 o sel1) (upd1 o upd2)
proof –
  interpret lense1: lense sel1 upd1 by (rule lense1)
  interpret lense2: lense sel2 upd2 by (rule lense2)
  show ?thesis
    apply (unfold-locales)
    subgoal by (simp add: comp-def)
    subgoal using lense1.upd-same lense2.upd-same by simp
    subgoal using lense1.upd-compose lense2.upd-compose by (simp add: comp-def)
    done
qed

lemma lense-compose:
  assumes lense sel1 upd1 lense sel2 upd2
  shows lense ( $\lambda x. \text{sel2 } (\text{sel1 } x)$ ) ( $\lambda f. \text{upd1 } (\text{upd2 } f)$ )
  using lense-comp[OF assms] by (simp add: comp-def)

```

lemma *lense-of-fg-cons*:
 $fg\text{-cons } g \ s \implies lense\ g \ (\lambda f\ x.\ s \ (f \ (g \ x)) \ x)$
by (*simp add: fg-cons-def lense-def*)

lemma *lense-of-fg-cons'*:
 $fg\text{-cons } g \ (u \circ (\lambda x \ -. \ x)) \implies$
 $(\bigwedge f\ x.\ u \ f \ x = u \ (\lambda \cdot.\ f \ (g \ x)) \ x) \implies$
 $lense\ g \ u$
by (*simp-all add: fg-cons-def fun-eq-iff comp-def lense-def*) *metis*

11.17 Update for functions

definition *upd-fun* :: $'a \Rightarrow 'b \text{ upd} \Rightarrow ('a \Rightarrow 'b) \text{ upd}$ **where**
 $upd\text{-fun } i \ f \ g = g(i := f \ (g \ i))$

global-interpretation *upd-fun*: $lense \ \lambda f.\ f \ a \ \text{upd-fun } a$
by (*simp add: lense-def upd-fun-def*)

lemma *upd-fun-ne*: $i \neq j \implies upd\text{-fun } i \ f \ g \ j = g \ j$
by (*simp add: upd-fun-def fun-upd-def*)

lemma *upd-fun-commute*: $i \neq j \implies upd\text{-fun } i \ f \circ upd\text{-fun } j \ g = upd\text{-fun } j \ g \circ upd\text{-fun } i \ f$
by (*auto simp: upd-fun-def fun-eq-iff fun-upd-def*)

11.18 Scenes

Scenes allow us to represent a projection/lense without referring to a second type.

type-synonym $'a \ \text{scene} = 'a \Rightarrow 'a \Rightarrow 'a$

locale *is-scene* =
fixes $s :: 'a \ \text{scene}$
assumes *left[simp]*: $s \ (s \ a \ b) \ c = s \ a \ c$
assumes *right[simp]*: $s \ a \ (s \ b \ c) = s \ a \ c$
assumes *idem[simp]*: $s \ a \ a = a$

lemma *is-scene-all[simp]*: $is\text{-scene} \ (\lambda a \ b.\ a)$
by (*simp add: is-scene-def*)

lemma *is-scene-id[simp]*: $is\text{-scene} \ (\lambda a.\ id)$
by (*simp add: is-scene-def*)

lemma *is-scene-flip*: $is\text{-scene } m \implies is\text{-scene} \ (\lambda a \ b.\ m \ b \ a)$
by (*simp add: is-scene-def*)

lemma *is-scene-of-lense*: $lense \ r \ w \implies is\text{-scene} \ (\lambda a.\ w \ (\lambda \cdot.\ r \ a))$

by (*simp add: lense-def is-scene-def K-record-comp*)

lemma *is-scene-of-fg-cons*: $fg-cons\ r\ w \implies is-scene\ (\lambda a. w\ (r\ a))$
by (*simp add: fg-cons-def is-scene-def*)

lemma *scene-comp-idem*: $is-scene\ m \implies m\ a \circ m\ a = m\ a$
by (*simp add: fun-eq-iff is-scene.right*)

definition *comm-scene* :: $'a\ scene \Rightarrow 'a\ scene \Rightarrow bool$ **where**
 $comm-scene\ m1\ m2 \iff (\forall a\ b. m1\ a\ (m2\ a\ b) = m2\ a\ (m1\ a\ b))$

lemma *comm-scene-comm*: $comm-scene\ m1\ m2 \iff comm-scene\ m2\ m1$
by (*simp add: comm-scene-def metis*)

lemma *comm-scene-sym*[*intro*]: $comm-scene\ m1\ m2 \implies comm-scene\ m2\ m1$
by (*simp add: comm-scene-comm*)

lemma *comm-sceneD*: $comm-scene\ m1\ m2 \implies m1\ a\ (m2\ a\ c) = m2\ a\ (m1\ a\ c)$
by (*simp add: comm-scene-def*)

lemma *comm-scene-all-left*[*simp*]: $is-scene\ m \implies comm-scene\ (\lambda a\ b. a)\ m$
by (*simp add: comm-scene-def is-scene.idem*)

lemma *comm-scene-all-right*[*simp*]: $is-scene\ m \implies comm-scene\ m\ (\lambda a\ b. a)$
by (*simp add: comm-scene-def is-scene.idem*)

lemma *comm-scene-id-left*[*simp*]: $comm-scene\ (\lambda a. id)\ m$
by (*simp add: comm-scene-def*)

lemma *comm-scene-id-right*[*simp*]: $comm-scene\ m\ (\lambda a. id)$
by (*simp add: comm-scene-def*)

lemma *comm-scene-refl*[*intro, simp*]: $comm-scene\ m\ m$
by (*simp add: comm-scene-def*)

lemma *is-scene-comp*:
 $is-scene\ m1 \implies is-scene\ m2 \implies comm-scene\ m1\ m2 \implies is-scene\ (\lambda a. m1\ a \circ m2\ a)$
by (*simp add: is-scene-def comm-scene-def metis*)

lemma *comm-scene-comp-left*:
 $comm-scene\ m1\ m2 \implies comm-scene\ m1\ m \implies comm-scene\ m2\ m \implies comm-scene\ (\lambda a. m1\ a \circ m2\ a)\ m$
by (*simp add: is-scene-def comm-scene-def*)

lemma *comm-scene-comp-right*:
 $comm-scene\ m\ m1 \implies comm-scene\ m\ m2 \implies comm-scene\ m1\ m2 \implies comm-scene\ m\ (\lambda a. m1\ a \circ m2\ a)$
using *comm-scene-comp-left*[*of m1 m2 m*] **by** (*simp add: comm-scene-comm*)

lemma *comm-scene-fold*:

pairwise comm-scene (insert m (set ms)) \implies comm-scene (λa . fold (λm . m a) ms) m

by (*induction ms arbitrary: m*) (*auto intro!*: *comm-scene-comp-left simp: pairwise-def*)

lemma *is-scene-fold*:

list-all is-scene ms \implies pairwise comm-scene (set ms) \implies is-scene (λa . fold (λm . m a) ms)

by (*induction ms*) (*simp-all add: is-scene-comp comm-scene-fold pairwise-def*)

lemma *is-scene-fold'*:

list-all (λx . is-scene (m x)) ms \implies pairwise comm-scene (m ' set ms) \implies is-scene (λa . fold (λx . m x a) ms)

using *is-scene-fold*[*of map m ms*] **by** (*simp add: list.pred-map comp-def fold-map*)

This expresses "disjointness" on scenes, saying that two scenes occupy disjoint parts of the type.

It is stronger than "commutativity" $\forall a c. m1 a (m2 a b) = m2 a (m1 a b)$ which is enough to show composition, but in our cases we are interested in disjointness. Important difference: a scene m "commutes" with itself, but only $\lambda a. id$ is disjoint with itself.

definition *disjnt-scene* :: 'a scene \Rightarrow 'a scene \Rightarrow bool **where**

disjnt-scene m1 m2 \longleftrightarrow ($\forall a b c. m1 a (m2 b c) = m2 b (m1 a c)$)

lemma *comm-scene-of-disjnt-scene*[*intro, simp*]: *disjnt-scene m1 m2 \implies comm-scene m1 m2*

by (*simp add: disjnt-scene-def comm-scene-def*)

lemma *disjnt-scene-comm*: *disjnt-scene m1 m2 \longleftrightarrow disjnt-scene m2 m1*

by (*simp add: disjnt-scene-def metis*)

lemma *disjnt-scene-sym*[*intro*]: *disjnt-scene m1 m2 \implies disjnt-scene m2 m1*

by (*simp add: disjnt-scene-comm*)

lemma *disjnt-sceneD*: *disjnt-scene m1 m2 \implies m1 a (m2 b c) = m2 b (m1 a c)*

by (*simp add: disjnt-scene-def*)

lemma *disjnt-sceneD-left*: *is-scene m1 \implies disjnt-scene m1 m2 \implies m1 (m2 a b) c = m1 b c*

by (*simp add: disjnt-scene-def is-scene-def metis*)

lemma *disjnt-sceneD-right*: *is-scene m2 \implies disjnt-scene m1 m2 \implies m2 (m1 a b) c = m2 b c*

using *disjnt-sceneD-left*[*of m2 m1, OF - disjnt-scene-sym*] .

lemma *disjnt-scene-all-left*[*simp*]: *disjnt-scene ($\lambda a b. a$) m \longleftrightarrow m = ($\lambda a. id$)*

by (*auto simp add: disjnt-scene-def fun-eq-iff*)

lemma *disjnt-scene-all-right[simp]*: *disjnt-scene* m $(\lambda a b. a) \longleftrightarrow m = (\lambda a. id)$
by (*auto simp add: disjnt-scene-def fun-eq-iff*)

lemma *disjnt-scene-id-left[simp]*: *disjnt-scene* $(\lambda a. id)$ m
by (*simp add: disjnt-scene-def*)

lemma *disjnt-scene-id-right[simp]*: *disjnt-scene* m $(\lambda a. id)$
by (*simp add: disjnt-scene-def*)

lemma *disjnt-scene-comp-left*:
comm-scene $m1$ $m2 \implies disjnt-scene$ $m1$ $m \implies disjnt-scene$ $m2$ $m \implies$
disjnt-scene $(\lambda a. m1\ a \circ m2\ a)$ m
by (*simp add: is-scene-def disjnt-scene-def*)

lemma *disjnt-scene-comp-right*:
disjnt-scene m $m1 \implies disjnt-scene$ m $m2 \implies comm-scene$ $m1$ $m2 \implies$
disjnt-scene m $(\lambda a. m1\ a \circ m2\ a)$
using *disjnt-scene-comp-left[of m1 m2 m]* **by** (*simp add: disjnt-scene-comm*)

lemma *disjnt-scene-fold*:
list-all (*disjnt-scene* m) $ms \implies pairwise$ *comm-scene* (*set* ms) \implies
disjnt-scene $(\lambda a. fold$ $(\lambda m. m\ a)$ $ms)$ m
by (*induction ms arbitrary: m*)
(auto simp: pairwise-def intro!: disjnt-scene-comp-left comm-scene-fold)

lemma *fold-disjnt-scene*:
list-all is-scene $ms \implies pairwise$ *comm-scene* (*set* ms) \implies
list-all $(\lambda m. disjnt-scene\ m\ m' \vee m = m')$ $ms \implies m' \in set\ ms \implies$
fold $(\lambda m. m\ (m'\ a\ b))$ $ms\ c = m'\ a\ (fold$ $(\lambda m. m\ b)$ $ms\ c)$

proof (*induction ms rule: rev-induct*)

case (*snoc* $m\ ms$)

show *?case*

proof *cases*

assume $m' \in set\ ms$

with *snoc* **have** *eq[simp]*: *fold* $(\lambda m. m\ (m'\ a\ b))$ $ms\ c = m'\ a\ (fold$ $(\lambda m. m\ b)$ $ms\ c)$

and $m-m'$: *disjnt-scene* $m\ m' \vee m = m'$

and m : *is-scene* m **and** m' : *is-scene* m'

by (*auto simp: list-all-iff pairwise-def*)

show *?thesis*

using $m-m'\ m\ m'$

by (*auto simp: disjnt-sceneD-left disjnt-sceneD*

is-scene.right[OF m'] is-scene.left[OF m'])

next

assume $m' \notin set\ ms$

with *snoc.prem*s **have** $m'-ms$: *list-all* (*disjnt-scene* m') ms

and *[simp]*: *list-all is-scene* ms *pairwise comm-scene* (*set* ms)

```

    and [simp]: m = m' and m': is-scene m'
    by (auto simp: list-all-iff pairwise-def)

    have [simp]: fold (λm. m (m' a b)) ms c = fold (λm. m b) ms c
    by (rule disjnt-sceneD-left[OF is-scene-fold disjnt-scene-fold[OF m'-ms]]; simp)

    show ?thesis
    by (simp add: is-scene.right[OF m'] is-scene.left[OF m'])
  qed
qed simp

end

```

```

theory CompoundCTypes

```

```

  imports
    Vanilla32
    Padding
    Lens

```

```

begin

```

```

lemma simple-type-dest: ¬aggregate t ⇒ ∃ n sz align align' d. t = TypDesc align'
(TypScalar sz align d) n
  apply (cases t)
  subgoal for x1 st n
    apply (cases st)
    apply auto
  done
done

```

```

definition empty-typ-info :: nat ⇒ typ-name ⇒ ('a,'b) typ-desc where
  empty-typ-info algn tn ≡ TypDesc algn (TypAggregate []) tn

```

```

primrec

```

```

  extend-ti :: 'a xtyp-info ⇒ 'a xtyp-info ⇒ nat ⇒ field-name ⇒ 'a field-desc ⇒ 'a
xtyp-info and
  extend-ti-struct :: 'a xtyp-info-struct ⇒ 'a xtyp-info ⇒ field-name ⇒ 'a field-desc
⇒ 'a xtyp-info-struct

```

```

where

```

```

  et0: extend-ti (TypDesc algn' st nm) t algn fn d = TypDesc (max algn' (max
algn (align-td t))) (extend-ti-struct st t fn d) nm

```

```

| et1: extend-ti-struct (TypScalar n sz algn) t fn d = TypAggregate [DTuple t fn d]
| et2: extend-ti-struct (TypAggregate ts) t fn d = TypAggregate (ts@[DTuple t fn
d])

```

```

lemma aggregate-empty-typ-info [simp]:
  aggregate (empty-typ-info algn tn)

```

by (simp add: empty-typ-info-def)

lemma *aggregate-extend-ti* [simp]:
aggregate (extend-ti tag t algn f d)
apply(cases tag)
subgoal for x1 typ-struct xs
 apply(cases typ-struct, auto)
 done
done

lemma *typ-name-extend-ti* [simp]: typ-name (extend-ti T t algn fn d) = typ-name T
by (cases T) auto

definition *update-desc* :: ('a ⇒ 'b) ⇒ ('b ⇒ 'a ⇒ 'a) ⇒ 'b field-desc ⇒ 'a field-desc
where
 update-desc acc upd d ≡ (field-access = (field-access d) ∘ acc,
 field-update = λbs v. upd (field-update d bs (acc v)) v,
 field-sz = field-sz d)

lemma *update-desc-id*[simp]: update-desc id (λx -. x) = id
by (simp add: update-desc-def fun-eq-iff)

term *map-td* (λn algn. update-desc acc upd) (update-desc acc upd) t

definition *adjust-ti* :: 'b xtyp-info ⇒ ('a ⇒ 'b) ⇒ ('b ⇒ 'a ⇒ 'a) ⇒ 'a xtyp-info
where
 adjust-ti t acc upd ≡ map-td (λn algn. update-desc acc upd) (update-desc acc upd) t

lemma *adjust-ti-adjust-ti*:
adjust-ti (adjust-ti t g s) g' s' =
 adjust-ti t (g ∘ g') (λv x. s' (s v (g' x)) x)
by (simp add: adjust-ti-def map-td-map update-desc-def[abs-def] comp-def)

lemma *typ-desc-size-update-ti* [simp]:
(size-td (adjust-ti t acc upd) = size-td t)
by (simp add: adjust-ti-def)

lemma *export-tag-adjust-ti*[rule-format]:
∀ bs. fg-cons acc upd ⟶ wf-fd t ⟶
 export-uinfo (adjust-ti t acc upd) = export-uinfo t
∀ bs. fg-cons acc upd ⟶ wf-fd-struct st ⟶
 map-td-struct field-norm (λ-. ()) (map-td-struct (λn algn. update-desc acc upd)
(update-desc acc upd) st) =
 map-td-struct field-norm (λ-. ()) st
∀ bs. fg-cons acc upd ⟶ wf-fd-list ts ⟶
 map-td-list field-norm (λ-. ()) (map-td-list (λn algn. update-desc acc upd)

(update-desc acc upd) ts =
 map-td-list field-norm (λ-. ()) ts
 ∀ bs. fg-cons acc upd → wf-fd-tuple x →
 map-td-tuple field-norm (λ-. ()) (map-td-tuple (λn algn. update-desc acc upd)
 (update-desc acc upd) x) =
 map-td-tuple field-norm (λ-. ()) x
unfolding adjust-ti-def
by (induct t **and** st **and** ts **and** x, all ⟨clarsimp simp: export-uinfo-def⟩)
 (fastforce simp: update-desc-def field-norm-def fg-cons-def fd-cons-struct-def
 fd-cons-access-update-def fd-cons-desc-def)

definition (in c-type) ti-typp-combine ::
 'a itself ⇒ ('b ⇒ 'a) ⇒ ('a ⇒ 'b ⇒ 'b) ⇒ nat ⇒ field-name ⇒ 'b xtyp-info ⇒
 'b xtyp-info
where
 ti-typp-combine t-b acc upd algn fn tag ≡
 let ft = adjust-ti (typ-info-t TYPE('a)) acc upd
 in extend-ti tag ft algn fn (field-access = xto-bytes o acc, field-update = upd o
 xfrom-bytes, field-sz = size-of TYPE('a))

primrec

 padding-fields :: ('a,'b) typ-desc ⇒ field-name list **and**
 padding-fields-struct :: ('a,'b) typ-struct ⇒ field-name list

where

 pf0: padding-fields (TypDesc algn st tn) = padding-fields-struct st

| pf1: padding-fields-struct (TypScalar n algn d) = []
 | pf2: padding-fields-struct (TypAggregate xs) = filter (λx. hd x = CHR '!') (map
 dt-snd xs)

primrec

 non-padding-fields :: ('a,'b) typ-desc ⇒ field-name list **and**
 non-padding-fields-struct :: ('a,'b) typ-struct ⇒ field-name list

where

 npf0: non-padding-fields (TypDesc algn st tn) = non-padding-fields-struct st

| npf1: non-padding-fields-struct (TypScalar n algn d) = []
 | npf2: non-padding-fields-struct (TypAggregate xs) = filter (λx. hd x ≠ CHR '!')
 (map dt-snd xs)

definition field-names-list :: ('a,'b) typ-desc ⇒ field-name list **where**
 field-names-list tag ≡ non-padding-fields tag @ padding-fields tag

definition ti-pad-combine :: nat ⇒ 'a xtyp-info ⇒ 'a xtyp-info **where**
 ti-pad-combine n tag ≡

```

let
  fn = foldl (@) "\pad-" (field-names-list tag);
  td = padding-desc n;
  nf = TypDesc 0 (TypScalar n 0 td) "\pad-typ"
in extend-ti tag nf 0 fn td

```

lemma *aggregate-ti-pad-combine* [simp]:
aggregate (ti-pad-combine n tag)
by (simp add: ti-pad-combine-def Let-def)

definition (in *c-type*) *ti-typ-pad-combine* ::
'*a* itself \Rightarrow ('*b* \Rightarrow '*a*) \Rightarrow ('*a* \Rightarrow '*b* \Rightarrow '*b*) \Rightarrow nat \Rightarrow field-name \Rightarrow '*b* xtyp-info \Rightarrow
'*b* xtyp-info
where
ti-typ-pad-combine t acc upd algn fn tag \equiv
let
 pad = padup (max (2 ^ algn) (align-of TYPE('a))) (size-td tag);
 ntag = if 0 < pad then ti-pad-combine pad tag else tag
in
 ti-typ-combine t acc upd algn fn ntag

definition *map-align* f t = (case t of TypDesc algn st n \Rightarrow TypDesc (f algn) st n)
lemma *map-align-simp* [simp]: *map-align* f (TypDesc algn st n) = TypDesc (f algn) st n
by (simp add: map-align-def)

definition *final-pad* :: nat \Rightarrow '*a* xtyp-info \Rightarrow '*a* xtyp-info **where**
final-pad algn tag \equiv
let n = padup (2 ^ (max algn (align-td tag))) (size-td tag)
in map-align (max algn) (if 0 < n then ti-pad-combine n tag else tag)

lemma *field-names-list-empty-typ-info* [simp]:
set (field-names-list (empty-typ-info algn tn)) = {}
by (simp add: empty-typ-info-def field-names-list-def)

lemma *field-names-list-extend-ti* [simp]:
set (field-names-list (extend-ti tag t algn fn d)) = set (field-names-list tag) \cup {fn}
unfolding field-names-list-def
apply(cases tag)
subgoal for x1 typ-struct xs
apply(cases typ-struct; simp)
done
done

lemma (in *c-type*) *field-names-list-ti-typ-combine* [simp]:
set (field-names-list (ti-typ-combine (t::'*a* itself) f f-upd algn fn tag))
= set (field-names-list tag) \cup {fn}
by (clarsimp simp: ti-typ-combine-def Let-def)

lemma *field-names-list-ti-pad-combine* [simp]:
 set (field-names-list (ti-pad-combine n tag))
 = set (field-names-list tag) \cup {foldl (@) "pad-" (field-names-list tag)}
by (clarsimp simp: ti-pad-combine-def Let-def)

— matches on padding

lemma *hd-string-hd-fold-eq* [simp]:
 $\llbracket s \neq []; \text{hd } s = \text{CHR } "!" \rrbracket \implies \text{hd } (\text{foldl } (@) s xs) = \text{CHR } "!"$
by (induct xs arbitrary: s; clarsimp)

lemma *field-names-list-ti-tyr-pad-combine* [simp]:
 $\text{hd } x \neq \text{CHR } "!" \implies$
 $x \in \text{set } (\text{field-names-list } (\text{ti-tyr-pad-combine } \text{align } t\text{-b } f\text{-ab } f\text{-upd-ab } fn \text{ tag}))$
 $= (x \in \text{set } (\text{field-names-list } \text{tag}) \cup \{fn\})$
by (auto simp: ti-tyr-pad-combine-def Let-def)

lemma *wf-desc-empty-tyr-info* [simp]:
 wf-desc (empty-tyr-info algn tn)
by (simp add: empty-tyr-info-def)

lemma *wf-desc-extend-ti*:
 $\llbracket \text{wf-desc } \text{tag}; \text{wf-desc } t; f \notin \text{set } (\text{field-names-list } \text{tag}) \rrbracket \implies$
 $\text{wf-desc } (\text{extend-ti } \text{tag } t \text{ algn } f d)$
unfolding field-names-list-def
apply(cases tag)
subgoal for $x1$ *typ-struct* xs
apply(cases *typ-struct*; clarsimp)
done
done

lemma *foldl-append-length*:
 $\text{length } (\text{foldl } (@) s xs) \geq \text{length } s$
apply(induct xs arbitrary: s, clarsimp)
subgoal for a list s
apply(drule meta-spec [where $x=s@a$])
apply clarsimp
done
done

lemma *foldl-append-nmem*:
 $s \neq [] \implies \text{foldl } (@) s xs \notin \text{set } xs$
apply(induct xs arbitrary: s, clarsimp)
subgoal for a list s
apply(drule meta-spec [where $x=s@a$])
apply clarsimp
by (metis add-le-same-cancel2 foldl-append-length le-zero-eq length-0-conv length-append)
done

lemma *wf-desc-ti-pad-combine*:
wf-desc tag \implies *wf-desc (ti-pad-combine n tag)*
apply(*clarsimp simp: ti-pad-combine-def Let-def*)
apply(*erule wf-desc-extend-ti*)
apply *simp*
apply(*rule foldl-append-nmem, simp*)
done

lemma *wf-desc-adjust-ti* [*simp*]:
wf-desc (adjust-ti t f g) = wf-desc (t::'a xtyp-info)
by (*simp add: adjust-ti-def wf-desc-map*)

lemma (**in** *wf-type*) *wf-desc-ti-typ-combine*:
 \llbracket *wf-desc tag; fn \notin set (field-names-list tag)* $\rrbracket \implies$
wf-desc (ti-typ-combine (t-b::'a itself) acc upd-ab algn fn tag)
by (*fastforce simp: ti-typ-combine-def Let-def elim: wf-desc-extend-ti*)

lemma (**in** *wf-type*) *wf-desc-ti-typ-pad-combine*:
 \llbracket *wf-desc tag; fn \notin set (field-names-list tag); hd fn \neq CHR "!"* $\rrbracket \implies$
wf-desc (ti-typ-pad-combine (t-b::'a itself) acc upd algn fn tag)
unfolding *ti-typ-pad-combine-def Let-def*
by (*auto intro!: wf-desc-ti-typ-combine wf-desc-ti-pad-combine*)

lemma *wf-desc-map-align*: *wf-desc tag* \implies *wf-desc (map-align f tag)*
by (*cases tag*) (*simp*)

lemma *wf-desc-final-pad*:
wf-desc tag \implies *wf-desc (final-pad algn tag)*
by (*auto simp: final-pad-def Let-def wf-desc-map-align wf-desc-ti-pad-combine*)

lemma *wf-size-desc-extend-ti*:
 \llbracket *wf-size-desc tag; wf-size-desc t* $\rrbracket \implies$ *wf-size-desc (extend-ti tag t algn fn d)*
apply(*cases tag*)
subgoal for *x1 typ-struct list*
apply(*cases typ-struct, auto*)
done
done

lemma *wf-size-desc-ti-pad-combine*:
 \llbracket *wf-size-desc tag; 0 < n* $\rrbracket \implies$ *wf-size-desc (ti-pad-combine n tag)*
by (*fastforce simp: ti-pad-combine-def Let-def elim: wf-size-desc-extend-ti*)

lemma *wf-size-desc-adjust-ti*:
wf-size-desc (adjust-ti t f g) = wf-size-desc (t::'a xtyp-info)
by (*simp add: adjust-ti-def wf-size-desc-map*)

lemma (**in** *wf-type*) *wf-size-desc-ti-typ-combine*:
wf-size-desc tag \implies *wf-size-desc (ti-typ-combine (t-b::'a itself) acc upd-ab algn fn tag)*

by (*fastforce simp: wf-size-desc-adjust-ti ti-typ-combine-def Let-def elim: wf-size-desc-extend-ti*)

lemma (*in wf-type*) *wf-size-desc-ti-typ-pad-combine*:

wf-size-desc tag \implies

wf-size-desc (ti-typ-pad-combine (t-b::'a itself) acc upd algn fn tag)

by (*auto simp: ti-typ-pad-combine-def Let-def*

intro: wf-size-desc-ti-typ-combine

elim: wf-size-desc-ti-pad-combine)

lemma (*in wf-type*) *wf-size-desc-ti-typ-combine-empty* [*simp*]:

wf-size-desc (ti-typ-combine (t-b::'a itself) acc upd algn fn (empty-typ-info algn' tn))

by (*clarsimp simp: ti-typ-combine-def Let-def empty-typ-info-def wf-size-desc-adjust-ti*)

lemma (*in wf-type*) *wf-size-desc-ti-typ-pad-combine-empty* [*simp*]:

wf-size-desc (ti-typ-pad-combine (t-b::'a itself) acc upd algn fn

(empty-typ-info algn' tn))

by (*clarsimp simp: ti-typ-pad-combine-def Let-def ti-typ-combine-def empty-typ-info-def ti-pad-combine-def wf-size-desc-adjust-ti*)

lemma *wf-size-desc-msp-align*:

wf-size-desc tag \implies *wf-size-desc (map-align f tag)*

by (*cases tag*) (*simp add: wf-size-desc-def*)

lemma *wf-size-desc-final-pad*:

wf-size-desc tag \implies *wf-size-desc (final-pad algn tag)*

by (*fastforce simp: final-pad-def Let-def wf-size-desc-msp-align wf-size-desc-ti-pad-combine*)

lemma *wf-fdp-set-comp-simp* [*simp*]:

wf-fdp {(a, n # b) | a b. (a, b) ∈ tf-set t} = wf-fdp (tf-set t)

unfolding *wf-fdp-def* **by** *fastforce*

lemma *lf-set-adjust-ti'*:

$\forall d \text{ fn. } d \in \text{lf-set (map-td (\lambda n \text{ algn. update-desc acc upd) (update-desc acc upd) t) fn} \longrightarrow$

$(\exists y. \text{lf-fd } d = \text{update-desc acc upd (lf-fd } y) \wedge \text{lf-sz } d = \text{lf-sz } y \wedge \text{lf-fn } d = \text{lf-fn } y \wedge y \in \text{lf-set } t \text{ fn})$

$\forall d \text{ fn. } d \in \text{lf-set-struct (map-td-struct (\lambda n \text{ algn. update-desc acc upd) (update-desc acc upd) st) fn} \longrightarrow$

$(\exists y. \text{lf-fd } d = \text{update-desc acc upd (lf-fd } y) \wedge \text{lf-sz } d = \text{lf-sz } y \wedge \text{lf-fn } d = \text{lf-fn } y \wedge y \in \text{lf-set-struct } st \text{ fn})$

$\forall d \text{ fn. } d \in \text{lf-set-list (map-td-list (\lambda n \text{ algn. update-desc acc upd) (update-desc acc upd) ts) fn} \longrightarrow$

$(\exists y. \text{lf-fd } d = \text{update-desc acc upd (lf-fd } y) \wedge \text{lf-sz } d = \text{lf-sz } y \wedge \text{lf-fn } d = \text{lf-fn } y \wedge y \in \text{lf-set-list } ts \text{ fn})$

$\forall d \text{ fn. } d \in \text{lf-set-tuple (map-td-tuple (\lambda n \text{ algn. update-desc acc upd) (update-desc acc upd) x) fn} \longrightarrow$

$(\exists y. \text{lf-fd } d = \text{update-desc acc upd (lf-fd } y) \wedge \text{lf-sz } d = \text{lf-sz } y \wedge \text{lf-fn } d = \text{lf-fn } y \wedge y \in \text{lf-set-tuple } x \text{ fn})$

unfolding *update-desc-def*
by (*induct t and st and ts and x*) *fastforce+*

lemma *lf-set-adjust-ti*:

$\llbracket d \in \text{lf-set } (\text{adjust-ti } t \text{ acc upd}) \text{ fn}; \bigwedge y. \text{upd } (\text{acc } y) \text{ } y = y \rrbracket \implies$
 $(\exists y. \text{lf-fd } d = \text{update-desc acc upd } (\text{lf-fd } y) \wedge \text{lf-sz } d = \text{lf-sz } y \wedge \text{lf-fn } d = \text{lf-fn } y \wedge$
 $y \in \text{lf-set } t \text{ fn})$
by (*simp add: lf-set-adjust-ti' adjust-ti-def*)

lemma *fd-cons-struct-id-simp* [*simp*]:

fd-cons-struct (*TypScalar n algn* (*field-access = $\lambda v. id$, field-update = $\lambda bs. id$, field-sz = m*))
by (*auto simp: fd-cons-struct-def fd-cons-double-update-def*
fd-cons-update-access-def fd-cons-access-update-def fd-cons-length-def
fd-cons-update-normalise-def fd-cons-desc-def)

lemma *field-desc-adjust-ti*:

fg-cons acc upd \longrightarrow
field-desc (*adjust-ti* (*t::'a xtyp-info*) *acc upd*) =
update-desc acc upd (*field-desc t*)
fg-cons acc upd \longrightarrow
field-desc-struct (*map-td-struct* ($\lambda n \text{ algn. update-desc acc upd}$) (*update-desc*
acc upd) *st*) =
update-desc acc upd (*field-desc-struct st*)
fg-cons acc upd \longrightarrow
field-desc-list (*map-td-list* ($\lambda n \text{ algn. update-desc acc upd}$) (*update-desc acc upd*)
ts) =
update-desc acc upd (*field-desc-list ts*)
fg-cons acc upd \longrightarrow
field-desc-tuple (*map-td-tuple* ($\lambda n \text{ algn. update-desc acc upd}$) (*update-desc acc*
upd) *x*) =
update-desc acc upd (*field-desc-tuple x*)
unfolding *adjust-ti-def*
by (*induct t and st and ts and x*) (*fastforce simp: fg-cons-def update-desc-def*
update-ti-t-struct-t) $+$

lemma *update-ti-t-adjust-ti*:

fg-cons acc upd \implies *update-ti-t* (*adjust-ti t acc upd*) *bs v* = *upd* (*update-ti-t t bs*
(acc v)) *v*
using *field-desc-adjust-ti(1)* [*of acc upd t*]
by (*clarsimp simp: update-desc-def*)

declare *field-desc-def* [*simp del*]

lemma (**in** *c-type*) *aggregate-ti-typ-combine* [*simp*]:

aggregate (*ti-typ-combine* (*t-b::'a itself*) *acc upd algn fn tag*)
by (*simp add: ti-typ-combine-def Let-def*)

lemma (in *c-type*) *aggregate-ti-typ-pad-combine* [simp]:
aggregate (ti-typ-pad-combine (t-b::'a itself) acc upd algn fn tag)
by (*simp add: ti-typ-pad-combine-def Let-def*)

lemma *align-of-empty-typ-info* [simp]:
align-td-wo-align (empty-typ-info algn tn) = 0
by (*simp add: empty-typ-info-def*)

lemma *align-of-empty-typ-info'* [simp]:
align-td (empty-typ-info algn tn) = algn
by (*simp add: empty-typ-info-def*)

lemma *align-of-tag-list* [simp]:
align-td-wo-align-list (xs @ [DTuple t fn d]) = max (align-td-wo-align-list xs)
(align-td-wo-align t)
by (*induct xs*) *auto*

lemma *align-of-tag-list'* [simp]:
align-td-list (xs @ [DTuple t fn d]) = max (align-td-list xs) (align-td t)
by (*induct xs*) *auto*

lemma *align-of-extend-ti* [simp]:
aggregate ti \implies align-td-wo-align (extend-ti ti t algn fn d) = max (align-td-wo-align ti) (align-td-wo-align t)
apply (*cases ti*)
subgoal for *x1 typ-struct xs*
apply (*cases typ-struct; clarsimp*)
done
done

lemma *align-of-extend-ti'* [simp]:
aggregate ti \implies align-td (extend-ti ti t algn fn d) = max (align-td ti) (max algn (align-td t))
apply (*cases ti*)
subgoal for *x1 typ-struct xs*
apply (*cases typ-struct; clarsimp*)
done
done

lemma *align-of-adjust-ti* [simp]:
align-td-wo-align (adjust-ti t acc upd) = align-td-wo-align (t::'a xtyp-info)
by (*simp add: adjust-ti-def*)

lemma *align-of-adjust-ti'* [simp]:
align-td (adjust-ti t acc upd) = align-td (t::'a xtyp-info)
by (*simp add: adjust-ti-def*)

lemma (in *c-type*) *align-of-ti-typ-combine* [simp]:
aggregate ti \implies

$align\text{-}td\text{-}wo\text{-}align (ti\text{-}typ\text{-}combine (t::'a\text{ itself})\ acc\ upd\ algn\ fn\ ti) =$
 $max (align\text{-}td\text{-}wo\text{-}align\ ti) (align\text{-}td\text{-}wo\text{-}align (typ\text{-}info\text{-}t (TYPE('a))))$
by (clarsimp simp: ti-typ-combine-def Let-def align-of-def)

lemma (in c-type) align-of-ti-typ-combine' [simp]:
 $aggregate\ ti \implies$
 $align\text{-}td (ti\text{-}typ\text{-}combine (t::'a\text{ itself})\ acc\ upd\ algn\ fn\ ti) =$
 $max (align\text{-}td\ ti) (max\ algn (align\text{-}td (typ\text{-}info\text{-}t\ TYPE('a))))$
by (clarsimp simp: ti-typ-combine-def Let-def align-of-def)

lemma align-of-ti-pad-combine [simp]:
 $aggregate\ ti \implies align\text{-}td\text{-}wo\text{-}align (ti\text{-}pad\text{-}combine\ n\ ti) = (align\text{-}td\text{-}wo\text{-}align\ ti)$
by (clarsimp simp: ti-pad-combine-def Let-def max-def)

lemma align-of-ti-pad-combine' [simp]:
 $aggregate\ ti \implies align\text{-}td (ti\text{-}pad\text{-}combine\ n\ ti) = (align\text{-}td\ ti)$
by (clarsimp simp: ti-pad-combine-def Let-def max-def)

lemma max-2-exp: $max ((2::nat) ^ a) (2 ^ b) = 2 ^ (max\ a\ b)$
by (simp add: max-def)

lemma align-td-wo-align-map-align: $align\text{-}td\text{-}wo\text{-}align (map\text{-}align\ f\ t) = align\text{-}td\text{-}wo\text{-}align\ t$
by (cases t) simp

lemma align-td-wo-align-final-pad:
 $aggregate\ ti \implies$
 $align\text{-}td\text{-}wo\text{-}align (final\text{-}pad\ algn\ ti) = (align\text{-}td\text{-}wo\text{-}align\ ti)$
by (simp add: final-pad-def Let-def padup-def align-td-wo-align-map-align)

lemma align-td-map-align [simp]: $align\text{-}td (map\text{-}align\ f\ t) = f (align\text{-}td\ t)$
by (cases t) simp

lemma align-of-final-pad:
 $aggregate\ ti \implies$
 $align\text{-}td (final\text{-}pad\ algn\ ti) = max\ algn (align\text{-}td\ ti)$
by (simp add: final-pad-def Let-def padup-def align-td-map-align)

lemma (in c-type) align-td-wo-align-ti-typ-pad-combine [simp]:
 $aggregate\ ti \implies$
 $align\text{-}td\text{-}wo\text{-}align (ti\text{-}typ\text{-}pad\text{-}combine (t::'a\text{ itself})\ acc\ upd\ algn\ fn\ ti) =$
 $max (align\text{-}td\text{-}wo\text{-}align\ ti) (align\text{-}td\text{-}wo\text{-}align (typ\text{-}info\text{-}t\ TYPE('a)))$
by (clarsimp simp: ti-typ-pad-combine-def Let-def)

lemma (in c-type) align-td-ti-typ-pad-combine [simp]:
 $aggregate\ ti \implies$
 $align\text{-}td (ti\text{-}typ\text{-}pad\text{-}combine (t::'a\text{ itself})\ acc\ upd\ algn\ fn\ ti) =$
 $max (align\text{-}td\ ti) (max\ algn (align\text{-}td (typ\text{-}info\text{-}t\ TYPE('a))))$

by (clarsimp simp: ti-typ-pad-combine-def Let-def)

definition *fu-s-comm-set* :: (byte list \Rightarrow 'a \Rightarrow 'a) set \Rightarrow (byte list \Rightarrow 'a \Rightarrow 'a) set \Rightarrow bool

where

fu-s-comm-set X Y $\equiv \forall x y. x \in X \wedge y \in Y \longrightarrow (\forall v bs bs'. x bs (y bs' v) = y bs' (x bs v))$

lemma *fc-empty-ti* [simp]:

fu-commutes (update-ti-t (empty-typ-info algn tn)) f

by (auto simp: fu-commutes-def empty-typ-info-def)

lemma *fc-extend-ti*:

\llbracket *fu-commutes* (update-ti-t s) h; *fu-commutes* (update-ti-t t) h \rrbracket

\implies *fu-commutes* (update-ti-t (extend-ti s t algn fn d)) h

apply(cases s)

subgoal for x1 typ-struct xs

apply(cases typ-struct, auto simp: fu-commutes-def)

done

done

lemma *fc-update-ti*:

\llbracket *fu-commutes* (update-ti-t ti) h; fg-cons acc upd;

$\forall v bs bs'. \text{upd } bs (h bs' v) = h bs' (\text{upd } bs v); \forall bs v. \text{acc } (h bs v) = \text{acc } v \rrbracket$

\implies *fu-commutes* (update-ti-t (adjust-ti t acc upd)) h

by (auto simp: fu-commutes-def update-ti-t-adjust-ti)

lemma (in c-type) *fc-ti-typ-combine*:

\llbracket *fu-commutes* (update-ti-t ti) h; fg-cons acc upd;

$\forall v bs bs'. \text{upd } bs (h bs' v) = h bs' (\text{upd } bs v); \forall bs v. \text{acc } (h bs v) = \text{acc } v \rrbracket$

\implies *fu-commutes* (update-ti-t (ti-typ-combine (t::'a itself) acc upd algn fn ti)) h

apply(clarsimp simp: ti-typ-combine-def Let-def)

apply(rule fc-extend-ti, assumption)

apply(rule fc-update-ti; simp)

done

lemma *fc-ti-pad-combine*:

fu-commutes (update-ti-t ti) f \implies *fu-commutes* (update-ti-t (ti-pad-combine n ti)) f

apply(clarsimp simp: ti-pad-combine-def Let-def)

apply(rule fc-extend-ti, assumption)

apply(auto simp: fu-commutes-def padding-desc-def)

done

lemma (in c-type) *fc-ti-typ-pad-combine*:

\llbracket *fu-commutes* (update-ti-t ti) h; fg-cons acc upd;

$\forall v bs bs'. \text{upd } bs (h bs' v) = h bs' (\text{upd } bs v); \forall bs v. \text{acc } (h bs v) = \text{acc } v \rrbracket$

\implies *fu-commutes* (update-ti-t (ti-typ-pad-combine (t::'a itself) acc upd algn fn ti)) h

```

apply(clarsimp simp: ti-typ-pad-combine-def Let-def)
apply(rule conjI; clarsimp)
apply(rule fc-ti-typ-combine; assumption?)
apply(erule fc-ti-pad-combine)
apply(erule (3) fc-ti-typ-combine)
done

```

definition *fu-eq-mask* :: 'a *xtyp-info* \Rightarrow ('a \Rightarrow 'a) \Rightarrow bool **where**
fu-eq-mask *ti f* \equiv
 \forall *bs v v'*. *length* *bs* = *size-td* *ti* \longrightarrow *update-ti-t* *ti* *bs* (*f v*) = *update-ti-t* *ti* *bs* (*f v'*)

lemma *fu-eq-mask*:
 \llbracket *length* *bs* = *size-td* *ti*; *fu-eq-mask* *ti id* $\rrbracket \Longrightarrow$
 $\text{update-ti-t } ti \text{ } bs \text{ } v = \text{update-ti-t } ti \text{ } bs \text{ } w$
by (*clarsimp simp: fu-eq-mask-def update-ti-t-def*)

lemma *fu-eq-mask-ti-pad-combine*:
 \llbracket *fu-eq-mask* *ti f*; *aggregate* *ti* $\rrbracket \Longrightarrow$ *fu-eq-mask* (*ti-pad-combine* *n ti*) *f*
unfolding *ti-pad-combine-def Let-def*
apply(*cases* *ti*)
subgoal for *x1 typ-struct xs*
apply(*cases typ-struct, auto simp: fu-eq-mask-def update-ti-list-t-def padding-desc-def*)
done
done

lemma *fu-eq-mask-map-align*: *fu-eq-mask* *t f* \Longrightarrow *fu-eq-mask* (*map-align* *g t*) *f*
by (*cases* *t*) (*auto simp add: fu-eq-mask-def*)

lemma *fu-eq-mask-final-pad*:
 \llbracket *fu-eq-mask* *ti f*; *aggregate* *ti* $\rrbracket \Longrightarrow$ *fu-eq-mask* (*final-pad* *algn ti*) *f*
by(*auto simp: final-pad-def Let-def fu-eq-mask-map-align fu-eq-mask-ti-pad-combine*)

definition *upd-local* :: ('b \Rightarrow 'a \Rightarrow 'a) \Rightarrow bool **where**
upd-local *g* $\equiv \forall$ *j k v v'*. *g k v* = *g k v'* \longrightarrow *g j v* = *g j v'*

lemma *fg-cons-upd-local*:
fg-cons *f g* \Longrightarrow *upd-local* *g*
apply(*clarsimp simp: fg-cons-def upd-local-def*)
subgoal for *j k v v'*
apply(*drule arg-cong [where f=g j]*)
apply *simp*
done
done

lemma (**in** *mem-type*) *fu-eq-mask-ti-typ-combine*:
 \llbracket *fu-eq-mask* *ti* (λv . (*upd* (*acc* *undefined*) (*h v*))); *fg-cons* *acc* *upd*;
fu-commutes (*update-ti-t* *ti*) *upd*; *aggregate* *ti* $\rrbracket \Longrightarrow$
fu-eq-mask (*ti-typ-combine* (*t::'a* *itself*) *acc* *upd* *algn* *fn* *ti*) *h*

```

apply(frule fg-cons-upd-local)
apply(clarsimp simp: ti-typ-combine-def Let-def)
apply(cases ti)
subgoal for x1 typ-struct xs
  apply(cases typ-struct; clarsimp)
  subgoal for xs'
    apply(clarsimp simp: fu-eq-mask-def update-ti-t-adjust-ti)
    apply(clarsimp simp: update-ti-list-t-def size-of-def)
    apply(subst upd [where w=acc undefined])
    apply(simp add: size-of-def)
    apply(subst upd [where w=acc undefined and v=acc (h v') for v'])
    apply(simp add: size-of-def)
  apply (smt (verit, ccfv-threshold) fu-commutes-def length-take min-ll upd-local-def)
  done
done
done

```

```

lemma (in mem-type) fu-eq-mask-ti-typ-pad-combine:
   $\llbracket fu-eq-mask\ ti\ (\lambda v. (upd\ (acc\ undefined)\ (h\ v))); fg-cons\ acc\ upd;$ 
   $fu-commutes\ (update-ti-t\ ti)\ upd; aggregate\ ti \rrbracket \implies$ 
   $fu-eq-mask\ (ti-typ-pad-combine\ (t::'a\ itself)\ acc\ upd\ algn\ fn\ ti)\ h$ 
by (fastforce simp: ti-typ-pad-combine-def Let-def)
  intro: fu-eq-mask-ti-typ-combine fu-eq-mask-ti-pad-combine fc-ti-pad-combine)

```

```

lemma fu-eq-mask-empty-typ-info-g:
   $\exists k. \forall v. f\ v = k \implies fu-eq-mask\ t\ f$ 
by (auto simp: fu-eq-mask-def)

```

```

lemma fu-eq-mask-empty-typ-info:
   $\forall v. f\ v = undefined \implies fu-eq-mask\ t\ f$ 
by (auto simp: fu-eq-mask-def)

```

```

lemma size-td-extend-ti:
   $aggregate\ s \implies size-td\ (extend-ti\ s\ t\ algn\ fn\ d) = size-td\ s + size-td\ t$ 
apply (cases s)
subgoal for x1 typ-struct xs
  apply (cases typ-struct; simp)
  done
done

```

```

lemma size-td-ti-pad-combine:
   $aggregate\ ti \implies size-td\ (ti-pad-combine\ n\ ti) = n + size-td\ ti$ 
unfolding ti-pad-combine-def Let-def by (simp add: size-td-extend-ti)

```

```

lemma size-td-map-align [simp]:  $size-td\ (map-align\ f\ ti) = size-td\ ti$ 
by (cases ti) auto

```

```

lemma align-of-dvd-size-of-final-pad [simp]:
   $aggregate\ ti \implies 2^{\wedge}align-td\ (final-pad\ algn\ ti)\ dvd\ size-td\ (final-pad\ algn\ ti)$ 

```

unfolding *final-pad-def Let-def*
apply (*cases ti*)
apply (*auto simp: size-td-ti-pad-combine ac-simps padup-dvd power-le-dvd intro: dvd-padup-add*)
done

lemma *size-td-lt-ti-pad-combine*:
 $aggregate\ t \implies size-td\ (ti-pad-combine\ n\ t) = size-td\ t + n$
by (*metis add.commute size-td-ti-pad-combine*)

lemma (**in** *c-type*) *size-td-lt-ti-typ-combine*:
 $aggregate\ ti \implies$
 $size-td\ (ti-typ-combine\ (t::'a\ itself)\ f\ g\ algn\ fn\ ti) =$
 $size-td\ ti + size-td\ (typ-info-t\ TYPE('a))$
by (*clarsimp simp: ti-typ-combine-def Let-def size-td-extend-ti*)

lemma (**in** *c-type*) *size-td-lt-ti-typ-pad-combine*:
 $aggregate\ ti \implies$
 $size-td\ (ti-typ-pad-combine\ (t::'a\ itself)\ f\ g\ algn\ fn\ ti) =$
 $(let\ k = size-td\ ti\ in$
 $k + size-td\ (typ-info-t\ TYPE('a)) + padup\ (2^{\wedge}(max\ algn\ (align-td$
 $(typ-info-t\ TYPE('a))))\ k)$
unfolding *ti-typ-pad-combine-def Let-def*
by (*auto simp: size-td-lt-ti-typ-combine size-td-ti-pad-combine align-of-def max-2-exp*)

lemma *size-td-lt-final-pad*:
 $aggregate\ tag \implies$
 $size-td\ (final-pad\ align\ tag) = (let\ k=size-td\ tag\ in\ k + padup\ (2^{\wedge}(max\ align$
 $(align-td\ tag)))\ k)$
by (*auto simp: final-pad-def Let-def size-td-ti-pad-combine*)

lemma *size-td-empty-typ-info [simp]*:
 $size-td\ (empty-typ-info\ algn\ tn) = 0$
by (*clarsimp simp: empty-typ-info-def*)

lemma *wf-lf-empty-typ-info [simp]*:
 $wf-lf\ \{\}$
by (*auto simp: wf-lf-def empty-typ-info-def*)

lemma *lf-fn-disj-fn*:
 $fn \notin set\ (field-names-list\ (TypDesc\ algn\ (TypAggregate\ xs)\ tn)) \implies$
 $lf-fn\ 'lf-set-list\ xs\ [] \cap\ lf-fn\ 'lf-set\ t\ [fn] = \{\}$
apply (*induct xs arbitrary: fn t tn, clarsimp*)
subgoal for *a list fn t tn*
apply (*cases a, clarsimp*)
apply (*drule meta-spec [where x=fn]*)
apply (*drule meta-spec [where x=t]*)
apply (*drule meta-spec [where x=tn]*)
apply (*drule meta-mp, fastforce simp: field-names-list-def split: if-split-asm*)


```

apply(safe)
apply(fastforce dest!: lf-set-fn simp: field-names-list-def prefix-def less-eq-list-def
      split: if-split-asm)
  by force
done

```

```

lemma wf-lf-extend-ti:
   $\llbracket \text{wf-lf } (lf\text{-set } t \ \square); \text{wf-lf } (lf\text{-set } ti \ \square); \text{wf-desc } t; fn \notin \text{set } (field\text{-names-list } ti);$ 
   $\text{ti-ind } (lf\text{-set } ti \ \square) (lf\text{-set } t \ \square) \rrbracket \implies$ 
   $\text{wf-lf } (lf\text{-set } (extend\text{-ti } ti \ t \ \text{algn } fn \ d) \ \square)$ 
apply(cases ti)
subgoal for x1 typ-struct xs
  apply(cases typ-struct; clarsimp)
  apply(subst wf-lf-fn; simp)
  apply(subst wf-lf-list, erule lf-fn-disj-fn)
  apply(subst ti-ind-sym2)
  apply(subst ti-ind-fn)
  apply(subst ti-ind-sym2)
  apply clarsimp
  apply(subst wf-lf-fn; simp)
done
done

```

```

lemma wf-lf-ti-pad-combine:
   $\text{wf-lf } (lf\text{-set } ti \ \square) \implies \text{wf-lf } (lf\text{-set } (ti\text{-pad-combine } n \ ti) \ \square)$ 
apply(clarsimp simp: ti-pad-combine-def Let-def padding-desc-def)
apply(rule wf-lf-extend-ti)
  apply(clarsimp simp: wf-lf-def fd-cons-desc-def fd-cons-double-update-def
      fd-cons-update-access-def fd-cons-access-update-def
      fd-cons-length-def)
  apply assumption
  apply(clarsimp)
  apply(rule foldl-append-nmem)
  apply clarsimp
apply(clarsimp simp: ti-ind-def fu-commutes-def fa-fu-ind-def)
done

```

```

lemma lf-set-map-align [simp]: lf-set (map-align f ti) = lf-set ti
by (cases ti) auto

```

```

lemma wf-lf-final-pad:
   $\text{wf-lf } (lf\text{-set } ti \ \square) \implies \text{wf-lf } (lf\text{-set } (final\text{-pad } \text{algn } ti) \ \square)$ 
by (auto simp: final-pad-def Let-def elim: wf-lf-ti-pad-combine)

```

```

lemma wf-lf-adjust-ti:
   $\llbracket \text{wf-lf } (lf\text{-set } t \ \square); \bigwedge v. g (f v) v = v;$ 
   $\bigwedge bs \ bs' \ v. g \ bs (g \ bs' v) = g \ bs v; \bigwedge bs \ v. f (g \ bs v) = bs \rrbracket$ 
   $\implies \text{wf-lf } (lf\text{-set } (adjust\text{-ti } t \ f \ g) \ \square)$ 

```

```

apply(clarsimp simp: wf-lf-def)
apply(drule lf-set-adjust-ti; clarsimp)
apply(rule conjI)
apply(fastforce simp: fd-cons-desc-def fd-cons-double-update-def update-desc-def
      fd-cons-update-access-def fd-cons-access-update-def
fd-cons-length-def)
apply(fastforce simp: fu-commutes-def update-desc-def fa-fu-ind-def dest!: lf-set-adjust-ti)
done

```

lemma *ti-ind-empty-typ-info* [simp]:
 $ti-ind (lf-set (empty-typ-info algn tn) []) (lf-set (adjust-ti k f g) [])$
by (clarsimp simp: ti-ind-def empty-typ-info-def)

lemma *ti-ind-extend-ti*:
 $[ti-ind (lf-set t []) (lf-set (adjust-ti k acc upd) []);$
 $ti-ind (lf-set ti []) (lf-set (adjust-ti k acc upd) [])]$
 $\implies ti-ind (lf-set (extend-ti ti t algn n d) []) (lf-set (adjust-ti k acc upd) [])$
apply(cases ti)
subgoal for *x1 typ-struct xs*
apply(cases typ-struct; clarsimp, subst ti-ind-fn, simp)
done
done

lemma *ti-ind-ti-pad-combine*:
 $ti-ind (lf-set ti []) (lf-set (adjust-ti k acc upd) []) \implies$
 $ti-ind (lf-set (ti-pad-combine n ti) []) (lf-set (adjust-ti k acc upd) [])$
apply(clarsimp simp: ti-pad-combine-def Let-def padding-desc-def)
apply(rule ti-ind-extend-ti)
apply(clarsimp simp: ti-ind-def fu-commutes-def fa-fu-ind-def)
apply assumption
done

definition *acc-ind* :: $'a \Rightarrow 'b \Rightarrow 'a$ field-desc set \Rightarrow bool **where**
 $acc-ind\ acc\ X \equiv \forall x\ bs\ v. x \in X \longrightarrow acc (field-update\ x\ bs\ v) = acc\ v$

definition *fu-s-comm-k* :: $'a$ leaf-desc set $\Rightarrow ('b \Rightarrow 'a \Rightarrow 'a) \Rightarrow$ bool **where**
 $fu-s-comm-k\ X\ upd \equiv \forall x. x \in field-update\ 'lf-fd\ 'X \longrightarrow fu-commutes\ x\ upd$

definition *upd-ind* :: $'a$ leaf-desc set $\Rightarrow ('b \Rightarrow 'a \Rightarrow 'a) \Rightarrow$ bool **where**
 $upd-ind\ X\ upd \equiv fu-s-comm-k\ X\ upd$

definition *fa-ind* :: $'a$ field-desc set $\Rightarrow ('b \Rightarrow 'a \Rightarrow 'a) \Rightarrow$ bool **where**
 $fa-ind\ X\ upd \equiv \forall x\ bs\ v. x \in X \longrightarrow field-access\ x\ (upd\ bs\ v) = field-access\ x\ v$

lemma *lf-fd-fn*:
 $\forall fn. lf-fd\ ' (lf-set (t::'a\ xtyp-info)\ fn) = lf-fd\ ' (lf-set\ t\ [])$
 $\forall fn. lf-fd\ ' (lf-set-struct (st::'a\ xtyp-info-struct)\ fn) = lf-fd\ ' (lf-set-struct\ st\ [])$
 $\forall fn. lf-fd\ ' (lf-set-list (ts::'a\ xtyp-info-tuple\ list)\ fn) = lf-fd\ ' (lf-set-list\ ts\ [])$
 $\forall fn. lf-fd\ ' (lf-set-tuple (x::'a\ xtyp-info-tuple)\ fn) = lf-fd\ ' (lf-set-tuple\ x\ [])$

by (induct t and st and ts and x, all <clarsimp simp: image-Un>) metis+

lemma *lf-set-empty-typ-info* [simp]:

lf-set (empty-typ-info algn tn) fn = {}

by (clarsimp simp: empty-typ-info-def)

lemma *upd-ind-empty* [simp]:

upd-ind {} upd

by (clarsimp simp: upd-ind-def fu-s-comm-k-def)

lemma *upd-ind-extend-ti*:

$\llbracket \text{upd-ind } (lf\text{-set } s \ \square) \ \text{upd}; \text{upd-ind } (lf\text{-set } t \ \square) \ \text{upd} \rrbracket \implies$

$\text{upd-ind } (lf\text{-set } (extend\text{-ti } s \ t \ algn \ fn \ d) \ \square) \ \text{upd}$

apply (cases s)

subgoal for x1 typ-struct xs

apply (cases typ-struct)

subgoal

apply (simp add: upd-ind-def image-Un fu-s-comm-k-def)

by (metis lf-fd-fn(1))

subgoal

apply (simp add: upd-ind-def image-Un fu-s-comm-k-def)

by (metis lf-fd-fn(1))

done

done

lemma (in *c-type*) *upd-ind-ti-typ-combine*:

$\llbracket \text{upd-ind } (lf\text{-set } ti \ \square) \ h; \bigwedge w \ u \ v. \text{upd } w \ (h \ u \ v) = h \ u \ (\text{upd } w \ v);$

$\bigwedge w \ v. \text{acc } (h \ w \ v) = \text{acc } v; \bigwedge v. \text{upd } (\text{acc } v) \ v = v \rrbracket$

$\implies \text{upd-ind } (lf\text{-set } (ti\text{-typ-combine } (t::'a \ \text{itself}) \ \text{acc } \text{upd } \text{algn } \text{fn } \ ti) \ \square) \ h$

apply (clarsimp simp: ti-typ-combine-def Let-def)

apply (erule upd-ind-extend-ti)

apply (clarsimp simp: upd-ind-def fu-s-comm-k-def)

apply (drule lf-set-adjust-ti)

apply clarsimp

apply (clarsimp simp: update-desc-def fu-commutes-def)

done

lemma *upd-ind-ti-pad-combine*:

$\text{upd-ind } ((lf\text{-set } ti \ \square)) \ \text{upd} \implies \text{upd-ind } ((lf\text{-set } (ti\text{-pad-combine } n \ ti) \ \square)) \ \text{upd}$

apply (clarsimp simp: ti-pad-combine-def Let-def padding-desc-def)

apply (erule upd-ind-extend-ti)

apply (auto simp: upd-ind-def fu-s-comm-k-def fu-commutes-def)

done

lemma (in *c-type*) *upd-ind-ti-typ-pad-combine*:

$\llbracket \text{upd-ind } (lf\text{-set } ti \ \square) \ h; \bigwedge w \ u \ v. \text{upd } w \ (h \ u \ v) = h \ u \ (\text{upd } w \ v);$

$\bigwedge w \ v. \text{acc } (h \ w \ v) = \text{acc } v; \bigwedge v. \text{upd } (\text{acc } v) \ v = v \rrbracket$

$\implies \text{upd-ind } (lf\text{-set } (ti\text{-typ-pad-combine } (t::'a \ \text{itself}) \ \text{acc } \text{upd } \text{algn } \ \text{fn } \ ti) \ \square) \ h$

unfolding *ti-typ-pad-combine-def Let-def*
by (*fastforce intro!: upd-ind-ti-typ-combine upd-ind-ti-pad-combine*)

lemma *acc-ind-empty [simp]:*
acc-ind acc {}
by (*clarsimp simp: acc-ind-def*)

lemma *acc-ind-extend-ti:*
 $\llbracket \text{acc-ind acc (lf-fd ' lf-set s []); acc-ind acc (lf-fd ' lf-set t [])} \rrbracket \implies$
 $\text{acc-ind acc (lf-fd ' lf-set (extend-ti s t algn fn d) [])}$
apply (*cases s*)
subgoal for *x1 typ-struct xs*
apply (*cases typ-struct*)
subgoal
apply (*simp add: acc-ind-def image-Un fu-s-comm-k-def*)
by (*metis lf-fd-fn(1)*)
subgoal
apply (*simp add: acc-ind-def image-Un fu-s-comm-k-def*)
by (*metis lf-fd-fn(1)*)
done
done

lemma (*in c-type*) *acc-ind-ti-typ-combine:*
 $\llbracket \text{acc-ind h (lf-fd ' lf-set ti []); } \bigwedge v w. h (\text{upd } w v) = h v;$
 $\bigwedge v. \text{upd (acc } v) v = v \rrbracket$
 $\implies \text{acc-ind h (lf-fd ' lf-set (ti-typ-combine (t::'a itself) acc upd algn fn ti) [])}$
apply(*clarsimp simp: ti-typ-combine-def Let-def*)
apply(*erule acc-ind-extend-ti*)
apply(*clarsimp simp: acc-ind-def*)
apply(*drule lf-set-adjust-ti*)
apply *clarsimp*
apply(*clarsimp simp: update-desc-def*)
done

lemma *acc-ind-ti-pad-combine:*
 $\text{acc-ind acc (lf-fd ' (lf-set t []))} \implies \text{acc-ind acc (lf-fd ' (lf-set (ti-pad-combine n$
 $t) [])}$
apply(*clarsimp simp: ti-pad-combine-def Let-def padding-desc-def*)
apply(*erule acc-ind-extend-ti*)
apply(*auto simp: acc-ind-def*)
done

lemma (*in c-type*) *acc-ind-ti-typ-pad-combine:*
 $\llbracket \text{acc-ind h (lf-fd ' lf-set ti []); } \bigwedge v w. h (\text{upd } w v) = h v;$
 $\bigwedge v. \text{upd (acc } v) v = v \rrbracket$
 $\implies \text{acc-ind h (lf-fd ' lf-set (ti-typ-pad-combine (t::'a itself) acc upd algn fn ti)$
 $[])}$
by (*auto simp: ti-typ-pad-combine-def Let-def intro: acc-ind-ti-typ-combine acc-ind-ti-pad-combine*)

lemma *fa-ind-empty [simp]:*

fa-ind {} *upd*
by (*clarsimp simp: fa-ind-def*)

lemma *fa-ind-extend-ti*:

[[*fa-ind* (*lf-fd* ' *lf-set* *s* []) *upd*; *fa-ind* (*lf-fd* ' *lf-set* *t* []) *upd*]] \implies
fa-ind (*lf-fd* ' *lf-set* (*extend-ti s t algn fn d*) []) *upd*
apply (*cases s*)
subgoal for *x1 typ-struct xs*
apply (*cases typ-struct*)
subgoal
apply (*simp add: fa-ind-def image-Un fu-s-comm-k-def*)
by (*metis lf-fd-fn(1)*)
subgoal
apply (*simp add: fa-ind-def image-Un fu-s-comm-k-def*)
by (*metis lf-fd-fn(1)*)
done
done

lemma (**in** *c-type*) *fa-ind-ti-typ-combine*:

[[*fa-ind* (*lf-fd* ' *lf-set* *ti* []) *h*; $\bigwedge v w. \text{acc } (h w v) = \text{acc } v$;
 $\bigwedge v. \text{upd } (\text{acc } v) v = v$]] \implies *fa-ind* (*lf-fd* ' *lf-set* (*ti-typ-combine* (*t::'a itself*) *acc upd algn fn ti*) []) *h*
apply(*clarsimp simp: ti-typ-combine-def Let-def*)
apply(*erule fa-ind-extend-ti*)
apply(*clarsimp simp: fa-ind-def fu-s-comm-k-def*)
apply(*drule lf-set-adjust-ti*)
apply *clarsimp*
apply(*clarsimp simp: update-desc-def fu-commutes-def*)
done

lemma *fa-ind-ti-pad-combine*:

fa-ind (*lf-fd* ' (*lf-set* *ti* [])) *upd* \implies *fa-ind* (*lf-fd* ' (*lf-set* (*ti-pad-combine n ti*) [])) *upd*
apply(*clarsimp simp: ti-pad-combine-def Let-def padding-desc-def*)
apply(*erule fa-ind-extend-ti*)
apply(*auto simp: fa-ind-def*)
done

lemma (**in** *c-type*) *fa-ind-ti-typ-pad-combine*:

[[*fa-ind* (*lf-fd* ' *lf-set* *ti* []) *h*; $\bigwedge v w. f (h w v) = f v$;
 $\bigwedge v. g (f v) v = v$]] \implies *fa-ind* (*lf-fd* ' *lf-set* (*ti-typ-pad-combine* (*t::'a itself*) *f g algn fn ti*) []) *h*
by (*auto simp: ti-typ-pad-combine-def Let-def intro: fa-ind-ti-typ-combine fa-ind-ti-pad-combine*)

lemma (**in** *wf-type*) *wf-lf-ti-typ-combine*:

[[*wf-lf* (*lf-set* *ti* []); *fn* \notin *set* (*field-names-list* *ti*);
 $\bigwedge v. \text{upd } (\text{acc } v) v = v$; $\bigwedge w u v. \text{upd } w (\text{upd } u v) = \text{upd } w v$;
 $\bigwedge w v. \text{acc } (\text{upd } w v) = w$;
upd-ind (*lf-set* *ti* []) *upd*; *acc-ind* *acc* (*lf-fd* ' *lf-set* *ti* []);

$fa-ind (lf-fd \text{ ' } lf-set \ ti \ \square) \ upd \ \square \implies$
 $wf-lf (lf-set (ti-typ-combine (t::'a \ itself) \ acc \ upd \ algn \ fn \ ti) \ \square)$
apply(*clarsimp simp: ti-typ-combine-def Let-def*)
apply(*rule wf-lf-extend-ti; simp?*)
apply(*rule wf-lf-adjust-ti; simp*)
apply(*clarsimp simp: ti-ind-def*)
apply(*drule lf-set-adjust-ti, simp*)
apply(*clarsimp simp: fu-commutes-def update-desc-def upd-ind-def acc-ind-def*
fu-s-comm-k-def
 $fa-fu-ind-def \ fa-ind-def$)
done

lemma (*in wf-type*) *wf-lf-ti-typ-pad-combine*:
 $\llbracket wf-lf (lf-set \ ti \ \square); \ fn \notin \ set \ (field-names-list \ ti); \ hd \ fn \neq \ CHR \ '!\';$
 $\bigwedge v. \ upd \ (acc \ v) \ v = v; \ \bigwedge w \ u \ v. \ upd \ w \ (upd \ u \ v) = upd \ w \ v;$
 $\bigwedge w \ v. \ acc \ (upd \ w \ v) = w;$
 $upd-ind \ (lf-set \ ti \ \square) \ upd; \ acc-ind \ acc \ (lf-fd \ \text{ ' } lf-set \ ti \ \square);$
 $fa-ind \ (lf-fd \ \text{ ' } lf-set \ ti \ \square) \ upd \ \square \implies$
 $wf-lf (lf-set (ti-typ-pad-combine (t::'a \ itself) \ acc \ upd \ algn \ fn \ ti) \ \square)$
apply(*clarsimp simp: ti-typ-pad-combine-def Let-def*)
apply (*fastforce intro!: wf-lf-ti-typ-combine wf-lf-ti-typ-pad-combine upd-ind-ti-typ-pad-combine*
 $acc-ind-ti-typ-pad-combine \ fa-ind-ti-typ-pad-combine$)
done

definition $upd-fa-ind \ X \ upd \equiv \ upd-ind \ X \ upd \wedge \ fa-ind \ (lf-fd \ \text{ ' } X) \ upd$

lemma (*in wf-type*) *wf-lf-ti-typ-pad-combine'*:
 $\llbracket wf-lf (lf-set \ ti \ \square); \ fn \notin \ set \ (field-names-list \ ti); \ hd \ fn \neq \ CHR \ '!\';$
 $\bigwedge v. \ upd \ (acc \ v) \ v = v; \ \bigwedge w \ u \ v. \ upd \ w \ (upd \ u \ v) = upd \ w \ v;$
 $\bigwedge w \ v. \ acc \ (upd \ w \ v) = w;$
 $upd-fa-ind \ (lf-set \ ti \ \square) \ upd; \ acc-ind \ acc \ (lf-fd \ \text{ ' } lf-set \ ti \ \square)\rrbracket$
 \implies
 $wf-lf (lf-set (ti-typ-pad-combine (t::'a \ itself) \ acc \ upd \ algn \ fn \ ti) \ \square)$
unfolding $upd-fa-ind-def$
by (*erule conjE*) (*rule wf-lf-ti-typ-pad-combine*)

lemma (*in c-type*) *upd-fa-ind-ti-typ-pad-combine*:
 $\llbracket upd-fa-ind \ (lf-set \ ti \ \square) \ h; \ \bigwedge w \ u \ v. \ upd \ w \ (h \ u \ v) = h \ u \ (upd \ w \ v);$
 $\bigwedge w \ v. \ acc \ (h \ w \ v) = acc \ v; \ \bigwedge v. \ upd \ (acc \ v) \ v = v \rrbracket$
 $\implies \ upd-fa-ind \ (lf-set \ (ti-typ-pad-combine \ (t::'a \ itself) \ acc \ upd \ algn \ fn \ ti) \ \square) \ h$
unfolding $upd-fa-ind-def$
by (*auto intro: upd-ind-ti-typ-pad-combine fa-ind-ti-typ-pad-combine*)

lemma *upd-fa-ind-empty* [*simp*]:
 $upd-fa-ind \ \{\} \ h$
by (*simp add: upd-fa-ind-def*)

lemma *wf-align-empty-typ-info*: $wf-align \ (empty-typ-info \ algn \ tn)$

by (*simp add: wf-align-def empty-typ-info-def*)

lemma *wf-align-list*: $wf\text{-align } t \implies wf\text{-align-list } fs \implies wf\text{-align-list } (fs @ [DTuple\ t\ f\ d])$
by (*induct fs*) *auto*

lemma *wf-align-struct*: $wf\text{-align } t \implies wf\text{-align-struct } st \implies wf\text{-align-struct } (extend\text{-ti-struct } st\ t\ f\ d)$
apply (*cases st*)
apply *simp*
apply (*simp add: wf-align-list*)
done

lemma *align-td-extend-ti*: $align\text{-td } (extend\text{-ti } tag\ t\ algn\ f\ d) = max (align\text{-td } tag)$
(*max algn (align-td t)*)
apply (*cases tag*)
apply (*simp*)
done

lemma *align-td-struct-extend-ti*: $aggregate\text{-struct } st \implies align\text{-td-struct } (extend\text{-ti-struct } st\ t\ f\ d) = max (align\text{-td-struct } st) (align\text{-td } t)$
by (*cases st*) *auto*

lemma *wf-align-extend-ti'*:
assumes *wf-t*: $wf\text{-align } t$
assumes *agg*: $aggregate\ tag$
assumes *wf-tag*: $wf\text{-align } tag$
shows $wf\text{-align } (extend\text{-ti } tag\ t\ algn\ f\ d)$

proof (*cases tag*)
case (*TypDesc algn' st n*)
with *wf-tag agg* **obtain** *le*: $align\text{-td-wo-align-struct } st \leq algn'$
and *le'*: $align\text{-td-struct } st \leq algn'$
and *wf-st*: $wf\text{-align-struct } st$
and *agg-st*: $aggregate\text{-struct } st$ **by** *auto*
from *wf-align-struct* [*OF wf-t wf-st*]
have *wf-st*: $wf\text{-align-struct } (extend\text{-ti-struct } st\ t\ f\ d)$.
from *align-td-wo-align-le-align-td* (2) [*OF this*]
have $align\text{-td-wo-align-struct } (extend\text{-ti-struct } st\ t\ f\ d) \leq align\text{-td-struct } (extend\text{-ti-struct } st\ t\ f\ d)$.
also from *align-td-struct-extend-ti* [*OF agg-st*]
have $\dots = max (align\text{-td-struct } st) (align\text{-td } t)$.
finally
have $align\text{-td-wo-align-struct } (extend\text{-ti-struct } st\ t\ f\ d) \leq max\ algn' (max\ algn (align\text{-td } t))$
using *le'*
by (*metis (full-types) le-max-iff-disj max.orderE*)
moreover from *align-td-struct-extend-ti* [*OF agg-st*] *le'*
have $align\text{-td-struct } (extend\text{-ti-struct } st\ t\ f\ d) \leq max\ algn' (max\ algn (align\text{-td } t))$
by (*metis max.cobounded2 max.mono*)
ultimately show *?thesis*

by (*simp add: TypDesc wf-st*)
qed

lemma (*in mem-type*) *wf-align-extend-ti*:
 assumes *agg: aggregate tag*
 assumes *wf-tag: wf-align tag*
 shows *wf-align (extend-ti tag (typ-info-t (TYPE('a))) algn f d)*
proof –
 have *wf-align (typ-info-t (TYPE('a)))* **by** (*rule wf-align*)
 from *wf-align-extend-ti' [OF this agg wf-tag]* **show** *?thesis* .
qed

lemma *wf-align-map-td [simp]*:
wf-align (map-td f g d) = wf-align d
wf-align-struct (map-td-struct f g ts) = wf-align-struct (ts)
wf-align-list (map-td-list f g fs) = wf-align-list fs
wf-align-tuple (map-td-tuple f g fd) = wf-align-tuple fd
by (*induct d and ts and fs and fd*) *auto*

lemma *wf-align-adjust-ti[simp]*: *wf-align (adjust-ti t acc upd) = wf-align t*
by (*simp add: adjust-ti-def*)

lemma (*in mem-type*) *wf-align-ti-typ-combine*:
aggregate tag \implies wf-align tag \implies wf-align (ti-typ-combine (TYPE('a)) acc upd
algn fn tag)
apply (*simp add: ti-typ-combine-def Let-def*)
apply (*rule wf-align-extend-ti'*)
apply (*simp add: wf-align*)
apply *assumption*
apply *assumption*
done

lemma *wf-align-ti-pad-combine*:
aggregate tag \implies wf-align tag \implies wf-align (ti-pad-combine n tag)
apply (*simp add: ti-pad-combine-def Let-def*)
apply (*rule wf-align-extend-ti'*)
apply *simp*
apply *assumption*
apply *assumption*
done

lemma (*in mem-type*) *wf-align-ti-typ-pad-combine*:
aggregate tag \implies wf-align tag \implies wf-align (ti-typ-pad-combine (TYPE('a)) acc
upd algn fn tag)
by (*simp add: ti-typ-pad-combine-def Let-def wf-align-ti-pad-combine wf-align-ti-typ-combine*)

lemma *wf-align-map-align*:
 assumes *wf-tag: wf-align tag*
 assumes *mono: $\bigwedge a. a \leq f a$*

shows *wf-align (map-align f tag)*
using *wf-tag mono*
apply (*cases tag*)
using *order-trans* **apply** *auto*
done

lemma *wf-align-final-pad: aggregate tag \implies wf-align tag \implies wf-align (final-pad align tag)*

by (*auto simp add: final-pad-def Let-def max-2-exp wf-align-map-align wf-align-ti-pad-combine*)

lemmas *wf-align-simps =*
wf-align-empty-typ-info
wf-align-ti-typ-pad-combine
wf-align-ti-typ-combine
wf-align-ti-pad-combine
wf-align-final-pad

lemma *align-field-empty-typ-info [simp]:*

align-field (empty-typ-info align tn)

by (*clarsimp simp: empty-typ-info-def align-field-def*)

lemma *align-td-wo-align-field-lookup:*

$\forall f m s n. \text{field-lookup } (t::('a,'b) \text{ typ-desc}) f m = \text{Some } (s,n) \longrightarrow \text{align-td-wo-align } s \leq \text{align-td-wo-align } t$

$\forall f m s n. \text{field-lookup-struct } (st::('a,'b) \text{ typ-struct}) f m = \text{Some } (s,n) \longrightarrow \text{align-td-wo-align } s \leq \text{align-td-wo-align-struct } st$

$\forall f m s n. \text{field-lookup-list } (ts::('a,'b) \text{ typ-tuple list}) f m = \text{Some } (s,n) \longrightarrow \text{align-td-wo-align } s \leq \text{align-td-wo-align-list } ts$

$\forall f m s n. \text{field-lookup-tuple } (x::('a,'b) \text{ typ-tuple}) f m = \text{Some } (s,n) \longrightarrow \text{align-td-wo-align } s \leq \text{align-td-wo-align-tuple } x$

by (*induct t and st and ts and x, all <clarsimp>*)

(fastforce simp: max-def split: option.splits)

lemma *align-td-field-lookup[rule-format]:*

$\forall f m s n. \text{wf-align } t \longrightarrow \text{field-lookup } (t::('a,'b) \text{ typ-desc}) f m = \text{Some } (s,n) \longrightarrow \text{align-td } s \leq \text{align-td } t$

$\forall f m s n. \text{wf-align-struct } st \longrightarrow \text{field-lookup-struct } (st::('a,'b) \text{ typ-struct}) f m = \text{Some } (s,n) \longrightarrow \text{align-td } s \leq \text{align-td-struct } st$

$\forall f m s n. \text{wf-align-list } ts \longrightarrow \text{field-lookup-list } (ts::('a,'b) \text{ typ-tuple list}) f m = \text{Some } (s,n) \longrightarrow \text{align-td } s \leq \text{align-td-list } ts$

$\forall f m s n. \text{wf-align-tuple } x \longrightarrow \text{field-lookup-tuple } (x::('a,'b) \text{ typ-tuple}) f m = \text{Some } (s,n) \longrightarrow \text{align-td } s \leq \text{align-td-tuple } x$

apply (*induct t and st and ts and x, all <clarsimp>*)

apply (*fastforce simp: max-def split: option.splits*)

done

lemma (*in mem-type*) *align-td-field-lookup-mem-type: field-lookup (typ-info-t (TYPE('a))) f m = Some (s, n) \implies*

align-td s \leq align-td (typ-info-t (TYPE('a)))

```

apply (rule align-td-field-lookup(1))
apply (rule wf-align)
apply simp
done

```

lemma *align-field-extend-ti*:

```

[[ align-field s; align-field t; wf-align t; 2^(align-td t) dvd size-td s ]] =>
  align-field (extend-ti s t algn fn d)
apply(cases s, clarsimp)
subgoal for x1 typ-struct xs
  apply(cases typ-struct, clarsimp)
  apply(clarsimp simp: align-field-def split: option.splits)
  apply(clarsimp simp: align-field-def)
  apply(subst (asm) field-lookup-list-append)
  apply(clarsimp split: if-split-asm option.splits)
subgoal for x2 f s n
  apply(cases f, clarsimp)
  apply clarsimp
  apply(frule field-lookup-offset2)
  apply (meson align-td-field-lookup(1) dvd-diffD field-lookup-offset-le(1) power-le-dvd)
  done
subgoal for x2 f s n
  by(cases f) auto
done
done

```

lemma *align-field-ti-pad-combine*:

```

align-field ti => align-field (ti-pad-combine n ti)
apply(clarsimp simp: ti-pad-combine-def Let-def)
apply(erule align-field-extend-ti)
apply(clarsimp simp: align-field-def)
apply clarsimp
apply clarsimp
done

```

lemma *align-field-map-align* [simp]: *align-field* (map-align f t) = *align-field* t
by (cases t) (auto simp add: align-field-def)

lemma *align-field-final-pad*:

```

align-field ti => align-field (final-pad algn ti)
apply(clarsimp simp: final-pad-def Let-def split: if-split-asm)
apply(erule align-field-ti-pad-combine)
done

```

lemma *field-lookup-adjust-ti-None*:

```

∀ fn m s n. field-lookup (adjust-ti t acc upd) fn m = None →
  (field-lookup t fn m = None)
  ∀ fn m s n. field-lookup-struct (map-td-struct (λn algn. update-desc acc upd)
(update-desc acc upd) st)

```

$fn\ m = None \longrightarrow$
 $(field_lookup_struct\ st\ fn\ m = None)$
 $\forall fn\ m\ s\ n. field_lookup_list\ (map_td_list\ (\lambda n\ algn. update_desc\ acc\ upd)\ (update_desc\ acc\ upd)\ ts)\ fn\ m = None \longrightarrow$
 $(field_lookup_list\ ts\ fn\ m = None)$
 $\forall fn\ m\ s\ n. field_lookup_tuple\ (map_td_tuple\ (\lambda n\ algn. update_desc\ acc\ upd)\ (update_desc\ acc\ upd)\ x)\ fn\ m = None \longrightarrow$
 $(field_lookup_tuple\ x\ fn\ m = None)$
proof *(induct t and st and ts and x)*
case *(TypDesc nat typ-struct list)*
then show *?case by (clarsimp simp: adjust-ti-def split: option.splits)*
next
case *(TypScalar nat1 nat2 a)*
then show *?case by auto*
next
case *(TypAggregate list)*
then show *?case by auto*
next
case *Nil-typ-desc*
then show *?case by auto*
next
case *(Cons-typ-desc dt-tuple list)*
then show *?case*
apply *(clarsimp simp: adjust-ti-def split: option.splits)*
apply *(cases dt-tuple)*
apply *clarsimp*
done
next
case *(DTuple-typ-desc typ-desc list b)*
then show *?case by (clarsimp simp: adjust-ti-def split: option.splits)*
qed

lemma *field-lookup-adjust-ti' [rule-format]:*

$\forall fn\ m\ s\ n. field_lookup\ (adjust_ti\ t\ acc\ upd)\ fn\ m = Some\ (s,n) \longrightarrow$
 $(\exists s'. field_lookup\ t\ fn\ m = Some\ (s',n) \wedge align_td_wo_align\ s = align_td_wo_align\ s')$
 $\forall fn\ m\ s\ n. field_lookup_struct\ (map_td_struct\ (\lambda n\ algn. update_desc\ acc\ upd)\ (update_desc\ acc\ upd)\ st)$
 $fn\ m = Some\ (s,n) \longrightarrow$
 $(\exists s'. field_lookup_struct\ st\ fn\ m = Some\ (s',n) \wedge align_td_wo_align\ s = align_td_wo_align\ s')$
 $\forall fn\ m\ s\ n. field_lookup_list\ (map_td_list\ (\lambda n\ algn. update_desc\ acc\ upd)\ (update_desc\ acc\ upd)\ ts)\ fn\ m = Some\ (s,n) \longrightarrow$
 $(\exists s'. field_lookup_list\ ts\ fn\ m = Some\ (s',n) \wedge align_td_wo_align\ s = align_td_wo_align\ s')$
 $\forall fn\ m\ s\ n. field_lookup_tuple\ (map_td_tuple\ (\lambda n\ algn. update_desc\ acc\ upd)\ (update_desc\ acc\ upd)\ x)\ fn\ m = Some\ (s,n) \longrightarrow$
 $(\exists s'. field_lookup_tuple\ x\ fn\ m = Some\ (s',n) \wedge align_td_wo_align\ s = align_td_wo_align\ s')$

```

proof (induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case by (clarsimp simp: adjust-ti-def)
next
  case (TypScalar nat1 nat2 a)
  then show ?case by (clarsimp simp: adjust-ti-def)
next
  case (TypAggregate list)
  then show ?case by (clarsimp simp: adjust-ti-def)
next
  case Nil-typ-desc
  then show ?case by (clarsimp simp: adjust-ti-def)
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply (clarsimp simp: adjust-ti-def)
    apply (clarsimp split: option.splits)
    apply (rule conjI; clarsimp)
    apply (cases dt-tuple, clarsimp)
    apply (cases dt-tuple, clarsimp split: if-split-asm)
  subgoal for fn
    apply (drule spec [where x=fn])
    apply clarsimp
    apply (fold adjust-ti-def)
    apply (subst (asm) field-lookup-adjust-ti-None; simp)
    done
  apply fastforce
  done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by (clarsimp simp: adjust-ti-def)
qed

```

lemma *field-lookup-adjust-ti'''* [rule-format]:

$$\forall fn\ m\ s\ n. \text{field-lookup (adjust-ti } t\ \text{acc upd) } fn\ m = \text{Some } (s,n) \longrightarrow$$

$$(\exists s'. \text{field-lookup } t\ fn\ m = \text{Some } (s',n) \wedge \text{align-td } s = \text{align-td } s')$$

$$\forall fn\ m\ s\ n. \text{field-lookup-struct (map-td-struct } (\lambda n\ \text{algn. update-desc acc upd) (\text{update-desc acc upd) } st) \text{ } fn\ m = \text{Some } (s,n) \longrightarrow$$

$$(\exists s'. \text{field-lookup-struct } st\ fn\ m = \text{Some } (s',n) \wedge \text{align-td } s = \text{align-td } s')$$

$$\forall fn\ m\ s\ n. \text{field-lookup-list (map-td-list } (\lambda n\ \text{algn. update-desc acc upd) (\text{update-desc acc upd) } ts) \text{ } fn\ m = \text{Some } (s,n) \longrightarrow$$

$$(\exists s'. \text{field-lookup-list } ts\ fn\ m = \text{Some } (s',n) \wedge \text{align-td } s = \text{align-td } s')$$

$$\forall fn\ m\ s\ n. \text{field-lookup-tuple (map-td-tuple } (\lambda n\ \text{algn. update-desc acc upd) (\text{update-desc acc upd) } x) \text{ } fn\ m = \text{Some } (s,n) \longrightarrow$$

$$(\exists s'. \text{field-lookup-tuple } x\ fn\ m = \text{Some } (s',n) \wedge \text{align-td } s = \text{align-td } s')$$

```

proof(induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case by (clarsimp simp: adjust-ti-def)

```

```

next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply(clarsimp simp: adjust-ti-def)
    apply(clarsimp split: option.splits)
    apply(rule conjI; clarsimp)
    apply(cases dt-tuple, clarsimp)
    apply(cases dt-tuple, clarsimp split: if-split-asm)
  subgoal for fn
    apply(drule spec [where x=fn])
    apply clarsimp
    apply(fold adjust-ti-def)
    apply(subst (asm) field-lookup-adjust-ti-None; simp)
    done
  apply fastforce
  done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by(clarsimp simp: adjust-ti-def)
qed

lemma field-lookup-adjust-ti:
  [[ field-lookup (adjust-ti t acc upd) fn m = Some (s,n) ]] ==>
  (∃ s'. field-lookup t fn m = Some (s',n) ∧ align-td-wo-align s = align-td-wo-align
s')
  by (rule field-lookup-adjust-ti')

lemma field-lookup-adjust-ti1:
  [[ field-lookup (adjust-ti t acc upd) fn m = Some (s,n) ]] ==>
  (∃ s'. field-lookup t fn m = Some (s',n) ∧ align-td s = align-td s')
  by (rule field-lookup-adjust-ti''')

lemma align-adjust-ti:
  align-field ti ==> align-field (adjust-ti ti acc upd)
  apply(clarsimp simp: align-field-def)
  apply(drule field-lookup-adjust-ti1)
  apply clarsimp
  done

lemma (in mem-type) align-field-ti-typ-combine:

```

```

[[ align-field ti; 2 ^ align-td (typ-info-t TYPE('a)) dvd size-td ti ]] ==>
  align-field (ti-typ-combine (t::'a itself) acc upd algn fn ti)
apply(clarsimp simp: ti-typ-combine-def Let-def)
apply(rule align-field-extend-ti, assumption)
apply(rule align-adjust-ti)
apply(rule align-field)
apply (simp add: wf-align)
apply simp
done

```

```

lemma (in mem-type) align-td-wo-align-type-info-t-le-align-td:
  align-td-wo-align (typ-info-t TYPE('a)) ≤ align-td (typ-info-t TYPE('a))
proof –
  have wf-align (typ-info-t TYPE('a)) by (rule wf-align)
  then show ?thesis thm wf-align.wfal0 by (rule align-td-wo-align-le-align-td(1))
qed

```

```

lemma (in mem-type) align-field-ti-typ-pad-combine:

```

```

  [[wf-align ti; align-field ti; aggregate ti]] ==>
  align-field (ti-typ-pad-combine (t::'a itself) acc upd algn fn ti)

```

```

unfolding ti-typ-pad-combine-def Let-def
apply (rule align-field-ti-typ-combine)
subgoal
  apply clarsimp
  apply (rule align-field-ti-pad-combine)
  apply assumption
  done
apply clarsimp
apply (rule conjI)
subgoal
  apply (clarsimp simp add: align-of-def)
  apply (metis (no-types, lifting) align-td-wo-align-type-info-t-le-align-td dvd-padup-add
max.cobounded2 max-2-exp
  power-le-dvd size-td-lt-ti-pad-combine zero-less-numeral zero-less-power)
  done
apply (clarsimp simp add: align-of-def)
by (simp add: max-2-exp padup-dvd power-le-dvd)

```

```

lemma npf-extend-ti [simp]:

```

```

  non-padding-fields (extend-ti s t algn fn d) = non-padding-fields s @
  (if hd fn = CHR "!" then [] else [fn])

```

```

apply (cases s)
subgoal for x1 typ-struct xs
  apply (cases typ-struct; simp)
  done
done

```

```

lemma npf-ti-pad-combine [simp]:

```

non-padding-fields (ti-pad-combine n tag) = *non-padding-fields* tag
by (clarsimp simp: ti-pad-combine-def Let-def)

lemma (in c-type) npf-ti-typ-combine [simp]:
 hd fn ≠ CHR "!" ⇒
non-padding-fields (ti-typ-combine (t::'a itself) acc upd algn fn tag) = *non-padding-fields*
 tag @ [fn]
by (clarsimp simp: ti-typ-combine-def Let-def)

lemma (in c-type) npf-ti-typ-pad-combine [simp]:
 hd fn ≠ CHR "!" ⇒
non-padding-fields (ti-typ-pad-combine (t::'a itself) acc upd algn fn tag) = *non-padding-fields*
 tag @ [fn]
by (clarsimp simp: ti-typ-pad-combine-def Let-def)

lemma *non-padding-fields-map-align* [simp]:
non-padding-fields (map-align f t) = *non-padding-fields* t
by (cases t) simp

lemma *npf-final-pad* [simp]:
non-padding-fields (final-pad algn tag) = *non-padding-fields* tag
by (clarsimp simp: final-pad-def Let-def)

lemma *npf-empty-typ-info* [simp]:
non-padding-fields (empty-typ-info algn tn) = []
by (clarsimp simp: empty-typ-info-def)

definition field-fd' :: 'a xtyp-info ⇒ qualified-field-name → 'a field-desc **where**
 field-fd' t f ≡ case field-lookup t f 0 of None ⇒ None | Some x ⇒ Some (field-desc
 (fst x))

lemma *padup-zero* [simp]:
 padup n 0 = 0
by (clarsimp simp: padup-def)

lemma *padup-same* [simp]:
 padup n n = 0
by (clarsimp simp: padup-def)

lemmas *size-td-simps-0* = *size-td-lt-final-pad size-td-lt-ti-typ-pad-combine*

lemmas *size-td-simps-1* = *size-td-simps-0*
aggregate-ti-typ-pad-combine aggregate-empty-typ-info

lemmas *size-td-simps-2* = *padup-def align-of-final-pad align-of-def*

lemmas *size-td-simps* = *size-td-simps-1 size-td-simps-2*

lemmas *size-td-simps-3* = *size-td-simps-0 size-td-simps-2*

lemma *fu-commutes-sym*: *fu-commutes* $x\ y = \text{fu-commutes } y\ x$
by (*auto simp add: fu-commutes-def*)

lemma *wf-lf-insert-recursion*:

assumes *wf-D*: *wf-lf* D

assumes *cons-x*: *fd-cons-desc* (*lf-fd* x) (*lf-sz* x)

assumes *comm-D*: $\bigwedge y. y \in D \implies \text{fu-commutes } (\text{field-update } (\text{lf-fd } x)) (\text{field-update } (\text{lf-fd } y)) \wedge$

$\text{fa-fu-ind } (\text{lf-fd } x) (\text{lf-fd } y) (\text{lf-sz } y) (\text{lf-sz } x) \wedge$

$\text{fa-fu-ind } (\text{lf-fd } y) (\text{lf-fd } x) (\text{lf-sz } x) (\text{lf-sz } y)$

shows *wf-lf* (*insert* $x\ D$)

proof –

have *comm-x-D*:

fu-commutes (*field-update* (*lf-fd* x')) (*field-update* (*lf-fd* y)) **and**

fa-fu-ind (*lf-fd* x') (*lf-fd* y) (*lf-sz* y) (*lf-sz* x')

if *x'-in*: $x' \in \text{insert } x\ D$ **and** *y-in*: $y \in \text{insert } x\ D$ **and** *neq*: *lf-fn* $y \neq \text{lf-fn } x'$

for x' **and** y

proof –

show *fu-commutes* (*field-update* (*lf-fd* x')) (*field-update* (*lf-fd* y))

proof (*cases* $x' = x$)

case *True*

from *True y-in neq comm-D x'-in* **show** *?thesis*

by *auto*

next

case *False*

note *neq-x'-x = this*

with *x'-in* **have** *x'-D*: $x' \in D$

by *auto*

from *comm-D [OF this]* **have** *fu-commutes* (*field-update* (*lf-fd* x)) (*field-update* (*lf-fd* x'))

by *auto*

with *fu-commutes-sym* **have** *comm-x'-x*: *fu-commutes* (*field-update* (*lf-fd* x')) (*field-update* (*lf-fd* x))

by *auto*

from *neq* **have** *neq'*: $y \neq x'$

by *auto*

show *?thesis*

proof (*cases* $y = x$)

case *True*

with *comm-x'-x* **show** *?thesis* **by** *simp*

next

case *False*

from *False y-in* **have** *y-D*: $y \in D$ **by** *simp*

with *x'-D neq wf-D*

show *?thesis*

by (*auto simp add: wf-lf-def*)


```

      qed
    qed
  next
  show fa-fu-ind (lf-fd  $x'$ ) (lf-fd  $y$ ) (lf-sz  $y$ ) (lf-sz  $x'$ )
  proof (cases  $x' = x$ )
    case True
    from True y-in neq comm-D x'-in show ?thesis
      by auto
    next
    case False
    note neq-x'-x = this
    with  $x'$ -in have  $x'$ -D:  $x' \in D$ 
      by auto
    from comm-D [OF this] have fa-fu-ind (lf-fd  $x$ ) (lf-fd  $x'$ ) (lf-sz  $x'$ ) (lf-sz  $x$ )
and
    fa-fu-x'-x: fa-fu-ind (lf-fd  $x'$ ) (lf-fd  $x$ ) (lf-sz  $x$ ) (lf-sz  $x'$ ) by auto
    from neq have neq':  $y \neq x'$ 
      by auto
    show ?thesis
    proof (cases  $y = x$ )
      case True
      with fa-fu-x'-x show ?thesis by simp
    next
    case False
    from False y-in have  $y$ -D:  $y \in D$  by simp
    with  $x'$ -D neq wf-D
    show ?thesis
      by (auto simp add: wf-lf-def)
    qed
  qed
  qed
  with cons-x wf-D
  show ?thesis
    by (auto simp add: wf-lf-def)
qed

```

lemma *wf-lf-insert-recursion'*:

```

  assumes cons-x: fd-cons-desc (lf-fd  $x$ ) (lf-sz  $x$ )
  assumes comm-D:  $\bigwedge y. y \in D \implies fu-commutes$  (field-update (lf-fd  $x$ )) (field-update
(lf-fd  $y$ ))  $\wedge$ 
    fa-fu-ind (lf-fd  $x$ ) (lf-fd  $y$ ) (lf-sz  $y$ ) (lf-sz  $x$ )  $\wedge$ 
    fa-fu-ind (lf-fd  $y$ ) (lf-fd  $x$ ) (lf-sz  $x$ ) (lf-sz  $y$ )
  assumes wf-D: wf-lf  $D$ 
  shows wf-lf (insert  $x$   $D$ )
  using wf-D cons-x comm-D
  by (rule wf-lf-insert-recursion)

```

```

lemma wf-lf-insert-recursion'':
  assumes wf-D: wf-lf D
  assumes cons-x: fd-cons-desc (lf-fd x) (lf-sz x)
  assumes comm-D:  $\bigwedge y. y \in D \implies \text{lf-fn } y \neq \text{lf-fn } x \implies \text{fu-commutes (field-update (lf-fd x)) (field-update (lf-fd y))} \wedge$ 
    fa-fu-ind (lf-fd x) (lf-fd y) (lf-sz y) (lf-sz x)  $\wedge$ 
    fa-fu-ind (lf-fd y) (lf-fd x) (lf-sz x) (lf-sz y)
  shows wf-lf (insert x D)
proof -
  have comm-x-D:
    fu-commutes (field-update (lf-fd x')) (field-update (lf-fd y)) and
    fa-fu-ind (lf-fd x') (lf-fd y) (lf-sz y) (lf-sz x')
    if x'-in: x'  $\in$  insert x D and y-in: y  $\in$  insert x D and neq: lf-fn y  $\neq$  lf-fn x'
    for x' and y
  proof -
    show fu-commutes (field-update (lf-fd x')) (field-update (lf-fd y))
    proof (cases x' = x)
      case True
      from True y-in neq comm-D x'-in show ?thesis
      by auto
    next
      case False
      note neq-x'-x = this
      with x'-in have x'-D: x'  $\in$  D
      by auto
      from neq have neq': y  $\neq$  x'
      by auto
      show ?thesis
      proof (cases y = x)
        case True
        with neq neq-x'-x neq'
        have lf-fn x'  $\neq$  lf-fn x by simp
        from comm-D [OF x'-D this] have fu-commutes (field-update (lf-fd x))
          (field-update (lf-fd x'))
          by auto
        with fu-commutes-sym have comm-x'-x: fu-commutes (field-update (lf-fd
          x')) (field-update (lf-fd x))
          by auto
        from True comm-x'-x show ?thesis by simp
      next
        case False
        from False y-in have y-D: y  $\in$  D by simp
        with x'-D neq wf-D
        show ?thesis
        by (auto simp add: wf-lf-def)
      qed
    qed
  next

```

```

show fa-fu-ind (lf-fd  $x'$ ) (lf-fd  $y$ ) (lf-sz  $y$ ) (lf-sz  $x'$ )
proof (cases  $x' = x$ )
  case True
    from True y-in neq comm-D  $x'$ -in show ?thesis
      by auto
  next
    case False
    note neq-x'-x = this
    with  $x'$ -in have  $x'$ -D:  $x' \in D$ 
      by auto
    from neq have neq':  $y \neq x'$ 
      by auto
    show ?thesis
    proof (cases  $y = x$ )
      case True
        with neq neq-x'-x neq'
        have lf-fn  $x' \neq$  lf-fn  $x$  by simp
        from comm-D [OF  $x'$ -D this] have fa-fu-ind (lf-fd  $x$ ) (lf-fd  $x'$ ) (lf-sz  $x'$ )
(lf-sz  $x$ ) and
          fa-fu-x'-x: fa-fu-ind (lf-fd  $x'$ ) (lf-fd  $x$ ) (lf-sz  $x$ ) (lf-sz  $x'$ ) by auto
        from True fa-fu-x'-x show ?thesis by simp
      next
        case False
        from False y-in have  $y$ -D:  $y \in D$  by simp
        with  $x'$ -D neq wf-D
        show ?thesis
          by (auto simp add: wf-lf-def)
    qed
  qed
qed
with cons-x wf-D
show ?thesis
  by (auto simp add: wf-lf-def)
qed

```

lemma *wf-field-desc-empty* [*simp*]:
wf-field-desc (\downarrow *field-access* = λv *bs*. [], *field-update* = λ *bs*. *id*, *field-sz* = 0)
by (*unfold-locals*) (*auto simp add: padding-base.eq-padding-def padding-base.eq-upto-padding-def*)

lemma *field-desc-independent-subset*:
 $D \subseteq E \implies$ *field-desc-independent acc upd* $E \implies$ *field-desc-independent acc upd* D
by (*auto simp add: field-desc-independent-def*)

lemma *field-desc-independent-union-iff*:
field-desc-independent acc upd ($D \cup E$) =
(*field-desc-independent acc upd* $D \wedge$ *field-desc-independent acc upd* E)

by (*auto simp add: field-desc-independent-def*)

lemma *field-desc-independent-unionI*:

field-desc-independent acc upd D \implies *field-desc-independent acc upd E* \implies
field-desc-independent acc upd (D \cup E)

by (*simp add: field-desc-independent-union-iff*)

lemma *field-desc-independent-unionD1*:

field-desc-independent acc upd (D \cup E) \implies
field-desc-independent acc upd D

by (*simp add: field-desc-independent-union-iff*)

lemma *field-desc-independent-unionD2*:

field-desc-independent acc upd (D \cup E) \implies
field-desc-independent acc upd E

by (*simp add: field-desc-independent-union-iff*)

lemma *field-desc-independent-insert-iff*:

field-desc-independent acc upd (insert d D) =
(field-desc-independent acc upd {d}) \wedge *field-desc-independent acc upd D*

apply (*subst insert-is-Un*)

apply (*simp only: field-desc-independent-union-iff*)

done

lemma *field-desc-independent-insertI*:

field-desc-independent acc upd {d} \implies *field-desc-independent acc upd D* \implies
field-desc-independent acc upd (insert d D)

apply (*subst insert-is-Un*)

apply (*simp only: field-desc-independent-union-iff*)

done

lemma *field-desc-independent-insertD1*:

assumes *ins: field-desc-independent acc upd (insert d D)*

shows *field-desc-independent acc upd {d}*

proof –

from *ins* **have** *field-desc-independent acc upd ({d} \cup D)*

by (*simp*)

from *field-desc-independent-unionD1* [*OF this*]

show *?thesis* .

qed

lemma *field-desc-independent-insertD2*:

assumes *ins: field-desc-independent acc upd (insert d D)*

shows *field-desc-independent acc upd D*

proof –

from *ins* **have** *field-desc-independent acc upd ({d} \cup D)*

by (*simp*)

from *field-desc-independent-unionD2* [*OF this*]

show *?thesis* .

qed

lemma *field-descs-independent-append-first*: $\text{field-descs-independent } (xs \text{ @ } ys) \implies \text{field-descs-independent } xs$
by (*induct xs arbitrary: ys*) (*auto intro: field-desc-independent-subset*)

lemma *field-descs-independent-append-second*: $\text{field-descs-independent } (xs \text{ @ } ys) \implies \text{field-descs-independent } ys$
by (*induct xs arbitrary: ys*) (*auto intro: field-desc-independent-subset*)

lemma *field-descs-independent-append-first-ind*:
 $\text{field-descs-independent } (xs \text{ @ } ys) \implies x \in \text{set } xs \implies \text{field-desc-independent } (\text{field-access } x) (\text{field-update } x) (\text{set } ys)$
by (*induct xs arbitrary: ys*) (*auto simp add: field-desc-independent-union-iff*)

lemma *field-descs-independent-appendI1*:
 $\text{field-descs-independent } xs \implies \text{field-descs-independent } ys \implies (\forall x \in \text{set } xs. \text{field-desc-independent } (\text{field-access } x) (\text{field-update } x) (\text{set } ys)) \implies \text{field-descs-independent } (xs \text{ @ } ys)$
by (*induct xs arbitrary: ys*) (*auto simp add: field-desc-independent-union-iff*)

lemma *field-desc-independent-symmetric*:
 $(\forall x \in X. \text{field-desc-independent } (\text{field-access } x) (\text{field-update } x) Y) \implies (\forall y \in Y. \text{field-desc-independent } (\text{field-access } y) (\text{field-update } y) X)$
by (*simp add: field-desc-independent-def fu-commutes-def*)

lemma *field-desc-independent-symmetric-singleton*:
 $\text{field-desc-independent } (\text{field-access } x) (\text{field-update } x) Y \implies y \in Y \implies \text{field-desc-independent } (\text{field-access } y) (\text{field-update } y) \{x\}$
using *field-desc-independent-symmetric* **by** *blast*

lemma *field-descs-independent-appendI2*:
assumes *ind-xs*: $\text{field-descs-independent } xs$
assumes *ind-ys*: $\text{field-descs-independent } ys$
assumes *ind-xs-ys*: $\forall y \in \text{set } ys. \text{field-desc-independent } (\text{field-access } y) (\text{field-update } y) (\text{set } xs)$
shows $\text{field-descs-independent } (xs \text{ @ } ys)$
using *field-desc-independent-symmetric* [*OF ind-xs-ys*] *ind-xs ind-ys*
by (*auto intro: field-descs-independent-appendI1*)

lemma *field-desc-indepenent-empty* [*simp*]:
 $\text{field-desc-independent } (\text{field-access } d) (\text{field-update } d) \{\}$ **by** (*simp add: field-desc-independent-def*)

```

lemma field-update-apply-field-updates-commute:
  fixes d::'a field-desc
  fixes ds::'a field-desc list
assumes ind-d:field-desc-independent (field-access d) (field-update d) (set ds)
assumes ind-ds: field-descs-independent ds
shows field-update d bs (fst (apply-field-updates ds bs' s)) =
  fst (apply-field-updates ds bs' (field-update d bs s))
using ind-d ind-ds
proof (induct ds arbitrary: d bs bs' s)
case Nil
then show ?case by simp
next
  case (Cons d' ds')
  from Cons.prems obtain
    ind-d1: field-desc-independent (field-access d) (field-update d) (insert d' (set ds')) and
    ind-d': field-desc-independent (field-access d') (field-update d') (set ds') and
    ind-d: field-desc-independent (field-access d) (field-update d) (set ds') and
    ind-ds': field-descs-independent ds'
    by (auto intro: field-desc-independent-subset)

  from ind-d1 interpret fi: field-desc-independent field-access d field-update d
insert d' (set ds') .
  from fi.fu-commutes [of d']
  have fu-commutes (field-update d) (field-update d') by simp
  then
    have d-d'-com: field-update d bs (field-update d' bs' s) = (field-update d' bs' (field-update d bs s))
    by (auto simp add: fu-commutes-def)

  from Cons.prems obtain
    ind-d': field-desc-independent (field-access d') (field-update d') (set ds') and
    ind-d: field-desc-independent (field-access d) (field-update d) (set ds') and
    ind-ds': field-descs-independent ds'
    by (auto intro: field-desc-independent-subset)

  note hyp = Cons.hyps [OF ind-d ind-ds']

  show ?case by (simp add: hyp d-d'-com)
qed

lemma (in wf-field-desc) is-padding-byte-acc-upd-compose:
assumes acc-upd:  $\bigwedge v s. acc (upd v s) = v$ 
shows padding-base.is-padding-byte (field-access d  $\circ$  acc)
  ( $\lambda bs v. upd (field-update d bs (acc v)) v (field-sz d) i =$ 
is-padding-byte i)
apply (simp add: is-padding-byte-def padding-base.is-padding-byte-def)
by (metis acc-upd)

```

```

lemma (in wf-field-desc) is-value-byte-acc-upd-compose:
  assumes acc-upd:  $\bigwedge v s. \text{acc } (\text{upd } v s) = v$ 
  shows padding-base.is-value-byte (field-access d  $\circ$  acc)
    ( $\lambda bs v. \text{upd } (\text{field-update } d bs (\text{acc } v)) v$ ) (field-sz d) i =
    is-value-byte i
  apply (simp add: is-value-byte-def padding-base.is-value-byte-def)
  by (metis acc-upd)

lemma (in wf-field-desc) wf-field-desc-update-desc:
  assumes double-update-upd:  $\bigwedge v w s. \text{upd } v (\text{upd } w s) = \text{upd } v s$ 
  assumes acc-upd:  $\bigwedge v s. \text{acc } (\text{upd } v s) = v$ 
  assumes upd-acc:  $\bigwedge s. \text{upd } (\text{acc } s) s = s$ 
  shows wf-field-desc (update-desc acc upd d) (is wf-field-desc ?upd)
proof (unfold-locales)
  fix i
  assume i-bound:  $i < \text{field-sz } (\text{update-desc } \text{acc } \text{upd } d)$ 
  show padding-base.is-padding-byte (field-access (update-desc acc upd d))
    (field-update (update-desc acc upd d))
    (field-sz (update-desc acc upd d)) i  $\neq$ 
    padding-base.is-value-byte (field-access (update-desc acc upd d))
    (field-update (update-desc acc upd d))
    (field-sz (update-desc acc upd d)) i
  using complement i-bound
  by (simp add: update-desc-def is-padding-byte-acc-upd-compose [of acc upd, OF
acc-upd]
    is-value-byte-acc-upd-compose [of acc upd, OF acc-upd])
next
  fix bs bs' s
  show field-update ?upd bs (field-update ?upd bs' s) = field-update ?upd bs s
  by (simp add: update-desc-def double-update double-update-upd acc-upd)
next
  fix bs s bs' s'
  show field-access ?upd (field-update ?upd bs s) bs' = field-access ?upd (field-update
?upd bs s') bs'
  by (simp add: update-desc-def access-update acc-upd)
next
  fix s bs
  show field-update ?upd (field-access ?upd s bs) s = s
  by (simp add: update-desc-def update-access upd-acc)
next
  fix s bs
  show (field-access ?upd s (take (field-sz ?upd) bs)) = field-access ?upd s bs
  by (simp add: update-desc-def access-size)
next
  fix s bs
  show length (field-access ?upd s bs) = field-sz ?upd
  by (simp add: update-desc-def access-result-size)
next
  fix bs

```

show $\text{field-update } ?\text{upd } (\text{take } (\text{field-sz } ?\text{upd}) \text{ bs}) = \text{field-update } ?\text{upd } \text{bs}$
by (*simp add: update-desc-def update-size*)
qed

lemma *toplevel-field-descs-subset*:
fixes $t::'a \text{ xtyp-info}$ **and**
 $st::'a \text{ xtyp-info-struct}$ **and**
 $fs::'a \text{ xtyp-info-tuple list}$ **and**
 $f::'a \text{ xtyp-info-tuple}$
shows
 $\text{set } (\text{toplevel-field-descs } t) \subseteq \text{set } (\text{field-descs } t)$
 $\text{set } (\text{toplevel-field-descs-struct } st) \subseteq \text{set } (\text{field-descs-struct } st)$
 $\text{set } (\text{toplevel-field-descs-list } fs) \subseteq \text{set } (\text{field-descs-list } fs)$
 $\text{set } (\text{toplevel-field-descs-tuple } f) \subseteq \text{set } (\text{field-descs-tuple } f)$
by (*induct t and st and fs and f*) *auto*

lemma
fixes $t::'a \text{ xtyp-info}$ **and**
 $st::'a \text{ xtyp-info-struct}$ **and**
 $fs::'a \text{ xtyp-info-tuple list}$ **and**
 $f::'a \text{ xtyp-info-tuple}$
shows
 $\text{lf-fd } ' \text{ lf-set } t \text{ xs} \subseteq \text{set } (\text{field-descs } t)$
 $\text{lf-fd } ' \text{ lf-set-struct } st \text{ xs} \subseteq \text{set } (\text{field-descs-struct } st)$
 $\text{lf-fd } ' \text{ lf-set-list } fs \text{ xs} \subseteq \text{set } (\text{field-descs-list } fs)$
 $\text{lf-fd } ' \text{ lf-set-tuple } f \text{ xs} \subseteq \text{set } (\text{field-descs-tuple } f)$
apply (*induction t and st and fs and f arbitrary: xs*) **apply** *auto*
oops

lemma *lf-set-subset-field-descs*:
fixes $t::'a \text{ xtyp-info}$ **and**
 $st::'a \text{ xtyp-info-struct}$ **and**
 $fs::'a \text{ xtyp-info-tuple list}$ **and**
 $f::'a \text{ xtyp-info-tuple}$
shows
 $\wedge xs. \text{lf-fd } ' \text{ lf-set } t \text{ xs} \subseteq \text{set } (\text{field-descs } t)$
 $\wedge xs. \text{lf-fd } ' \text{ lf-set-struct } st \text{ xs} \subseteq \text{set } (\text{field-descs-struct } st)$
 $\wedge xs. \text{lf-fd } ' \text{ lf-set-list } fs \text{ xs} \subseteq \text{set } (\text{field-descs-list } fs)$
 $\wedge xs. \text{lf-fd } ' \text{ lf-set-tuple } f \text{ xs} \subseteq \text{set } (\text{field-descs-tuple } f)$
by (*induction t and st and fs and f*) (*auto simp add: image-subset-iff*)

lemma *wf-field-descs-empty-typ-info* [*simp*]: $\text{wf-field-descs } (\text{set } (\text{field-descs } (\text{empty-typ-info } \text{algn } t)))$

by (simp add: empty-typ-info-def wf-field-descs-def)

lemma *field-descs-map*:

field-descs (map-td (λ- -. update-desc acc upd) (update-desc acc upd) t) =
 map (update-desc acc upd) (*field-descs* t)
field-descs-struct (map-td-struct (λ- -. update-desc acc upd) (update-desc acc upd)
 st) =
 map (update-desc acc upd) (*field-descs-struct* st)
field-descs-list (map-td-list (λ- -. update-desc acc upd) (update-desc acc upd) fs)
 =
 map (update-desc acc upd) (*field-descs-list* fs)
field-descs-tuple (map-td-tuple (λ- -. update-desc acc upd) (update-desc acc upd)
 f) =
 map (update-desc acc upd) (*field-descs-tuple* f)
 by (induct t and st and fs and f) auto

lemma *component-update-access*:

assumes *wf:wf-field-descs* (set (*field-descs* t))
 shows *field-update* (*component-desc* t)
 (*field-access* (*component-desc* t) s bs) s = s
proof (cases t)
 case (*TypDesc* *algn* *st* *n*)
 show ?thesis
proof (cases *st*)
 case (*TypScalar* *sz* *align* *d*)
 from *TypDesc TypScalar* have *d* ∈ set (*field-descs* t) by simp
 with *wf* have *wf-field-desc* *d* by (simp add: *wf-field-descs-def*)
 then interpret *wf-d: wf-field-desc* *d* .
 from *TypDesc TypScalar* *wf-d.update-access* show ?thesis
 by simp
 next
 case (*TypAggregate* *fs*)
 from *wf* have *wf-field-descs* (set (*field-descs-list* *fs*)) by (simp add: *TypDesc*
TypAggregate)
 then
 have *fst* (*apply-field-updates* (map *dt-trd* *fs*)
 (concat (*split-map* (*component-access-tuple* s) *fs* bs)) s) = s
proof (induct *fs* arbitrary: s bs)
 case Nil
 then show ?case by simp
 next
 case (*Cons* *f* *fs'*)
 obtain *ft* *fn* *d* where *f*: *f* = *DTuple* *ft* *fn* *d* by (cases *f*) simp
 from *Cons* *f* obtain
wf-d: wf-field-desc *d* and

```

wf-ft: wf-field-descs (set (field-descs ft)) and
wf-fs': wf-field-descs (set (field-descs-list fs')) by auto

from wf-d interpret wf-d: wf-field-desc d .
from Cons.hyps [OF wf-fs']
have fst (apply-field-updates (map dt-trd fs')
      (concat (split-map (component-access-tuple s) fs' (drop (field-sz d) bs))))
s) = s .
then
show ?case
by (simp add: Cons f wf-d.access-result-size wf-d.update-access-append)
qed
with TypDesc TypAggregate show ?thesis by simp
qed
qed

lemma toplevel-field-descs-tuple-dt-trd: toplevel-field-descs-tuple t = [dt-trd t]
by (cases t) simp
lemma toplevel-field-descs-list-map: toplevel-field-descs-list fs = map dt-trd fs
by (induct fs) (auto simp add: toplevel-field-descs-tuple-dt-trd)

lemma component-double-update:
assumes wf:wf-field-descs (set (field-descs t))
assumes ind: field-descs-independent (toplevel-field-descs t)
shows field-update (component-desc t) bs (field-update (component-desc t) bs' s)
=
field-update (component-desc t) bs s
proof (cases t)
case (TypDesc algn st n)
show ?thesis
proof (cases st)
case (TypScalar sz align d)
from TypDesc TypScalar have d ∈ set (field-descs t) by simp
with wf have wf-field-desc d by (simp add: wf-field-descs-def)
then interpret wf-d: wf-field-desc d .
from TypDesc TypScalar wf-d.double-update show ?thesis
by simp
next
case (TypAggregate fs)
from wf have wf: wf-field-descs (set (field-descs-list fs)) by (simp add: TypDesc
TypAggregate)
from ind have ind: field-descs-independent (toplevel-field-descs-list fs) by (simp
add: TypDesc TypAggregate)
from wf ind
have fst (apply-field-updates (map dt-trd fs) bs
      (fst (apply-field-updates (map dt-trd fs) bs' s))) =
fst (apply-field-updates (map dt-trd fs) bs s)
proof (induct fs arbitrary: s bs bs')

```

```

case Nil
then show ?case by simp
next
case (Cons f fs')
obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp

from Cons.prems f obtain
  wf-d: wf-field-desc d and
  wf-ft: wf-field-descs (set (field-descs ft)) and
  wf-fs': wf-field-descs (set (field-descs-list fs')) and
  ind-d: field-desc-independent (field-access d) (field-update d)
    (set (toplevel-field-descs-list fs')) and
  ind-fs': field-descs-independent (toplevel-field-descs-list fs')
by auto

from wf-d interpret wf-d: wf-field-desc d .
from field-update-apply-field-updates-commute [OF ind-d ind-fs']
have d-ds'-commute: field-update d bs
  (fst (apply-field-updates (map dt-trd fs') (drop (field-sz d) bs')
    (field-update d bs' s))) =
  fst (apply-field-updates (map dt-trd fs') (drop (field-sz d) bs') (field-update
d bs s))
by (simp add: toplevel-field-descs-list-map wf-d.double-update)

note hyp = Cons.hyps [OF wf-fs' ind-fs', where bs=(drop (field-sz d) bs)]
and s=(field-update d bs s)
show ?case
by (simp add: Cons f wf-d.access-result-size d-ds'-commute hyp)
qed
then show ?thesis by (simp add: TypDesc TypAggregate)
qed
qed

lemma component-access-update:
  assumes wf:wf-field-descs (set (field-descs t))
  assumes ind: field-descs-independent (toplevel-field-descs t)
  shows field-access (component-desc t) (field-update (component-desc t) bs s) bs'
  =
  field-access (component-desc t) (field-update (component-desc t) bs s') bs'
proof (cases t)
case (TypDesc algn st n)
show ?thesis
proof (cases st)
case (TypScalar sz align d)
from TypDesc TypScalar have d ∈ set (field-descs t) by simp
with wf have wf-field-desc d by (simp add: wf-field-descs-def)

```

```

then interpret wf-d: wf-field-desc d .
from TypDesc TypScalar wf-d.access-update show ?thesis
  by simp
next
  case (TypAggregate fs)
  from wf have wf: wf-field-descs (set (field-descs-list fs)) by (simp add: TypDesc
TypAggregate)
  from ind have ind: field-descs-independent (toplevel-field-descs-list fs) by (simp
add: TypDesc TypAggregate)
  from wf ind
  have concat (split-map
    (component-access-tuple (fst (apply-field-updates (map dt-trd fs) bs s))) fs
bs^ ) =
    concat (split-map
    (component-access-tuple (fst (apply-field-updates (map dt-trd fs) bs s'))) fs
bs^ )
  proof (induct fs arbitrary: bs s bs' s')
  case Nil
  then show ?case by simp
next
  case (Cons f fs')
  obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp

from Cons.premis f obtain
  wf-d: wf-field-desc d and
  wf-ft: wf-field-descs (set (field-descs ft)) and
  wf-fs': wf-field-descs (set (field-descs-list fs')) and
  ind-d: field-desc-independent (field-access d) (field-update d)
    (set (toplevel-field-descs-list fs')) and
  ind-fs': field-descs-independent (toplevel-field-descs-list fs')
  by auto

from wf-d interpret wf-d: wf-field-desc d .

note hyp = Cons.hyps [OF wf-fs' ind-fs']
from field-update-apply-field-updates-commute [OF ind-d ind-fs']
have eq1: fst ((apply-field-updates (map dt-trd fs') (drop (field-sz d) bs))
  (field-update d bs s)) =
  field-update d bs
  (fst (apply-field-updates (map dt-trd fs') (drop (field-sz d) bs) s))
  (is - = ?rhs)
  by (simp add: topLevel-field-descs-list-map)
have eq2: field-access d ?rhs bs' = field-access d (field-update d bs s) bs'
  by (simp add: wf-d.access-update)
from field-update-apply-field-updates-commute [OF ind-d ind-fs']
have eq3: fst (apply-field-updates (map dt-trd fs') (drop (field-sz d) bs)
  (field-update d bs s')) =
  field-update d bs
  (fst (apply-field-updates (map dt-trd fs') (drop (field-sz d) bs) s'))

```

```

      (is - = ?rhs)
    by (simp add: toplevel-field-descs-list-map)
  have eq4: field-access d ?rhs bs' = field-access d (field-update d bs s) bs'
    by (simp add: wf-d.access-update)
  show ?case
    apply (simp add: f eq1 eq2 eq3 eq4 wf-d.access-result-size)
    apply (simp only: eq1 [symmetric] )
    apply (simp only: eq3 [symmetric] )
    apply (simp add: hyp)
  done
qed

  then show ?thesis
    by (simp add: TypDesc TypAggregate)
  qed
qed

lemma sum-nat-foldl: (n::nat) + foldl (+) m xs = foldl (+) (n + m) xs
  by (induct xs arbitrary: n m) (auto simp add: semiring-normalization-rules(25))

lemma sum-nat-foldl-le: (n::nat) ≤ foldl (+) n xs
  by (induct xs arbitrary: n) (auto elim: add-leE)

lemma sum-add-sub-foldl: foldl (+) (m + n) ns - n = foldl (+) (m::nat) ns
  apply (induct ns arbitrary: m n)
  apply simp
  apply (metis diff-add-inverse2 semiring-normalization-rules(24) sum-nat-foldl)
  done

lemma sum-sub-zero-foldl: foldl (+) n ns - (n::nat) = foldl (+) 0 ns
  using sum-add-sub-foldl [where m=0, simplified]
  by simp

lemma drop-take-sub: drop n (take (foldl (+) n ns) bs) = take (foldl (+) 0 ns)
(drop n bs)
  by (simp add: drop-take sum-sub-zero-foldl)

lemma component-access-size:
  assumes wf: wf-field-descs (set (field-descs t))
  shows field-access (component-desc t) s (take (field-sz (component-desc t)) bs)
=
  field-access (component-desc t) s bs
proof (cases t)
  case (TypDesc algn st n)
  show ?thesis
  proof (cases st)
    case (TypScalar sz align d)
    from TypDesc TypScalar have d ∈ set (field-descs t) by simp

```

```

with wf have wf-field-desc d by (simp add: wf-field-descs-def)
then interpret wf-d: wf-field-desc d .
from TypDesc TypScalar wf-d.access-size show ?thesis
  by simp
next
  case (TypAggregate fs)
from wf have wf: wf-field-descs (set (field-descs-list fs)) by (simp add: TypDesc
TypAggregate)
from wf
have concat (split-map (component-access-tuple s) fs
  (take (foldl (+) 0 (map (field-sz ∘ dt-trd) fs)) bs)) =
  concat (split-map (component-access-tuple s) fs bs)
proof (induct fs arbitrary: s bs)
  case Nil
  then show ?case by simp
next
  case (Cons f fs')
obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp

from Cons.premis f obtain
  wf-d: wf-field-desc d and
  wf-ft: wf-field-descs (set (field-descs ft)) and
  wf-fs': wf-field-descs (set (field-descs-list fs'))
  by auto

from wf-d interpret wf-d: wf-field-desc d .

have field-sz d ≤ foldl (+) (field-sz d) (map (λf'. field-sz (dt-trd f')) fs') (is
?n ≤ ?m)
  by (rule sum-nat-foldl-le)
then
have eq1: take ?n (take ?m bs) = take ?n bs
  by (simp add: min-def)

from wf-d.access-size [where bs=take ?m bs]
have field-access d s (take ?m bs) = field-access d s (take ?n (take ?m bs))
  by (simp)
also have ... = field-access d s (take ?n bs) by (simp only: eq1)
also have ... = field-access d s bs by (simp add: wf-d.access-size)
finally
have eq-head: field-access d s (take (foldl (+) (field-sz d) (map (λf'. field-sz
(dt-trd f')) fs')) bs) =
  field-access d s bs .

from Cons.hyps [OF wf-fs']
show ?case
  by (simp add: f eq-head wf-d.access-result-size drop-take-sub)
qed

```

```

    then show ?thesis
      by (simp add: TypDesc TypAggregate)
  qed
qed

lemma component-access-result-size:
  assumes wf: wf-field-descs (set (field-descs t))
  shows length (field-access (component-desc t) s bs) = field-sz (component-desc
t)
proof (cases t)
  case (TypDesc algn st n)
  show ?thesis
  proof (cases st)
    case (TypScalar sz align d)
    from TypDesc TypScalar have d ∈ set (field-descs t) by simp
    with wf have wf-field-desc d by (simp add: wf-field-descs-def)
    then interpret wf-d: wf-field-desc d .
    from TypDesc TypScalar wf-d.access-result-size show ?thesis
      by simp
  next
  case (TypAggregate fs)
  from wf have wf: wf-field-descs (set (field-descs-list fs)) by (simp add: TypDesc
TypAggregate)
  from wf
  have length (concat (split-map (component-access-tuple s) fs bs)) =
    foldl (+) 0 (map (field-sz ∘ dt-trd) fs)
  proof (induct fs arbitrary: s bs)
    case Nil
    then show ?case by simp
  next
  case (Cons f fs')
  obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp

  from Cons.prem1 f obtain
    wf-d: wf-field-desc d and
    wf-ft: wf-field-descs (set (field-descs ft)) and
    wf-fs': wf-field-descs (set (field-descs-list fs'))
  by auto

  from wf-d interpret wf-d: wf-field-desc d .

  note hyp = Cons.hyps [OF wf-fs']
  show ?case
    by (simp add: hyp wf-d.access-result-size f sum-nat-foldl)
  qed
  then show ?thesis by (simp add: TypDesc TypAggregate)
qed
qed

```

```

lemma component-update-size:
  assumes wf: wf-field-descs (set (field-descs t))
  shows field-update (component-desc t) (take (field-sz (component-desc t)) bs) s
  =
    field-update (component-desc t) bs s
proof (cases t)
  case (TypDesc algn st n)
  show ?thesis
  proof (cases st)
    case (TypScalar sz align d)
    from TypDesc TypScalar have d ∈ set (field-descs t) by simp
    with wf have wf-field-desc d by (simp add: wf-field-descs-def)
    then interpret wf-d: wf-field-desc d .
    from TypDesc TypScalar wf-d.update-size show ?thesis
    by simp
  next
    case (TypAggregate fs)
    from wf have wf: wf-field-descs (set (field-descs-list fs)) by (simp add: TypDesc
TypAggregate)
    then have fst (apply-field-updates (map dt-trd fs)
      (take (foldl (+) 0 (map (field-sz ∘ dt-trd) fs)) bs) s) =
      fst (apply-field-updates (map dt-trd fs) bs s)
    proof (induct fs arbitrary: s bs)
      case Nil
      then show ?case by simp
    next
      case (Cons f fs')
      obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp

      from Cons.prems f obtain
        wf-d: wf-field-desc d and
        wf-ft: wf-field-descs (set (field-descs ft)) and
        wf-fs': wf-field-descs (set (field-descs-list fs'))
        by auto

      from wf-d interpret wf-d: wf-field-desc d .

      have field-sz d ≤ foldl (+) (field-sz d) (map (λa. field-sz (dt-trd a)) fs') (is
      ?n ≤ ?m)
      by (rule sum-nat-foldl-le)
      then
      have eq1: take ?n (take ?m bs) = take ?n bs
      by (simp add: min-def)

      from wf-d.update-size [where bs=take ?m bs]
      have field-update d ((take ?m) bs) s = field-update d (take ?n (take ?m bs))
      by simp
      also have ... = field-update d (take ?n bs) s by (simp only: eq1)
      also from wf-d.update-size [where bs=bs]

```



```

have ... = field-update d bs s by simp
finally have eq: field-update d ((take ?m) bs) s = field-update d bs s .

note hyp = Cons.hyps [OF wf-fs', simplified comp-def]

show ?case
  apply (simp add: f )
  apply (simp only: eq)
  apply (simp only: drop-take-sub)
  apply (simp only: hyp)
  done
qed
then show ?thesis by (simp add: TypDesc TypAggregate)
qed
qed

lemma apply-field-updates-access-cancel:
  assumes wf: wf-field-descs (set (field-descs-list fs))
  assumes ind: field-descs-independent (toplevel-field-descs-list fs)
  shows fst (apply-field-updates (map dt-trd fs) (concat (split-map (component-access-tuple
v) fs bs)) v) = v
  using wf ind
proof (induct fs arbitrary: v bs)
  case Nil
  then show ?case by simp
next
  case (Cons f fs')
  obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp
  from Cons.prem1 obtain
    wf-d: wf-field-desc d and
    wf-ft: wf-field-descs (set (field-descs ft)) and
    wf-fs': wf-field-descs (set (field-descs-list fs')) and
    ind-d: field-desc-independent (field-access d) (field-update d)
      (set (toplevel-field-descs-list fs')) and
    ind-fs': field-descs-independent (toplevel-field-descs-list fs')
  by auto

from wf-d interpret wf-d: wf-field-desc d .

from wf-d.update-size [of (field-access d v bs @
  concat (split-map (component-access-tuple v) fs' (drop (length (field-access d v
bs)) bs)))]
have eq1: field-update d
  (field-access d v bs @
  concat
  (split-map (component-access-tuple v) fs'
  (drop (length (field-access d v bs)) bs)))

```

$v = \text{field-update } d \ (\text{field-access } d \ v \ bs) \ v$

by (*simp add: wf-d.access-result-size*)

note $\text{hyp} = \text{Cons.hyps} \ [OF \ \text{wf-fs}' \ \text{ind-fs}']$
show *?case* **apply** (*simp add: Cons.premis f*)
apply (*simp add: eq1*)
apply (*simp add: wf-d.access-result-size wf-d.update-access hyp*)
done

qed

lemma *apply-field-updates-double*:
assumes $\text{wf}: \text{wf-field-descs} \ (\text{set} \ (\text{field-descs-list} \ fs))$
assumes $\text{ind}: \text{field-descs-independent} \ (\text{toplevel-field-descs-list} \ fs)$
shows $\text{fst} \ (\text{apply-field-updates} \ (\text{map} \ \text{dt-trd} \ fs) \ bs$
 $\quad (\text{fst} \ (\text{apply-field-updates} \ (\text{map} \ \text{dt-trd} \ fs) \ bs' \ v))) =$
 $\quad \text{fst} \ (\text{apply-field-updates} \ (\text{map} \ \text{dt-trd} \ fs) \ bs \ v)$
using wf ind

proof (*induct fs arbitrary: v bs bs'*)
case *Nil*
then show *?case* **by** *simp*

next
case (*Cons f fs'*)
obtain $\text{ft fn } d$ **where** $f = \text{DTuple } \text{ft } \text{fn } d$ **by** (*cases f*) *simp*
from *Cons.premis f* **obtain**
 $\text{wf-d}: \text{wf-field-desc } d$ **and**
 $\text{wf-ft}: \text{wf-field-descs} \ (\text{set} \ (\text{field-descs } \text{ft}))$ **and**
 $\text{wf-fs}': \text{wf-field-descs} \ (\text{set} \ (\text{field-descs-list } \text{fs}'))$ **and**
 $\text{ind-d}: \text{field-desc-independent} \ (\text{field-access } d) \ (\text{field-update } d)$
 $\quad (\text{set} \ (\text{toplevel-field-descs-list } \text{fs}'))$ **and**
 $\text{ind-fs}': \text{field-descs-independent} \ (\text{toplevel-field-descs-list } \text{fs}')$
by *auto*

note $\text{commute} = \text{field-update-apply-field-updates-commute} \ [OF \ \text{ind-d } \text{ind-fs}', \ \text{simp-}$
 $\text{plified } \text{toplevel-field-descs-list-map}]$
note $\text{hyp} = \text{Cons.hyps} \ [OF \ \text{wf-fs}' \ \text{ind-fs}']$
from wf-d **interpret** $\text{wf-d}: \text{wf-field-desc } d$.
show *?case* **apply** (*simp add: Cons.premis f*)
by (*simp add: commute hyp wf-d.double-update*)

qed

lemma *list-append-eq-split-conv*: $\text{length } xs1 = \text{length } xs2 \implies xs1 \ @ \ ys1 = xs2 \ @ \ ys2 \iff xs1 = xs2 \wedge ys1 = ys2$
by *auto*

lemma *component-complement-padding*:
assumes $\text{wf}: \text{wf-field-descs} \ (\text{set} \ (\text{field-descs } t))$
assumes $\text{ind}: \text{field-descs-independent} \ (\text{toplevel-field-descs } t)$
assumes $i\text{-bound}: i < \text{field-sz} \ (\text{component-desc } t)$

```

shows padding-base.is-padding-byte (field-access (component-desc t))
      (field-update (component-desc t)) (field-sz (component-desc t)) i ≠
padding-base.is-value-byte (field-access (component-desc t))
      (field-update (component-desc t)) (field-sz (component-desc t)) i
proof (cases t)
  case (TypDesc algn st n)
  show ?thesis
  proof (cases st)
    case (TypScalar sz align d)
    from TypDesc TypScalar have d ∈ set (field-descs t) by simp
    with wf have wf-field-desc d by (simp add: wf-field-descs-def)
    then interpret wf-d: wf-field-desc d .
    from TypDesc TypScalar wf-d.complement show ?thesis
    using i-bound
    by auto
  next
    case (TypAggregate fs)
    from wf have wf: wf-field-descs (set (field-descs-list fs)) by (simp add: TypDesc
TypAggregate)
    from ind have ind: field-descs-independent (toplevel-field-descs-list fs) by (simp
add: TypDesc TypAggregate)
    from i-bound have i-bound': i < foldl (+) 0 (map (field-sz ∘ dt-trd) fs) by
(simp add: TypDesc TypAggregate)
    from wf ind i-bound'
    have padding-base.is-padding-byte (component-access-struct (TypAggregate fs))
      (component-update-struct (TypAggregate fs))
      (foldl (+) 0 (map (field-sz ∘ dt-trd) fs)) i =
      (¬ padding-base.is-value-byte (component-access-struct (TypAggregate fs))
      (component-update-struct (TypAggregate fs))
      (foldl (+) 0 (map (field-sz ∘ dt-trd) fs)) i)
  proof (induct fs arbitrary: i)
    case Nil
  then show ?case by (simp add: padding-base.is-padding-byte-def padding-base.is-value-byte-def)
  next
    case (Cons f fs')
    obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp
    from Cons.premis f obtain
      wf-d: wf-field-desc d and
      wf-ft: wf-field-descs (set (field-descs ft)) and
      wf-fs': wf-field-descs (set (field-descs-list fs')) and
      ind-d: field-desc-independent (field-access d) (field-update d)
        (set (toplevel-field-descs-list fs')) and
      ind-fs': field-descs-independent (toplevel-field-descs-list fs') and
      i-bound: i < foldl (+) (field-sz d) (map (λa. field-sz (dt-trd a)) fs')
    by (auto)

  have split-foldl:
    (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) =
      field-sz d + (foldl (+) 0 (map (field-sz ∘ dt-trd) fs'))

```

```

    by (auto simp add: f sum-nat-foldl)

  from wf-d interpret wf-d: wf-field-desc d .

  note commute = field-update-apply-field-updates-commute [OF ind-d ind-fs',
symmetric, simplified toplevel-field-descs-list-map]
  note hyp = Cons.hyps [OF wf-fs' ind-fs']
  show ?case
  proof (cases i < field-sz d)
    case True
    note i-in-d = this
    show ?thesis
    proof (cases wf-d.is-padding-byte i)
      case True
      note is-padding-d = this
      with True wf-d.complement i-in-d have not-value-d: ¬ wf-d.is-value-byte
i by simp
      have padding-base.is-padding-byte
        (component-access-struct (TypAggregate (f # fs')))
        (component-update-struct (TypAggregate (f # fs')))
        (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i
      proof (rule padding-base.is-padding-byteI)
        show i < foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
          using i-bound
          by (simp add: f comp-def)
      next
      fix v::'a and bs::byte list
      assume i < foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
      assume length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
      hence lbs: length bs = foldl (+) (field-sz d) (map (λx. field-sz (dt-trd x))
fs')
        by (simp add: f comp-def)
      hence lbs': field-sz d ≤ length bs
        using sum-nat-foldl-le by auto
      hence ltake: length (take (field-sz d) bs) = field-sz d by simp
      from wf-d.access-size
      have acc-d-take: field-access d v bs = field-access d v (take (field-sz d)
bs) by simp
      from wf-d.access-result-size have acc-d-sz: length (field-access d v bs) =
field-sz d .
      have field-access d v bs ! i = bs ! i
        using i-in-d
      by (simp add: acc-d-take wf-d.access-result-size wf-d.is-padding-byte-acc-id
[OF is-padding-d ltake])
      then
      show component-access-struct (TypAggregate (f # fs')) v bs ! i = bs ! i
        using i-bound lbs i-in-d
        by (simp add: Cons f nth-append acc-d-sz)
      next

```

```

fix v::'a and bs::byte list and b::byte
assume i < foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs^))
assume length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs^))
hence lbs: length bs = foldl (+) (field-sz d) (map (λx. field-sz (dt-trd x))
fs^)
  by (simp add: f comp-def)
hence lbs': field-sz d ≤ length bs
  using sum-nat-foldl-le by auto
with i-in-d have drop-eq: (drop (field-sz d) (bs[i := b])) = drop (field-sz
d) bs
  by (meson drop-update-cancel)
from lbs' have ltake: length (take (field-sz d) bs) = field-sz d by simp
have (field-update d (bs[i := b]) v) = field-update d bs v
  apply (subst (1 2) wf-d.update-size [symmetric])
  using wf-d.is-padding-byte-upd-eq [OF is-padding-d ltake] ltake
  by (metis take-update-swap)
then
show component-update-struct (TypAggregate (f # fs^)) bs v =
  component-update-struct (TypAggregate (f # fs^)) (bs[i := b]) v
  by (simp add: Cons f drop-eq)
qed
moreover
{
  assume is-value: padding-base.is-value-byte
    (component-access-struct (TypAggregate (f # fs^)))
    (component-update-struct (TypAggregate (f # fs^)))
    (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs^))) i
  have wf-d.is-value-byte i
  proof (rule wf-d.is-value-byteI)
    show i < field-sz d by (rule i-in-d)
  next
  fix v::'a and bs::byte list and bs'::byte list
  assume lbs: length bs = field-sz d
  assume lbs': length bs' = field-sz d
  obtain xbs where
    lxs:length xbs = (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs^))) and
    bs: bs = take (field-sz d) xbs and
    split-xbs: xbs = take (field-sz d) xbs @ drop (field-sz d) xbs
  using lbs append-eq-conv-conj
  unfolding split-foldl
  by (metis (no-types, opaque-lifting) Ex-list-of-length length-append)
  obtain xbs' where
    lxs':length xbs' = (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs^))) and
    bs': bs' = take (field-sz d) xbs' and
    split-xbs': xbs' = take (field-sz d) xbs' @ drop (field-sz d) xbs'
  using lbs' append-eq-conv-conj
  unfolding split-foldl
  by (metis (no-types, opaque-lifting) Ex-list-of-length length-append)
  have xbs-i: xbs ! i = bs ! i

```

```

using bs split-xbs'
by (simp add: bs i-in-d)

have upd-xbs: field-update d xbs v = field-update d bs v
apply (subst split-xbs)
apply (subst wf-d.update-size [symmetric])
apply simp
apply (simp add: bs [symmetric])
done

from commute [where bs'=(drop (field-sz d) xbs) and bs = bs and
s=v]
have acc-upd: field-access d
  (fst (apply-field-updates (map dt-trd fs') (drop (field-sz d) xbs)
  (field-update d bs v)))
  bs' = field-access d (field-update d bs v) bs'
using wf-d.access-update by auto

from padding-base.is-value-byte-acc-upd-cancel [OF is-value lxs lxs']
have component-access-struct (TypAggregate (f # fs'))
  (component-update-struct (TypAggregate (f # fs')) xbs v) xbs' !
  i = xbs ! i .
then
show field-access d (field-update d bs v) bs' ! i = bs ! i
  using lbs' lbs' i-in-d
  apply (simp add: f xbs-i)
  apply (subst (asm) wf-d.access-size [symmetric])
  apply (simp add: bs' [symmetric])
  apply (simp add: wf-d.access-result-size nth-append upd-xbs acc-upd)
done

next
fix v::'a and bs::byte list and b::byte
assume lbs: length bs = field-sz d
obtain xbs where
  lxs:length xbs = (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) and
  bs: bs = take (field-sz d) xbs and
  split-xbs: xbs = take (field-sz d) xbs @ drop (field-sz d) xbs
using lbs append-eq-conv-conj
unfolding split-foldl
by (metis (no-types, opaque-lifting) Ex-list-of-length length-append)
have eq1: field-access d v xbs = field-access d v bs
using bs wf-d.access-size by simp
have eq2: field-access d v (xbs[i := b]) = field-access d v (bs[i := b])
using i-in-d bs
by (metis take-update-swap wf-d.access-size)

from padding-base.is-value-byte-acc-eq [OF is-value lxs]

```

```

have component-access-struct (TypAggregate (f # fs')) v xbs =
  component-access-struct (TypAggregate (f # fs')) v (xbs[i := b]) .
then
show field-access d v bs = field-access d v (bs[i := b])
  apply (simp add: f)
  apply (subst (asm) list-append-eq-split-conv)
  apply (simp add: wf-d.access-result-size)
  apply (clarsimp simp add: eq1 eq2)
done
qed
}
then
have ¬ padding-base.is-value-byte
  (component-access-struct (TypAggregate (f # fs')))
  (component-update-struct (TypAggregate (f # fs')))
  (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i
  using not-value-d
  by blast
ultimately show ?thesis by blast
next
case False
note not-padding = this
with wf-d.complement i-in-d have is-value: wf-d.is-value-byte i by simp
have padding-base.is-value-byte
  (component-access-struct (TypAggregate (f # fs')))
  (component-update-struct (TypAggregate (f # fs')))
  (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i
proof (rule padding-base.is-value-byteI)
  show i < foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
    using i-bound by (simp add: f comp-def)
next
fix v::'a and bs::byte list and bs'::byte list
assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
assume lbs': length bs' = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
from lbs have lbs-take: length (take (field-sz d) bs) = field-sz d
  using lbs split-foldl by (simp add: f comp-def)
from lbs' have lbs'-take: length (take (field-sz d) bs') = field-sz d
  using lbs split-foldl by (simp add: f comp-def)

show component-access-struct (TypAggregate (f # fs'))
  (component-update-struct (TypAggregate (f # fs')) bs v) bs' !
  i = bs ! i
apply (simp add: f)
apply (subst wf-d.access-size [symmetric])
using i-in-d
apply (simp add: wf-d.access-result-size nth-append)
apply (subst commute)
apply (subst wf-d.access-update [where s'=v])
using wf-d.is-value-byte-acc-upd-cancel [OF is-value lbs-take lbs'-take,

```

of v]

```
    apply (simp add: wf-d.update-size)
  done
next
fix v::'a and bs::byte list and b::byte
assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs^))
from lbs have lbs-take: length (take (field-sz d) bs) = field-sz d
  using lbs split-foldl by (simp add: f comp-def)
have acc-eq: field-access d v bs = field-access d v (bs[i:=b])
  apply (subst (1 2) wf-d.access-size [symmetric])
  using wf-d.is-value-byte-acc-eq [OF is-value lbs-take] i-in-d
  by (metis take-update-swap)

show component-access-struct (TypAggregate (f # fs^)) v bs =
  component-access-struct (TypAggregate (f # fs^)) v (bs[i := b])
  apply (simp add: f acc-eq)
  using lbs-take i-in-d
  by (simp add: wf-d.access-result-size)
qed
moreover
{
  assume is-padding: padding-base.is-padding-byte
    (component-access-struct (TypAggregate (f # fs^)))
    (component-update-struct (TypAggregate (f # fs^)))
    (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs^))) i
  have wf-d.is-padding-byte i
  proof (rule wf-d.is-padding-byteI)
    from i-in-d show i < field-sz d .
  next
  fix v::'a and bs::byte list
  assume lbs: length bs = field-sz d
  obtain xbs where
    lbs:length xbs = (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs^))) and
    bs: bs = take (field-sz d) xbs and
    split-xbs: xbs = take (field-sz d) xbs @ drop (field-sz d) xbs
  using lbs append-eq-conv-conj
  unfolding split-foldl
  by (metis (no-types, opaque-lifting) Ex-list-of-length length-append)
  from padding-base.is-padding-byte-acc-eq [OF is-padding lbs, where
v=v]

  show field-access d v bs ! i = bs ! i
  apply (simp add: f)
by (simp add: bs i-in-d nth-append wf-d.access-result-size wf-d.access-size)
next
fix v::'a and bs::byte list and b::byte
assume lbs: length bs = field-sz d
obtain xbs where
  lbs:length xbs = (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs^))) and
```



```

    bs: bs = take (field-sz d) xbs and
    split-xbs: xbs = take (field-sz d) xbs @ drop (field-sz d) xbs
    using lbs append-eq-conv-conj
    unfolding split-foldl
    by (metis (no-types, opaque-lifting) Ex-list-of-length length-append)
    from padding-base.is-padding-byte-upd-eq [OF is-padding lxs, where
v=v]
    show field-update d bs v = field-update d (bs[i := b]) v
    apply (simp add: f)
    by (metis bs commute take-update-swap wf-d.access-update wf-d.double-update
        wf-d.update-access wf-d.update-size-ext)
    qed
  }
  with not-padding
  have ¬ padding-base.is-padding-byte
    (component-access-struct (TypAggregate (f # fs')))
    (component-update-struct (TypAggregate (f # fs')))
    (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i by blast
  ultimately show ?thesis by blast
  qed
next
  case False
  note i-notin-d = this
  with i-bound obtain n j where
    j-bound: j < foldl (+) 0 (map (field-sz ∘ dt-trd) fs') and
    j: j = i - field-sz d and
    i: i = j + field-sz d
  apply (subst (asm) (2) sum-nat-foldl [where m = 0, simplified, symmetric])
  by (fastforce simp add: comp-def)
  note complement-fs'-j = hyp [OF j-bound]
  show ?thesis
  proof (cases padding-base.is-padding-byte (component-access-struct (TypAggregate
fs')))
    (component-update-struct (TypAggregate fs'))
    (foldl (+) 0 (map (field-sz ∘ dt-trd) fs')) j
  case True
  note is-padding-j = this
  with complement-fs'-j have not-is-value-j:
    ¬ padding-base.is-value-byte (component-access-struct (TypAggregate fs'))
    (component-update-struct (TypAggregate fs'))
    (foldl (+) 0 (map (field-sz ∘ dt-trd) fs')) j by blast
  have padding-base.is-padding-byte
    (component-access-struct (TypAggregate (f # fs')))
    (component-update-struct (TypAggregate (f # fs')))
    (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i
  proof (rule padding-base.is-padding-byteI)
    from i-bound show i < foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
by (simp add: f comp-def)
  next

```

```

    fix v::'a and bs::byte list
    assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs^))
    from lbs have lbs-drop: length (drop (field-sz d) bs) = foldl (+) 0 (map
(field-sz ∘ dt-trd) fs^)
      apply (subst (asm) split-foldl)
      apply (simp add: f)
      done
    show component-access-struct (TypAggregate (f # fs^)) v bs ! i = bs ! i
      apply (simp add: f)
      using i-notin-d i
      apply (simp add: nth-append wf-d.access-result-size)
      using padding-base.is-padding-byte-acc-eq [OF is-padding-j lbs-drop, of
v]
      apply simp
      by (metis add.commute lbs le-add-same-cancel1 nth-drop split-foldl
sum-nat-foldl-le)
    next
    fix v::'a and bs::byte list and b::byte
    assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs^))
    from lbs have lbs-drop: length (drop (field-sz d) bs) = foldl (+) 0 (map
(field-sz ∘ dt-trd) fs^)
      apply (subst (asm) split-foldl)
      apply (simp add: f)
      done
    from i-notin-d
    have tk-eq: take (field-sz d) (bs[i := b]) = take (field-sz d) bs
      by (meson not-less take-update-cancel)
    have upd-eq: field-update d bs v = field-update d (bs[i := b]) v
      apply (subst (1 2) wf-d.update-size [symmetric])
      apply (simp add: tk-eq)
      done
    show component-update-struct (TypAggregate (f # fs')) bs v =
      component-update-struct (TypAggregate (f # fs')) (bs[i := b]) v
      apply (simp add: f)
      apply (simp add: upd-eq [symmetric])
      using padding-base.is-padding-byte-upd-eq [OF is-padding-j lbs-drop,
where v = (field-update d bs v) and b = b]
      apply (simp)
      by (metis drop-update-swap i j le-add2)
    qed
  moreover
  {
    assume is-value: padding-base.is-value-byte
      (component-access-struct (TypAggregate (f # fs^)))
      (component-update-struct (TypAggregate (f # fs^)))
      (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs^))) i
    have padding-base.is-value-byte (component-access-struct (TypAggregate
fs'))
      (component-update-struct (TypAggregate fs'))

```

```

      (foldl (+) 0 (map (field-sz ∘ dt-trd) fs^)) j
proof (rule padding-base.is-value-byteI)
  from j-bound show j < foldl (+) 0 (map (field-sz ∘ dt-trd) fs^) .
next
  fix v::'a and bs::byte list and bs'::byte list
  assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) fs^)
  assume lbs': length bs' = foldl (+) 0 (map (field-sz ∘ dt-trd) fs^)
  obtain d-bs v-bs where v-bs: v-bs = field-access d v d-bs and
    l-v-bs: length v-bs = field-sz d
  using wf-d.access-result-size by auto
  obtain xbs where
    xbs-def: xbs = v-bs @ bs and
    xbs: xbs = take (field-sz d) xbs @ bs and
    lxs: length xbs = field-sz d + length bs
  by (simp add: l-v-bs)
  obtain xbs' where
    xbs': xbs' = take (field-sz d) xbs' @ bs' and
    lxs': length xbs' = field-sz d + length bs'
  by (metis Ex-list-of-length append-eq-conv-conj length-append)
  have xbs-i: xbs ! i = bs ! j
    by (metis add.commute append-take-drop-id i le-add2 lxs nth-drop
same-append-eq xbs)
  have drop-xbs': drop (field-sz d) xbs' = bs'
    by (metis append-take-drop-id same-append-eq xbs')
  have drop-xbs: drop (field-sz d) xbs = bs
    by (metis append-take-drop-id same-append-eq xbs)
  have upd-xbs: field-update d xbs v = v
    by (simp add: v-bs wf-d.update-access-append xbs-def)
  show component-access-struct (TypAggregate fs^)
    (component-update-struct (TypAggregate fs') bs v) bs' !
    j = bs ! j
  apply (simp)
  using padding-base.is-value-byte-acc-upd-cancel
    [OF is-value, where bs=xbs and bs'=xbs', simplified split-foldl,
    OF lxs [simplified lbs] lxs' [simplified lbs'], where v = v]
  apply (simp add: xbs-i f nth-append wf-d.access-result-size i-notin-d
drop-xbs' drop-xbs
j [symmetric] upd-xbs)
  done
next
  fix v::'a and bs::byte list and b::byte
  assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) fs^)
  obtain d-bs v-bs where v-bs: v-bs = field-access d v d-bs and
    l-v-bs: length v-bs = field-sz d
  using wf-d.access-result-size by auto
  obtain xbs where
    xbs-def: xbs = v-bs @ bs and
    xbs: xbs = take (field-sz d) xbs @ bs and
    lxs: length xbs = field-sz d + length bs

```

```

    by (simp add: l-v-bs)
  have xbs-i: xbs[i := b] = bs [j := b]

  by (metis calculation component-sz-struct-aggregate is-value l-v-bs lbs
len-gt-0
      length-append less-irrefl padding-base.is-padding-byte-def
padding-base.is-value-byte-upd-neq
      split-foldl word-power-less-1 xbs-def)
  have drop-xbs: drop (length (field-access d v xbs)) xbs = bs
  by (simp add: l-v-bs wf-d.access-result-size xbs-def)

  show component-access-struct (TypAggregate fs') v bs =
    component-access-struct (TypAggregate fs') v (bs[j := b])
  apply (simp)
  using padding-base.is-value-byte-acc-eq [OF is-value, where bs=xbs,
simplified split-foldl,
      OF lxs [simplified lbs], where v=v and b=b]
  apply (simp add: f)
  apply (subst (asm) list-append-eq-split-conv)
  apply (simp add: wf-d.access-result-size)
  apply (clarsimp simp add: drop-xbs)
  apply (simp add: xbs-i)
  by (metis (no-types, opaque-lifting) append-Nil append-eq-append-conv
append-eq-conv-conj
      l-v-bs length-list-update wf-d.access-result-size xbs-def xbs-i)
  qed
}
with not-is-value-j
have ¬ padding-base.is-value-byte
  (component-access-struct (TypAggregate (f # fs')))
  (component-update-struct (TypAggregate (f # fs')))
  (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i by blast

ultimately show ?thesis by blast
next
case False
note not-is-padding-j = this
with complement-fs'-j have is-value-j:
padding-base.is-value-byte (component-access-struct (TypAggregate fs'))
  (component-update-struct (TypAggregate fs'))
  (foldl (+) 0 (map (field-sz ∘ dt-trd) fs')) j by blast
have padding-base.is-value-byte
  (component-access-struct (TypAggregate (f # fs')))
  (component-update-struct (TypAggregate (f # fs')))
  (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i
proof (rule padding-base.is-value-byteI)
  from i-bound show i < foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
by (simp add: f comp-def)
next

```

```

fix v::'a and bs::byte list and bs'::byte list
assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs^))
assume lbs': length bs' = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'^))
from lbs have lbs-drop: length (drop (field-sz d) bs) = foldl (+) 0 (map
(field-sz ∘ dt-trd) fs^)
  apply (subst (asm) split-foldl)
  apply (simp add: f)
  done
from lbs' have lbs'-drop: length (drop (field-sz d) bs') = foldl (+) 0 (map
(field-sz ∘ dt-trd) fs'^)
  apply (subst (asm) split-foldl)
  apply (simp add: f)
  done
have drop-j: drop (field-sz d) bs ! j = bs ! i
  by (metis add.commute i lbs le-add-same-cancel1 nth-drop split-foldl
zero-le)
show component-access-struct (TypAggregate (f # fs^))
  (component-update-struct (TypAggregate (f # fs'^)) bs v) bs' !
  i = bs ! i
  apply (simp add: f nth-append i-notin-d wf-d.access-result-size
j[symmetric])
  using padding-base.is-value-byte-acc-upd-cancel [OF is-value-j lbs-drop
lbs'-drop, where v = field-update d bs v]
  apply (simp add: drop-j)
  done
next
fix v::'a and bs::byte list and b::byte
assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs^))
from lbs have lbs-drop: length (drop (field-sz d) bs) = foldl (+) 0 (map
(field-sz ∘ dt-trd) fs^)
  apply (subst (asm) split-foldl)
  apply (simp add: f)
  done
have acc-eq: field-access d v bs = field-access d v (bs[i := b])
  by (metis i le-add2 take-update-cancel wf-d.access-size)
show component-access-struct (TypAggregate (f # fs^)) v bs =
  component-access-struct (TypAggregate (f # fs'^)) v (bs[i := b])
  apply (simp add: f)
  apply (subst list-append-eq-split-conv)
  apply (simp add: wf-d.access-result-size)
  apply (simp add: acc-eq)
  using padding-base.is-value-byte-acc-eq [OF is-value-j lbs-drop, where
v=v and b=b]
  apply simp
  by (metis drop-update-swap i j le-add2 wf-d.access-result-size)
qed
moreover
{
  assume is-padding: padding-base.is-padding-byte

```

```

      (component-access-struct (TypAggregate (f # fs')))
      (component-update-struct (TypAggregate (f # fs')))
      (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i
have padding-base.is-padding-byte (component-access-struct (TypAggregate
fs'))
      (component-update-struct (TypAggregate fs'))
      (foldl (+) 0 (map (field-sz ∘ dt-trd) fs')) j
proof (rule padding-base.is-padding-byteI)
  from j-bound show j < foldl (+) 0 (map (field-sz ∘ dt-trd) fs') .
next
  fix v::'a and bs::byte list
  assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) fs')
  obtain d-bs v-bs where v-bs: v-bs = field-access d v d-bs and
    l-v-bs: length v-bs = field-sz d
    using wf-d.access-result-size by auto
  obtain xbs where
    xbs-def: xbs = v-bs @ bs and
    xbs: xbs = take (field-sz d) xbs @ bs and
    lbs: length xbs = field-sz d + length bs
    by (simp add: l-v-bs)
  have drop-eq: (drop (field-sz d) xbs) = bs
    by (simp add: l-v-bs xbs-def)
  have xbs-i: xbs ! i = bs ! j
    by (simp add: i-notin-d j l-v-bs nth-append xbs-def)
  show component-access-struct (TypAggregate fs') v bs ! j = bs ! j
    using padding-base.is-padding-byte-acc-eq [OF is-padding, where
bs=xbs, simplified split-foldl,
      OF lbs [simplified lbs], where v = v]
    apply (simp add: f nth-append wf-d.access-result-size i-notin-d j
[symmetric] drop-eq xbs-i)
  done
next
  fix v::'a and bs::byte list and b::byte
  assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) fs')
  obtain d-bs v-bs where v-bs: v-bs = field-access d v d-bs and
    l-v-bs: length v-bs = field-sz d
    using wf-d.access-result-size by auto
  obtain xbs where
    xbs-def: xbs = v-bs @ bs and
    xbs: xbs = take (field-sz d) xbs @ bs and
    lbs: length xbs = field-sz d + length bs
    by (simp add: l-v-bs)
  have drop-eq: (drop (field-sz d) xbs) = bs
    by (simp add: l-v-bs xbs-def)

  have xbs-i: xbs [i := b] = bs [j := b]
    by (metis calculation is-padding l-v-bs lbs len-gt-0 length-append
less-irrefl
      padding-base.is-padding-byte-acc-neq padding-base.is-value-byte-def

```

```

split-foldl word-power-less-1 xbs-def)
  have upd-eq: (field-update d xbs v) = v
    by (simp add: v-bs wf-d.update-access-append xbs-def)
  have eq: fst (apply-field-updates (map dt-trd fs') (bs[j := b]) v) = fst
(apply-field-updates (map dt-trd fs') bs v)
  by (metis calculation is-padding lbs list-update-id lxs padding-base.is-padding-byte-acc-neq
padding-base.is-value-byte-acc-eq split-foldl)
  show component-update-struct (TypAggregate fs') bs v =
component-update-struct (TypAggregate fs') (bs[j := b]) v
  using padding-base.is-padding-byte-upd-eq [OF is-padding, where
bs=xbs, simplified split-foldl,
OF lxs [simplified lbs], where v = v and b = b]
  apply (simp add: f drop-eq xbs-i upd-eq eq)
  done
qed
}
with not-is-padding-j
have ¬ padding-base.is-padding-byte
(component-access-struct (TypAggregate (f # fs')))
(component-update-struct (TypAggregate (f # fs')))
(foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i by blast
ultimately show ?thesis by blast
qed
qed
qed
then show ?thesis
by (simp add: TypDesc TypAggregate)
qed
qed

```

```

theorem wf-field-desc-component-desc:
  assumes wf:wf-field-descs (set (field-descs t))
  assumes ind: field-descs-independent (toplevel-field-descs t)
  shows wf-field-desc (component-desc t)
  apply (unfold-locales)
  apply (rule component-complement-padding [OF wf ind], assumption)
  apply (rule component-double-update [OF wf ind])
  apply (rule component-access-update [OF wf ind])
  apply (rule component-update-access [OF wf])
  apply (rule component-access-size [OF wf])
  apply (rule component-access-result-size [OF wf])
  apply (rule ext, rule component-update-size [OF wf])
  done

```

lemma *field-desc-list-append* [*simp*]: *field-descs-list* (*fs1* @ *fs2*) = *field-descs-list* *fs1* @ *field-descs-list* *fs2*

by (*induct fs1 arbitrary: fs2*) *auto*

lemma *toplevel-field-desc-list-append* [*simp*]:

toplevel-field-descs-list (*fs1* @ *fs2*) = *toplevel-field-descs-list* *fs1* @ *toplevel-field-descs-list* *fs2*

by (*induct fs1 arbitrary: fs2*) *auto*

corollary *wf-field-descs-struct-append-first*:

assumes *wf:wf-field-descs* (*set* (*field-descs-struct* (*TypAggregate* (*fs1* @ *fs2*))))

assumes *ind: field-descs-independent* (*toplevel-field-descs-struct* (*TypAggregate* (*fs1* @ *fs2*)))

shows *wf-field-desc* (*component-desc-struct* (*TypAggregate* *fs1*))

proof –

fix *algn*

from *wf* **have** *wf'*: *wf-field-descs* (*set* (*field-descs* (*TypDesc* *algn* (*TypAggregate* *fs1*) [])))

by *simp*

from *ind*

have *ind'*: *field-descs-independent* (*toplevel-field-descs* (*TypDesc* *algn* (*TypAggregate* *fs1*) []))

by (*simp add: field-descs-independent-append-first*)

from *wf-field-desc-component-desc* [*OF wf' ind'*]

show *?thesis*

by *simp*

qed

corollary *wf-field-descs-struct-append-second*:

assumes *wf:wf-field-descs* (*set* (*field-descs-struct* (*TypAggregate* (*fs1* @ *fs2*))))

assumes *ind: field-descs-independent* (*toplevel-field-descs-struct* (*TypAggregate* (*fs1* @ *fs2*)))

shows *wf-field-desc* (*component-desc-struct* (*TypAggregate* *fs2*))

proof –

fix *algn*

from *wf* **have** *wf'*: *wf-field-descs* (*set* (*field-descs* (*TypDesc* *algn* (*TypAggregate* *fs2*) [])))

by *simp*

from *ind*

have *ind'*: *field-descs-independent* (*toplevel-field-descs* (*TypDesc* *algn* (*TypAggregate* *fs2*) []))

by (*simp add: field-descs-independent-append-second*)

from *wf-field-desc-component-desc* [*OF wf' ind'*]

show *?thesis*

by *simp*

qed

corollary *wf-field-descs-component-desc*:
assumes *wf*:*wf-field-descs* (*set* (*field-descs* *t*))
assumes *ind*: *field-descs-independent* (*toplevel-field-descs* *t*)
shows *wf-field-descs* (*insert* (*component-desc* *t*) (*set* (*field-descs* *t*)))
using *wf-field-desc-component-desc* [*OF wf ind*] *wf*
by *simp*

lemma *size-td-field-sz*:
fixes
 t::'a *xtyp-info* **and**
 st::'a *xtyp-info-struct* **and**
 fs::'a *xtyp-info-tuple list* **and**
 f::'a *xtyp-info-tuple*
shows
wf-component-descs *t* \implies *size-td* *t* = *field-sz* (*component-desc* *t*)
wf-component-descs-struct *st* \implies *size-td-struct* *st* = *field-sz* (*component-desc-struct* *st*)
wf-component-descs-list *fs* \implies *size-td-list* *fs* = *field-sz* (*component-desc-struct* (*TypAggregate* *fs*))
wf-component-descs-tuple *f* \implies *size-td-tuple* *f* = *field-sz* (*dt-trd* *f*)
proof (*induct* *t* **and** *st* **and** *fs* **and** *f*)
 case (*Cons-typ-desc* *dt-tuple list*)
 then show ?*case*
 apply (*cases* *dt-tuple*)
 apply (*clarsimp simp add: sum-nat-foldl*)
 done
qed *auto*

lemma (**in** *c-type*) *simple-type-to-xto-eq*:
fixes *v*::'a
assumes *simple*: \neg *aggregate* (*typ-info-t* *TYPE*('a))
shows *to-bytes* *v* = *xto-bytes* *v*
proof –
 from *simple-type-dest* [*OF simple*]
 show ?*thesis*
 by (*clarsimp simp add: xto-bytes-def to-bytes-def*)
qed

lemma *field-desc-independent-empty* [*simp*]: *field-desc-independent* *acc upd* {}
by (*simp add: field-desc-independent-def*)

lemma *component-descs-field-descs-independent*:
component-descs-independent *t* \implies *field-descs-independent* (*toplevel-field-descs* *t*)
apply (*cases* *t*)

```

subgoal for  $x1\ st\ nm$ 
  apply ( $cases\ st$ )
  apply ( $simp$ )
  subgoal for  $fs$ 
    apply ( $cases\ fs$ )
    apply  $simp$ 
    apply  $simp$ 
  done
done
done

```

lemma *component-descs-list-field-descs-independent:*
component-descs-independent-list fs \implies field-descs-independent (toplevel-field-descs-list fs)

```

  apply ( $cases\ fs$ )
  apply  $simp$ 
  apply  $simp$ 
done

```

theorem *access-ti-component-desc-compatible:*

```

 $\bigwedge bs\ v::'a.\ \llbracket wf\text{-component-descs}\ t;\ component\text{-descs-independent}\ t;$ 
 $\quad wf\text{-field-descs}\ (set\ (field\text{-descs}\ t))\rrbracket$ 
 $\implies access\text{-ti}\ t\ v\ bs = field\text{-access}\ (component\text{-desc}\ t)\ v\ bs$ 
 $\bigwedge bs\ v::'a.\ \llbracket wf\text{-component-descs-struct}\ st;\ component\text{-descs-independent-struct}\ st;$ 
 $\quad wf\text{-field-descs}\ (set\ (field\text{-descs-struct}\ st))\rrbracket$ 
 $\implies access\text{-ti-struct}\ st\ v\ bs = field\text{-access}\ (component\text{-desc-struct}\ st)\ v\ bs$ 
 $\bigwedge bs\ v::'a.\ \llbracket wf\text{-component-descs-list}\ fs;\ component\text{-descs-independent-list}\ fs;$ 
 $\quad wf\text{-field-descs}\ (set\ (field\text{-descs-list}\ fs))\rrbracket$ 
 $\implies access\text{-ti-list}\ fs\ v\ bs = field\text{-access}\ (component\text{-desc-struct}\ (TypAggregate\ fs))$ 
 $v\ bs$ 
 $\bigwedge bs\ v::'a.\ \llbracket wf\text{-component-descs-tuple}\ f;\ component\text{-descs-independent-tuple}\ f;$ 
 $\quad wf\text{-field-descs}\ (set\ (field\text{-descs}\ (dt\text{-fst}\ f)))\rrbracket$ 
 $\implies access\text{-ti-tuple}\ f\ v\ bs = field\text{-access}\ (dt\text{-trd}\ f)\ v\ bs$ 

```

proof (*induct t and st and fs and f*)

```

  case ( $TypDesc\ algn\ st\ n$ )
    then show ?case by simp
  next
    case ( $TypScalar\ sz\ align\ d$ )
      then show ?case by simp
  next
    case ( $TypAggregate\ fs$ )
      then show ?case by simp
  next
    case  $Nil\text{-typ-desc}$ 
      then show ?case by simp
  next
    case ( $Cons\text{-typ-desc}\ f\ fs'$ )
      obtain  $ft\ fn\ d$  where  $f = DTuple\ ft\ fn\ d$  by ( $cases\ f$ ) simp

```

from *Cons-typ-desc.prem*s **obtain**
wf-descs-f: *wf-component-descs-tuple f* **and** *wf-descs-fs'*: *wf-component-descs-list fs'* **and**
ind: *field-descs-independent (toplevel-field-descs-tuple f @ toplevel-field-descs-list fs')* **and**
ind-cf: *component-descs-independent-tuple f* **and**
ind-cfs': *component-descs-independent-list fs'* **and**
wf-field-desc-f: *wf-field-descs (set (field-descs (dt-fst f)))* **and**
wf-d: *wf-field-desc d* **and** *wf-ft*: *wf-field-descs (set (field-descs ft))* **and**
wf-field-descs-fs': *wf-field-descs (set (field-descs-list fs'))*
by (*auto simp add: f*)

note *ind-fs' = field-descs-independent-append-second [OF ind]*

from *wf-d* **interpret** *wf-d*: *wf-field-desc d*.

have *wf-fd-fs'*: *wf-field-descs (insert (component-desc-struct (TypAggregate fs')) (set (field-descs-list fs')))*
proof –
fix *algn*
from *wf-field-descs-fs'*
have *wf:wf-field-descs (set (field-descs (TypDesc algn (TypAggregate fs') []))*
by *simp*
from *ind-fs'*
have *ind*: *field-descs-independent (toplevel-field-descs (TypDesc algn (TypAggregate fs') []))*
by *simp*
from *wf-field-descs-component-desc [OF wf ind]* **show** *?thesis*
by *simp*
qed

from *Cons-typ-desc.hyps(1)* [*OF wf-descs-f ind-cf wf-field-desc-f*]
have *eq-f*: *access-ti-tuple f v (take (size-td ft) bs) = field-access (dt-trd f) v (take (size-td ft) bs)*.
from *Cons-typ-desc.hyps(2)* [*OF wf-descs-fs' ind-cfs' wf-field-descs-fs'*]
have *eq-fs'*: *access-ti-list fs' v (drop (size-td ft) bs) = field-access (component-desc-struct (TypAggregate fs')) v (drop (size-td ft) bs)* .

from *wf-descs-f* **have** *sz-eq*: *field-sz d = size-td ft* **by** (*simp add: f size-td-field-sz*)

from *sz-eq wf-d.access-size*
have *eq-access-take*: *field-access d v (take (size-td ft) bs) = field-access d v bs*
by (*simp*)

```

from wf-descs-f have sz-eq: field-sz d = size-td ft by (simp add: f size-td-field-sz)

from eq-f eq-fs' eq-access-take
  show ?case by (simp add: f wf-d.access-result-size size-td-field-sz sz-eq)
next
case (DTuple-typ-desc ft fn d)
from DTuple-typ-desc.premis obtain
  d: d = component-desc ft and
  wf-ft: wf-component-descs ft and
  ind-cf: component-descs-independent ft and
  wf-descs-ft: wf-field-descs (set (field-descs ft))
  by auto

from wf-field-descs-component-desc [OF wf-descs-ft] ind-cf
have wf-desc-ft: wf-field-desc (component-desc ft)
  by (auto intro: component-descs-field-descs-independent)
from component-descs-field-descs-independent [OF ind-cf]
have field-descs-independent (toplevel-field-descs ft) .

from DTuple-typ-desc.hyps [OF wf-ft ind-cf wf-descs-ft ]

  show ?case by (simp add: d)
qed

```

theorem update-ti-component-desc-compatible:

```

 $\bigwedge bs v::'a. \llbracket wf\text{-component-descs } t; component\text{-descs-independent } t;
  wf\text{-field-descs } (set (field\text{-descs } t)) \rrbracket$ 
 $\implies update\text{-ti } t \text{ bs } v = field\text{-update } (component\text{-desc } t) \text{ bs } v$ 
 $\bigwedge bs v::'a. \llbracket wf\text{-component-descs-struct } st; component\text{-descs-independent-struct } st;
  wf\text{-field-descs } (set (field\text{-descs-struct } st)) \rrbracket$ 
 $\implies update\text{-ti-struct } st \text{ bs } v = field\text{-update } (component\text{-desc-struct } st) \text{ bs } v$ 
 $\bigwedge bs v::'a. \llbracket wf\text{-component-descs-list } fs; component\text{-descs-independent-list } fs;
  wf\text{-field-descs } (set (field\text{-descs-list } fs)) \rrbracket$ 
 $\implies update\text{-ti-list } fs \text{ bs } v = field\text{-update } (component\text{-desc-struct } (TypAggregate$ 
fs))  $\text{ bs } v$ 
 $\bigwedge bs v::'a. \llbracket wf\text{-component-descs-tuple } f; component\text{-descs-independent-tuple } f;
  wf\text{-field-descs } (set (field\text{-descs } (dt\text{-fst } f))) \rrbracket$ 
 $\implies update\text{-ti-tuple } f \text{ bs } v = field\text{-update } (dt\text{-trd } f) \text{ bs } v$ 
proof (induct t and st and fs and f )
  case (TypDesc algn st n)
  then show ?case by simp
next
  case (TypScalar sz align d)
  then show ?case by simp
next
  case (TypAggregate fs)
  then show ?case by simp
next

```

```

case Nil-typ-desc
then show ?case by simp
next
case (Cons-typ-desc f fs')
obtain ft fn d where f = DTuple ft fn d by (cases f) simp

from Cons-typ-desc.prems obtain
  wf-descs-f: wf-component-descs-tuple f and wf-descs-fs': wf-component-descs-list
fs' and
  ind: field-descs-independent (toplevel-field-descs-tuple f @ toplevel-field-descs-list
fs') and
  ind-cf: component-descs-independent-tuple f and
  ind-cfs': component-descs-independent-list fs' and
  ind-d: field-desc-independent (field-access d) (field-update d) (set (toplevel-field-descs-list
fs')) and
  wf-field-desc-f: wf-field-descs (set (field-descs (dt-fst f))) and
  wf-d: wf-field-desc d and wf-ft: wf-field-descs (set (field-descs ft)) and
  wf-field-descs-fs': wf-field-descs (set (field-descs-list fs'))
by (auto simp add: f)

note ind-fs' = field-descs-independent-append-second [OF ind]

from wf-d interpret wf-d: wf-field-desc d.

from Cons-typ-desc.hyps(2) [OF wf-descs-fs' ind-cfs' wf-field-descs-fs']
have eq-fs': update-ti-list fs' (drop (size-td ft) bs) v =
  fst (apply-field-updates (map dt-trd fs') (drop (size-td ft) bs) v) (is ?upd-ti-fs'
= ?upd-fs')
by simp

from Cons-typ-desc.hyps(1) [OF wf-descs-f ind-cf wf-field-desc-f]
have eq-f: update-ti ft (take (size-td ft) bs) ?upd-ti-fs' =
  field-update d (take (size-td ft) bs) ?upd-fs' by (simp add: f eq-fs')

also
from field-update-apply-field-updates-commute [OF ind-d ind-fs', where bs=
(take (size-td ft) bs) and bs'=(drop (size-td ft) bs)]
have ... = fst (apply-field-updates (toplevel-field-descs-list fs') (drop (size-td ft)
bs)
  (field-update d (take (size-td ft) bs) v))
by (simp add: toplevel-field-descs-list-map)
finally have eq-ft: update-ti ft (take (size-td ft) bs) ?upd-ti-fs' =
  fst (apply-field-updates (toplevel-field-descs-list fs') (drop (size-td ft) bs)
  (field-update d (take (size-td ft) bs) v)) .

from size-td-field-sz(1) [of ft] wf-descs-f
have sz-d: field-sz d = size-td ft
by (simp add: f)

```

```

show ?case
  apply (simp add: f )
  apply (simp add: eq-ft toplevel-field-descs-list-map wf-d.update-size [simplified
sz-d] sz-d)
  done
next
  case (DTuple-typ-desc ft fn d)
  then show ?case by simp
qed

```

```

context xmem-type
begin

```

```

lemma wf-field-desc-component-desc: wf-field-desc (component-desc (typ-info-t (TYPE('a))))
  using wf-field-descs component-descs-independent
  by (auto intro: wf-field-desc-component-desc component-descs-field-descs-independent)

```

```

lemma wf-field-descs': wf-field-descs (insert (component-desc (typ-info-t (TYPE('a))))
  (set (field-descs (typ-info-t (TYPE('a')))))
  using wf-field-descs
  by (auto intro: wf-field-desc-component-desc)

```

```

lemma wf-field-desc-t: wf-field-desc (component-desc (typ-info-t (TYPE('a))))
  using wf-field-descs' by (simp)

```

```

sublocale xmem-type-wf-field-desc: wf-field-desc component-desc (typ-info-t (TYPE('a)))
  by (rule wf-field-desc-t)

```

```

lemma xfrom-bytes-xto-bytes-inv: xfrom-bytes (xto-bytes (v::'a) bs) = v
proof –

```

```

  define t where t = (typ-info-t TYPE('a))
  have wf-comp-descs-t: wf-component-descs t by (simp add: wf-component-descs
t-def)
  have ind-t: component-descs-independent t by (simp add: component-descs-independent
t-def)
  from wf-field-descs
  have wf-field-descs-t: wf-field-descs (set (field-descs t))
    by (simp add: t-def)
  have xto: xto-bytes v bs = field-access (component-desc t) v bs (is (- = ?acc))
    by (simp add: xto-bytes-def t-def)

```

```

  from wf-field-desc-component-desc have wf-field-desc (component-desc t) by
(simp add: t-def)
  then interpret wf-t: wf-field-desc component-desc t.
  from wf-t.access-result-size
  have len-acc: length ?acc = field-sz (component-desc t) .

```

```

from size-td-field-sz(1) [OF wf-comp-descs-t]
have size-t: size-td t = field-sz (component-desc t).
have len-acc': size-td t = length ?acc by (simp add: len-acc size-t)
from size-t len-acc
have len-acc'': length ?acc = size-of TYPE('a) by (simp add: size-of-def t-def)
from len-acc'' len-acc'
have len-acc''': size-of TYPE('a) = size-td (typ-info-t TYPE('a)) by (simp add:
t-def)

from update-ti-component-desc-compatible(1) [OF wf-comp-descs-t ind-t wf-field-descs-t
]
have field-update (component-desc t) (field-access (component-desc t) v bs) un-
defined =
  update-ti t (field-access (component-desc t) v bs) undefined by simp
also
from upd [rule-format, OF len-acc'', folded t-def, where v= undefined and w
= v] len-acc' t-def
have ... = update-ti t (field-access (component-desc t) v bs) v by (simp add:
update-ti-t-def)
also
from update-ti-component-desc-compatible(1) [OF wf-comp-descs-t ind-t wf-field-descs-t
]
have ... = field-update (component-desc t) (field-access (component-desc t) v bs)
v .
also
from wf-t.update-access
have ... = v .
finally
have field-update (component-desc t) (field-access (component-desc t) v bs) un-
defined = v .

with len-acc show ?thesis
by (simp add: xto xfrom-bytes-def t-def)
qed

```

lemma xto-bytes-inj:

```

xto-bytes (v::'a) = xto-bytes (v'::'a)  $\implies$  v = v'
apply (drule fun-cong [where x=replicate (size-of TYPE('a)) 0])
apply (drule arg-cong [where f=xfrom-bytes::byte list  $\Rightarrow$  'a])
apply (simp add: xfrom-bytes-xto-bytes-inv)
done

```

lemma field-update-update-ti-t:

```

assumes len: length bs = size-td (typ-info-t (TYPE('a)))
shows field-update (component-desc (typ-info-t (TYPE('a)))) bs = update-ti-t
(typ-info-t (TYPE('a))) bs

```

proof –

```

from len
  update-ti-component-desc-compatible(1) [OF wf-component-descs component-descs-independent

```

$wf\text{-field}\text{-descs}]$
show *?thesis*
by (*simp add: update-ti-t-def fun-eq-iff*)
qed

lemma *field-update-update-ti*:
shows *field-update (component-desc (typ-info-t (TYPE('a)))) = update-ti (typ-info-t (TYPE('a)))*
proof –
from
update-ti-component-desc-compatible(1) [OF wf-component-descs component-descs-independent wf-field-descs]
show *?thesis*
by (*simp add: fun-eq-iff*)
qed

lemma *field-access-access-ti*:
shows *field-access (component-desc (typ-info-t (TYPE('a)))) = access-ti (typ-info-t (TYPE('a)))*
proof –
from
access-ti-component-desc-compatible(1) [OF wf-component-descs component-descs-independent wf-field-descs]
show *?thesis*
by (*simp add: fun-eq-iff*)
qed

lemma *field-sz-size-td*:
field-sz (component-desc (typ-info-t (TYPE('a)))) = size-td (typ-info-t (TYPE('a)))
by (*simp add: local.wf-component-descs size-td-field-sz(1)*)

lemma *field-sz-size-of*:
field-sz (component-desc (typ-info-t (TYPE('a)))) = size-of (TYPE('a))
by (*simp add: size-of-def field-sz-size-td*)

lemma *xto-bytes-size*: *xto-bytes (v::'a) (take (size-td (typ-info-t TYPE('a))) bs) = xto-bytes v bs*
proof –
from *xmem-type-wf-field-desc.access-size size-td-field-sz(1) [OF wf-component-descs]*
show *?thesis*
by (*simp add: xto-bytes-def*)
qed

lemma *xto-bytes-result-size*: *length (xto-bytes (v::'a) bs) = size-td (typ-info-t TYPE('a))*
proof –
from *xmem-type-wf-field-desc.access-result-size size-td-field-sz(1) [OF wf-component-descs]*
show *?thesis*

by (simp add: xto-bytes-def)
qed

lemma *xfrom-bytes-size*: (xfrom-bytes::byte list \Rightarrow 'a) (take (size-td (typ-info-t TYPE('a))) bs) = xfrom-bytes bs

proof –

from xmem-type-wf-field-desc.update-size [where bs=bs] size-td-field-sz(1) [OF wf-component-descs]

show ?thesis

by (simp add: xfrom-bytes-def)

qed

lemma *entire-update*:

shows field-update (component-desc (typ-info-t (TYPE('a)))) bs v = field-update (component-desc (typ-info-t (TYPE('a)))) bs w

by (metis xmem-type-wf-field-desc.double-update xfrom-bytes-def xfrom-bytes-xto-bytes-inv)

lemma *update-access-complete*:

shows field-update (component-desc (typ-info-t (TYPE('a)))) (field-access (component-desc (typ-info-t (TYPE('a)))) v bs) w = v

by (metis entire-update xmem-type-wf-field-desc.update-access)

end

lemma *contained-field-descs-list-all*: contained-field-descs-list xs = list-all contained-field-descs-tuple xs

by (induct xs) auto

lemma *toplevel-field-descs-map*:

toplevel-field-descs (map-td (λ - -. update-desc acc upd) (update-desc acc upd) t)

=

map (update-desc acc upd) (toplevel-field-descs t)

toplevel-field-descs-struct (map-td-struct (λ - -. update-desc acc upd) (update-desc acc upd) st) =

map (update-desc acc upd) (toplevel-field-descs-struct st)

toplevel-field-descs-list (map-td-list (λ - -. update-desc acc upd) (update-desc acc upd) fs) =

map (update-desc acc upd) (toplevel-field-descs-list fs)

toplevel-field-descs-tuple (map-td-tuple (λ - -. update-desc acc upd) (update-desc acc upd) f) =

map (update-desc acc upd) (toplevel-field-descs-tuple f)

by (induct t and st and fs and f) auto

lemma *toplevel-field-descs-adjust-ti*: toplevel-field-descs (adjust-ti t acc upd) = map (update-desc acc upd) (toplevel-field-descs t)

by (*simp add: adjust-ti-def toplevel-field-descs-map*)

theorem *contained-field-descs-empty-typ-info* [*simp*]: *contained-field-descs (empty-typ-info algn n)*
by (*simp add: empty-typ-info-def*)

lemma *fields-contained-update-desc*:
assumes *contained-t: fields-contained (field-access d) (field-update d) (set (toplevel-field-descs t))*
shows *fields-contained (field-access (update-desc acc upd d)) (field-update (update-desc acc upd d)) (update-desc acc upd ' set (toplevel-field-descs t))*
proof (*rule fields-contained.intro*)
fix *d' s s'*
assume *d'-in: d' ∈ update-desc acc upd ' set (toplevel-field-descs t)*
assume *acc-contained: field-access (update-desc acc upd d) s = field-access (update-desc acc upd d) s'*

interpret *ft: fields-contained (field-access d) (field-update d) set (toplevel-field-descs t)*
by (*rule contained-t*)

from *acc-contained* **have** *acc: field-access d (acc s) = field-access d (acc s')*
by (*simp add: update-desc-def*)

from *d'-in* **obtain** *d''* **where**
d''-in: d'' ∈ set (toplevel-field-descs t) **and**
d': d' = update-desc acc upd d''
by *auto*

from *ft.access-contained* [*OF d''-in acc*]
have *field-access d'' (acc s) = field-access d'' (acc s')*.
with *d'*
show *field-access d' s = field-access d' s'*
by (*simp add: update-desc-def*)

next
fix *d' bs s*
assume *d'-in: d' ∈ update-desc acc upd ' set (toplevel-field-descs t)*

interpret *ft: fields-contained (field-access d) (field-update d) set (toplevel-field-descs t)*
by (*rule contained-t*)

from *d'-in* **obtain** *d''* **where**
d''-in: d'' ∈ set (toplevel-field-descs t) **and**
d': d' = update-desc acc upd d''
by *auto*
from *ft.update-contained* [*OF d''-in*]
obtain *bs'* **where** *cont: ∀ s'. field-access d (acc s') = field-access d (acc s) →*

$field\text{-}update\ d''\ bs\ (acc\ s') = field\text{-}update\ d\ bs'\ (acc\ s')$

by *fastforce*

from *cont*

show $\exists bs'. \forall s'. field\text{-}access\ (update\text{-}desc\ acc\ upd\ d)\ s' =$
 $field\text{-}access\ (update\text{-}desc\ acc\ upd\ d)\ s \longrightarrow field\text{-}update\ d'\ bs\ s' =$
 $field\text{-}update\ (update\text{-}desc\ acc\ upd\ d)\ bs'\ s'$

by (*auto simp add: update-desc-def d'*)

qed

lemma *contained-field-descs-update-desc:*

$contained\text{-}field\text{-}descs\ t \implies$
 $contained\text{-}field\text{-}descs\ (map\text{-}td\ (\lambda\ -. update\text{-}desc\ acc\ upd)\ (update\text{-}desc\ acc\ upd)$
 $t)$
 $contained\text{-}field\text{-}descs\text{-}struct\ st \implies$
 $contained\text{-}field\text{-}descs\text{-}struct\ (map\text{-}td\text{-}struct\ (\lambda\ -. update\text{-}desc\ acc\ upd)\ (update\text{-}desc$
 $acc\ upd)\ st)$
 $contained\text{-}field\text{-}descs\text{-}list\ fs \implies$
 $contained\text{-}field\text{-}descs\text{-}list\ (map\text{-}td\text{-}list\ (\lambda\ -. update\text{-}desc\ acc\ upd)\ (update\text{-}desc$
 $acc\ upd)\ fs)$
 $contained\text{-}field\text{-}descs\text{-}tuple\ f \implies$
 $contained\text{-}field\text{-}descs\text{-}tuple\ (map\text{-}td\text{-}tuple\ (\lambda\ -. update\text{-}desc\ acc\ upd)\ (update\text{-}desc$
 $acc\ upd)\ f)$

proof (*induct t and st and fs and f*)

case (*TypDesc st n*)

then show *?case by simp*

next

case (*TypScalar n sz d*)

then show *?case by simp*

next

case (*TypAggregate fs*)

then show *?case by simp*

next

case *Nil-typ-desc*

then show *?case by simp*

next

case (*Cons-typ-desc f fs*)

then show *?case by simp*

next

case (*DTuple-typ-desc ft fn d*)

note $\langle contained\text{-}field\text{-}descs\text{-}tuple\ (DTuple\ ft\ fn\ d) \rangle$

then obtain
 $contained\text{-}ft: contained\text{-}field\text{-}descs\ ft$ **and**
 $fields\text{-}contained\text{-}ft: fields\text{-}contained\ (field\text{-}access\ d)\ (field\text{-}update\ d)\ (set\ (toplevel\text{-}field\text{-}descs$
 $ft))$

by *simp*

from $contained\text{-}ft\ DTuple\text{-}typ\text{-}desc(1)$

```

have contained-map: contained-field-descs
  (map-td (λ- -. update-desc acc upd) (update-desc acc upd) ft)
by simp

from fields-contained-update-desc [OF fields-contained-ft]
have fields-contained (field-access (update-desc acc upd d))
  (field-update (update-desc acc upd d))
  (update-desc acc upd ' set (toplevel-field-descs ft)) .

then have fields-contained (field-access (update-desc acc upd d))
  (field-update (update-desc acc upd d))
  (set (toplevel-field-descs (map-td (λ- -. update-desc acc upd) (update-desc
acc upd) ft)))
  by (simp add: toplevel-field-descs-map)
with DTuple-tyt-desc show ?case by simp
qed

theorem contained-field-descs-adjust-ti:
  fixes t::'a xtyp-info
  assumes contained-t: contained-field-descs t
  shows contained-field-descs (adjust-ti t acc upd)
  using contained-field-descs-update-desc(1) [OF contained-t]
  by (simp add: adjust-ti-def)

theorem contained-field-descs-extend-ti:
  assumes contained-t: contained-field-descs t
  assumes contained-ft: contained-field-descs ft
  assumes fields-contained-ft: fields-contained (field-access d) (field-update d) (set
(toplevel-field-descs ft))
  shows contained-field-descs (extend-ti t ft algn n d)
  using contained-t contained-ft fields-contained-ft
  apply (cases t)
  subgoal for x1 x2 x3
    apply simp
    apply (cases x2)
    apply simp
    apply (simp add: contained-field-descs-list-all)
  done
done

theorem contained-field-descs-ti-pad-combine:
  fixes t::'a xtyp-info
  assumes cont-t: contained-field-descs t
  shows contained-field-descs (ti-pad-combine n t)
proof -
  define d::'a field-desc where
    d ≡ (|field-access = λv bs. if n ≤ length bs then take n bs else replicate n 0,
field-update = λbs. id, field-sz = n)
  define ft::'a xtyp-info where ft ≡ TypDesc 0 (TypScalar n 0 d) "pad-tyt"

```

```

define fn where fn  $\equiv$  (foldl (@) "!pad-" (field-names-list t))
have contained-field-descs (extend-ti t ft 0 fn d )
by (rule contained-field-descs-extend-ti [OF cont-t]) (auto simp add: ft-def d-def
fields-contained-def)

```

```

then show ?thesis
by (simp add: ti-pad-combine-def Let-def d-def ft-def fn-def padding-desc-def)
qed

```

```

lemma contained-field-descs-simp[simp]:
contained-field-descs (map-align f t) = contained-field-descs t
by (cases t) (simp)

```

```

theorem contained-field-descs-final-pad:
fixes t::'a xtyp-info
assumes cont-t: contained-field-descs t
shows contained-field-descs (final-pad algn t)
using cont-t
by (simp add: final-pad-def contained-field-descs-ti-pad-combine Let-def)

```

```

lemma (in xmem-type) fields-contained-update-desc-mem-type:
fixes acc::'b  $\Rightarrow$  'a
fixes D::'a field-desc set
shows fields-contained (xto-bytes  $\circ$  acc) (upd  $\circ$  xfrom-bytes) (update-desc acc upd
' D)

```

```

proof (rule fields-contained.intro)
fix d s s'
assume d-in: d  $\in$  update-desc acc upd ' D
assume to-bytes-eq: (xto-bytes  $\circ$  acc) s = (xto-bytes  $\circ$  acc) s'

```

```

hence acc s = acc s'
by (auto intro: xto-bytes-inj)

```

```

with d-in
show field-access d s = field-access d s'
by (auto simp add: update-desc-def)

```

```

next
fix d bs s
assume d-in: d  $\in$  update-desc acc upd ' D

```

```

from d-in obtain d' where d'-in: d'  $\in$  D and d: d = update-desc acc upd d'
by auto

```

```

define x where x  $\equiv$   $\lambda s$ . field-update d' bs (acc s)

```

```

from xfrom-bytes-xto-bytes-inv obtain bs' where
cont:  $\forall s'$ . acc s' = acc s  $\longrightarrow$  xfrom-bytes (xto-bytes (x s') bs') = (x s')
by auto

```

```

from d'-in cont
show  $\exists bs'. \forall s'. (xto\text{-}bytes \circ acc) s' = (xto\text{-}bytes \circ acc) s \longrightarrow$ 
   $field\text{-}update\ d\ bs\ s' = (upd \circ xfrom\text{-}bytes) bs' s'$ 
  by (metis (no-types, lifting) comp-apply d field-desc.simps(2) update-desc-def
x-def xto-bytes-inj)
qed

```

```

locale wf-field =
  fixes acc::'s  $\Rightarrow$  'a
  fixes upd::'a  $\Rightarrow$  's  $\Rightarrow$  's
  assumes double-update-upd:  $\bigwedge v\ w\ s. upd\ v\ (upd\ w\ s) = upd\ v\ s$ 
  assumes acc-upd:  $\bigwedge v\ s. acc\ (upd\ v\ s) = v$ 
  assumes upd-acc:  $\bigwedge s. upd\ (acc\ s) s = s$ 

```

```

locale wf-cfield = wf-field +
  constrains acc::'s  $\Rightarrow$  'a::c-type
locale wf-xfield = wf-cfield +
  constrains acc::'s  $\Rightarrow$  'a::xmem-type

```

```

lemma (in wf-field) wf-field-descs-adjust-ti:
  assumes wf-t: wf-field-descs (set (field-descs t))
  shows wf-field-descs (set (field-descs (adjust-ti t acc upd)))
proof –
  have wf-field-descs (update-desc acc upd ‘ set (field-descs t))
  proof (rule wf-field-descs.intro)
    fix d
    assume d  $\in$  update-desc acc upd ‘ set (field-descs t)
    then obtain d' where d'-in: d'  $\in$  set (field-descs t) and d: d = update-desc
acc upd d'
    by auto
    from wf-t d'-in have wf-field-desc d'
    by (rule wf-field-descs.wf-desc)
    then interpret wf-d': wf-field-desc d' .
    from wf-d'.wf-field-desc-update-desc [of upd acc, OF double-update-upd acc-upd
upd-acc]
    have wf-field-desc (update-desc acc upd d') .
    then
    show wf-field-desc d
    by (simp add: d)
  qed
then show ?thesis
  by (simp add: adjust-ti-def field-descs-map)
qed

```

lemma *field-descs-list-append* [simp]: *field-descs-list* (*fs* @ *fs'*) = *field-descs-list* *fs*
 @ *field-descs-list* *fs'*
 by (*induct fs arbitrary: fs'*) *auto*

lemma *wf-field-descs-extend-ti*:
 assumes *wf-t*: *wf-field-descs* (*set* (*field-descs* *t*))
 assumes *wf-ft*: *wf-field-descs* (*set* (*field-descs* *ft*))
 assumes *wf-d*: *wf-field-desc* *d*
 shows *wf-field-descs* (*set* (*field-descs* (*extend-ti* *t* *ft* *algn* *fn* *d*)))
proof (*cases t*)
 case (*TypDesc algn st n*)
 note *desc = this*
 show ?*thesis*
proof (*cases st*)
 case (*TypScalar sz align d*)
 with *wf-ft wf-d* show ?*thesis*
 by (*simp add: desc*)
 next
 case (*TypAggregate fs*)
 note *aggr = this*
 show ?*thesis*
proof (*cases fs*)
 case *Nil*
 then show ?*thesis*
 using *wf-ft wf-d* by (*simp add: desc aggr*)
 next
 case (*Cons f fs'*)
 note *cons = this*
 obtain *ft' fn fd* where *f: f = DTuple ft' fn fd* by (*cases f*)
 show ?*thesis*
 using *wf-ft wf-t wf-d* by (*auto simp add: desc aggr cons f wf-field-descs-def*)
 qed
 qed
 qed

lemma *padding-pad*: *complement-padding* (λv *bs*. if $n \leq \text{length } bs$ then take *n* *bs*
 else replicate *n* 0) (λbs . *id*) *n*
 apply (*unfold-locales*)
 by (*auto simp add: padding-base.is-padding-byte-def padding-base.is-value-byte-def*)

(*metis* (*mono-tags*, *opaque-lifting*) *Ex-list-of-length* *len8* *len-not-eq-0*
less-irrefl *neq0-conv* *nth-list-update-eq* *word-power-less-1*)

lemma *wf-field-desc-pad*:

wf-field-desc

(*field-access* = λv *bs*. if $n \leq \text{length } bs$ then take n *bs* else replicate n 0,
field-update = λbs . *id*, *field-sz* = n)

apply (*intro-locales*)

apply *simp*

apply (*rule padding-pad*)

apply (*unfold-locales*)

apply *auto*

done

lemma *wf-field-descs-ti-pad-combine*: *wf-field-descs* (set (*field-descs* *t*)) \implies

wf-field-descs (set (*field-descs* (*ti-pad-combine* n *t*)))

apply (*simp add: ti-pad-combine-def Let-def*)

apply (*rule wf-field-descs-extend-ti*)

apply (*auto simp add: wf-field-desc-pad padding-desc-def*)

done

lemma *set-field-descs-map-align*[*simp*]: set (*field-descs* (*map-align* *f* *t*)) = set (*field-descs* *t*)

by (*cases* *t*) *simp*

lemma *wf-field-descs-final-pad*: *wf-field-descs* (set (*field-descs* *t*)) \implies

wf-field-descs (set (*field-descs* (*final-pad* *algn* *t*)))

apply (*clarsimp simp add: final-pad-def Let-def*)

apply (*rule wf-field-descs-ti-pad-combine*)

apply *simp*

done

lemma (in *wf-xfield*) *padding-lift*:

shows *complement-padding* (*xto-bytes* \circ *acc*) (*upd* \circ *xfrom-bytes*) (*size-of* *TYPE*('a))

proof –

thm *xmem-type-wf-field-desc.access-update*

thm *xmem-type-wf-field-desc.is-padding-byte-acc-upd-compose*

show *?thesis*

proof

fix *i*

assume *i-bound*: $i < \text{size-of } \text{TYPE}('a)$

{

fix *bs* *v*

have *upd* (*field-update* (*component-desc* (*typ-info-t* *TYPE*('a))) *bs* (*acc* *v*)) *v*

=

upd (*field-update* (*component-desc* (*typ-info-t* *TYPE*('a))) *bs* *undefined*) *v*

by (*simp add: entire-update* [**where** *bs* = *bs* **and** *v* = *acc* *v* **and** *w* = *undefined*])


```

} note tweak-eq = this

have sz-eq: field-sz (component-desc (typ-info-t TYPE('a))) = size-of TYPE('a)
  by (simp add: size-of-def size-td-field-sz(1) wf-component-descs)

from xmem-type-wf-field-desc.is-padding-byte-acc-upd-compose [where acc=acc
and upd=upd, OF acc-upd, of i]
  xmem-type-wf-field-desc.is-value-byte-acc-upd-compose [where acc=acc and
upd=upd, OF acc-upd, of i]
  show padding-base.is-padding-byte (xto-bytes ∘ acc) (upd ∘ xfrom-bytes)
    (size-of TYPE('a)) i ≠
    padding-base.is-value-byte (xto-bytes ∘ acc) (upd ∘ xfrom-bytes)
    (size-of TYPE('a)) i
  apply (simp add: xto-bytes-def xfrom-bytes-def [abs-def] comp-def tweak-eq
sz-eq)
  using xmem-type-wf-field-desc.complement i-bound sz-eq
  by metis
qed
qed

```

```

lemma (in wf-xfield) wf-field-desc-lift:
  shows wf-field-desc (field-access = xto-bytes ∘ acc,
    field-update = upd ∘ xfrom-bytes,
    field-sz = size-of (TYPE('a))) (is wf-field-desc ?lift)
  apply (intro-locales, simp, rule padding-lift, unfold-locales)
proof -
  fix bs bs' s
  show field-update ?lift bs (field-update ?lift bs' s) = field-update ?lift bs s
    by (simp add: double-update-upd)
next
  fix bs s bs' s'
  show field-access ?lift (field-update ?lift bs s) bs' = field-access ?lift (field-update
?lift bs s') bs'
    by (simp add: acc-upd)
next
  fix s bs
  show field-update ?lift (field-access ?lift s bs) s = s
    by (simp add: upd-acc xfrom-bytes-xto-bytes-inv)
next
  fix s bs
  show field-access ?lift s (take (field-sz ?lift) bs) = field-access ?lift s bs
    by (simp add: xto-bytes-size size-of-def)
next
  fix s bs
  show length (field-access ?lift s bs) = field-sz ?lift
    by (simp add: xto-bytes-result-size size-of-def)

```

```

next
  fix bs
  show field-update ?lift (take (field-sz ?lift) bs) = field-update ?lift bs
    by (simp add: xfrom-bytes-size size-of-def)
qed

```

```

lemma (in wf-xfield) wf-field-descs-ti-typ-combine:
  assumes wf-ft: wf-field-descs (set (field-descs ft))
  shows wf-field-descs (set (field-descs (ti-typ-combine (TYPE('a)) acc upd algn
fn ft)))
  apply (simp add: ti-typ-combine-def)
  apply (rule wf-field-descs-extend-ti)
  apply (rule wf-ft)
  apply (rule wf-field-descs-adjust-ti)
  apply (rule wf-field-descs)
  apply (rule wf-field-desc-lift)
done

```

```

lemma (in wf-xfield) wf-field-descs-ti-typ-pad-combine:
  assumes wf-ft: wf-field-descs (set (field-descs ft))
  shows wf-field-descs (set (field-descs (ti-typ-pad-combine (TYPE('a)) acc upd
algn fn ft)))
proof (cases 0 < padup (max (2 ^ algn) (align-of TYPE('a))) (size-td ft))
  case True
  then show ?thesis
    apply (simp add: ti-typ-pad-combine-def Let-def)
    apply (rule wf-field-descs-ti-typ-combine)
    apply (rule wf-field-descs-ti-pad-combine)
    apply (rule wf-ft)
  done
next
  case False
  then show ?thesis
    apply (simp add: ti-typ-pad-combine-def Let-def)
    apply (rule wf-field-descs-ti-typ-combine)
    apply (rule wf-ft)
  done
qed

```

```

lemma wf-component-descs-empty-typ-info [simp]: wf-component-descs (empty-typ-info
algn n)
  by (simp add: empty-typ-info-def)

```

```

lemma wf-component-descs-list-append [simp]:
  wf-component-descs-list (fs @ fs') = (wf-component-descs-list fs  $\wedge$  wf-component-descs-list
fs')

```

by (induct fs arbitrary: fs') auto

lemma *wf-component-descs-extend-ti*:
assumes *wf-t*: *wf-component-descs t*
assumes *wf-ft*: *wf-component-descs ft*
assumes *d*: *d = (component-desc ft)*
shows *wf-component-descs (extend-ti t ft algn fn d)*
apply (cases t)
subgoal for *x1 st n*
 apply (cases st)
 apply (simp add: *wf-ft d*)
 subgoal for *fs*
 apply (cases fs)
 apply (simp add: *wf-ft d*)
 subgoal for *f fs'*
 apply (cases f)
 apply (insert *wf-t*)
 apply (simp add: *d wf-ft*)
 done
 done
done
done

lemma *wf-component-descs-ti-pad-combine*:
assumes *wf-t*: *wf-component-descs t*
shows *wf-component-descs (ti-pad-combine n t)*
apply (simp add: *ti-pad-combine-def Let-def padding-desc-def*)
apply (rule *wf-component-descs-extend-ti*)
 apply (rule *wf-t*)
 apply simp
apply simp
done

lemma *wf-component-descs-map-align [simp]*: *wf-component-descs (map-align f t)*
= *wf-component-descs t*
by (cases t) simp

lemma *wf-component-descs-final-pad*:
assumes *wf-t*: *wf-component-descs t*
shows *wf-component-descs (final-pad algn t)*
by (clarsimp simp add: *final-pad-def Let-def wf-t wf-component-descs-ti-pad-combine*)

lemma *field-sz-update-desc [simp]*: *field-sz (update-desc acc upd d) = field-sz d*
by (simp add: *update-desc-def*)

lemma *component-sz-struct-update-desc-commutes*:

```

(component-sz-struct (TypAggregate fs)) =
  component-sz-struct (map-td-struct (λn algn. update-desc acc upd) (update-desc
acc upd) (TypAggregate fs))
proof (induct fs)
  case Nil
  then show ?case by simp
next
  case (Cons f fs')
  obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp
  have map-eq: map (field-sz ∘ dt-trd) (map-td-list (λn algn. update-desc acc upd)
(update-desc acc upd) fs') =
    map (field-sz ∘ dt-trd) fs'
  apply (induct fs')
  apply simp
  subgoal for f' fs''
  apply (cases f')
  apply clarsimp
  done
  done
  show ?case by (simp add: f map-eq)
qed

```

```

lemma component-access-struct-update-desc-commutes':
component-access-struct (TypAggregate fs) (acc v) bs =
  component-access-struct (map-td-struct (λn algn. update-desc acc upd) (update-desc
acc upd) (TypAggregate fs)) v bs
proof (induct fs arbitrary: bs)
  case Nil
  then show ?case by simp
next
  case (Cons f fs')
  obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp

  have d-commutes: field-access (update-desc acc upd d) v bs = field-access d (acc
v) bs
    by (simp add: update-desc-def)

  note hyp = Cons.hyps [where bs=(drop (length (field-access d (acc v) bs)) bs) ]
  then
  show ?case
    by (simp add: f d-commutes hyp)
qed

```

```

lemma component-access-struct-update-desc-commutes:
(component-access-struct (TypAggregate fs)) ∘ acc =
  component-access-struct (map-td-struct (λn algn. update-desc acc upd) (update-desc
acc upd) (TypAggregate fs))
apply -

```

```

apply (rule ext)
apply (rule ext)
apply (simp only: comp-def)
apply (rule component-access-struct-update-desc-commutes')
done

lemma (in wf-field) lemma-component-update-struct-update-desc-commutes:
  shows upd (component-update-struct (TypAggregate fs) bs (acc v)) v =
    component-update-struct (map-td-struct (λn algn. update-desc acc upd)
      (update-desc acc upd) (TypAggregate fs)) bs v
proof (induct fs arbitrary: v bs)
  case Nil
  then show ?case by (simp add: upd-acc)
next
  case (Cons f fs')
  obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp

  have acc-update-desc: acc (field-update (update-desc acc upd d) bs v) = field-update
    d bs (acc v)
    by (simp add: update-desc-def acc-upd)

  have upd-eq: upd (fst (apply-field-updates (map dt-trd fs') (drop (field-sz d) bs)
    (field-update d bs (acc v)))) (field-update (update-desc acc upd d) bs
    v) =
    upd (fst (apply-field-updates (map dt-trd fs') (drop (field-sz d) bs)
      (field-update d bs (acc v)))) v
    by (simp add: update-desc-def double-update-upd)

  note hyp = Cons.hyps [where bs = (drop (field-sz d) bs) and v=field-update
    (update-desc acc upd d) bs v]

  from hyp
  show ?case
    by (simp add: f acc-update-desc upd-eq)
qed

lemma (in wf-field) update-desc-component-desc-struct-commutes:
  shows update-desc acc upd (component-desc-struct (TypAggregate fs)) =
    component-desc-struct (map-td-struct (λn algn. update-desc acc upd)
      (update-desc acc upd) (TypAggregate fs))
  using lemma-component-update-struct-update-desc-commutes
    component-access-struct-update-desc-commutes [where acc=acc and upd=upd
    and fs=fs]
    component-sz-struct-update-desc-commutes [where acc=acc and upd=upd and
    fs=fs]
  apply (simp only: update-desc-def)
  apply (fastforce simp add: component-desc-struct-def)
  done

```

lemma (in *wf-field*) *update-desc-component-desc-commutes*:
shows $update_desc\ acc\ upd\ (component_desc\ t) =$
 $component_desc\ (map_td\ (\lambda n\ algn.\ update_desc\ acc\ upd))\ (update_desc\ acc$
 $upd)\ t)$
using *update-desc-component-desc-struct-commutes*
apply (*cases t*)
apply *simp*
subgoal for *x1 st n*
apply (*cases st*)
apply *simp*
apply *simp*
done
done

lemma (in *wf-field*) *wf-component-descs-map-td-update-desc*:
shows
 $wf_component_descs\ t$
 $\implies wf_component_descs\ (map_td\ (\lambda n\ algn.\ update_desc\ acc\ upd))\ (update_desc$
 $acc\ upd)\ t)$
 $wf_component_descs_struct\ st$
 $\implies wf_component_descs_struct\ (map_td_struct\ (\lambda n\ algn.\ update_desc\ acc\ upd)$
 $(update_desc\ acc\ upd)\ st)$
 $wf_component_descs_list\ fs$
 $\implies wf_component_descs_list\ (map_td_list\ (\lambda n\ algn.\ update_desc\ acc\ upd))\ (update_desc$
 $acc\ upd)\ fs)$
 $wf_component_descs_tuple\ f$
 $\implies wf_component_descs_tuple\ (map_td_tuple\ (\lambda n\ algn.\ update_desc\ acc\ upd))\ (update_desc$
 $acc\ upd)\ f)$
apply (*induct t and st and fs and f*)
apply (*auto simp add: update-desc-component-desc-commutes*)
done

lemma (in *wf-field*) *wf-component-descs-adjust-ti*:
assumes *wf-t: wf-component-descs t*
shows $wf_component_descs\ (adjust_ti\ t\ acc\ upd)$
using *wf-component-descs-map-td-update-desc(1)* [*OF wf-t*]
by (*simp add: adjust-ti-def*)

lemma (in *wf-xfield*) *update-desc-component-desc*:
shows
 $update_desc\ acc\ upd\ (component_desc\ (typ_info_t\ TYPE('a))) =$
 $(\|field_access = xto_bytes\ \circ\ acc,\ field_update = upd\ \circ\ xfrom_bytes,$
 $field_sz = size_of\ TYPE('a)\|)$
proof –
have *wf: wf-component-descs (typ-info-t TYPE('a))* **by** (*rule wf-component-descs*)
have *eq: $\bigwedge bs\ v.\ field_update\ (component_desc\ (typ_info_t\ TYPE('a)))\ bs\ (acc\ v)$*
 $=$
 $field_update\ (component_desc\ (typ_info_t\ TYPE('a)))\ bs\ undefined$

```

  by (simp add: entire-update)
  show ?thesis
  using size-td-field-sz(1) [OF wf]
  apply (simp add: xto-bytes-def size-of-def update-desc-def)
  apply (rule ext)
  apply (rule ext)
  apply (simp add: xfrom-bytes-def eq)
  done
qed

```

```

lemma (in wf-xfield) wf-component-descs-ti-typ-combine:
  assumes wf-t: wf-component-descs t
  shows wf-component-descs (ti-typ-combine ft acc upd algn fn t)
proof -
  have wf-ft: wf-component-descs (typ-info-t TYPE('a)) by (rule wf-component-descs)
  show ?thesis
  apply (simp add: ti-typ-combine-def Let-def)
  apply (rule wf-component-descs-extend-ti)
  apply (rule wf-t)
  apply (rule wf-component-descs-adjust-ti)
  apply (rule wf-ft)
  apply (simp add: adjust-ti-def Let-def update-desc-component-desc
    update-desc-component-desc-commutes[symmetric])
  done
qed

```

```

lemma (in wf-xfield) wf-component-descs-ti-typ-pad-combine:
  assumes wf-t: wf-component-descs t
  shows wf-component-descs (ti-typ-pad-combine ft acc upd algn fn t)
  apply (cases 0 < padup (max (2 ^ algn) (align-of TYPE('a))) (size-td t))
  apply (simp add: ti-typ-pad-combine-def Let-def)
  apply (rule wf-component-descs-ti-typ-combine)
  apply (rule wf-component-descs-ti-pad-combine [OF wf-t])
  apply (simp add: ti-typ-pad-combine-def Let-def)
  apply (rule wf-component-descs-ti-typ-combine [OF wf-t])
  done

```

```

lemma component-descs-independent-empty-typ-info [simp]:
  component-descs-independent (empty-typ-info algn n)
  by (simp add: empty-typ-info-def)

```

```

lemma component-descs-independent-list-appendI:
  assumes ind-xs-ys:  $\forall x \in \text{set (toplevel-field-descs-list xs)}$ .
    field-desc-independent (field-access x) (field-update x) (set (toplevel-field-descs-list
  ys))
  assumes ind-xs: component-descs-independent-list xs

```

assumes *ind-ys: component-descs-independent-list ys*
shows *component-descs-independent-list (xs @ ys)*
using *ind-xs-ys ind-xs ind-ys*
proof (*induct xs arbitrary: ys*)
case *Nil*
then show *?case by simp*
next
case (*Cons x xs*)
obtain *ft' fn' d'* **where** *x: x = DTuple ft' fn' d'* **by** (*cases x*) *simp*

from *Cons.prem*s **obtain**

find-d'-ys: field-desc-independent (field-access d') (field-update d')
(set (toplevel-field-descs-list ys)) and
find-xs-ys: $\forall x \in \text{set (toplevel-field-descs-list xs)}$.
field-desc-independent (field-access x) (field-update x) (set
(*toplevel-field-descs-list ys*)) **and**
find-d'-xs: field-desc-independent (field-access d') (field-update d') (set (toplevel-field-descs-list
xs)) and
find-xs: field-descs-independent (toplevel-field-descs-list xs) and
ind-ft': component-descs-independent ft' and
ind-ys: component-descs-independent-list ys and
ind-xs: component-descs-independent-list xs
by (*clarsimp simp add: x*)

from *find-d'-xs find-d'-ys*
have *find-d': field-desc-independent (field-access d') (field-update d')*
(set (toplevel-field-descs-list xs) \cup set (toplevel-field-descs-list ys))
by (*simp add: field-desc-independent-union-iff*)

from *component-descs-list-field-descs-independent [OF ind-xs]*
have *tl-xs: field-descs-independent (toplevel-field-descs-list xs).*
from *component-descs-list-field-descs-independent [OF ind-ys]*
have *tl-ys: field-descs-independent (toplevel-field-descs-list ys).*

from *field-descs-independent-appendI1 [OF tl-xs tl-ys find-xs-ys]*
have *tl-xs-ys: field-descs-independent*
(toplevel-field-descs-list xs @ toplevel-field-descs-list ys) .

from *Cons.hyps [OF find-xs-ys ind-xs ind-ys]*
have *ind-xs-ys: component-descs-independent-list (xs @ ys) .*
from *find-d' tl-xs-ys ind-ft' ind-xs-ys*
show *?case*
by (*simp add: x*)
qed

lemma *component-descs-independent-extend-ti:*
assumes *ind-t: component-descs-independent t*


```

assumes ind-ft: component-descs-independent ft
assumes ind-d-t: field-desc-independent (field-access d) (field-update d) (set (toplevel-field-descs t))
shows component-descs-independent (extend-ti t ft algn fn d)
proof (cases t)
  case (TypDesc algn st nn)
  then show ?thesis
  proof (cases st)
    case (TypScalar sz align d)
    from ind-ft show ?thesis
    by (simp add: TypDesc TypScalar)
  next
  case (TypAggregate fs)
  then show ?thesis
  proof (cases fs)
    case Nil
    from ind-ft show ?thesis
    by (simp add: TypDesc TypAggregate Nil)
  next
  case (Cons f fs')
  obtain ft' fn' d' where f: f = DTuple ft' fn' d' by (cases f) simp
  from ind-t obtain
    field-ind-d'-fs': field-desc-independent (field-access d') (field-update d')
    (set (toplevel-field-descs-list fs')) and
    field-ind-fs': field-descs-independent (toplevel-field-descs-list fs') and
    comp-ind-ft': component-descs-independent ft' and
    comp-ind-fs': component-descs-independent-list fs'
    by (simp add: TypDesc TypAggregate Cons f)

  from ind-d-t obtain ind-d-d': field-desc-independent (field-access d) (field-update
d) {d'} and
    ind-d-fs': field-desc-independent (field-access d) (field-update d) (set (toplevel-field-descs-list
fs'))
    apply (simp add: TypDesc TypAggregate Cons f)
    supply field-desc-independent-insert-iff[iff]
    by fastforce

  from field-desc-independent-symmetric [of {d} {d'} ind-d-d'
  have field-ind-d'-d: field-desc-independent (field-access d') (field-update d')
{d}
    by auto

  from field-ind-d'-fs' field-ind-d'-d
  have field-ind-d'-d-fs': field-desc-independent (field-access d') (field-update d')
(insert d (set (toplevel-field-descs-list fs')))
    apply (subst insert-is-Un)
    apply (simp only: field-desc-independent-union-iff)
    done

```

```

have ind-d: field-descs-independent [d] by simp
from field-desc-independent-symmetric [of set [d] set (toplevel-field-descs-list
fs')] ind-d-fs'
have ind-fs'-d:  $\forall x \in \text{set (toplevel-field-descs-list fs')}$ .
    field-desc-independent (field-access x) (field-update x) (set [d]) by simp
from field-descs-independent-appendI1 [OF field-ind-fs' ind-d ind-fs'-d]
have field-ind-fs'-d:field-descs-independent (toplevel-field-descs-list fs' @ [d]) .

from ind-fs'-d ind-ft
have comp-ind-fs'-d: component-descs-independent-list (fs' @ [DTuple ft fn
d])
    apply –
    apply (rule component-descs-independent-list-appendI [OF - comp-ind-fs' ])
)
    apply simp
    apply simp
    done
from field-ind-d'-d-fs' field-ind-fs'-d comp-ind-ft' comp-ind-fs'-d
show ?thesis
    by (simp add: TypDesc TypAggregate Cons f)
qed
qed
qed

```

```

lemma fu-commutes-id1 [simp]: fu-commutes ( $\lambda bs. id$ ) upd
by (auto simp add: fu-commutes-def)

```

```

lemma fu-commutes-id2 [simp]: fu-commutes upd ( $\lambda bs. id$ )
by (auto simp add: fu-commutes-def)

```

```

lemma field-desc-independent-pad: field-desc-independent ( $\lambda v bs. \text{if } n \leq \text{length } bs$ 
then take n bs else replicate n 0) ( $\lambda bs. id$ ) D
by (rule field-desc-independent.intro) auto

```

```

lemma component-descs-independent-ti-pad-combine:
assumes ind-t: component-descs-independent t
shows component-descs-independent (ti-pad-combine n t)
apply (simp add: ti-pad-combine-def Let-def padding-desc-def)
apply (rule component-descs-independent-extend-ti [OF ind-t])
apply simp
apply (simp add: field-desc-independent-pad)
done

```

```

lemma (in wf-field) field-desc-independent-update-desc:

```

```

assumes ind: field-desc-independent (field-access d) (field-update d) D
shows field-desc-independent (field-access (update-desc acc upd d)) (field-update
(update-desc acc upd d))
      (update-desc acc upd ‘ D) (is field-desc-independent ?acc ?upd ?U)
proof –
from ind
interpret ind: field-desc-independent field-access d field-update d D .
show ?thesis
proof
  fix ud
  assume ud ∈ ?U
  show fu-commutes ?upd (field-update ud)
  by (smt ⟨ud ∈ update-desc acc upd ‘ D⟩ acc-upd double-update-upd field-desc.select-convs(2)
      field-desc-independent.fu-commutes fu-commutes-def imageE ind.field-desc-independent-axioms
update-desc-def)
  next
  fix ud bs v bs'
  assume ud ∈ ?U
  show ?acc (field-update ud bs v) bs' = ?acc v bs'
  by (smt ⟨ud ∈ update-desc acc upd ‘ D⟩ acc-upd field-desc.select-convs(1)
      field-desc.select-convs(2)
      field-desc-independent.acc-upd-old imageE ind.field-desc-independent-axioms
o-apply update-desc-def)
  next
  fix ud bs' v bs
  assume ud ∈ ?U
  show field-access ud (?upd bs' v) bs = field-access ud v bs
  by (smt ⟨ud ∈ update-desc acc upd ‘ D⟩ acc-upd field-desc.select-convs(1)
      field-desc.select-convs(2)
      field-desc-independent.acc-upd-new imageE ind.field-desc-independent-axioms
o-apply update-desc-def)
  qed
qed

```

```

lemma (in wf-field) field-descs-independent-update-desc:
  field-descs-independent (toplevel-field-descs t)
  ⇒ field-descs-independent (toplevel-field-descs
      (map-td (λn algn. update-desc acc upd) (update-desc acc upd) t))
  field-descs-independent (toplevel-field-descs-struct st)
  ⇒ field-descs-independent (toplevel-field-descs-struct
      (map-td-struct (λn algn. update-desc acc upd) (update-desc acc upd) st))
  field-descs-independent (toplevel-field-descs-list fs)
  ⇒ field-descs-independent (toplevel-field-descs-list
      (map-td-list (λn algn. update-desc acc upd) (update-desc acc upd) fs))
  field-descs-independent (toplevel-field-descs-tuple f)
  ⇒ field-descs-independent (toplevel-field-descs-tuple
      (map-td-tuple (λn algn. update-desc acc upd) (update-desc acc upd) f))

```

```

proof (induct t and st and fs and f)
  case (TypDesc st n)
  then show ?case by simp
next
  case (TypScalar sz align d)
  then show ?case by simp
next
  case (TypAggregate fs)
  then show ?case by simp
next
  case Nil-typ-desc
  then show ?case by simp
next
  case (Cons-typ-desc f fs)

  obtain ft' fn' d' where f: f = DTuple ft' fn' d' by (cases f) simp

  from Cons-typ-desc.prem
  have ind-f-fs: field-descs-independent
    (toplevel-field-descs-tuple f @ toplevel-field-descs-list fs) by simp

  from field-descs-independent-append-first [OF ind-f-fs]
  have ind-f: field-descs-independent (toplevel-field-descs-tuple f) .

  from field-descs-independent-append-second [OF ind-f-fs]
  have ind-fs: field-descs-independent (toplevel-field-descs-list fs) .

  from Cons-typ-desc.hyps(1) [OF ind-f]
  have ind-upd-f: field-descs-independent
    (toplevel-field-descs-tuple
      (map-td-tuple (λn alg. update-desc acc upd) (update-desc acc
upd) f)).

  from Cons-typ-desc.hyps(2) [OF ind-fs]
  have ind-upd-fs: field-descs-independent
    (toplevel-field-descs-list
      (map-td-list (λn alg. update-desc acc upd) (update-desc acc
upd) fs)) .

  from field-descs-independent-append-first-ind [OF ind-f-fs]
  have field-desc-independent (field-access d') (field-update d') (set (toplevel-field-descs-list
fs))
  by (auto simp add: f)
  from field-desc-independent-update-desc [OF this]
  have field-desc-independent (field-access (update-desc acc upd d')) (field-update
(update-desc acc upd d'))
    (update-desc acc upd ' set (toplevel-field-descs-list fs)) .

```

```

then
  have upd-d'-fs: field-desc-independent (field-access (update-desc acc upd d'))
(field-update (update-desc acc upd d'))
  (set (toplevel-field-descs-list
      (map-td-list ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd)
fs)))
  by (simp add: toplevel-field-descs-map)
from ind-upd-fs upd-d'-fs
have ind-upd-f-fs: field-descs-independent
  (toplevel-field-descs-tuple
  (map-td-tuple ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) f) @
  toplevel-field-descs-list
  (map-td-list ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) fs))
  by (simp add: f)
from Cons-typ-desc ind-upd-f-fs
show ?case by simp
next
  case (DTuple-typ-desc ft fn d)
  then show ?case by simp
qed

```

```

lemma (in wf-field) component-descs-independent-update-desc:
  component-descs-independent t
 $\implies$  component-descs-independent
  (map-td ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) t)
  component-descs-independent-struct st
 $\implies$  component-descs-independent-struct
  (map-td-struct ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) st)
  component-descs-independent-list fs
 $\implies$  component-descs-independent-list
  (map-td-list ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) fs)
  component-descs-independent-tuple f
 $\implies$  component-descs-independent-tuple
  (map-td-tuple ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) f)
proof (induct t and st and fs and f)
  case (TypDesc st n)
then show ?case by simp
next
  case (TypScalar sz align d)
  then show ?case by simp
next
  case (TypAggregate fs)
then show ?case by simp
next
  case Nil-typ-desc
  then show ?case by simp

```

```

next
  case (Cons-typ-desc f fs)

  obtain ft' fn' d' where f: f = DTuple ft' fn' d' by (cases f) simp

  from Cons-typ-desc.prems obtain
    find-f-fs: field-descs-independent
      (toplevel-field-descs-tuple f @ toplevel-field-descs-list fs) and
    ind-f: component-descs-independent-tuple f and
    ind-fs: component-descs-independent-list fs
  by simp

  from field-descs-independent-append-first [OF find-f-fs]
  have fi-f: field-descs-independent (toplevel-field-descs-tuple f) .

  from Cons-typ-desc.hyps(1) [OF ind-f]
  have ind-upd-f: component-descs-independent-tuple
    (map-td-tuple ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) f) .
  from field-descs-independent-update-desc(4) [OF fi-f]
  have field-descs-independent (toplevel-field-descs-tuple
    (map-td-tuple ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) f)) .
  with ind-upd-f
  have ind-upd-f': component-descs-independent-list
    (map-td-list ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) [f]) by
simp

  from Cons-typ-desc.hyps(2) [OF ind-fs]
  have ind-upd-fs: component-descs-independent-list
    (map-td-list ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) fs) .
  from component-descs-list-field-descs-independent [OF this]
  have find-upd-fs: field-descs-independent (toplevel-field-descs-list
    (map-td-list ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) fs)) .
  from field-descs-independent-append-first-ind [OF find-f-fs]
  have find-fs: field-desc-independent (field-access d') (field-update d') (set (toplevel-field-descs-list
fs))
    by (auto simp add: f)
  from field-desc-independent-update-desc [OF this]
  have field-desc-independent (field-access (update-desc acc upd d')) (field-update
(update-desc acc upd d'))
    (update-desc acc upd ' set (toplevel-field-descs-list fs)) .
  then
    have upd-d'-fs: field-desc-independent (field-access (update-desc acc upd d'))
(field-update (update-desc acc upd d'))
    (set (toplevel-field-descs-list
      (map-td-list ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd)
fs)))
    by (simp add: toplevel-field-descs-map)

```

```

from upd-d'-fs find-upd-fs
have ind-f-fs: field-descs-independent
  (toplevel-field-descs-tuple
   (map-td-tuple ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) f) @
   toplevel-field-descs-list
   (map-td-list ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) fs))
by (simp add: f)
from Cons-typ-desc ind-f-fs show ?case
by (simp)
next
  case (DTuple-typ-desc ft fn d)
  then show ?case by simp
qed

```

```

lemma (in wf-field) component-descs-independent-adjust-ti:
  assumes ind-t: component-descs-independent t
  shows component-descs-independent (adjust-ti t acc upd)
  apply (simp add: adjust-ti-def)
  apply (rule component-descs-independent-update-desc(1) [OF ind-t])
  done

```

```

theorem (in wf-xfield) component-descs-independent-ti-typ-combine:
  fixes
    ft::'a::xmem-type itself and
    t:: 's xtyp-info
  assumes ind-t: component-descs-independent t
  assumes ind-acc-upd: field-desc-independent (xto-bytes  $\circ$  acc) (upd  $\circ$  xfrom-bytes)
    (set (toplevel-field-descs t))
  shows component-descs-independent (ti-typ-combine ft acc upd algn fn t)
  apply (simp add: ti-typ-combine-def)
  apply (rule component-descs-independent-extend-ti [OF ind-t])
  apply (rule component-descs-independent-adjust-ti [OF component-descs-independent])
  apply simp
  apply (rule ind-acc-upd)
  done

```

```

lemma (in wf-xfield) wf-field-desc-extend-ti:
  assumes ind-t: field-desc-independent (xto-bytes  $\circ$  acc) (upd  $\circ$  xfrom-bytes) (set
(toplevel-field-descs t))
  assumes ind-d: field-desc-independent (xto-bytes  $\circ$  acc) (upd  $\circ$  xfrom-bytes) {d}
  shows
    field-desc-independent (xto-bytes  $\circ$  acc) (upd  $\circ$  xfrom-bytes)
      (set (toplevel-field-descs (extend-ti t ft algn fn d)))
proof (cases t)

```

```

case (TypDesc align' st nn)
then show ?thesis
proof (cases st)
  case (TypScalar sz align d)
  from ind-d show ?thesis
  by (simp add: TypDesc TypScalar)
next
  case (TypAggregate fs)
  then show ?thesis
  proof (cases fs)
  case Nil
  from ind-d show ?thesis
  by (simp add: TypDesc TypAggregate Nil)
next
  case (Cons f fs')
  obtain ft' fn' d' where f: f = DTuple ft' fn' d' by (cases f) simp
  from ind-d ind-t
  show ?thesis
  apply (simp add: TypDesc TypAggregate Cons f)
  by (meson field-desc-independent-insert-iff)
qed
qed
qed

```

lemma (in wf-xfield) field-desc-independent-ti-pad-combine:
assumes ind-acc-upd: field-desc-independent (xto-bytes \circ acc) (upd \circ xfrom-bytes)
 (set (toplevel-field-descs t))
shows field-desc-independent (xto-bytes \circ acc) (upd \circ xfrom-bytes)
 (set (toplevel-field-descs (ti-pad-combine n t)))
apply (simp add: ti-pad-combine-def Let-def padding-desc-def)
apply (rule wf-field-desc-extend-ti [OF ind-acc-upd])
by (auto simp add: field-desc-independent-def)

theorem (in wf-xfield) component-desc-independent-ti-typ-pad-combine:
fixes
 ft::'a::xmem-type itself **and**
 t:: 's xtyp-info
assumes ind-t: component-descs-independent t
assumes ind-acc-upd: field-desc-independent (xto-bytes \circ acc) (upd \circ xfrom-bytes)
 (set (toplevel-field-descs t))
shows component-descs-independent (ti-typ-pad-combine ft acc upd align fn t)

apply (cases 0 < padup (max (2 ^ align) (align-of TYPE('a))) (size-td t))
apply (simp add: ti-typ-pad-combine-def Let-def)
apply (rule component-descs-independent-ti-typ-combine)
apply (rule component-descs-independent-ti-pad-combine [OF ind-t])
apply (rule field-desc-independent-ti-pad-combine [OF ind-acc-upd])
apply (simp add: ti-typ-pad-combine-def Let-def)
apply (rule component-descs-independent-ti-typ-combine [OF ind-t ind-acc-upd])


```

done

lemma component-descs-independent-map-align[simp]:
  component-descs-independent (map-align f t) = component-descs-independent t
  by (cases t) simp

lemma component-descs-independent-final-pad:
  assumes ind-t: component-descs-independent t
  shows component-descs-independent (final-pad algn t)
  apply (clarsimp simp add: final-pad-def Let-def ind-t)
  apply (rule component-descs-independent-ti-pad-combine [OF ind-t])
  done

theorem (in xmem-contained-type) contained-field-descs-ti-typ-combine:
  fixes
    ft::'a itself and
    t:: 'b xtyp-info
  assumes contained-t: contained-field-descs t
  shows contained-field-descs (ti-typ-combine ft acc upd algn fn t)
  proof -
    from contained-field-descs-adjust-ti [OF contained-field-descs]
    have contained-field-descs (adjust-ti (typ-info-t TYPE('a)) acc upd) .
    moreover
    define d::'b field-desc
      where d ≡ (|field-access = xto-bytes ∘ acc, field-update = upd ∘ xfrom-bytes,
        field-sz = size-of TYPE('a))

    from fields-contained-update-desc-mem-type [of acc upd]
    have fields-contained (field-access d) (field-update d)
      (set (toplevel-field-descs (adjust-ti (typ-info-t TYPE('a)) acc upd)))
    by (simp add: toplevel-field-descs-adjust-ti d-def)
    ultimately
    show ?thesis
      using contained-t
      by (simp add: ti-typ-combine-def contained-field-descs-extend-ti d-def)
  qed

theorem (in xmem-contained-type) contained-field-descs-ti-typ-pad-combine:
  fixes
    ft::'a itself and
    t:: 'b xtyp-info
  assumes contained-t: contained-field-descs t
  shows contained-field-descs (ti-typ-pad-combine ft acc upd algn fn t)
  using contained-t contained-field-descs
  by (simp add: ti-typ-pad-combine-def Let-def contained-field-descs-ti-typ-combine
    contained-field-descs-ti-pad-combine)

```

```

locale ti-ind' =
  fixes X :: 'a leaf-desc set
  fixes Y :: 'a leaf-desc set
  assumes fu-commutes:  $x \in X \implies y \in Y \implies \text{fu-commutes (field-update (lf-fd } x)) \text{ (field-update (lf-fd } y))}$ 
  assumes fa-fu-ind-X:  $x \in X \implies y \in Y \implies \text{fa-fu-ind (lf-fd } x) \text{ (lf-fd } y) \text{ (lf-sz } y) \text{ (lf-sz } x)}$ 
  assumes fa-fu-ind-Y:  $x \in X \implies y \in Y \implies \text{fa-fu-ind (lf-fd } y) \text{ (lf-fd } x) \text{ (lf-sz } x) \text{ (lf-sz } y)}$ 

```

```

lemma ti-ind'-ti-ind:  $\text{ti-ind' } X \ Y = \text{ti-ind } X \ Y$ 
  by (auto simp add: ti-ind-def ti-ind'-def)

```

```

lemma fields-contained-transitive:
  assumes d-in:  $d \in D$ 
  assumes d-contains :  $\text{fields-contained (field-access } d) \text{ (field-update } d) \ X}$ 
  assumes contained:  $\text{fields-contained acc upd } D$ 
  shows  $\text{fields-contained acc upd } X$ 
  using d-in d-contains contained unfolding fields-contained-def
  by metis

```

```

lemma fields-contained-singleton [simp]:
   $\text{fields-contained (field-access } d) \text{ (field-update } d) \ \{d\}$ 
  by (auto simp add: fields-contained-def)

```

```

lemma contained-field-descs-leaf:
   $\text{contained-field-descs } t \implies \text{ld} \in \text{lf-fd ' (lf-set } t \ \[] \implies$ 
     $\exists d \in \text{set (toplevel-field-descs } t). \text{fields-contained (field-access } d) \text{ (field-update } d) \ \{ld\}}$ 
   $\text{contained-field-descs-struct } st \implies \text{ld} \in \text{lf-fd ' (lf-set-struct } st \ \[] \implies$ 
     $\exists d \in \text{set (toplevel-field-descs-struct } st). \text{fields-contained (field-access } d) \text{ (field-update } d) \ \{ld\}}$ 
   $\text{contained-field-descs-list } fs \implies \text{ld} \in \text{lf-fd ' (lf-set-list } fs \ \[] \implies$ 
     $\exists d \in \text{set (toplevel-field-descs-list } fs). \text{fields-contained (field-access } d) \text{ (field-update } d) \ \{ld\}}$ 
   $\text{contained-field-descs-tuple } f \implies \text{ld} \in \text{lf-fd ' (lf-set-tuple } f \ \[] \implies$ 
     $\exists d \in \text{set (toplevel-field-descs-tuple } f). \text{fields-contained (field-access } d) \text{ (field-update } d) \ \{ld\}}$ 
  proof (induct t and st and fs and f)
    case (TypDesc st n)
    then show ?case by auto
  next
    case (TypScalar n sz d)
    then show ?case by auto
  next
    case (TypAggregate fs)

```

```

then show ?case by auto
next
case Nil-typ-desc
then show ?case by auto
next
case (Cons-typ-desc f fs)
then show ?case by auto
next
case (DTuple-typ-desc ft fn d)
then show ?case
by (metis contained-field-descs-tuple.simps toplevel-field-descs-tuple.simps fields-contained-transitive
insertI1 lf-fd-fn(1) lf-set-tuple.simps list.simps(15))
qed

```

lemma *wf-field-desc-wf-lf*:
 $wf\text{-field-desc } d \implies sz = field\text{-sz } d \implies wf\text{-lf } \{(lf\text{-fd} = d, lf\text{-sz} = sz, lf\text{-fn} = xs)\}$
by (*simp add: wf-field-desc-def*
fd-cons-access-update-def fd-cons-desc-def fd-cons-double-update-def fd-cons-length-def
fd-cons-update-access-def padding-lense.access-result-size padding-lense.access-update
padding-lense.double-update padding-lense.update-access wf-lf-def)

lemma *wf-lf-unionI*: $wf\text{-lf } A \implies wf\text{-lf } B \implies ti\text{-ind}' A B \implies wf\text{-lf } (A \cup B)$
by (*auto simp add: wf-lf-def ti-ind'-def fu-commutes-def*)

lemma *fields-contained-subset*:
assumes *cont-X*: *fields-contained acc upd X*
assumes *subs*: $Y \subseteq X$
shows *fields-contained acc upd Y*
proof –
from *cont-X* **interpret** *X*: *fields-contained acc upd X* .
show ?thesis
proof
fix *d s s'*
assume *d*: $d \in Y$
assume *acc*: $acc\ s = acc\ s'$
show $field\text{-access } d\ s = field\text{-access } d\ s'$
using *d subs acc X.access-contained* **by** *blast*
next
fix *d bs s*
assume *d*: $d \in Y$
show $\exists bs'. \forall s'. acc\ s' = acc\ s \longrightarrow field\text{-update } d\ bs\ s' = upd\ bs'\ s'$
using *d subs X.update-contained* **by** *blast*
qed
qed

```

lemma fields-contained-unionD1:
  assumes fields-contained acc upd ( $X \cup Y$ )
  shows fields-contained acc upd  $X$ 
  using assms
  by (auto intro: fields-contained-subset)

```

```

lemma fields-contained-unionD2:
  assumes fields-contained acc upd ( $X \cup Y$ )
  shows fields-contained acc upd  $Y$ 
  using assms
  by (auto intro: fields-contained-subset)

```

```

lemma fields-contained-unionI:
  assumes cont-X: fields-contained acc upd  $X$ 
  assumes cont-Y: fields-contained acc upd  $Y$ 
  shows fields-contained acc upd ( $X \cup Y$ )
proof –
  from cont-X interpret  $X$ : fields-contained acc upd  $X$  .
  from cont-Y interpret  $Y$ : fields-contained acc upd  $Y$  .
  show ?thesis
  proof
    fix  $d s s'$ 
    assume  $d$ :  $d \in X \cup Y$ 
    assume acc:  $acc\ s = acc\ s'$ 
    show field-access  $d\ s = field-access\ d\ s'$ 
      using  $d\ acc\ X.access-contained\ Y.access-contained$  by blast
  next
    fix  $d\ bs\ s$ 
    assume  $d$ :  $d \in X \cup Y$ 
    show  $\exists bs'. \forall s'. acc\ s' = acc\ s \longrightarrow field-update\ d\ bs\ s' = upd\ bs'\ s'$ 
      using  $d\ X.update-contained\ Y.update-contained$  by blast
  qed
qed

```

```

lemma fields-contained-insertI:
  assumes cont-x: fields-contained acc upd  $\{x\}$ 
  assumes cont-Y: fields-contained acc upd  $Y$ 
  shows fields-contained acc upd (insert  $x\ Y$ )
  using fields-contained-unionI [OF cont-x cont-Y]
  by simp

```

```

lemma fields-contained-toplevel-to-field-descs:
   $\wedge(acc::'a \Rightarrow byte\ list \Rightarrow byte\ list)\ upd.$ 

```

```

    contained-field-descs t  $\implies$  fields-contained acc upd (set (toplevel-field-descs t))
     $\implies$  fields-contained acc upd (set (field-descs t))
     $\wedge$ (acc::'a  $\implies$  byte list  $\implies$  byte list) upd.
    contained-field-descs-struct st  $\implies$  fields-contained acc upd (set (toplevel-field-descs-struct
st))
     $\implies$  fields-contained acc upd (set (field-descs-struct st))
     $\wedge$ (acc::'a  $\implies$  byte list  $\implies$  byte list) upd.
    contained-field-descs-list fs  $\implies$  fields-contained acc upd (set (toplevel-field-descs-list
fs))
     $\implies$  fields-contained acc upd (set (field-descs-list fs))
     $\wedge$ (acc::'a  $\implies$  byte list  $\implies$  byte list) upd.
    contained-field-descs-tuple f  $\implies$  fields-contained acc upd (set (toplevel-field-descs-tuple
f))
     $\implies$  fields-contained acc upd (set (field-descs-tuple f))
proof (induct t and st and fs and f)
case (TypDesc st n)
  then show ?case by simp
next
  case (TypScalar st align d)
  then show ?case by simp
next
  case (TypAggregate fs)
  then show ?case by simp
next
  case Nil-typ-desc
  then show ?case by simp
next
  case (Cons-typ-desc f fs)
  from Cons-typ-desc.prem obtain
    fcont-f-fs: fields-contained acc upd
      (set (toplevel-field-descs-tuple f)  $\cup$  set (toplevel-field-descs-list fs))
and
  cont-f: contained-field-descs-tuple f and
  cont-fs: contained-field-descs-list fs by clarsimp

  note fcont-f = fields-contained-unionD1 [OF fcont-f-fs]
  note fcont-fs = fields-contained-unionD2 [OF fcont-f-fs]

  from Cons-typ-desc.hyps(1) [OF cont-f fcont-f] Cons-typ-desc.hyps(2) [OF cont-fs
fcont-fs]
  have fields-contained acc upd (set (field-descs-tuple f)  $\cup$  set (field-descs-list fs))
    by (rule fields-contained-unionI)

  then show ?case
    by clarsimp
next
  case (DTuple-typ-desc ft fn d)
  then show ?case

```

by (*fastforce intro: fields-contained-insertI fields-contained-transitive*)
qed

lemma *fu-commutes-intro*:
assumes $\bigwedge v \text{ bs bs}'. f \text{ bs } (g \text{ bs}' v) = g \text{ bs}' (f \text{ bs } v)$
shows *fu-commutes f g*
using *assms* **by** (*simp add: fu-commutes-def*)

lemma (**in** *wf-field-desc*) *access-same-update-id*:
assumes *field-access d (field-update d bs v) bs' = field-access d v bs'*
shows *field-update d bs v = v*
by (*metis assms double-update update-access*)

lemma (**in** *wf-field-desc*) *access-same-update-id'*:
assumes *upd-inv: field-update d bs v = v*
assumes *acc-same: field-access d v bs = field-access d v' bs*
shows *field-update d bs v' = v'*
by (*metis acc-same access-update double-update upd-inv update-access*)

lemma (**in** *wf-field-desc*) *update-access-unequal*:
assumes *neq-upd: field-update d bs v \neq v*
shows *field-access d v bs' \neq field-access d (field-update d bs v) bs'*
by (*metis access-same-update-id neq-upd*)

lemma (**in** *wf-field-desc*) *access-eq-update-eq*:
assumes $\bigwedge \text{bs}'. \text{field-access } d \text{ (field-update } d \text{ bs } v) \text{ bs}' = \text{field-access } d \text{ v bs}'$
shows *field-update d bs v = v*
by (*metis assms double-update update-access*)

lemma *field-desc-independent-contained-transitive*:
assumes *cont: fields-contained (field-access e) (field-update e) D*
assumes *ind: field-desc-independent (field-access d) (field-update d) {e}*
shows *field-desc-independent (field-access d) (field-update d) D*
proof –
from *cont* **interpret** *cont-e: fields-contained field-access e field-update e D* .
from *ind* **interpret** *ind-e: field-desc-independent field-access d field-update d {e}*
show *?thesis*
proof
fix *f*
assume *f-in: f \in D*
show *fu-commutes (field-update d) (field-update f)*

```

proof (rule fu-commutes-intro)
  fix v bs bs'

  from ind-e.acc-upd-new [of e]
  have e-d-ind: field-access e (field-update d bs v) = field-access e v
    by (rule ext) auto

  from cont-e.update-contained [OF f-in, where s=v] e-d-ind
  obtain bs1 where f-e-v: field-update f bs' v = field-update e bs1 v and
    f-e-upd-v: field-update f bs' (field-update d bs v) = field-update e
bs1 (field-update d bs v)

    by blast
  have field-update d bs (field-update f bs' v) = field-update d bs (field-update e
bs1 v)
    by (simp add: f-e-v)
  also
  from ind-e.fu-commutes [of e] have ... = field-update e bs1 (field-update d
bs v)
    by (auto simp add: fu-commutes-def)
  also have ... = field-update f bs' (field-update d bs v) by (simp add: f-e-upd-v)
  finally
  show field-update d bs (field-update f bs' v) =
    field-update f bs' (field-update d bs v).

qed
next
fix f bs v bs'
assume f-in: f ∈ D
show field-access d (field-update f bs v) bs' = field-access d v bs'
  by (metis cont-e.update-contained f-in ind-e.acc-upd-old insertI1)
next
fix f bs' v bs
assume f-in: f ∈ D
show field-access f (field-update d bs' v) bs = field-access f v bs
proof –
  from ind-e.acc-upd-new [of e]
  have e-d-ind: field-access e (field-update d bs' v) = field-access e v
    by (rule ext) auto
  from cont-e.access-contained [OF f-in e-d-ind]
  show ?thesis by simp
qed
qed
qed

```

```

lemma field-desc-independent-contained-toplevel-to-field-descs:
assumes cont-fs: contained-field-descs-list fs
assumes ind-tl: field-desc-independent (field-access d) (field-update d)
  (set (toplevel-field-descs-list fs))

```

```

shows field-desc-independent (field-access d) (field-update d)
  (set (field-descs-list fs))
using cont-fs ind-tl
proof (induct fs)
  case Nil
  then show ?case by simp
next
  case (Cons f fs)
  obtain ft fn d' where f: f = DTuple ft fn d' by (cases f) simp

from Cons.prems obtain
  ind-d-tl: field-desc-independent (field-access d) (field-update d)
    (insert d' (set (toplevel-field-descs-list fs))) and
  cont-ft: contained-field-descs ft and
  cont-d'-ft: fields-contained (field-access d') (field-update d')
    (set (toplevel-field-descs ft)) and
  cont-fs: contained-field-descs-list fs
  by (simp add: f)

note ind-d-fs-tl = field-desc-independent-insertD2 [OF ind-d-tl]

from Cons.hyps [OF cont-fs ind-d-fs-tl]
have ind-d-fs: field-desc-independent (field-access d) (field-update d)
  (set (field-descs-list fs)).

from fields-contained-toplevel-to-field-descs(1) [OF cont-ft cont-d'-ft]
have fcont-d'-ft: fields-contained (field-access d') (field-update d')
  (set (field-descs ft)) .

from field-desc-independent-insertD1 [OF ind-d-tl]
have ind-d-d': field-desc-independent (field-access d) (field-update d) {d'} .

from field-desc-independent-contained-transitive [OF fcont-d'-ft ind-d-d']
have field-desc-independent (field-access d) (field-update d) (set (field-descs ft))
.

from field-desc-independent-unionI [OF this ind-d-fs] field-desc-independent-insertD1
[OF ind-d-tl]
have field-desc-independent (field-access d) (field-update d)
  (insert d' (set (field-descs ft)  $\cup$  set (field-descs-list fs)))
  by (blast intro: field-desc-independent-insertI)
then show ?case
  by (clarsimp simp add: f)
qed

theorem component-descs-independent-contained-wf-lf:
  fixes t::'a xtyp-info and

```



```

  st::'a xtyp-info-struct and
  fs::'a xtyp-info-tuple list and
  f::'a xtyp-info-tuple
shows
 $\wedge ps. \llbracket wf\text{-desc } t; wf\text{-component-descs } t; wf\text{-field-descs } (set (field\text{-descs } t)); component\text{-descs-independent } t; contained\text{-field-descs } t \rrbracket$ 
 $\implies wf\text{-lf } (lf\text{-set } t ps)$ 
 $\wedge ps. \llbracket wf\text{-desc-struct } st; wf\text{-component-descs-struct } st; wf\text{-field-descs } (set (field\text{-descs-struct } st)); component\text{-descs-independent-struct } st; contained\text{-field-descs-struct } st \rrbracket$ 
 $\implies wf\text{-lf } (lf\text{-set-struct } st ps)$ 
 $\wedge ps. \llbracket wf\text{-desc-list } fs; wf\text{-component-descs-list } fs; wf\text{-field-descs } (set (field\text{-descs-list } fs)); component\text{-descs-independent-list } fs; contained\text{-field-descs-list } fs \rrbracket$ 
 $\implies wf\text{-lf } (lf\text{-set-list } fs ps)$ 
 $\wedge ps. \llbracket wf\text{-desc-tuple } f; wf\text{-component-descs-tuple } f; wf\text{-field-descs } (set (field\text{-descs-tuple } f)); component\text{-descs-independent-tuple } f; contained\text{-field-descs-tuple } f \rrbracket$ 
 $\implies wf\text{-lf } (lf\text{-set-tuple } f ps)$ 
proof (induct t and st and fs and f)
case (TypDesc align st n)
  then show ?case by simp
next
  case (TypScalar sz align d)
  then show ?case by (simp add: wf-field-desc-wf-lf)
next
  case (TypAggregate fs)
then show ?case by simp
next
  case Nil-typ-desc
  then show ?case by simp
next
  case (Cons-typ-desc f fs)
obtain ft' fn' d' where f: f = DTuple ft' fn' d' by (cases f) simp
from Cons-typ-desc.prem obtain
  wf-f: wf-desc-tuple f and
  wf-fs: wf-desc-list fs and
  wf-cd-f: wf-component-descs-tuple f and
  wf-cd-fs: wf-component-descs-list fs and
  wf-fd-f: wf-field-descs (set (field-descs-tuple f)) and
  wf-fd-fs: wf-field-descs (set (field-descs-list fs)) and
  ind-f-fs: field-descs-independent
  (toplevel-field-descs-tuple f @ toplevel-field-descs-list fs) and
  cont-f: contained-field-descs-tuple f and
  cont-fs: contained-field-descs-list fs and
  dist-names: dt-snd f  $\notin$  dt-snd ' set fs and
  ind-f: component-descs-independent-tuple f and
  ind-fs: component-descs-independent-list fs

```

by *clarsimp*

from *Cons-typ-desc.hyps(1)* [*OF wf-f wf-cd-f wf-fd-f ind-f cont-f*]
have *wf-lf-ft'*: *wf-lf (lf-set ft' (ps @ [fn']))* **by** (*simp add: f*)

from *Cons-typ-desc.hyps(2)* [*OF wf-fs wf-cd-fs wf-fd-fs ind-fs cont-fs*]
have *wf-lf-fs*: *wf-lf (lf-set-list fs ps)* .

from *field-descs-independent-append-first-ind* [*OF ind-f-fs, of d'*]
have *field-desc-independent (field-access d') (field-update d')*
(set (toplevel-field-descs-list fs)) **by** (*simp add: f*)

from *cont-f field-desc-independent-contained-toplevel-to-field-descs* [*OF cont-fs,*
where *d=d'*] *this*
have *ind-d'-fs*: *field-desc-independent (field-access d') (field-update d')*
(set (field-descs-list fs))
by (*simp add: f*)

from *cont-f fields-contained-toplevel-to-field-descs*
have *cont-d'-ft'*: *fields-contained (field-access d') (field-update d') (set (field-descs*
ft'))
by (*auto simp add: f*)

have *ti-ind*: *ti-ind' (lf-set ft' (ps @ [fn'])) (lf-set-list fs ps)*
proof
fix *x y*
assume *x-in*: *x ∈ lf-set ft' (ps @ [fn'])*
assume *y-in*: *y ∈ lf-set-list fs ps*
from *lf-set-subset-field-descs(1)* *x-in*
have *x-in'*: *lf-fd x ∈ set (field-descs ft')* **by** *blast*
from *lf-set-subset-field-descs(3)* *y-in*
have *y-in'*: *lf-fd y ∈ set (field-descs-list fs)* **by** *blast*

from *field-desc-independent-symmetric-singleton* [*OF ind-d'-fs y-in'*]
have *field-desc-independent (field-access (lf-fd y)) (field-update (lf-fd y)) {d'}* .

from *field-desc-independent-contained-transitive* [*OF cont-d'-ft' this*]
have *field-desc-independent (field-access (lf-fd y)) (field-update (lf-fd y))*
(set (field-descs ft')) .

from *field-desc-independent.fu-commutes* [*OF this x-in'*] *fu-commutes-sym*
show *fu-commutes (field-update (lf-fd x)) (field-update (lf-fd y))*
by *blast*

next
fix *x y*
assume *x-in*: *x ∈ lf-set ft' (ps @ [fn'])*
assume *y-in*: *y ∈ lf-set-list fs ps*

```

from lf-set-subset-field-descs(1) x-in
have x-in': lf-fd x ∈ set (field-descs ft') by blast
from lf-set-subset-field-descs(3) y-in
have y-in': lf-fd y ∈ set (field-descs-list fs) by blast

from field-desc-independent-symmetric-singleton [OF ind-d'-fs y-in']
have field-desc-independent (field-access (lf-fd y)) (field-update (lf-fd y)) {d'} .

from field-desc-independent-contained-transitive [OF cont-d'-ft' this]
have field-desc-independent (field-access (lf-fd y)) (field-update (lf-fd y))
  (set (field-descs ft')) .
from field-desc-independent.acc-upd-new [OF this x-in']
show fa-fu-ind (lf-fd x) (lf-fd y) (lf-sz y) (lf-sz x)
  by (auto simp add: fa-fu-ind-def)
next
fix x y
assume x-in: x ∈ lf-set ft' (ps @ [fn'])
assume y-in: y ∈ lf-set-list fs ps
from lf-set-subset-field-descs(1) x-in
have x-in': lf-fd x ∈ set (field-descs ft') by blast
from lf-set-subset-field-descs(3) y-in
have y-in': lf-fd y ∈ set (field-descs-list fs) by blast

from field-desc-independent-symmetric-singleton [OF ind-d'-fs y-in']
have field-desc-independent (field-access (lf-fd y)) (field-update (lf-fd y)) {d'} .

from field-desc-independent-contained-transitive [OF cont-d'-ft' this]
have field-desc-independent (field-access (lf-fd y)) (field-update (lf-fd y))
  (set (field-descs ft')) .
from field-desc-independent.acc-upd-old [OF this x-in']
show fa-fu-ind (lf-fd y) (lf-fd x) (lf-sz x) (lf-sz y)
  by (auto simp add: fa-fu-ind-def)
qed

from wf-lf-unionI [OF wf-lf-ft' wf-lf-fs ti-ind]
have wf-lf (lf-set ft' (ps @ [fn']) ∪ lf-set-list fs ps) .

then show ?case by (simp add: f)

next
case (DTuple-typ-desc ft fn d)
then show ?case by simp
qed

definition wf-align-field ti ≡ wf-align ti ∧ align-field ti

lemma wf-align-field-empty-typ-info: wf-align-field (empty-typ-info algn n)

```

by (*simp add: wf-align-field-def wf-align-simps align-field-empty-typ-info*)

lemma (*in mem-type*) *wf-align-field-ti-typ-pad-combine*:

aggregate ti \implies *wf-align-field ti* \implies

wf-align-field (ti-typ-pad-combine (TYPE('a)) acc upd algn fn ti)

by (*auto simp add: wf-align-field-def wf-align-ti-typ-pad-combine align-field-ti-typ-pad-combine*)

lemma (*in mem-type*) *wf-align-field-ti-typ-combine*:

aggregate ti \implies *wf-align-field ti* \implies 2^{\wedge} *align-td (typ-info-t TYPE('a)) dvd size-td ti* \implies

wf-align-field (ti-typ-combine (TYPE('a)) acc upd algn fn ti)

by (*auto simp add: wf-align-field-def wf-align-ti-typ-combine align-field-ti-typ-combine*)

lemma *wf-align-field-ti-pad-combine*: *aggregate ti* \implies *wf-align-field ti* \implies *wf-align-field (ti-pad-combine n ti)*

by (*auto simp add: wf-align-field-def wf-align-ti-pad-combine align-field-ti-pad-combine*)

lemma *wf-align-field-final-pad*: *aggregate ti* \implies *wf-align-field ti* \implies *wf-align-field (final-pad algn ti)*

by (*auto simp add: wf-align-field-def wf-align-final-pad align-field-final-pad*)

lemmas *wf-align-field-simps* =

wf-align-field-empty-typ-info

wf-align-field-ti-typ-pad-combine

wf-align-field-ti-typ-combine

wf-align-field-ti-pad-combine

wf-align-field-final-pad

The following theorem is used to prove that a new type is in class *xmem-contained-type*, for which we have constructed the extended type info with:

- *empty-typ-info*
- *ti-typ-pad-combine (ti-typ-combine, ti-pad-combine)*
- *final-pad*.

Note that the field-types are already in *xmem-contained-type*.

theorem *tuned-xmem-contained-type-class-intro*:

assumes *wf-desc*: *wf-desc (typ-info-t TYPE('a))*

assumes *wf-size-desc*: *wf-size-desc (typ-info-t TYPE('a))*

assumes *align-dvd*: *align-of TYPE('a) dvd size-of TYPE('a)*

assumes *wf-align-field*: *wf-align-field (typ-info-t TYPE('a))*

assumes *size*: *size-of TYPE('a) < addr-card*

assumes *entire-update*: $\bigwedge bs v w. \text{length } bs = \text{size-of } TYPE('a)$

\implies *field-update (component-desc (typ-info-t TYPE('a))) bs v =*

field-update (component-desc (typ-info-t TYPE('a))) bs w

assumes *wf-component-descs*: *wf-component-descs (typ-info-t TYPE('a))*

```

assumes ind: component-descs-independent (typ-info-t TYPE('a))
assumes wf-field-descs: wf-field-descs (set (field-descs (typ-info-t TYPE('a))))
assumes contained-field-descs: contained-field-descs (typ-info-t TYPE('a))
assumes wf-padding: wf-padding (typ-info-t TYPE('a))
shows OFCLASS('a::c-type, xmem-contained-type-class)
proof
  show wf-desc (typ-info-t TYPE('a))
    by (rule wf-desc)
next
  show wf-size-desc (typ-info-t TYPE('a))
    by (rule wf-size-desc)
next
  show wf-lf (lf-set (typ-info-t TYPE('a)) [])
  proof –

```

We can show that all leaves of the tree are disjoint, by exploiting the correct nesting of fields in the *toplevel-structure*.

```

from component-descs-independent-contained-wf-lf(1) [OF wf-desc wf-component-descs
  wf-field-descs ind contained-field-descs]
  show ?thesis .
qed
next
  fix bs v w
  show length bs = size-of TYPE('a) →
    update-ti-t (typ-info-t TYPE('a)) bs v =
    update-ti-t (typ-info-t TYPE('a)) bs w
  proof

```

The updates are captured by the component-descriptor of the toplevel structure.

```

  assume len: length bs = size-of TYPE('a)
  from entire-update [OF len] wf-field-descs len wf-component-descs
  show update-ti-t (typ-info-t TYPE('a)) bs v =
    update-ti-t (typ-info-t TYPE('a)) bs w
    by (simp add: ind size-of-def update-ti-component-desc-compatible(1) up-
date-ti-t-def)
  qed
next
  show align-of TYPE('a) dvd size-of TYPE('a) by (rule align-dvd)
next
  from wf-align-field
  show align-field (typ-info-t TYPE('a)) by (simp add: wf-align-field-def)
next
  show size-of TYPE('a) < addr-card by (rule size)
next
  show wf-component-descs (typ-info-t TYPE('a)) by (rule wf-component-descs)
next
  show component-descs-independent (typ-info-t TYPE('a)) by (rule ind)
next

```

```

show wf-field-descs (set (field-descs (typ-info-t TYPE('a)))) by (rule wf-field-descs)
next
show contained-field-descs (typ-info-t TYPE('a)) by (rule contained-field-descs)
next
from wf-align-field
show wf-align (typ-info-t TYPE('a))
by (simp add: wf-align-field-def)
next
from wf-padding
show wf-padding (typ-info-t TYPE('a)) .
qed

```

```

lemma field-sz-extend-ti: aggregate t  $\implies$ 
  field-sz (component-desc (extend-ti t ft algn fn d)) = field-sz (component-desc t)
+ field-sz d
apply (cases t)
subgoal for x1 st n
apply (cases st)
apply (simp)
subgoal for fs
apply simp
done
done
done

```

```

lemma field-sz-empty-typ-info: field-sz (component-desc (empty-typ-info algn n))
= 0
by (simp add: empty-typ-info-def)

```

```

lemma (in c-type) field-sz-ti-typ-combine:
fixes acc:: 's  $\Rightarrow$  'a
assumes agg: aggregate t
shows field-sz (component-desc (ti-typ-combine ft acc upd algn fn t)) =
  field-sz (component-desc t) + size-of TYPE('a)
using agg
by (simp add: ti-typ-combine-def Let-def field-sz-extend-ti )

```

```

lemma (in c-type) field-sz-ti-pad-combine:
fixes acc:: 's  $\Rightarrow$  'a
assumes agg: aggregate t
shows field-sz (component-desc (ti-pad-combine n t)) =
  field-sz (component-desc t) + n
using agg
by (simp add: ti-pad-combine-def Let-def field-sz-extend-ti padding-desc-def)

```

```

lemma (in c-type) field-sz-ti-typ-pad-combine:
fixes acc:: 's  $\Rightarrow$  'a
assumes agg: aggregate t
shows field-sz (component-desc (ti-typ-pad-combine ft acc upd algn fn t)) =

```

$field\text{-}sz\ (component\text{-}desc\ t) + size\text{-}of\ TYPE('a) + padup\ (max\ (2 \wedge\ algn)$
 $(align\text{-}of\ TYPE('a)))\ (size\text{-}td\ t)$
using *agg*
by (*simp add: ti-typ-pad-combine-def Let-def field-sz-ti-pad-combine field-sz-ti-typ-combine*)

lemma *component-desc-map-align[simp]:component-desc (map-align f t) = component-desc t*
by (*cases t) simp*

lemma *field-sz-final-pad:*
assumes *agg: aggregate t*
shows $field\text{-}sz\ (component\text{-}desc\ (final\text{-}pad\ algn\ t)) =$
 $field\text{-}sz\ (component\text{-}desc\ t) + padup\ (2 \wedge (max\ algn\ (align\text{-}td\ t)))\ (size\text{-}td\ t)$
using *agg*
by (*simp add: final-pad-def Let-def field-sz-ti-pad-combine*)

lemma *split-fold-append: split-fold f (xs@ys) bs s =*
(let (s', bs') = split-fold f xs bs s in split-fold f ys bs' s')
apply (*induct xs arbitrary: bs ys s*)
apply *simp*
apply *clarsimp*
by (*metis prod.case-distrib*)

lemma *apply-field-updates-append: apply-field-updates (xs@ys) bs s =*
(let (s', bs') = apply-field-updates xs bs s in apply-field-updates ys bs' s')
by (*simp add: apply-field-updates-def split-fold-append*)

lemma *snd-apply-field-updates: snd (apply-field-updates ds bs s) = (drop (foldl (+)*
 $0\ (map\ field\text{-}sz\ ds))\ bs)$
apply (*induct ds arbitrary: bs s*)
apply *simp*
apply (*simp add: add.commute sum-nat-foldl*)
done

lemma *field-update-extend-ti:*
assumes *agg: aggregate t*
shows $field\text{-}update\ (component\text{-}desc\ (extend\text{-}ti\ t\ ft\ algn\ fn\ d))\ bs\ v =$
 $field\text{-}update\ d\ ((drop\ (field\text{-}sz\ (component\text{-}desc\ t)))\ bs)$
 $(field\text{-}update\ (component\text{-}desc\ t)\ bs\ v)$
proof (*cases t*)
case (*TypDesc algn st n*)
show *?thesis*
proof (*cases st*)
case (*TypScalar sz align d*)
with *agg show ?thesis by (simp add: TypDesc)*

```

next
  case (TypAggregate fs)
  from snd-apply-field-updates
  have fst (apply-field-updates (map dt-trd fs @ [d]) bs v) =
    field-update d (drop (foldl (+) 0 (map (field-sz ∘ dt-trd) fs)) bs)
    (fst (apply-field-updates (map dt-trd fs) bs v))
  by (simp add: apply-field-updates-append case-prod-beta snd-apply-field-updates)

  then show ?thesis by (simp add: TypDesc TypAggregate)
qed

```

```

lemma field-update-empty-typ-info: field-update (component-desc (empty-typ-info
  algn n)) bs s = s
  by (simp add: empty-typ-info-def)

```

```

lemma (in c-type) field-update-ti-typ-combine:
  assumes agg: aggregate t
  shows field-update (component-desc (ti-typ-combine (ft::'a itself) acc upd algn fn
  t)) bs v =
    (upd o xfrom-bytes) ((drop (field-sz (component-desc t))) bs)
    (field-update (component-desc t) bs v)
  using agg
  by (simp add: ti-typ-combine-def field-update-extend-ti)

```

```

lemma field-update-ti-pad-combine:
  assumes agg: aggregate t
  shows field-update (component-desc (ti-pad-combine n t)) bs v =
    field-update (component-desc t) bs v
  using agg
  by (simp add: ti-pad-combine-def Let-def field-update-extend-ti padding-desc-def)

```

```

lemma (in c-type) field-update-ti-typ-pad-combine:
  fixes acc:: 's ⇒ 'a
  assumes agg: aggregate t
  shows field-update (component-desc (ti-typ-pad-combine ft acc upd algn fn t)) bs
  v =
    (upd o xfrom-bytes) (drop (field-sz (component-desc t) + padup (max (2 ^
  algn) (align-of TYPE('a))) (size-td t)) bs)
    (field-update (component-desc t) bs v)
  using agg
  apply (clarsimp simp add: ti-typ-pad-combine-def Let-def field-update-ti-pad-combine
  field-update-ti-typ-combine)
  apply (clarsimp simp add: ti-pad-combine-def field-update-extend-ti Let-def field-sz-extend-ti
  padding-desc-def)
  done

```


lemma *field-update-final-pad*:
assumes *agg*: aggregate *t*
shows *field-update* (*component-desc* (*final-pad* *algn* *t*)) *bs* *v* =
field-update (*component-desc* *t*) *bs* *v*
using *agg*
by (*simp* *add*: *final-pad-def* *field-update-ti-pad-combine* *Let-def*)

lemma *set-toplevel-field-descs-extend-ti*:
 $set\ (toplevel-field-descs\ (extend-ti\ t\ ft\ algn\ fn\ d)) \subseteq insert\ d\ (set\ (toplevel-field-descs\ t))$
apply (*cases* *t*)
subgoal **for** *x1* *st* *n*
apply (*cases* *st*)
apply *simp*
subgoal **for** *fs*
apply (*cases* *fs*)
apply *simp*
subgoal **for** *f* *fs'*
apply *simp*
done
done
done
done

lemma *set-toplevel-field-descs-extend-ti-aggregate*:
 $aggregate\ t \implies set\ (toplevel-field-descs\ (extend-ti\ t\ ft\ algn\ fn\ d)) = insert\ d\ (set\ (toplevel-field-descs\ t))$
apply (*cases* *t*)
subgoal **for** *x1* *st* *n*
apply (*cases* *st*)
apply *simp*
subgoal **for** *fs*
apply (*cases* *fs*)
apply *simp*
subgoal **for** *f* *fs'*
apply *simp*
done
done
done
done

lemma (**in** *c-type*) *set-toplevel-field-descs-ti-typ-combine*:
 $set\ (toplevel-field-descs\ (ti-typ-combine\ ft\ (acc::'b \Rightarrow 'a)\ upd\ algn\ fn\ t)) \subseteq$
 $insert\ (\lfloor field-access = xto-bytes \circ acc, field-update = upd \circ xfrom-bytes, field-sz =$
 $size-of\ TYPE('a) \rfloor)$
 $(set\ (toplevel-field-descs\ t))$
apply (*simp* *add*: *ti-typ-combine-def*)
apply (*rule* *set-toplevel-field-descs-extend-ti*)
done

```

lemma (in c-type) set-toplevel-field-descs-ti-typ-combine-aggregate:
  fixes acc::'s  $\Rightarrow$  'a
  assumes agg: aggregate t
  shows set (toplevel-field-descs (ti-typ-combine ft acc upd algn fn t)) =
    insert (field-access = xto-bytes  $\circ$  acc, field-update = upd  $\circ$  xfrom-bytes,
field-sz = size-of TYPE('a))
    (set (toplevel-field-descs t))
  using agg
  apply (simp add: ti-typ-combine-def)
  apply (rule set-toplevel-field-descs-extend-ti-aggregate)
  apply simp
  done

```

```

lemma set-field-descs-extend-ti-aggregate:
aggregate t  $\Rightarrow$  set (field-descs (extend-ti t ft algn fn d)) = insert d (set (field-descs
ft)  $\cup$  set (field-descs t))
  apply (cases t)
  subgoal for x1 st n
    apply (cases st)
    apply simp
  subgoal for fs
    apply (cases fs)
    apply simp
  subgoal for f fs'
    apply auto
  done
  done
  done
  done

```

```

lemma (in c-type) set-field-descs-ti-typ-combine-aggregate:
  fixes acc::'s  $\Rightarrow$  'a
  assumes agg: aggregate t
  shows set (field-descs (ti-typ-combine ft acc upd algn fn t)) =
    insert (field-access = xto-bytes  $\circ$  acc, field-update = upd  $\circ$  xfrom-bytes,
field-sz = size-of TYPE('a))
    (set (field-descs (adjust-ti (typ-info-t TYPE('a)) acc upd))  $\cup$  set (field-descs
t))
  using agg
  apply (simp add: ti-typ-combine-def)
  apply (simp add: set-field-descs-extend-ti-aggregate)
  done

```

```

locale padding=
  fixes d::'a field-desc
  assumes independent: field-desc-independent acc upd {d}

```

```

lemma pad-is-padding: padding (padding-desc n)
  unfolding padding-desc-def
  apply (rule padding.intro)
  apply (rule field-desc-independent.intro)
  apply (auto simp add: fu-commutes-def)
  done

definition PAD::'a field-desc set where PAD = {d. padding d}

lemma in-PAD-iff[iff]: d ∈ PAD ↔ (padding d) by (auto simp add: PAD-def)

lemma (in c-type) set-toplevel-field-descs-ti-pad-combine-aggregate:
  fixes acc::'s ⇒ 'a
  assumes agg: aggregate t
  shows set (toplevel-field-descs (ti-pad-combine n t)) ∪ PAD =
    set (toplevel-field-descs t) ∪ PAD
  using agg
  by (auto simp add: ti-pad-combine-def Let-def set-toplevel-field-descs-extend-ti-aggregate
    pad-is-padding)

lemma (in c-type) set-toplevel-field-descs-ti-typ-pad-combine-aggregate:
  fixes acc::'s ⇒ 'a
  assumes agg: aggregate t
  shows set (toplevel-field-descs (ti-typ-pad-combine ft acc upd algn fn t)) ∪ PAD
  =
    insert (|field-access = xto-bytes ∘ acc, field-update = upd ∘ xfrom-bytes,
field-sz = size-of TYPE('a)|)
    (set (toplevel-field-descs t)) ∪ PAD)
  using agg
  apply (simp add: ti-typ-pad-combine-def Let-def)
  apply (cases  $0 < \text{padup } (\max (2 \wedge \text{algn}) (\text{align-of TYPE('a)})) (\text{size-td } t))$ )
  apply clarsimp
  using set-toplevel-field-descs-ti-pad-combine-aggregate [OF agg, of (padup ( $\max$ 
 $(2 \wedge \text{algn}) (\text{align-of TYPE('a)})) (\text{size-td } t)$ )]
    set-toplevel-field-descs-ti-typ-combine-aggregate [OF aggregate-ti-pad-combine
  [of (padup ( $\max (2 \wedge \text{algn}) (\text{align-of TYPE('a)})) (\text{size-td } t)$ ) t], of ft acc upd algn
  fn]
  apply clarsimp
  apply (simp add: set-toplevel-field-descs-ti-typ-combine-aggregate)
  done

lemma toplevel-field-descs-map-align[simp]: toplevel-field-descs (map-align f t) =
  toplevel-field-descs t
  by (cases t) simp

lemma set-toplevel-field-descs-final-pad-aggregate:
  assumes agg: aggregate t
  shows set (toplevel-field-descs (final-pad algn t)) ∪ PAD =

```

$set (toplevel-field-descs t) \cup PAD$
using *agg*
by (*auto simp add: final-pad-def Let-def set-toplevel-field-descs-ti-pad-combine-aggregate*)

lemma *set-toplevel-field-descs-empty-typ-info*: $set (toplevel-field-descs (empty-typ-info\ align\ n)) = \{\}$
by (*simp add: empty-typ-info-def*)

lemmas *set-toplevel-field-descs-combinator-simps* =
set-toplevel-field-descs-empty-typ-info
set-toplevel-field-descs-ti-typ-combine-aggregate
set-toplevel-field-descs-ti-typ-pad-combine-aggregate
set-toplevel-field-descs-final-pad-aggregate

lemma *field-descs-independent-PAD*:
 $field-desc-independent\ acc\ upd\ (D \cup PAD) = field-desc-independent\ acc\ upd\ D$
by (*auto simp add: field-desc-independent-def padding-def*)

lemma *field-desc-independent-PAD-expand*:
 $field-desc-independent\ acc\ upd\ (D \cup PAD) \implies field-desc-independent\ acc\ upd\ D$
by (*simp add: field-descs-independent-PAD*)

lemma *field-desc-independent-PAD-collapse*:
 $field-desc-independent\ acc\ upd\ D \implies field-desc-independent\ acc\ upd\ (D \cup PAD)$
by (*simp add: field-descs-independent-PAD*)

lemma *insert-union-out*: $insert\ d\ (X \cup Y) = insert\ d\ X \cup Y$
by *blast*

lemmas *field-sz-typ-combinators-simps* =
field-sz-final-pad
field-sz-ti-typ-pad-combine
field-sz-ti-typ-combine
field-sz-empty-typ-info

lemmas *field-update-typ-combinators-simps* =
field-update-final-pad
field-update-ti-typ-pad-combine
field-update-ti-typ-combine
field-update-empty-typ-info

lemma *aggregate-map-align[simp]*: $aggregate\ (map-align\ f\ t) = aggregate\ t$
by (*cases\ t\ simp*)

lemma *aggregate-final-pad[simp]*: $aggregate\ t \implies aggregate\ (final-pad\ align\ t)$
by (*simp\ add: final-pad-def Let-def*)

```

lemmas aggregate-typ-combinators-simps =
  aggregate-empty-typ-info
  aggregate-ti-pad-combine
  aggregate-ti-typ-pad-combine
  aggregate-final-pad

lemma wf-padding-empty-typ-info: wf-padding (empty-typ-info algn tn)
  by (simp add: empty-typ-info-def)

lemma is-padding-tag-padding-tag[simp]: is-padding-tag (padding-tag n)
  by (simp add: is-padding-tag-def padding-tag-def)

lemma is-padding-tag-pad-typ[simp]: is-padding-tag (TypDesc 0 (TypScalar n 0
(padding-desc n)) "\pad-typ")
  by (simp add: is-padding-tag-def padding-tag-def)

lemma wf-padding-padding-tag: wf-padding (padding-tag n)
  by (simp add: padding-tag-def)

lemma wf-padding-tuple-padI:
  is-padding-tag s  $\implies$  wf-padding-tuple (DTuple s (CHR "\'##xs) d)
  by (auto simp add: is-padding-tag-def wf-padding-padding-tag)

lemma wf-padding-tuple-no-padI:
   $\neg$  padding-field-name fn  $\implies$  wf-padding s  $\implies$  wf-padding-tuple (DTuple s fn d)
  by (simp)

lemma wf-padding-extend-ti:
  fixes
  t :: 'a xtyp-info and
  st :: 'a xtyp-info-struct and
  ts :: 'a xtyp-info-tuple list and
  x :: 'a xtyp-info-tuple
  shows
  wf-padding t  $\implies$  wf-padding-tuple (DTuple s fn d)  $\implies$ 
    wf-padding (extend-ti t s n fn d)

  wf-padding-struct st  $\implies$  wf-padding-tuple (DTuple s fn d)  $\implies$ 
    wf-padding-struct (extend-ti-struct st s fn d)

  wf-padding-list ts  $\implies$  wf-padding-tuple (DTuple s fn d)  $\implies$ 
    wf-padding-list (ts @ [DTuple s fn d])

  wf-padding-tuple x  $\implies$  wf-padding-tuple x
  by (induct t and st and ts and x) auto

lemma is-padding-tag-update-desc: is-padding-tag t  $\implies$ 

```

$(\bigwedge x. \text{upd } (\text{acc } x) x = x) \implies$
is-padding-tag
 $(\text{map-td } (\lambda n \text{ algn. update-desc acc upd}) (\text{update-desc acc upd}) t)$
by (*auto simp add: is-padding-tag-def padding-tag-def padding-desc-def update-desc-def fun-eq-iff*)

lemma *wf-padding-adjust-ti:*

fixes

$t :: 'a \text{ xtyp-info}$ **and**

$st :: 'a \text{ xtyp-info-struct}$ **and**

$ts :: 'a \text{ xtyp-info-tuple list}$ **and**

$x :: 'a \text{ xtyp-info-tuple}$

assumes *upd-acc-id:* $(\bigwedge x. \text{upd } (\text{acc } x) x = x)$

shows

wf-padding $t \implies$

wf-padding $(\text{map-td } (\lambda n \text{ algn. update-desc acc upd}) (\text{update-desc acc upd}) t)$

wf-padding-struct $st \implies$

wf-padding-struct $(\text{map-td-struct } (\lambda n \text{ algn. update-desc acc upd}) (\text{update-desc acc upd}) st)$

wf-padding-list $ts \implies$

wf-padding-list $(\text{map-td-list } (\lambda n \text{ algn. update-desc acc upd}) (\text{update-desc acc upd}) ts)$

wf-padding-tuple $x \implies$

wf-padding-tuple $(\text{map-td-tuple } (\lambda n \text{ algn. update-desc acc upd}) (\text{update-desc acc upd}) x)$

apply (*induct t and st and ts and x*)

by (*auto simp add: is-padding-tag-update-desc upd-acc-id*)

lemma *wf-padding-ti-pad-combine:* *wf-padding* $t \implies$ *wf-padding* $(\text{ti-pad-combine } n t)$

by (*simp add: ti-pad-combine-def Let-def wf-padding-extend-ti(1)*)

lemma (*in c-type*) *wf-padding-ti-tyt-combine:*

wf-padding $t \implies$ *wf-padding* $(\text{typ-info-t } \text{TYPE}('a)) \implies (\bigwedge xs. \text{fn} \neq \text{CHR } "!" \# xs)$
 $\implies (\bigwedge x. \text{upd } (\text{acc } x) x = x) \implies$

wf-padding $(\text{ti-tyt-combine } (\text{TYPE}('a)) \text{ acc upd algn fn } t)$

apply (*simp add: ti-tyt-combine-def Let-def*)

apply (*rule wf-padding-extend-ti(1)*)

apply *assumption*

apply (*simp add: adjust-ti-def wf-padding-adjust-ti(1) padding-field-name-def*)

done

lemma (*in c-type*) *wf-padding-ti-tyt-pad-combine:*

wf-padding $t \implies$ *wf-padding* $(\text{typ-info-t } \text{TYPE}('a)) \implies (\bigwedge xs. \text{fn} \neq \text{CHR } "!" \# xs)$

$\implies (\wedge x. \text{upd } (\text{acc } x) \ x = x) \implies$
 $\text{wf-padding } (\text{ti-typ-pad-combine } (\text{TYPE}('a)) \ \text{acc } \text{upd } \text{algn } \text{fn } t)$
by (*simp add: ti-typ-pad-combine-def wf-padding-ti-typ-combine wf-padding-ti-pad-combine Let-def*)

lemma *wf-padding-map-align*: $\text{wf-padding } t \implies \text{wf-padding } (\text{map-align } f \ t)$
by (*cases t*) (*simp add: map-align-def*)

lemma *wf-padding-final-pad*: $\text{wf-padding } t \implies \text{wf-padding } (\text{final-pad } n \ t)$
by (*simp add: final-pad-def wf-padding-ti-pad-combine Let-def wf-padding-map-align*)

lemmas *wf-padding-combinator-simps* =
wf-padding-empty-typ-info
wf-padding-final-pad
wf-padding-ti-typ-pad-combine
wf-padding-ti-typ-combine

end

theory *ArraysMemInstance*
imports *Arrays CompoundCTypes*
begin

primrec (**in** *c-type*)
 $\text{array-tag-n} :: \text{nat} \Rightarrow ('a, 'b :: \text{finite}) \ \text{array} \ \text{xtyp-info}$
where
atn-base:
 $\text{array-tag-n } 0 = ((\text{empty-typ-info } (\text{align-td } (\text{typ-uinfo-t } \text{TYPE}('a)))) (\text{typ-name } (\text{typ-uinfo-t } \text{TYPE}('a)) \ @ \ \text{"array-"} \ @ \ \text{nat-to-bin-string } (\text{CARD}('b :: \text{finite})))) :: ('a, 'b) \ \text{array} \ \text{xtyp-info}$
| *atn-rec*:
 $\text{array-tag-n } (\text{Suc } n) = ((\text{ti-typ-combine } \text{TYPE}('a) \ (\lambda x. \ \text{index } x \ n) \ (\lambda x \ f. \ \text{update } f \ n \ x) \ 0 \ (\text{replicate } n \ \text{CHR } \text{"1"})) (\text{array-tag-n } n)) :: ('a, 'b :: \text{finite}) \ \text{array} \ \text{xtyp-info}$

definition (**in** *c-type*) $\text{array-tag} :: ('a, 'b :: \text{finite}) \ \text{array} \ \text{itself} \Rightarrow ('a, 'b) \ \text{array} \ \text{xtyp-info}$
where
 $\text{array-tag } t \equiv \text{array-tag-n } (\text{CARD}('b))$

lemma (**in** *c-type*) *typ-name-array-tag*: $\text{typ-name } ((\text{array-tag-n } n) :: ('a, 'b :: \text{finite}) \ \text{array} \ \text{xtyp-info}) =$
 $(\text{typ-name } (\text{typ-uinfo-t } \text{TYPE}('a)) \ @ \ \text{"array-"} \ @ \ \text{nat-to-bin-string } (\text{CARD}('b)))$
apply (*induct n*)
apply (*simp add: empty-typ-info-def*)
apply (*simp add: ti-typ-combine-def Let-def adjust-ti-def*)

done

instantiation *array* :: (c-type, finite) c-type
begin

definition *typ-info-array*:
typ-info-t (w::('a::c-type,'b::finite) array itself) ≡ *array-tag* w

definition *typ-name-itself* (w::('a::c-type,'b::finite) array itself) = *typ-name* (*typ-info-t* w)

instance by (*intro-classes*) (*simp add: typ-name-itself-array-def*)
end

lemma (**in** c-type) *field-names-array-tag-length* [*rule-format*]:
 $x \in \text{set } (\text{field-names-list } (\text{array-tag-n } n)) \longrightarrow \text{length } x < n$
by (*induct n*) (*auto*)

lemma (**in** c-type) *replicate-mem-field-names-array-tag* [*simp*]:
 $\text{replicate } n \ x \notin \text{set } (\text{field-names-list } (\text{array-tag-n } n))$
by (*fastforce dest: field-names-array-tag-length*)

lemma (**in** c-type) *aggregate-array-tag* [*simp*]:
 $\text{aggregate } (\text{array-tag-n } n)$
by (*cases n*; *simp add: c-type.ti-typ-combine-def*)

lemma (**in** mem-type) *wf-desc-array-tag* [*simp*]:
 $\text{wf-desc } ((\text{array-tag-n } n)::('a,'b::\text{finite}) \text{ array } \text{xtyp-info})$
by (*induct n*; *simp*) (*fastforce elim: wf-desc-ti-typ-combine*)

lemma (**in** mem-type) *wf-size-desc-array-tag* [*simp*]:
 $0 < n \implies \text{wf-size-desc } ((\text{array-tag-n } n)::('a,'b::\text{finite}) \text{ array } \text{xtyp-info})$
apply (*induct n*; *simp*)
subgoal for *n*
apply (*cases n=0*; *simp*)
apply (*rule wf-size-desc-ti-typ-combine*)
apply *simp*
done
done

lemma (**in** mem-type) *upd-ind-array-tag-udpate* [*simp*]:
 $\llbracket n \leq m; n \leq \text{CARD}('b) \rrbracket \implies$
 $\text{upd-ind } (\text{lf-set } ((\text{array-tag-n } n)::('a,'b::\text{finite}) \text{ array } \text{xtyp-info}) \llbracket \rrbracket) (\lambda x f. \text{update } f$
 $m \ x)$
by (*induct n*) (*auto elim: upd-ind-ti-typ-combine*)

lemma (**in** mem-type) *fc-array-tag-udpate* [*simp*]:
 $\llbracket n \leq m; n \leq \text{CARD}('b) \rrbracket \implies$
 $\text{fu-commutes } (\text{update-ti-t } ((\text{array-tag-n } n)::('a,'b::\text{finite}) \text{ array } \text{xtyp-info})) (\lambda x f.$

update f m x
by (*induct n; fastforce elim: fc-ti-typ-combine simp: fg-cons-def*)

lemma (*in mem-type*) *acc-ind-array-tag-udpate* [*simp*]:
 $\llbracket n \leq m; m < \text{CARD}('b) \rrbracket \implies$
 $\text{acc-ind } (\lambda x. \text{index } x m) (\text{lf-fd } ' \text{ lf-set } ((\text{array-tag-n } n)::('a, 'b)::\text{finite}) \text{ array } x\text{typ-info})$
 $\llbracket \rrbracket$
by (*induct n; fastforce elim: acc-ind-ti-typ-combine*)

lemma (*in mem-type*) *fa-fu-g-array-tag-udpate* [*simp*]:
 $\llbracket n \leq m; m < \text{CARD}('b) \rrbracket \implies$
 $\text{fa-ind } (\text{lf-fd } ' \text{ lf-set } ((\text{array-tag-n } n)::('a, 'b)::\text{finite}) \text{ array } x\text{typ-info}) \llbracket \rrbracket (\lambda x f. \text{update } f m x)$
by (*induct n; fastforce elim: fa-ind-ti-typ-combine*)

lemma (*in mem-type*) *wf-fdp-array-tag* [*simp*]:
 $n \leq \text{CARD}('b) \implies \text{wf-lf } (\text{lf-set } ((\text{array-tag-n } n)::('a, 'b)::\text{finite}) \text{ array } x\text{typ-info})$
 $\llbracket \rrbracket$
by (*induct n; fastforce elim: wf-lf-ti-typ-combine*)

lemma *upd-local-update* [*simp*]:
 $\text{upd-local } (\lambda x f. \text{update } f n x)$
unfolding *upd-local-def*
by (*metis update-update*)

lemma (*in mem-type*) *fu-eq-mask-array-tag* [*simp, rule-format*]:
 $n \leq \text{CARD}('b) \longrightarrow (\forall m. (\forall k v. k < \text{CARD}('b) \longrightarrow$
 $\text{index } ((m v)::('a, 'b) \text{ array}) k = (\text{if } n \leq k \text{ then}$
 $\text{index } (\text{undefined}::('a, 'b)::\text{finite}) \text{ array}) k$
 $\text{else } \text{index } v k)) \longrightarrow \text{fu-eq-mask } (\text{array-tag-n } n) m$
apply(*induct n; clarsimp*)
apply(*rule fu-eq-mask-empty-typ-info*)
apply(*clarsimp simp: array-index-eq*)
subgoal for $n m$
apply(*rule fu-eq-mask-ti-typ-combine; clarsimp?*)
apply (*drule spec* [**where** $x = \lambda v. \text{update } (m v) n (\text{index } (\text{undefined}::'a['b]) n)$])
apply(*erule impE*)
apply *clarsimp*
subgoal for $k v$
by (*cases k=n*) *auto*
subgoal premises *prems*
proof –
from *prems*
have $\forall v bs. m (\text{update } v n bs) = \text{update } (m v) n bs$
apply(*clarsimp simp: array-index-eq*)
subgoal for $v bs i$
apply(*cases i=n; clarsimp*)
apply(*cases Suc n ≤ i; clarsimp*)
done

```

    done
  then show ?thesis using prems by clarsimp
qed
subgoal
  by(clarsimp simp: fg-cons-def)
done
done

lemma (in c-type) size-td-array-tag [simp]:
  size-td (((array-tag-n n)::('a,'b::finite) array xtyp-info)) =
    n * size-of TYPE('a)
  by (induct n; simp add: size-td-lt-ti-typ-combine size-of-def)

lemma (in c-type) align-td-wo-align-array-tag:
  0 < n  $\implies$ 
  align-td-wo-align ((array-tag-n n)::('a,'b::finite) array xtyp-info) = (align-td-wo-align
  (typ-info-t (TYPE('a))))
proof (induct n)
  case 0
  then show ?case by clarsimp
next
  case (Suc n)
  then show ?case
    by (cases n = 0) (auto simp: align-of-def max-def)
qed

lemma align-td-export-uinfo[simp]: align-td (export-uinfo t) = align-td t
  apply (cases t)
  apply (simp add: export-uinfo-def)
  done

lemma (in c-type) align-td-uinfo: align-td (typ-uinfo-t (TYPE('a))) = align-td
  (typ-info-t (TYPE('a)))
  by (simp add: typ-uinfo-t-def)

lemma (in c-type) align-td-array-tag:
  0 < n  $\implies$ 
  align-td ((array-tag-n n)::('a,'b::finite) array xtyp-info) = (align-td (typ-info-t
  (TYPE('a))))
proof (induct n)
  case 0
  then show ?case by clarsimp
next
  case (Suc n)
  then show ?case
    by (cases n = 0) (auto simp: align-of-def max-def align-td-uinfo)
qed

```

```

lemma align-of-array [simp]:
   $0 < \text{CARD}(b) \implies \text{align-of TYPE}('a, 'b::\text{finite}) \text{ array} = \text{align-of TYPE}('a::\text{c-type})$ 
  by (simp add: align-of-def typ-info-array array-tag-def align-td-array-tag)

lemma align-td-wo-align-array-info:  $0 < \text{CARD}(b) \implies \text{align-td-wo-align} (\text{typ-info-t}$ 
   $\text{TYPE}('a, 'b::\text{finite}) \text{ array})$ 
  = align-td-wo-align (typ-info-t  $\text{TYPE}('a::\text{c-type})$ )
  by (simp add: typ-info-array array-tag-def align-td-wo-align-array-tag)

lemma align-td-array-info:  $0 < \text{CARD}(b) \implies \text{align-td} (\text{typ-info-t } \text{TYPE}('a, 'b::\text{finite})$ 
   $\text{array})$ 
  = align-td (typ-info-t  $\text{TYPE}('a::\text{c-type})$ )
  by (simp add: typ-info-array array-tag-def align-td-array-tag)

lemma (in mem-type) align-field-array [simp]:
  align-field ((array-tag-n n)::('a, 'b::finite) array xtyp-info)
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then show ?case
    by clarsimp (metis align-field-ti-typ-combine align-of-def align-size-of dvd-mult
  size-td-array-tag)
qed

lemma (in mem-type) wf-align-array [simp]:
  wf-align ((array-tag-n n)::('a, 'b::finite) array xtyp-info)
proof (induct n)
  case 0
  then show ?case by (simp add: wf-align-simps)
next
  case (Suc n)
  then show ?case by (simp add: wf-align-ti-typ-combine)
qed

instance array :: (mem-type, finite) mem-type-sans-size
apply intro-classes
  apply (simp-all add: typ-info-array array-tag-def size-of-def norm-bytes-def)
apply clarsimp
apply (rule fu-eq-mask)
apply (simp add: size-of-def)
apply (rule fu-eq-mask-array-tag; simp)
apply (clarsimp simp: align-of-def typ-info-array array-tag-def align-td-wo-align-array-tag)
apply (metis align-of-def align-size-of dvd-mult size-of-def)
done

declare atn-base [simp del]

```

declare *atn-rec* [*simp del*]

lemma *size-of-array* [*simp*]:

size-of TYPE('a,'b::finite) array = *CARD('b) * size-of TYPE('a::c-type)*
by (*simp add: size-of-def typ-info-array array-tag-def*)

lemma *size-td-array*:

size-td (typ-info-t TYPE('a,'b::finite) array) = *CARD('b) * size-of TYPE('a::c-type)*
by (*simp add: size-of-def typ-info-array array-tag-def*)

lemma *align-td-array*:

2^{\wedge} *align-td (typ-info-t TYPE('a,'b::finite) array)* = *align-of TYPE('a::c-type)*
by (*simp add: align-of-def typ-info-array array-tag-def align-td-array-tag*)

lemma *wf-field-array*:

$n < \text{CARD}('b) \implies \text{wf-field } (\lambda x. x.[n]) (\lambda x f. \text{update } (f::('a,'b::finite) \text{ array}) n x)$
by (*simp add: wf-field-def*)

lemma *wf-cfield-array*:

$n < \text{CARD}('b) \implies \text{wf-cfield } (\lambda x. x.[n]) (\lambda x f. \text{update } (f::('a::c-type,'b::finite) \text{ array}) n x)$
by (*simp add: wf-cfield-def wf-field-array*)

lemma *wf-xfield-array*:

$n < \text{CARD}('b) \implies \text{wf-xfield } (\lambda x. x.[n]) (\lambda x f. \text{update } (f::('a::xmem-type,'b::finite) \text{ array}) n x)$
by (*simp add: wf-xfield-def wf-cfield-array*)

lemma *wf-component-descs-array-tag-n*: $n \leq \text{CARD}('b)$

$\implies \text{wf-component-descs } ((\text{array-tag-n}::\text{nat} \implies ('a::\text{xmem-type},'b::\text{finite}) \text{ array } \text{xtyp-info}) n)$

apply (*induct n*)

apply (*simp add: atn-base*)

apply (*simp add: atn-rec*)

apply (*rule wf-xfield.wf-component-descs-ti-tyt-combine*)

apply (*simp add: wf-xfield-array*)

apply *simp*

done

lemma *wf-component-descs-array*: *wf-component-descs (typ-info-t TYPE('a::xmem-type['b::finite]))*

apply (*simp add: typ-info-array array-tag-def*)

apply (*rule wf-component-descs-array-tag-n*)

apply *simp*

done

lemma (**in** *c-type*) *set-toplevel-field-descs-array-tag-n*:

```

(set (toplevel-field-descs ( (array-tag-n::nat ⇒ ('a,'b::finite) array xtyp-info) n)))
=
{d. ∃ m. m < n ∧ d = (|field-access = xto-bytes ∘ (λx. index x m),
  field-update = (λx f. update f m x) ∘ xfrom-bytes,
  field-sz = size-of TYPE('a)|)} (is - = ?D n)
proof (induct n)
  case 0
  then show ?case by (simp add: atn-base empty-typ-info-def)
next
  case (Suc n)
  from Suc.hyps have hyp: set (toplevel-field-descs (array-tag-n n)) = ?D n .

  show ?case
  proof
    show set (toplevel-field-descs (array-tag-n (Suc n))) ⊆ ?D (Suc n)
    proof
      fix d
      assume d-in: d ∈ set (toplevel-field-descs ((array-tag-n::nat ⇒ ('a,'b::finite)
array xtyp-info) (Suc n)))
      show d ∈ ?D (Suc n)
      proof -
        from d-in consider
          (d-new) d = (|field-access = xto-bytes ∘ (λx. x.[n]), field-update = (λx f.
update f n x) ∘ xfrom-bytes, field-sz = size-of TYPE('a)|) |
          (d-old) d ∈ set (toplevel-field-descs (array-tag-n n))
          by (auto simp add: set-toplevel-field-descs-ti-typ-combine-aggregate atn-rec)
          then show ?thesis
          proof (cases)
            case d-new
            then show ?thesis by (auto simp add: comp-def)
          next
            case d-old
            with hyp less-Suc-eq show ?thesis by (auto)
          qed
        qed
      qed
    next
    show ?D (Suc n) ⊆ set (toplevel-field-descs (array-tag-n (Suc n)))
    proof
      fix d
      assume d-in: d ∈ ?D (Suc n)
      show d ∈ set (toplevel-field-descs ((array-tag-n::nat ⇒ ('a,'b::finite) array
xtyp-info) (Suc n)))
      proof -
        from d-in obtain m where m-bound: m < Suc n and
          d: d = (|field-access = xto-bytes ∘ (λx. x.[m]),
            field-update = (λx f. update f m x) ∘ xfrom-bytes,
            field-sz = size-of TYPE('a)|) by (auto simp add: comp-def)
        from m-bound d show ?thesis

```

```

    using hyp
    apply (simp add: set-toplevel-field-descs-ti-typ-combine-aggregate atn-rec)
    using not-less-less-Suc-eq by fastforce
  qed
  qed
  qed
  qed

```

lemma (in *xmem-type*) *field-desc-independent-extend-array*:

```

n < CARD('b)  $\implies$ 
  field-desc-independent (xto-bytes  $\circ$  ( $\lambda x. x.[n]$ ))
  (( $\lambda x f. \text{update } (f::('a, 'b::\text{finite}) \text{array}) \ n \ x$ )  $\circ$  xfrom-bytes)
  (set (toplevel-field-descs (array-tag-n n)))
  apply (simp add: set-toplevel-field-descs-array-tag-n)
  apply (rule field-desc-independent.intro)
  apply (auto simp add: fu-commutes-def)
done

```

lemma *component-descs-independent-array-tag-n*: $n \leq \text{CARD}('b)$

```

 $\implies$  component-descs-independent ((array-tag-n::nat  $\implies$  ('a::xmem-type, 'b::finite)
array xtyp-info) n)
  apply (induct n)
  apply (simp add: atn-base)
  apply (simp add: atn-rec)
  apply (rule wf-xfield.component-descs-independent-ti-typ-combine)
  apply (simp add: wf-xfield-array)
  apply simp
  apply (rule field-desc-independent-extend-array)
  apply simp
done

```

lemma *component-descs-independent-array*: *component-descs-independent* (*typ-info-t*
TYPE('a::xmem-type['b::finite]))

```

  apply (simp add: typ-info-array array-tag-def)
  apply (rule component-descs-independent-array-tag-n)
  apply simp
done

```

lemma *complement-padding-extend-array*: $n < \text{CARD}('b) \implies$

```

  complement-padding (xto-bytes  $\circ$  ( $\lambda x. x.[n]$ ))
  (( $\lambda x f. \text{update } (f::('a::\text{xmem-type}, 'b::\text{finite}) \text{array}) \ n \ x$ )  $\circ$  xfrom-bytes) (size-of
TYPE('a))
  apply (unfold-locales)
  by (simp add: complement-padding.complement wf-cfield.intro wf-field-def wf-xfield.intro
wf-xfield.padding-lift)

```

lemma *wf-field-desc-extend-array*: $n < \text{CARD}(b) \implies \text{wf-field-desc}$
 $(\text{field-access} = \text{xto-bytes} \circ (\lambda x. x.[n]),$
 $\text{field-update} = (\lambda x f. \text{update } (f::('a::\text{xmem-type}, 'b::\text{finite}) \text{array}) n x) \circ$
 $\text{xfrom-bytes},$
 $\text{field-sz} = \text{size-of TYPE}('a::\text{xmem-type}))$
apply (*intro-locales*)
apply (*simp*)
apply (*rule complement-padding-extend-array, assumption*)
apply (*unfold-locales*)
by (*auto simp add: xfrom-bytes-xto-bytes-inv size-of-def xto-bytes-result-size xto-bytes-size*
xfrom-bytes-size)

lemma (**in** *xmem-type*) *wf-field-desc-adjust-array-field*: $n < \text{CARD}(b) \implies$
wf-field-descs
 $(\text{set } (\text{field-descs}$
 $(\text{adjust-ti } (\text{typ-info-t TYPE}('a)) (\lambda x. x.[n])$
 $(\lambda x f. \text{update } (f::('a, 'b::\text{finite}) \text{array}) n x))))$
apply (*rule wf-field.wf-field-descs-adjust-ti*)
apply (*rule wf-field-array, assumption*)
apply (*simp add: wf-field-descs*)
done

lemma *wf-field-descs-array-tag-n*: $n \leq \text{CARD}(b)$
 $\implies \text{wf-field-descs } (\text{set } (\text{field-descs } ((\text{array-tag-n}::\text{nat} \Rightarrow ('a::\text{xmem-type}, 'b::\text{finite})$
 $\text{array } \text{xtyp-info } n))))$
apply (*induct n*)
apply (*simp add: atn-base*)
apply (*simp add: atn-rec*)
apply (*simp add: set-field-descs-ti-typ-combine-aggregate*)
apply (*simp add: wf-field-desc-extend-array wf-field-desc-adjust-array-field*)
done

lemma *wf-field-descs-array*: $\text{wf-field-descs } (\text{set } (\text{field-descs } (\text{typ-info-t TYPE}('a::\text{xmem-type}['b::\text{finite}]))))$
apply (*simp add: typ-info-array array-tag-def*)
apply (*rule wf-field-descs-array-tag-n*)
apply (*simp*)
done

lemma (**in** *xmem-contained-type*) *contained-field-descs-array-tag-n*:
 $\text{contained-field-descs } ((\text{array-tag-n}::\text{nat} \Rightarrow ('a, 'b::\text{finite}) \text{array } \text{xtyp-info } n)$
 $\text{array } \text{xtyp-info } n)$
apply (*induct n*)
apply (*simp add: atn-base*)
apply (*simp add: atn-rec*)
apply (*rule contained-field-descs-ti-typ-combine*)
apply (*simp*)
done

```

lemma contained-field-descs-array: contained-field-descs (typ-info-t TYPE('a::xmem-contained-type['b::finite]))
  apply (simp add: typ-info-array array-tag-def)
  apply (rule contained-field-descs-array-tag-n)
  done

```

```

lemma replicate-1-neq-padding: replicate n CHR "1" ≠ CHR "!" # xs
  by (cases n) auto

```

```

lemma (in xmem-type) wf-padding-array-tag-n: n ≤ CARD('b)
⇒ wf-padding ((array-tag-n::nat ⇒ ('a,'b::finite) array xtyp-info) n)
  apply (induct n)
  apply (simp add: atn-base wf-padding-combinator-simps)
  apply (simp add: atn-rec wf-padding-ti-typ-combine wf-padding replicate-1-neq-padding)
  done

```

```

lemma wf-padding-array: wf-padding (typ-info-t TYPE('a::xmem-type['b::finite]))
  apply (simp add: typ-info-array array-tag-def)
  apply (rule wf-padding-array-tag-n)
  apply simp
  done

```

end

```

theory ArchArraysMemInstance
imports ArraysMemInstance
begin

```

```

class array-outer-max-size = xmem-contained-type +
  assumes array-outer-max-size-ax: size-of TYPE('a) < 2 ^ array-outer-max-size-exponent

```

```

class array-max-count = finite +
  assumes array-max-count-ax: CARD ('a) <= 2 ^ array-outer-max-count-exponent

```

```

instance array :: (array-outer-max-size, array-max-count) mem-type
apply intro-classes
apply simp
apply (subgoal-tac addr-card = 2 ^ (addr-bitsize - array-outer-max-size-exponent)
  * 2 ^ array-outer-max-size-exponent)
  apply (erule ssubst)
  apply (rule less-le-trans[where y = card (UNIV::'b set) * 2 ^ array-outer-max-size-exponent])
  apply (rule mult-less-mono2)
  apply (rule array-outer-max-size-ax)
  apply simp
apply (rule mult-le-mono1)
  apply (rule le-trans[where j = 2 ^ array-outer-max-count-exponent])

```



```

    apply (rule array-max-count-ax)
  apply simp
  apply simp
  apply (simp add: addr-card)
done

```

```

instance array :: (array-outer-max-size, array-max-count) xmem-contained-type
  apply intro-classes
    apply (rule wf-component-descs-array)
    apply (rule component-descs-independent-array)
    apply (rule wf-field-descs-array)
    apply (rule wf-padding-array)
    apply (rule contained-field-descs-array)
  done

```

```

class array-inner-max-size = array-outer-max-size +
  assumes array-inner-max-size-ax: size-of TYPE('a) < 2 ^ array-inner-max-size-exponent

```

```

instance array :: (array-inner-max-size, array-max-count) array-outer-max-size
  apply intro-classes
  apply simp
    apply (rule order-less-le-trans)
    apply (rule mult-le-less-imp-less)
    apply (rule array-max-count-ax)
    apply (rule array-inner-max-size-ax)
  apply simp
  apply simp
  apply simp
done

```

```

instance word :: (len8) array-outer-max-size
  apply intro-classes
  apply (simp add: size-of-def)
  apply (subgoal-tac len-of TYPE('a) ≤ 128)
    apply simp
  apply (rule len8-width)
done

```

```

instance word :: (len8) array-inner-max-size
  apply intro-classes
  apply (simp add: size-of-def)
  apply (subgoal-tac len-of TYPE('a) ≤ 128)
    apply simp
  apply (rule len8-width)
done

```

```

instance ptr :: (c-type) array-outer-max-size

```

```

apply intro-classes
apply (simp add: size-of-def)
done

instance ptr :: (c-type) array-inner-max-size
apply intro-classes
apply (simp add: size-of-def)
done

class lt19 = finite +
  assumes lt19-ax: CARD ('a) < 2 ^ 19
class lt18 = lt19 +
  assumes lt18-ax: CARD ('a) < 2 ^ 18
class lt17 = lt18 +
  assumes lt17-ax: CARD ('a) < 2 ^ 17
class lt16 = lt17 +
  assumes lt16-ax: CARD ('a) < 2 ^ 16
class lt15 = lt16 +
  assumes lt15-ax: CARD ('a) < 2 ^ 15
class lt14 = lt15 +
  assumes lt14-ax: CARD ('a) < 2 ^ 14
class lt13 = lt14 +
  assumes lt13-ax: CARD ('a) < 2 ^ 13
class lt12 = lt13 +
  assumes lt12-ax: CARD ('a) < 2 ^ 12
class lt11 = lt12 +
  assumes lt11-ax: CARD ('a) < 2 ^ 11
class lt10 = lt11 +
  assumes lt10-ax: CARD ('a) < 2 ^ 10
class lt9 = lt10 +
  assumes lt9-ax: CARD ('a) < 2 ^ 9
class lt8 = lt9 +
  assumes lt8-ax: CARD ('a) < 2 ^ 8
class lt7 = lt8 +
  assumes lt7-ax: CARD ('a) < 2 ^ 7
class lt6 = lt7 +
  assumes lt6-ax: CARD ('a) < 2 ^ 6
class lt5 = lt6 +
  assumes lt5-ax: CARD ('a) < 2 ^ 5
class lt4 = lt5 +
  assumes lt4-ax: CARD ('a) < 2 ^ 4
class lt3 = lt4 +
  assumes lt3-ax: CARD ('a) < 2 ^ 3
class lt2 = lt3 +
  assumes lt2-ax: CARD ('a) < 2 ^ 2
class lt1 = lt2 +
  assumes lt1-ax: CARD ('a) < 2 ^ 1

```

```

instance bit0 :: (lt19) array-max-count
  using lt19-ax[where 'a='a] if-architecture-by (ARM64, RISC64, X64) intro-classes simp

instance bit1 :: (lt19) array-max-count
  using lt19-ax[where 'a='a] if-architecture-by (ARM64, RISC64, X64) intro-classes simp

instance bit0 :: (lt18) lt19
  using lt18-ax[where 'a='a] if-architecture-by (ARM64, RISC64, X64) intro-classes simp

instance bit1 :: (lt18) lt19
  using lt18-ax[where 'a='a] if-architecture-by (ARM64, RISC64, X64) intro-classes simp

instance bit0 :: (lt17) lt18
  using lt17-ax[where 'a='a] if-architecture-by (ARM64, RISC64, X64) intro-classes simp

instance bit1 :: (lt17) lt18
  using lt17-ax[where 'a='a] if-architecture-by (ARM64, RISC64, X64) intro-classes simp

instance bit0 :: (lt16) lt17
  using lt16-ax[where 'a='a] if-architecture-by (ARM64, RISC64, X64) intro-classes simp

instance bit1 :: (lt16) lt17
  using lt16-ax[where 'a='a] if-architecture-by (ARM64, RISC64, X64) intro-classes simp

instance bit0 :: (lt15) lt16
  using lt15-ax[where 'a='a] if-architecture-by (ARM64, RISC64, X64) intro-classes simp

instance bit1 :: (lt15) lt16
  using lt15-ax[where 'a='a] if-architecture-by (ARM64, RISC64, X64) intro-classes simp

instance bit0 :: (lt14) lt15
  using lt14-ax[where 'a='a] if-architecture-by (ARM64, RISC64, X64) intro-classes simp

instance bit1 :: (lt14) lt15
  using lt14-ax[where 'a='a] if-architecture-by (ARM64, RISC64, X64) intro-classes simp

instance bit0 :: (lt13) lt14

```

```

using lt13-ax[where 'a='a] if-architecture-by (ARM64, RISCV64, X64) intro-classes simp

instance bit1 :: (lt13) lt14
  using lt13-ax[where 'a='a] if-architecture-by (ARM64, RISCV64, X64) intro-classes simp

instance bit0 :: (lt12) lt13
  using lt12-ax[where 'a='a] if-architecture-by (ARM64, RISCV64, X64) intro-classes simp

instance bit1 :: (lt12) lt13
  using lt12-ax[where 'a='a] if-architecture-by (ARM64, RISCV64, X64) intro-classes simp

instance bit0 :: (lt12) array-max-count
  using lt12-ax[where 'a='a] if-architecture-by (ARM, ARM-HYP) intro-classes simp

instance bit1 :: (lt12) array-max-count
  using lt12-ax[where 'a='a] if-architecture-by (ARM, ARM-HYP) intro-classes simp

instance bit0 :: (lt11) lt12
  using lt11-ax[where 'a='a] by intro-classes simp-all

instance bit1 :: (lt11) lt12
  using lt11-ax[where 'a='a] by intro-classes simp-all

instance bit0 :: (lt10) lt11
  using lt10-ax[where 'a='a] by intro-classes simp-all

instance bit1 :: (lt10) lt11
  using lt10-ax[where 'a='a] by intro-classes simp-all

instance bit0 :: (lt9) lt10
  using lt9-ax[where 'a='a] by intro-classes simp-all

instance bit1 :: (lt9) lt10
  using lt9-ax[where 'a='a] by intro-classes simp-all

instance bit0 :: (lt8) lt9
  using lt8-ax[where 'a='a] by intro-classes simp-all

instance bit1 :: (lt8) lt9
  using lt8-ax[where 'a='a] by intro-classes simp-all

instance bit0 :: (lt7) lt8

```

```

    using lt7-ax[where 'a='a] by intro-classes simp-all

instance bit1 :: (lt7) lt8
  using lt7-ax[where 'a='a] by intro-classes simp-all

instance bit0 :: (lt6) lt7
  using lt6-ax[where 'a='a] by intro-classes simp-all

instance bit1 :: (lt6) lt7
  using lt6-ax[where 'a='a] by intro-classes simp-all

instance bit0 :: (lt5) lt6
  using lt5-ax[where 'a='a] by intro-classes simp-all

instance bit1 :: (lt5) lt6
  using lt5-ax[where 'a='a] by intro-classes simp-all

instance bit0 :: (lt4) lt5
  using lt4-ax[where 'a='a] by intro-classes simp-all

instance bit1 :: (lt4) lt5
  using lt4-ax[where 'a='a] by intro-classes simp-all

instance bit0 :: (lt3) lt4
  using lt3-ax[where 'a='a] by intro-classes simp-all

instance bit1 :: (lt3) lt4
  using lt3-ax[where 'a='a] by intro-classes simp-all

instance bit0 :: (lt2) lt3
  using lt2-ax[where 'a='a] by intro-classes simp-all

instance bit1 :: (lt2) lt3
  using lt2-ax[where 'a='a] by intro-classes simp-all

instance bit0 :: (lt1) lt2
  using lt1-ax[where 'a='a] by intro-classes simp-all

instance bit1 :: (lt1) lt2
  using lt1-ax[where 'a='a] by intro-classes simp-all

instance num1 :: lt1
  by (intro-classes, simp-all)

instance num1 :: array-max-count
  by (intro-classes, simp)

```

end

theory *HeapRawState*
imports *CTypes*
begin

type-synonym *typ-base* = *bool*
datatype *s-heap-index* = *SIndexVal* | *SIndexTyp nat*
datatype *s-heap-value* = *SValue byte* | *STyp typ-uinto* × *typ-base*

primrec (*nonexhaustive*) *s-heap-tag* :: *s-heap-value* ⇒ *typ-uinto* × *typ-base* **where**
 s-heap-tag (*STyp t*) = *t*

type-synonym *typ-slice* = *nat* → *typ-uinto* × *typ-base*

type-synonym *s-addr* = *addr* × *s-heap-index*
type-synonym *heap-state* = *s-addr* → *s-heap-value*
type-synonym *heap-tyt-desc* = *addr* ⇒ *bool* × *typ-slice*
type-synonym *heap-raw-state* = *heap-mem* × *heap-tyt-desc*

definition *hrs-mem* :: *heap-raw-state* ⇒ *heap-mem* **where**
 hrs-mem ≡ *fst*

definition *hrs-mem-update* :: (*heap-mem* ⇒ *heap-mem*) ⇒ *heap-raw-state* ⇒ *heap-raw-state*
where
 hrs-mem-update *f* ≡ λ(*h,d*). (*f h,d*)

definition *hrs-htd* :: *heap-raw-state* ⇒ *heap-tyt-desc* **where**
 hrs-htd ≡ *snd*

definition *hrs-htd-update* :: (*heap-tyt-desc* ⇒ *heap-tyt-desc*) ⇒ *heap-raw-state* ⇒
heap-raw-state
where
 hrs-htd-update *f* ≡ λ(*h,d*). (*h,f d*)

lemma *hrs-comm*:
 hrs-htd-update *d* (*hrs-mem-update* *h s*) = *hrs-mem-update* *h* (*hrs-htd-update* *d s*)
by (*simp add: hrs-htd-update-def hrs-mem-update-def split-def*)

lemma *hrs-htd-update-htd-update*:
 (λ*s*. *hrs-htd-update* *d* (*hrs-htd-update* *d' s*)) = *hrs-htd-update* (*d* ∘ *d'*)
by (*simp add: hrs-htd-update-def split-def*)

```

lemma hrs-htd-mem-update [simp]:
  hrs-htd (hrs-mem-update f s) = hrs-htd s
  by (simp add: hrs-mem-update-def hrs-htd-def split-def)

lemma hrs-mem-htd-update [simp]:
  hrs-mem (hrs-htd-update f s) = hrs-mem s
  by (simp add: hrs-htd-update-def hrs-mem-def split-def)

lemma hrs-mem-update:
  hrs-mem (hrs-mem-update f s) = (f (hrs-mem s))
  by (simp add: hrs-mem-update-def hrs-mem-def split-def)

lemma hrs-htd-update:
  hrs-htd (hrs-htd-update f s) = (f (hrs-htd s))
  by (simp add: hrs-htd-update-def hrs-htd-def split-def)

lemmas hrs-update = hrs-mem-update hrs-htd-update

lemma hrs-htd-update-comp: hrs-htd-update f ∘ hrs-htd-update g = hrs-htd-update
(f ∘ g)
  by (auto simp add: hrs-htd-update-def split: prod.splits)

lemma hrs-mem-update-comp: hrs-mem-update f ∘ hrs-mem-update g = hrs-mem-update
(f ∘ g)
  by (auto simp add: hrs-mem-update-def split: prod.splits)

lemma hrs-update-commute:
  hrs-mem-update f ∘ hrs-htd-update g = hrs-htd-update g ∘ hrs-mem-update f
  by (auto simp add: hrs-mem-update-def hrs-htd-update-def split: prod.splits)

end

```

11.19 More properties of maps plus map disjunction.

```

theory MapExtra
imports Main
begin

```

BEWARE: we are not interested in using the $\text{dom } x \cap \text{dom } y = \{\}$ rules from Map for our separation logic proofs. As such, we overwrite the Map rules where that form of disjointness is in the assumption conflicts with a name we want to use with \perp .

A note on naming: Anything not involving heap disjunction can potentially be incorporated directly into Map.thy, thus uses m . Anything involving heap disjunction is not really mergeable with Map, is destined for use in separation logic, and hence uses h

Things that should go into Option Type

Misc option lemmas

lemma *None-not-eq*: $(None \neq x) = (\exists y. x = Some\ y)$ **by** *(cases x) auto*

lemma *None-com*: $(None = x) = (x = None)$ **by** *fast*

lemma *Some-com*: $(Some\ y = x) = (x = Some\ y)$ **by** *fast*

Things that should go into Map.thy

Map intersection: set of all keys for which the maps agree.

definition

map-inter :: $('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b) \Rightarrow 'a\ set$ (**infixl** \cap_m 70) **where**
 $m_1 \cap_m m_2 \equiv \{x \in dom\ m_1. m_1\ x = m_2\ x\}$

Map restriction via domain subtraction

definition

sub-restrict-map :: $('a \rightarrow 'b) \Rightarrow 'a\ set \Rightarrow ('a \rightarrow 'b)$ (**infixl** $'-$ 110)
where
 $m\ ' -\ S \equiv (\lambda x. if\ x \in S\ then\ None\ else\ m\ x)$

11.19.1 Properties of maps not related to restriction

lemma *empty-forall-equiv*: $(m = Map.empty) = (\forall x. m\ x = None)$
by *(rule fun-eq-iff)*

lemma *map-le-empty2* [*simp*]:
 $(m \subseteq_m Map.empty) = (m = Map.empty)$
by *(auto simp: map-le-def)*

lemma *dom-iff*:
 $(\exists y. m\ x = Some\ y) = (x \in dom\ m)$
by *auto*

lemma *non-dom-eval*:
 $x \notin dom\ m \implies m\ x = None$
by *auto*

lemma *non-dom-eval-eq*:
 $x \notin dom\ m = (m\ x = None)$
by *auto*

lemma *map-add-same-left-eq*:
 $m_1 = m_1' \implies (m_0 ++ m_1 = m_0 ++ m_1')$
by *simp*

lemma *map-add-left-cancelI* [*intro!*]:
 $m_1 = m_1' \implies m_0 ++ m_1 = m_0 ++ m_1'$
by *simp*

lemma *dom-empty-is-empty*:
 $(\text{dom } m = \{\}) = (m = \text{Map.empty})$
proof (*rule iffI*)
 assume $a: \text{dom } m = \{\}$
 { **assume** $m \neq \text{Map.empty}$
 hence $\text{dom } m \neq \{\}$
 by $-$ (*subst (asm) empty-forall-equiv, simp add: dom-def*)
 hence *False* **using** a **by** *blast*
 }
 thus $m = \text{Map.empty}$ **by** *blast*
next
 assume $a: m = \text{Map.empty}$
 thus $\text{dom } m = \{\}$ **by** *simp*
qed

lemma *map-add-dom-eq*:
 $\text{dom } m = \text{dom } m' \implies m ++ m' = m'$
by (*rule ext*) (*auto simp: map-add-def split: option.splits*)

lemma *map-add-right-dom-eq*:
 $\llbracket m_0 ++ m_1 = m_0' ++ m_1'; \text{dom } m_1 = \text{dom } m_1' \rrbracket \implies m_1 = m_1'$
unfolding *map-add-def*
apply (*rule ext*)
apply (*rule ccontr*)
subgoal for x
 apply (*drule fun-cong* [**where** $x=x$], *clarsimp split: option.splits*)
 apply (*drule sym, drule sym, force+*)
 done
done

lemma *map-le-same-dom-eq*:
 $\llbracket m_0 \subseteq_m m_1; \text{dom } m_0 = \text{dom } m_1 \rrbracket \implies m_0 = m_1$
by (*simp add: map-le-antisym map-le-def*)

11.19.2 Properties of map restriction

lemma *restrict-map-cancel*:
 $(m \upharpoonright S = m \upharpoonright T) = (\text{dom } m \cap S = \text{dom } m \cap T)$
by (*fastforce intro: set-eqI dest: fun-cong*
 simp: restrict-map-def None-not-eq
 split: if-split-asm)

lemma *map-add-restricted-self* [*simp*]:
 $m ++ m \mid' S = m$
by (*auto simp: restrict-map-def map-add-def split: option.splits*)

lemma *map-add-restrict-dom-right* [*simp*]:
 $(m ++ m') \mid' \text{dom } m' = m'$
by (*rule ext, auto simp: restrict-map-def map-add-def split: option.splits*)

lemma *restrict-map-UNIV* [*simp*]:
 $m \mid' \text{UNIV} = m$
by (*simp add: restrict-map-def*)

lemma *restrict-map-dom*:
 $S = \text{dom } m \implies m \mid' S = m$
by (*fastforce simp: restrict-map-def None-not-eq*)

lemma *restrict-map-subdom*:
 $\text{dom } m \subseteq S \implies m \mid' S = m$
by (*fastforce simp: restrict-map-def None-com*)

lemma *map-add-restrict*:
 $(m_0 ++ m_1) \mid' S = ((m_0 \mid' S) ++ (m_1 \mid' S))$
by (*force simp: map-add-def restrict-map-def*)

lemma *map-le-restrict*:
 $m \subseteq_m m' \implies m = m' \mid' \text{dom } m$
by (*force simp: map-le-def restrict-map-def None-com*)

lemma *restrict-map-le*:
 $m \mid' S \subseteq_m m$
by (*auto simp: map-le-def*)

lemma *restrict-map-remerge*:
 $\llbracket S \cap T = \{\} \rrbracket \implies m \mid' S ++ m \mid' T = m \mid' (S \cup T)$
by (*rule ext, clarsimp simp: restrict-map-def map-add-def split: option.splits*)

lemma *restrict-map-empty*:
 $\text{dom } m \cap S = \{\} \implies m \mid' S = \text{Map.empty}$
by (*fastforce simp: restrict-map-def*)

lemma *map-add-restrict-comp-right* [*simp*]:
 $(m \mid' S ++ m \mid' (\text{UNIV} - S)) = m$
by (*force simp: map-add-def restrict-map-def split: option.splits*)

lemma *map-add-restrict-comp-right-dom* [*simp*]:
 $(m \mid' S ++ m \mid' (\text{dom } m - S)) = m$
by (*fastforce simp: map-add-def restrict-map-def split: option.splits*)

lemma *map-add-restrict-comp-left* [*simp*]:

$$(m \mid' (UNIV - S) ++ m \mid' S) = m$$

by (*subst map-add-comm, auto*)

lemma *restrict-self-UNIV*:

$$m \mid' (dom\ m - S) = m \mid' (UNIV - S)$$

by (*fastforce simp: restrict-map-def*)

lemma *map-add-restrict-nonmember-right*:

$$x \notin dom\ m' \implies (m ++ m') \mid' \{x\} = m \mid' \{x\}$$

by (*rule ext, auto simp: restrict-map-def map-add-def split: option.splits*)

lemma *map-add-restrict-nonmember-left*:

$$x \notin dom\ m \implies (m ++ m') \mid' \{x\} = m' \mid' \{x\}$$

by (*rule ext, auto simp: restrict-map-def map-add-def split: option.splits*)

lemma *map-add-restrict-right*:

$$x \subseteq dom\ m' \implies (m ++ m') \mid' x = m' \mid' x$$

by (*rule ext, auto simp: restrict-map-def map-add-def split: option.splits*)

lemma *restrict-map-compose*:

$$\llbracket S \cup T = dom\ m ; S \cap T = \{\} \rrbracket \implies m \mid' S ++ m \mid' T = m$$

by (*fastforce simp: map-add-def restrict-map-def*)

lemma *map-le-dom-subset-restrict*:

$$\llbracket m' \subseteq_m m ; dom\ m' \subseteq S \rrbracket \implies m' \subseteq_m (m \mid' S)$$

by (*force simp: restrict-map-def map-le-def*)

lemma *map-le-dom-restrict-sub-add*:

$$m' \subseteq_m m \implies m \mid' (dom\ m - dom\ m') ++ m' = m$$

by (*metis map-add-restrict-comp-right-dom map-le-iff-map-add-commute map-le-restrict*)

lemma *subset-map-restrict-sub-add*:

$$T \subseteq S \implies m \mid' (S - T) ++ m \mid' T = m \mid' S$$

by (*fastforce simp: restrict-map-def map-add-def split: option.splits*)

lemma *restrict-map-sub-union*:

$$m \mid' (dom\ m - (S \cup T)) = (m \mid' (dom\ m - T)) \mid' (dom\ m - S)$$

by (*auto simp: restrict-map-def*)

lemma *prod-restrict-map-add*:

$$\llbracket S \cup T = U ; S \cap T = \{\} \rrbracket \implies m \mid' (X \times S) ++ m \mid' (X \times T) = m \mid' (X \times U)$$

by (*auto simp: map-add-def restrict-map-def split: option.splits*)

Things that should NOT go into Map.thy

11.20 Definitions

Map disjunction

definition

$map\text{-}disj :: ('a \multimap 'b) \Rightarrow ('a \multimap 'b) \Rightarrow bool$ (**infix** \perp 51) **where**
 $h_0 \perp h_1 \equiv dom\ h_0 \cap dom\ h_1 = \{\}$

declare *None-not-eq* [simp]

Heap monotonicity and the frame property

definition

$heap\text{-}mono :: (('a \multimap 'b) \Rightarrow 'c\ option) \Rightarrow bool$ **where**
 $heap\text{-}mono\ f \equiv \forall h\ h'\ v. h \perp h' \wedge f\ h = Some\ v \longrightarrow f\ (h\ ++\ h') = Some\ v$

lemma *heap-monoE*:

$\llbracket heap\text{-}mono\ f ; f\ h = Some\ v ; h \perp h' \rrbracket \Longrightarrow f\ (h\ ++\ h') = Some\ v$
unfolding *heap-mono-def* **by** *blast*

lemma *heap-mono-simp*:

$\llbracket heap\text{-}mono\ f ; f\ h = Some\ v ; h \perp h' \rrbracket \Longrightarrow f\ (h\ ++\ h') = f\ h$
by (*frule* (2) *heap-monoE*, *simp*)

definition

$heap\text{-}frame :: (('a \multimap 'b) \Rightarrow 'c\ option) \Rightarrow bool$ **where**
 $heap\text{-}frame\ f \equiv \forall h\ h'\ v. h \perp h' \wedge f\ (h\ ++\ h') = Some\ v$
 $\longrightarrow (f\ h = Some\ v \vee f\ h = None)$

lemma *heap-frameE*:

$\llbracket heap\text{-}frame\ f ; f\ (h\ ++\ h') = Some\ v ; h \perp h' \rrbracket$
 $\Longrightarrow f\ h = Some\ v \vee f\ h = None$
unfolding *heap-frame-def* **by** *fastforce*

11.21 Properties of (' -)

lemma *restrict-map-sub-disj*: $h \mid \text{' } S \perp h \text{' } \text{- } S$

by (*fastforce simp*: *sub-restrict-map-def restrict-map-def map-disj-def*
split: *option.splits if-split-asm*)

lemma *restrict-map-sub-add*: $h \mid \text{' } S\ ++\ h \text{' } \text{- } S = h$

by (*fastforce simp*: *sub-restrict-map-def restrict-map-def map-add-def*
split: *option.splits if-split*)

11.22 Properties of map disjunction

lemma *map-disj-empty-right* [simp]:

$h \perp Map.empty$
by (*simp add*: *map-disj-def*)

lemma *map-disj-empty-left* [*simp*]:
 $Map.empty \perp h$
by (*simp add: map-disj-def*)

lemma *map-disj-com*:
 $h_0 \perp h_1 = h_1 \perp h_0$
by (*simp add: map-disj-def, fast*)

lemma *map-disjD*:
 $h_0 \perp h_1 \implies dom\ h_0 \cap dom\ h_1 = \{\}$
by (*simp add: map-disj-def*)

lemma *map-disjI*:
 $dom\ h_0 \cap dom\ h_1 = \{\} \implies h_0 \perp h_1$
by (*simp add: map-disj-def*)

11.22.1 Map associativity-commutativity based on map disjunction

lemma *map-add-com*:
 $h_0 \perp h_1 \implies h_0 ++ h_1 = h_1 ++ h_0$
by (*drule map-disjD, rule map-add-comm, force*)

lemma *map-add-left-commute*:
 $h_0 \perp h_1 \implies h_0 ++ (h_1 ++ h_2) = h_1 ++ (h_0 ++ h_2)$
by (*simp add: map-add-com map-disj-com*)

lemma *map-add-disj*:
 $h_0 \perp (h_1 ++ h_2) = (h_0 \perp h_1 \wedge h_0 \perp h_2)$
by (*simp add: map-disj-def, fast*)

lemma *map-add-disj'*:
 $(h_1 ++ h_2) \perp h_0 = (h_1 \perp h_0 \wedge h_2 \perp h_0)$
by (*simp add: map-disj-def, fast*)

We redefine $(++)$ associativity to bind to the right, which seems to be the more common case. Note that when a theory includes `Map` again, *map-add-assoc* will return to the simpset and will cause infinite loops if its symmetric counterpart is added (e.g. via *map-ac-simps*)

declare *map-add-assoc* [*simp del*]

Since the associativity-commutativity of $(++)$ relies on map disjunction, we include some basic rules into the ac set.

lemmas *map-ac-simps* =
 $map-add-assoc[symmetric]\ map-add-com\ map-disj-com$
 $map-add-left-commute\ map-add-disj\ map-add-disj'$

11.22.2 Basic properties

lemma *map-disj-None-right*:

$\llbracket h_0 \perp h_1 ; x \in \text{dom } h_0 \rrbracket \implies h_1 x = \text{None}$
by (*auto simp: map-disj-def dom-def*)

lemma *map-disj-None-left*:

$\llbracket h_0 \perp h_1 ; x \in \text{dom } h_1 \rrbracket \implies h_0 x = \text{None}$
by (*auto simp: map-disj-def dom-def*)

lemma *map-disj-None-left'*:

$\llbracket h_0 x = \text{Some } y ; h_1 \perp h_0 \rrbracket \implies h_1 x = \text{None}$
by (*auto simp: map-disj-def*)

lemma *map-disj-None-right'*:

$\llbracket h_1 x = \text{Some } y ; h_1 \perp h_0 \rrbracket \implies h_0 x = \text{None}$
by (*auto simp: map-disj-def*)

lemma *map-disj-common*:

$\llbracket h_0 \perp h_1 ; h_0 p = \text{Some } v ; h_1 p = \text{Some } v' \rrbracket \implies \text{False}$
by (*frule (1) map-disj-None-left', simp*)

11.22.3 Map disjunction and addition

lemma *map-add-eval-left*:

$\llbracket x \in \text{dom } h ; h \perp h' \rrbracket \implies (h ++ h') x = h x$
by (*auto dest!: map-disj-None-right simp: map-add-def cong: option.case-cong*)

lemma *map-add-eval-right*:

$\llbracket x \in \text{dom } h' ; h \perp h' \rrbracket \implies (h ++ h') x = h' x$
by (*auto elim!: map-disjD simp: map-add-comm map-add-eval-left map-disj-com*)

lemma *map-add-eval-left'*:

$\llbracket x \notin \text{dom } h' ; h \perp h' \rrbracket \implies (h ++ h') x = h x$
by (*clarsimp simp: map-disj-def map-add-def split: option.splits*)

lemma *map-add-eval-right'*:

$\llbracket x \notin \text{dom } h ; h \perp h' \rrbracket \implies (h ++ h') x = h' x$
by (*clarsimp simp: map-disj-def map-add-def split: option.splits*)

lemma *map-add-left-dom-eq*:

assumes *eq*: $h_0 ++ h_1 = h_0' ++ h_1'$
assumes *etc*: $h_0 \perp h_1 \ h_0' \perp h_1' \ \text{dom } h_0 = \text{dom } h_0'$
shows $h_0 = h_0'$

proof –

from *eq* **have** $h_1 ++ h_0 = h_1' ++ h_0'$ **using** *etc* **by** (*simp add: map-ac-simps*)

thus *?thesis* **using** *etc*

by (*fastforce elim!: map-add-right-dom-eq simp: map-ac-simps*)

qed

lemma *map-add-left-eq*:
assumes *eq*: $h_0 ++ h = h_1 ++ h$
assumes *disj*: $h_0 \perp h$ $h_1 \perp h$
shows $h_0 = h_1$
proof (*rule ext*)
fix x
from *eq* **have** *eq'*: $(h_0 ++ h) x = (h_1 ++ h) x$ **by** *auto*
{ **assume** $x \in \text{dom } h$
hence $h_0 x = h_1 x$ **using** *disj* **by** (*simp add: map-disj-None-left*)
} **moreover** **{**
assume $x \notin \text{dom } h$
hence $h_0 x = h_1 x$ **using** *disj eq'* **by** (*simp add: map-add-eval-left'*)
}
ultimately show $h_0 x = h_1 x$ **by** *cases*
qed

lemma *map-add-right-eq*:
 $\llbracket h ++ h_0 = h ++ h_1; h_0 \perp h; h_1 \perp h \rrbracket \implies h_0 = h_1$
by (*rule map-add-left-eq [where h=h], auto simp: map-ac-simps*)

lemma *map-disj-add-eq-dom-right-eq*:
assumes *merge*: $h_0 ++ h_1 = h_0' ++ h_1'$ **and** $d: \text{dom } h_0 = \text{dom } h_0'$ **and**
 $ab\text{-disj}: h_0 \perp h_1$ **and** $cd\text{-disj}: h_0' \perp h_1'$
shows $h_1 = h_1'$
proof (*rule ext*)
fix x
from *merge* **have** *merge-x*: $(h_0 ++ h_1) x = (h_0' ++ h_1') x$ **by** *simp*
with d $ab\text{-disj}$ $cd\text{-disj}$ **show** $h_1 x = h_1' x$
apply (*cases h₁ x*)
apply (*cases h₁' x*)
apply *force*
apply (*fastforce simp: map-disj-def*)
apply (*cases h₁' x*)
apply *clarsimp*
apply (*simp add: Some-com*)
by (*force simp: map-disj-def*)
qed

lemma *map-disj-add-eq-dom-left-eq*:
assumes *add*: $h_0 ++ h_1 = h_0' ++ h_1'$ **and**
 $dom: \text{dom } h_1 = \text{dom } h_1'$ **and**
 $disj: h_0 \perp h_1$ $h_0' \perp h_1'$
shows $h_0 = h_0'$
proof –
have $h_1 ++ h_0 = h_1' ++ h_0'$ **using** *add disj* **by** (*simp add: map-ac-simps*)
thus *?thesis* **using** *dom disj*
by – (*rule map-disj-add-eq-dom-right-eq, auto simp: map-disj-com*)
qed

lemma *map-add-left-cancel*:
assumes *disj*: $h_0 \perp h_1$ $h_0 \perp h_1'$
shows $(h_0 ++ h_1 = h_0 ++ h_1') = (h_1 = h_1')$
proof (*rule iffI, rule ext*)
fix x
assume $(h_0 ++ h_1) = (h_0 ++ h_1')$
hence $(h_0 ++ h_1) x = (h_0 ++ h_1') x$ **by** *auto*
hence $h_1 x = h_1' x$ **using** *disj*
by - (*cases* $x \in \text{dom } h_0$,
simp-all add: map-disj-None-right map-add-eval-right')
thus $h_1 x = h_1' x$ **by** *auto*
qed *auto*

lemma *map-add-lr-disj*:
 $\llbracket h_0 ++ h_1 = h_0' ++ h_1'; h_1 \perp h_1' \rrbracket \implies \text{dom } h_1 \subseteq \text{dom } h_0'$
apply (*clarsimp simp: map-disj-def map-add-def*)
subgoal for $x y$
apply (*drule fun-cong [where x=x]*)
apply (*auto split: option.splits*)
done
done

11.22.4 Map disjunction and updates

lemma *map-disj-update-left* [*simp*]:
 $p \in \text{dom } h_1 \implies h_0 \perp h_1(p \mapsto v) = h_0 \perp h_1$
by (*clarsimp simp add: map-disj-def, blast*)

lemma *map-disj-update-right* [*simp*]:
 $p \in \text{dom } h_1 \implies h_1(p \mapsto v) \perp h_0 = h_1 \perp h_0$
by (*simp add: map-disj-com*)

lemma *map-add-update-left*:
 $\llbracket h_0 \perp h_1 ; p \in \text{dom } h_0 \rrbracket \implies (h_0 ++ h_1)(p \mapsto v) = (h_0(p \mapsto v) ++ h_1)$
by (*drule (1) map-disj-None-right*)
(auto simp: map-add-def cong: option.case-cong)

lemma *map-add-update-right*:
 $\llbracket h_0 \perp h_1 ; p \in \text{dom } h_1 \rrbracket \implies (h_0 ++ h_1)(p \mapsto v) = (h_0 ++ h_1(p \mapsto v))$
by (*drule (1) map-disj-None-left*)
(auto simp: map-add-def cong: option.case-cong)

lemma *map-add3-update*:
 $\llbracket h_0 \perp h_1 ; h_1 \perp h_2 ; h_0 \perp h_2 ; p \in \text{dom } h_0 \rrbracket$
 $\implies (h_0 ++ h_1 ++ h_2)(p \mapsto v) = h_0(p \mapsto v) ++ h_1 ++ h_2$
by (*auto simp: map-add-update-left[symmetric] map-ac-simps*)

11.22.5 Map disjunction and (\subseteq_m)

lemma *map-le-override* [*simp*]:

$\llbracket h \perp h' \rrbracket \implies h \subseteq_m h ++ h'$

by (*auto simp: map-le-def map-add-def map-disj-def split: option.splits*)

lemma *map-leI-left*:

$\llbracket h = h_0 ++ h_1 ; h_0 \perp h_1 \rrbracket \implies h_0 \subseteq_m h$ **by** *auto*

lemma *map-leI-right*:

$\llbracket h = h_0 ++ h_1 ; h_0 \perp h_1 \rrbracket \implies h_1 \subseteq_m h$ **by** *auto*

lemma *map-disj-map-le*:

$\llbracket h_0' \subseteq_m h_0 ; h_0 \perp h_1 \rrbracket \implies h_0' \perp h_1$

by (*force simp: map-disj-def map-le-def*)

lemma *map-le-on-disj-left*:

$\llbracket h' \subseteq_m h ; h_0 \perp h_1 ; h' = h_0 ++ h_1 \rrbracket \implies h_0 \subseteq_m h$

unfolding *map-le-def*

apply (*rule ballI*)

subgoal for *a*

apply (*erule ballE [where x=a], auto simp: map-add-eval-left*)
done

done

lemma *map-le-on-disj-right*:

$\llbracket h' \subseteq_m h ; h_0 \perp h_1 ; h' = h_1 ++ h_0 \rrbracket \implies h_0 \subseteq_m h$

by (*auto simp: map-le-on-disj-left map-ac-simps*)

lemma *map-le-add-cancel*:

$\llbracket h_0 \perp h_1 ; h_0' \subseteq_m h_0 \rrbracket \implies h_0' ++ h_1 \subseteq_m h_0 ++ h_1$

by (*auto simp: map-le-def map-add-def map-disj-def split: option.splits*)

lemma *map-le-override-bothD*:

assumes *subm*: $h_0' ++ h_1 \subseteq_m h_0 ++ h_1$

assumes *disj'*: $h_0' \perp h_1$

assumes *disj*: $h_0 \perp h_1$

shows $h_0' \subseteq_m h_0$

unfolding *map-le-def*

proof (*rule ballI*)

fix *a*

assume *a*: $a \in \text{dom } h_0'$

hence *sumeq*: $(h_0' ++ h_1) a = (h_0 ++ h_1) a$

using *subm* **unfolding** *map-le-def* **by** *auto*

from *a* **have** $a \notin \text{dom } h_1$ **using** *disj'* **by** (*auto dest!: map-disj-None-right*)

thus $h_0' a = h_0 a$ **using** *a sumeq disj disj'*

by (*simp add: map-add-eval-left map-add-eval-left'*)

qed

lemma *map-le-conv*:

$(h_0' \subseteq_m h_0 \wedge h_0' \neq h_0) = (\exists h_1. h_0 = h_0' ++ h_1 \wedge h_0' \perp h_1 \wedge h_0' \neq h_0)$
unfolding *map-le-def map-disj-def map-add-def*
using *exI[where x=λx. if x ∉ dom h_0' then h_0 x else None]*
by (*rule iffI, clarsimp*)
(fastforce intro: set-eqI split: option.splits if-split-asm)+

lemma *map-le-conv2*:
 $h_0' \subseteq_m h_0 = (\exists h_1. h_0 = h_0' ++ h_1 \wedge h_0' \perp h_1)$
by (*cases h_0'=h_0, insert map-le-conv, auto intro: exI[where x=Map.empty]*)

11.22.6 Map disjunction and restriction

lemma *map-disj-comp [simp]*:
 $h_0 \perp h_1 \mid' (UNIV - \text{dom } h_0)$
by (*force simp: map-disj-def*)

lemma *restrict-map-disj*:
 $S \cap T = \{\} \implies h \mid' S \perp h \mid' T$
by (*auto simp: map-disj-def restrict-map-def dom-def*)

lemma *map-disj-restrict-dom [simp]*:
 $h_0 \perp h_1 \mid' (\text{dom } h_1 - \text{dom } h_0)$
by (*force simp: map-disj-def*)

lemma *restrict-map-disj-dom-empty*:
 $h \perp h' \implies h \mid' \text{dom } h' = \text{Map.empty}$
by (*fastforce simp: map-disj-def restrict-map-def*)

lemma *restrict-map-univ-disj-eq*:
 $h \perp h' \implies h \mid' (UNIV - \text{dom } h') = h$
by (*rule ext, auto simp: map-disj-def restrict-map-def*)

lemma *restrict-map-disj-dom*:
 $h_0 \perp h_1 \implies h \mid' \text{dom } h_0 \perp h \mid' \text{dom } h_1$
by (*auto simp: map-disj-def restrict-map-def dom-def*)

lemma *map-add-restrict-dom-left*:
 $h \perp h' \implies (h ++ h') \mid' \text{dom } h = h$
by (*rule ext, auto simp: restrict-map-def map-add-def dom-def map-disj-def split: option.splits*)

lemma *restrict-map-disj-left*:
 $h_0 \perp h_1 \implies h_0 \mid' S \perp h_1$
by (*auto simp: map-disj-def*)

lemma *restrict-map-disj-right*:
 $h_0 \perp h_1 \implies h_0 \perp h_1 \mid' S$
by (*auto simp: map-disj-def*)

lemmas *restrict-map-disj-both* = *restrict-map-disj-right restrict-map-disj-left*

lemma *map-dom-disj-restrict-right*:

$h_0 \perp h_1 \implies (h_0 ++ h_0') \mid' \text{dom } h_1 = h_0' \mid' \text{dom } h_1$

by (*simp add: map-add-restrict restrict-map-empty map-disj-def*)

lemma *restrict-map-on-disj*:

$h_0' \perp h_1 \implies h_0 \mid' \text{dom } h_0' \perp h_1$

unfolding *map-disj-def* **by** *auto*

lemma *restrict-map-on-disj'*:

$h_0 \perp h_1 \implies h_0 \perp h_1 \mid' S$

by (*auto simp: map-disj-def map-add-def*)

lemma *map-le-sub-dom*:

$\llbracket h_0 ++ h_1 \subseteq_m h ; h_0 \perp h_1 \rrbracket \implies h_0 \subseteq_m h \mid' (\text{dom } h - \text{dom } h_1)$

by (*rule map-le-override-bothD, subst map-le-dom-restrict-sub-add*)
(*auto elim: map-add-le-mapE simp: map-ac-simps*)

lemma *map-submap-break*:

$\llbracket h \subseteq_m h' \rrbracket \implies h' = (h' \mid' (\text{UNIV} - \text{dom } h)) ++ h$

by (*fastforce split: option.splits*

simp: map-le-restrict restrict-map-def map-le-def map-add-def dom-def)

lemma *map-add-disj-restrict-both*:

$\llbracket h_0 \perp h_1 ; S \cap S' = \{\}; T \cap T' = \{\} \rrbracket$

$\implies (h_0 \mid' S) ++ (h_1 \mid' T) \perp (h_0 \mid' S') ++ (h_1 \mid' T')$

by (*auto simp: map-ac-simps intro!: restrict-map-disj-both restrict-map-disj*)

end

theory *MapExtraTrans*

imports *MapExtra*

begin

lemma *case-option-None-Some* [*simp*]:

case-option None Some P = P

by (*simp split: option.splits*)

lemma *heap-merge-dom-exact2*:

```

[[ a ++ b = c ++ d; dom a = dom c; a ⊥ b; c ⊥ d ]] ⇒ a=c ∧ b=d
apply (rule conjI)
apply (erule (3) map-add-left-dom-eq)
apply (erule (3) map-disj-add-eq-dom-right-eq)
done

lemma map-add-restrict-sub:
[[ dom s = X; dom t = X - Y ]] ⇒
s ++ (t |' (X - Y - Z)) = s ++ t ++ s |' Z
apply(rule ext)
apply(auto simp: restrict-map-def map-add-def split: option.splits)
done

lemma map-add-restrict-UNIV:
[[ dom g ∩ X = {}; dom f = dom h ]] ⇒ f ++ g = f |' (UNIV - X) ++ h |'
X ++ g ++ f |' X
apply(rule ext)
apply(force simp: restrict-map-def map-add-def split: option.splits)
done

end

theory TypHeap
imports
  Vanilla32
  ArchArraysMemInstance
  HeapRawState
  MapExtraTrans
begin

declare map-add-assoc [simp del]

definition wf-heap-val :: heap-state ⇒ bool where
  wf-heap-val s ≡
    ∀ x t n v. s (x, SIndexVal) ≠ Some (STyp t) ∧ s (x, SIndexTyp n) ≠ Some
(SValue v)

type-synonym typ-slice-list = (typ-uinfo × typ-base) list

primrec
  typ-slice-t :: typ-uinfo ⇒ nat ⇒ typ-slice-list and
  typ-slice-struct :: typ-uinfo-struct ⇒ nat ⇒ typ-slice-list and
  typ-slice-list :: (typ-uinfo, field-name, unit) dt-tuple list ⇒ nat ⇒ typ-slice-list and
  typ-slice-tuple :: (typ-uinfo, field-name, unit) dt-tuple ⇒ nat ⇒ typ-slice-list
where

```

$tl0: \text{typ-slice-t } (TypDesc \text{ algn } st \text{ nm}) \text{ m} = \text{typ-slice-struct } st \text{ m} \text{ @}$
 $[(if \text{ m} = 0 \text{ then } ((TypDesc \text{ algn } st \text{ nm}), True) \text{ else}$
 $((TypDesc \text{ algn } st \text{ nm}), False))]$

$| tl1: \text{typ-slice-struct } (TypScalar \text{ n } \text{algn } \text{d}) \text{ m} = []$
 $| tl2: \text{typ-slice-struct } (TypAggregate \text{ xs}) \text{ m} = \text{typ-slice-list } \text{xs } \text{m}$

$| tl3: \text{typ-slice-list } [] \text{ m} = []$
 $| tl4: \text{typ-slice-list } (\text{x}\#\text{xs}) \text{ m} = (if \text{ m} < \text{size-td } (\text{dt-fst } \text{x}) \vee \text{xs} = [] \text{ then}$
 $\text{typ-slice-tuple } \text{x } \text{m} \text{ else } \text{typ-slice-list } \text{xs } (\text{m} - \text{size-td } (\text{dt-fst } \text{x})))$

$| tl5: \text{typ-slice-tuple } (DTuple \text{ t } \text{n } \text{d}) \text{ m} = \text{typ-slice-t } \text{t } \text{m}$

definition $\text{list-map} :: 'a \text{ list} \Rightarrow (\text{nat} \rightarrow 'a) \text{ where}$
 $\text{list-map } \text{xs} \equiv \text{map-of } (\text{zip } [0..<\text{length } \text{xs}] \text{xs})$

definition $\text{s-footprint-untyped} :: \text{addr} \Rightarrow \text{typ-uinfo} \Rightarrow (\text{addr} \times \text{s-heap-index}) \text{ set}$
where
 $\text{s-footprint-untyped } \text{p } \text{t} \equiv$
 $\{(p + \text{of-nat } \text{x}, \text{k}) \mid \text{x } \text{k}. \text{x} < \text{size-td } \text{t} \wedge$
 $(\text{k} = \text{SIndexVal} \vee (\exists \text{n}. \text{k} = \text{SIndexTyp } \text{n} \wedge \text{n} < \text{length}$
 $(\text{typ-slice-t } \text{t } \text{x})))\}$

definition (in *c-type*) $\text{s-footprint} :: 'a \text{ ptr} \Rightarrow (\text{addr} \times \text{s-heap-index}) \text{ set where}$
 $\text{s-footprint } \text{p} \equiv \text{s-footprint-untyped } (\text{ptr-val } \text{p}) (\text{typ-uinfo-t } \text{TYPE}('a))$

definition $\text{empty-htd} :: \text{heap-typ-desc} \text{ where}$
 $\text{empty-htd} \equiv \lambda \text{x}. (\text{False}, \text{Map.empty})$

definition $\text{dom-s} :: \text{heap-typ-desc} \Rightarrow \text{s-addr set where}$
 $\text{dom-s } \text{d} \equiv \{(x, \text{SIndexVal}) \mid \text{x}. \text{fst } (\text{d } \text{x})\} \cup$
 $\{(x, \text{SIndexTyp } \text{n}) \mid \text{x } \text{n}. \text{snd } (\text{d } \text{x}) \text{ n} \neq \text{None}\}$

definition $\text{restrict-s} :: \text{heap-typ-desc} \Rightarrow \text{s-addr set} \Rightarrow \text{heap-typ-desc where}$
 $\text{restrict-s } \text{d } \text{X} \equiv$
 $\lambda \text{x}. ((x, \text{SIndexVal}) \in \text{X} \wedge \text{fst } (\text{d } \text{x}), (\lambda \text{y}. if (x, \text{SIndexTyp } \text{y}) \in \text{X} \text{ then } \text{snd } (\text{d } \text{x}) \text{ y} \text{ else } \text{None}))$

definition $\text{valid-footprint} :: \text{heap-typ-desc} \Rightarrow \text{addr} \Rightarrow \text{typ-uinfo} \Rightarrow \text{bool where}$
 $\text{valid-footprint } \text{d } \text{x } \text{t} \equiv$
 $\text{let } \text{n} = \text{size-td } \text{t} \text{ in}$
 $0 < \text{n} \wedge (\forall \text{y}. \text{y} < \text{n} \rightarrow$
 $\text{list-map } (\text{typ-slice-t } \text{t } \text{y}) \subseteq_m \text{snd } (\text{d } (\text{x} + \text{of-nat } \text{y})) \wedge \text{fst } (\text{d } (\text{x} +$
 $\text{of-nat } \text{y})))$

definition (in *c-type*) $\text{h-t-valid} ::$
 $\text{heap-typ-desc} \Rightarrow 'a \text{ ptr-guard} \Rightarrow 'a \text{ ptr} \Rightarrow \text{bool } (-, - \models_t - [99, 0, 99] 100) \text{ where}$

$d, g \models_t p \equiv \text{valid-footprint } d \text{ (ptr-val (p::'a ptr)) (typ-uinfo-t TYPE('a))} \wedge g p$

type-synonym $'a \text{ typ-heap} = 'a \text{ ptr} \rightarrow 'a$

definition $\text{proj-h} :: \text{heap-state} \Rightarrow \text{heap-mem}$ **where**
 $\text{proj-h } s \equiv \lambda x. \text{case-option undefined (case-s-heap-value id undefined) (s (x, SIndexVal))}$

definition $\text{lift-state} :: \text{heap-raw-state} \Rightarrow \text{heap-state}$ **where**
 $\text{lift-state} \equiv \lambda(h, d) (x, y).$
case y of
 $\text{SIndexVal} \Rightarrow \text{if fst (d x) then Some (SValue (h x)) else None}$
 $| \text{SIndexTyp } n \Rightarrow \text{case-option None (Some} \circ \text{STyp) (snd (d x) n)}$

definition $\text{fun2list} :: (\text{nat} \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow 'a \text{ list}$ **where**
 $\text{fun2list } f \ n \equiv \text{if } n=0 \text{ then } [] \text{ else map } f \ [0..<n]$

definition $\text{null-d} :: \text{heap-state} \Rightarrow \text{addr} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**
 $\text{null-d } s \ x \ y \equiv s \ (x, \text{SIndexTyp } y) = \text{None}$

definition $\text{max-d} :: \text{heap-state} \Rightarrow \text{addr} \Rightarrow \text{nat}$ **where**
 $\text{max-d } s \ x \equiv 1 + (\text{GREATEST } y. \neg \text{null-d } s \ x \ y)$

definition $\text{proj-d} :: \text{heap-state} \Rightarrow \text{heap-ty-desc}$ **where**
 $\text{proj-d } s \equiv \lambda x. (s \ (x, \text{SIndexVal}) \neq \text{None},$
 $\lambda n. \text{case-option None (Some} \circ \text{s-heap-tag) (s (x, SIndexTyp } n)))$

definition **(in c-type)** $\text{s-valid} ::$
 $\text{heap-state} \Rightarrow 'a \text{ ptr-guard} \Rightarrow 'a \text{ ptr} \Rightarrow \text{bool}$ $(-, - \models_s - [100, 0, 100] 100)$ **where**
 $s, g \models_s p \equiv \text{proj-d } s, g \models_t p$

definition $\text{heap-list-s} :: \text{heap-state} \Rightarrow \text{nat} \Rightarrow \text{addr} \Rightarrow \text{byte list}$ **where**
 $\text{heap-list-s } s \ n \ p \equiv \text{heap-list (proj-h } s) \ n \ p$

definition **(in c-type)** $\text{lift-ty-heap} :: 'a \text{ ptr-guard} \Rightarrow \text{heap-state} \Rightarrow 'a \text{ typ-heap}$
where
 $\text{lift-ty-heap } g \ s \equiv$
 $(\text{Some} \circ \text{from-bytes} \circ \text{heap-list-s } s \ (\text{size-of } \text{TYPE('a)}) \circ \text{ptr-val}) \mid \{p. s, g \models_s p\}$

definition **(in c-type)** $\text{heap-update-s} :: 'a \text{ ptr} \Rightarrow 'a \Rightarrow \text{heap-state} \Rightarrow \text{heap-state}$
where
 $\text{heap-update-s } n \ p \ s \equiv \text{lift-state (heap-update } n \ p \ (\text{proj-h } s), \text{proj-d } s)$

definition **(in c-type)** $\text{lift-t} :: 'a \text{ ptr-guard} \Rightarrow \text{heap-raw-state} \Rightarrow 'a \text{ typ-heap}$ **where**
 $\text{lift-t } g \equiv \text{lift-ty-heap } g \circ \text{lift-state}$

definition $\text{tag-disj} :: ('a, 'b) \text{ typ-desc} \Rightarrow ('a, 'b) \text{ typ-desc} \Rightarrow \text{bool}$ $(- \perp_t - [90, 90])$

90) **where**

$$f \perp_t g \equiv \neg (f \leq g \vee g \leq f)$$

definition *ladder-set* :: *typ-uinfo* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow (*typ-uinfo* \times *nat*) *set* **where**
ladder-set *s n p* \equiv $\{(t, n+p) \mid t. \exists k. (t, k) \in \text{set } (\text{typ-slice-t } s \ n)\}$

primrec

field-names :: ('a, 'b) *typ-info* \Rightarrow *typ-uinfo* \Rightarrow
 (*qualified-field-name*) *list* **and**
field-names-struct :: ('a *field-desc*, 'b) *typ-struct* \Rightarrow *typ-uinfo* \Rightarrow
 (*qualified-field-name*) *list* **and**
field-names-list :: (('a, 'b) *typ-info*, *field-name*, 'b) *dt-tuple list* \Rightarrow *typ-uinfo* \Rightarrow
 (*qualified-field-name*) *list* **and**
field-names-tuple :: (('a, 'b) *typ-info*, *field-name*, 'b) *dt-tuple* \Rightarrow *typ-uinfo* \Rightarrow
 (*qualified-field-name*) *list*

where

tfs0: *field-names* (*TypDesc* *algn st nm*) *t* = (if *t=export-uinfo* (*TypDesc* *algn st nm*) then
 \square else *field-names-struct* *st t*)

| *tfs1*: *field-names-struct* (*TypScalar m algn d*) *t* = \square
| *tfs2*: *field-names-struct* (*TypAggregate xs*) *t* = *field-names-list* *xs t*

| *tfs3*: *field-names-list* \square *t* = \square
| *tfs4*: *field-names-list* (*x#xs*) *t* = *field-names-tuple* *x t@field-names-list xs t*

| *tfs5*: *field-names-tuple* (*DTuple s f d*) *t* = *map* ($\lambda fs. f\#fs$) (*field-names s t*)

definition *field-typ-untyped* :: ('a, 'b) *typ-desc* \Rightarrow *qualified-field-name* \Rightarrow ('a, 'b) *typ-desc* **where**
field-typ-untyped *t n* \equiv (*fst* (*the* (*field-lookup* *t n 0*)))

definition (**in** *c-type*) *field-typ* :: 'a *itself* \Rightarrow *qualified-field-name* \Rightarrow 'a *xtyp-info* **where**
field-typ *t n* \equiv *field-typ-untyped* (*typ-info-t* *TYPE*('a)) *n*

definition (**in** *c-type*) *fs-consistent* ::

qualified-field-name list \Rightarrow 'a *itself* \Rightarrow 'b::*c-type* *itself* \Rightarrow *bool* **where**
fs-consistent *fs a b* \equiv *set* *fs* \subseteq *set* (*field-names* (*typ-info-t* *TYPE*('a)) (*typ-uinfo-t* *TYPE*('b)))

definition (**in** *c-type*) *field-offset-footprint* :: 'a *ptr* \Rightarrow (*qualified-field-name*) *list* \Rightarrow 'b *ptr set* **where**
field-offset-footprint *p fs* \equiv $\{Ptr \ \&(p \rightarrow k) \mid k. k \in \text{set } fs\}$

definition $sub\text{-}typ :: 'a::c\text{-}type\ itself \Rightarrow 'b::c\text{-}type\ itself \Rightarrow bool$ ($- \leq_\tau$ - [51, 51] 50) **where**
 $s \leq_\tau t \equiv typ\text{-}uinfo\text{-}t\ s \leq typ\text{-}uinfo\text{-}t\ t$

definition $sub\text{-}typ\text{-}proper :: 'a::c\text{-}type\ itself \Rightarrow 'b::c\text{-}type\ itself \Rightarrow bool$ ($- <_\tau$ - [51, 51] 50) **where**
 $s <_\tau t \equiv typ\text{-}uinfo\text{-}t\ s < typ\text{-}uinfo\text{-}t\ t$

definition $peer\text{-}typ :: 'a::c\text{-}type\ itself \Rightarrow 'b :: c\text{-}type\ itself \Rightarrow bool$ **where**
 $peer\text{-}typ\ a\ b \equiv typ\text{-}uinfo\text{-}t\ TYPE('a) = typ\text{-}uinfo\text{-}t\ TYPE('b) \vee$
 $typ\text{-}uinfo\text{-}t\ TYPE('a) \perp_t typ\text{-}uinfo\text{-}t\ TYPE('b)$

definition (**in** $c\text{-}type$) $guard\text{-}mono :: 'a\ ptr\text{-}guard \Rightarrow 'b::c\text{-}type\ ptr\text{-}guard \Rightarrow bool$ **where**
 $guard\text{-}mono\ g\ g' \equiv$
 $\forall n\ f\ p.\ g\ p \wedge$
 $field\text{-}lookup\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ f\ 0 = Some\ (typ\text{-}uinfo\text{-}t\ TYPE('b),n)$
 \longrightarrow
 $g'\ (Ptr\ (ptr\text{-}val\ p + of\text{-}nat\ n))$

primrec (**in** $c\text{-}type$) $sub\text{-}field\text{-}update\text{-}t ::$
 $qualified\text{-}field\text{-}name\ list \Rightarrow 'a\ ptr \Rightarrow 'a \Rightarrow 'b::c\text{-}type\ typ\text{-}heap \Rightarrow 'b\ typ\text{-}heap$ **where**
 $sft0: sub\text{-}field\text{-}update\text{-}t\ []\ p\ v\ s = s$
 $| sft1: sub\text{-}field\text{-}update\text{-}t\ (f\#\#fs)\ p\ (v::'a)\ s =$
 $(let\ s' = sub\text{-}field\text{-}update\text{-}t\ fs\ p\ v\ s\ in$
 $s'\ (Ptr\ \&(p \rightarrow f) \mapsto from\text{-}bytes\ (access\text{-}ti_0\ (field\text{-}typ\ TYPE('a)\ f)\ v)))\ |'$
 $dom\ (s::'b::c\text{-}type\ typ\text{-}heap)$

primrec (**in** $c\text{-}type$) $update\text{-}value\text{-}t :: (qualified\text{-}field\text{-}name)\ list \Rightarrow 'a \Rightarrow 'b \Rightarrow nat$
 $\Rightarrow 'b::c\text{-}type$

where
 $uvt0: update\text{-}value\text{-}t\ []\ v\ w\ x = w$
 $| uvt1: update\text{-}value\text{-}t\ (f\#\#fs)\ v\ (w::'b)\ x = (if\ x = field\text{-}offset\ TYPE('b)\ f\ then$
 $field\text{-}update\ (field\text{-}desc\ (field\text{-}typ\ TYPE('b)\ f))\ (to\text{-}bytes\text{-}p\ (v::'a))\ (w::'b::c\text{-}type)$
 $else\ update\text{-}value\text{-}t\ fs\ v\ w\ x)$

definition (**in** $c\text{-}type$) $super\text{-}field\text{-}update\text{-}t :: 'a\ ptr \Rightarrow 'a \Rightarrow 'b::c\text{-}type\ typ\text{-}heap \Rightarrow$
 $'b\ typ\text{-}heap$ **where**
 $super\text{-}field\text{-}update\text{-}t\ p\ v\ s \equiv \lambda q.$
 $if\ field\text{-}of\text{-}t\ p\ q$
 $then$
 $case\text{-}option\ None$
 $(\lambda w.\ Some\ (update\text{-}value\text{-}t\ (field\text{-}names\ (typ\text{-}info\text{-}t\ TYPE('b)))$


```

(typ-uinto-t TYPE('a)))
      v w (unat (ptr-val p - ptr-val q)))
    (s q)
  else s q

```

definition *heap-footprint* :: *heap-tyt-desc* ⇒ *typ-uinto* ⇒ *addr set* **where**
heap-footprint d t ≡ {*x*. ∃ *y*. *valid-footprint d y t* ∧ *x* ∈ {*y*} ∪ {*y*..*size-td t*}}

definition (in *c-type*) *ptr-safe* :: 'a *ptr* ⇒ *heap-tyt-desc* ⇒ *bool* **where**
ptr-safe p d ≡ *s-footprint p* ⊆ *dom-s d*

primrec

```

htd-update-list :: addr ⇒ typ-slice list ⇒ heap-tyt-desc ⇒ heap-tyt-desc
where
  hul0: htd-update-list p [] d = d
| hul1: htd-update-list p (x#xs) d = htd-update-list (p+1) xs (d(p := (True,snd
(d p) ++ x)))

```

definition *dom-tll* :: *addr* ⇒ *typ-slice list* ⇒ *s-addr set* **where**
dom-tll p xs ≡ {(*p* + *of-nat x, SIndexVal*) | *x*. *x* < *length xs*} ∪
 {(*p* + *of-nat x, SIndexTyp n*) | *x n*. *x* < *length xs* ∧ (*xs ! x*) *n* ≠
 None}

definition (in *c-type*) *typ-slices* :: 'a *itself* ⇒ *typ-slice list* **where**
typ-slices t ≡ *map* (λ*n*. *list-map* (*typ-slice-t* (*typ-uinto-t* TYPE('a)) *n*)) [0..*size-of*
 TYPE('a)]

definition (in *c-type*) *ptr-retyt* :: 'a *ptr* ⇒ *heap-tyt-desc* ⇒ *heap-tyt-desc* **where**
ptr-retyt p ≡ *htd-update-list* (*ptr-val p*) (*typ-slices* TYPE('a))

definition (in *c-type*) *field-fd* :: 'a *itself* ⇒ *qualified-field-name* ⇒ 'a *field-desc*
where
field-fd t n ≡ *field-desc* (*field-tyt t n*)

definition *tag-disj-tyt* :: 'a::*c-type* *itself* ⇒ 'b::*c-type* *itself* ⇒ *bool* (- ⊥_τ -) **where**
s ⊥_τ *t* ≡ *typ-uinto-t s* ⊥_t *typ-uinto-t t*

lemma *wf-heap-val-SIndexVal-STyp-simp* [*simp*]:
wf-heap-val s ⇒ *s* (*x, SIndexVal*) ≠ *Some* (*STyp t*)
apply (*clarsimp simp: wf-heap-val-def*)
apply (*drule spec [where x=x]*)
apply *clarsimp*
apply (*cases t, simp*)
apply *fast*
done

lemma *wf-heap-val-SIndexTyp-SValue-simp* [*simp*]:
wf-heap-val s $\implies s(x, SIndexTyp\ n) \neq Some\ (SValue\ v)$
apply (*unfold wf-heap-val-def*)
apply *clarify*
apply (*drule-tac x=x in spec*)
apply *clarsimp*
done

lemma (**in** *mem-type*) *field-tag-sub*:
field-lookup (typ-info-t TYPE('a)) f 0 = Some (t,n) \implies
 $\{\&(p \rightarrow f) .. +size-td\ t\} \subseteq \{ptr-val\ (p::'a\ ptr) .. +size-of\ TYPE('a)\}$
apply (*clarsimp simp: field-ti-def split: option.splits*)
apply (*drule intvlD, clarsimp simp: field-lvalue-def field-offset-def*)
apply (*drule field-lookup-export-uinfo-Some*)
apply (*subst add.assoc*)
apply (*subst Abs-fnat-homs*)
apply (*rule intvlI*)
apply (*simp add: size-of-def typ-uinfo-t-def*)
apply (*drule td-set-field-lookupD*)
apply (*drule td-set-offset-size*)
apply (*simp*)
done

lemma *typ-slice-t-not-empty* [*simp*]:
typ-slice-t t n $\neq []$
by (*cases t, simp*)

lemma *list-map-typ-slice-t-not-empty* [*simp*]:
list-map (typ-slice-t t n) $\neq Map.empty$
by (*simp add: list-map-def*)

lemma (**in** *c-type*) *s-footprint*:
s-footprint (p::'a ptr) =
 $\{(ptr-val\ p + of-nat\ x, k) \mid x\ k.$
 $x < size-of\ TYPE('a) \wedge$
 $(k = SIndexVal \vee (\exists n. k = SIndexTyp\ n \wedge n < length\ (typ-slice-t\ (typ-uinfo-t$
 $TYPE('a)\ x)))\}$
by (*auto simp: s-footprint-def s-footprint-untyped-def size-of-def*)

lemma (**in** *mem-type*) *ptr-val-SIndexVal-in-s-footprint* [*simp*]:
 $(ptr-val\ p, SIndexVal) \in s-footprint\ (p::'a\ ptr)$
apply (*simp add: s-footprint*)
apply (*rule exI [where x=0]*)
by *auto*

lemma (**in** *c-type*) *s-footprintI*:
 $\llbracket n < length\ (typ-slice-t\ (typ-uinfo-t\ TYPE('a)\ x)); x < size-of\ TYPE('a) \rrbracket \implies$
 $(ptr-val\ p + of-nat\ x, SIndexTyp\ n) \in s-footprint\ (p::'a\ ptr)$
apply (*simp add: s-footprint*)

apply(rule *exI* [where $x=x$])
apply *auto*
done

lemma (in *c-type*) *s-footprintI2*:
 $x < \text{size-of } \text{TYPE}('a) \implies (\text{ptr-val } p + \text{of-nat } x, \text{SIndexVal}) \in \text{s-footprint } (p::'a \text{ ptr})$
apply(simp add: *s-footprint*)
apply(rule *exI* [where $x=x$])
apply *auto*
done

lemma (in *c-type*) *s-footprintD*:
 $(x,k) \in \text{s-footprint } p \implies x \in \{\text{ptr-val } (p::'a \text{ ptr})..+\text{size-of } \text{TYPE}('a)\}$
by (*auto simp: s-footprint elim: intvlI*)

lemma (in *mem-type*) *s-footprintD2*:
 $(x,\text{SIndexTyp } n) \in \text{s-footprint } (p::'a \text{ ptr}) \implies$
 $n < \text{length } (\text{typ-slice-t } (\text{typ-uinfo-t } \text{TYPE}('a)) (\text{unat } (x - \text{ptr-val } p)))$
by (*clarsimp simp add: s-footprint*)
(metis len-of-addr-card max-size nat-less-le of-nat-inverse order-less-le-trans)

lemma (in *c-type*) *s-footprint-restrict*:
 $x \in \text{s-footprint } p \implies (s \mid' \text{s-footprint } p) x = s x$
by (*rule restrict-in*)

lemma *restrict-s-fst*:
 $\text{fst } (\text{restrict-s } d X x) \implies \text{fst } (d x)$
by (*clarsimp simp: restrict-s-def*)

lemma *restrict-s-map-le* [*simp*]:
 $\text{snd } (\text{restrict-s } d X x) \subseteq_m \text{snd } (d x)$
by (*auto simp: restrict-s-def map-le-def*)

lemma *dom-list-map* [*simp*]:
 $\text{dom } (\text{list-map } xs) = \{0..<\text{length } xs\}$
by (*auto simp: list-map-def*)

lemma *list-map* [*simp*]:
 $n < \text{length } xs \implies \text{list-map } xs n = \text{Some } (xs ! n)$
by (*force simp: list-map-def set-zip*)

lemma *list-map-eq*:
 $\text{list-map } xs n = (\text{if } n < \text{length } xs \text{ then } \text{Some } (xs ! n) \text{ else } \text{None})$
by (*force simp: list-map-def set-zip*)

lemma *valid-footprintI*:
 $\llbracket 0 < \text{size-td } t; \bigwedge y. y < \text{size-td } t \implies \text{list-map } (\text{typ-slice-t } t y) \subseteq_m \text{snd } (d (x + \text{of-nat } y)) \wedge$

$$\text{fst } (d \ (x + \text{of-nat } y)) \] \implies$$

$$\text{valid-footprint } d \ x \ t$$
by (*simp add: valid-footprint-def*)

lemma *valid-footprintD*:

$$\llbracket \text{valid-footprint } d \ x \ t; \ y < \text{size-td } t \rrbracket \implies$$

$$\text{list-map } (\text{typ-slice-t } t \ y) \subseteq_m \text{snd } (d \ (x + \text{of-nat } y)) \wedge$$

$$\text{fst } (d \ (x + \text{of-nat } y))$$
by (*simp add: valid-footprint-def Let-def*)

lemma (**in** *c-type*) *h-t-valid-taut*:

$$d, g \models_t p \implies d, (\lambda x. \text{True}) \models_t p$$
by (*simp add: h-t-valid-def*)

lemma (**in** *c-type*) *h-t-valid-restrict*:

$$\text{restrict-s } d \ (\text{s-footprint } p), g \models_t p = d, g \models_t p$$
apply (*simp add: h-t-valid-def valid-footprint-def Let-def*)
apply (*fastforce simp: restrict-s-def map-le-def size-of-def intro: s-footprintI s-footprintI2*)
done

lemma (**in** *c-type*) *h-t-valid-restrict2*:

$$\llbracket d, g \models_t p; \text{restrict-s } d \ (\text{s-footprint } p) = \text{restrict-s } d' \ (\text{s-footprint } p) \rrbracket \implies d', g \models_t (p::'a \ \text{ptr})$$
apply (*clarsimp simp: h-t-valid-def valid-footprint-def Let-def*)
apply (*rule conjI; clarsimp?*)
apply (*clarsimp simp: map-le-def*)
apply (*drule-tac x=(ptr-val p + of-nat y) in fun-cong*)
apply (*clarsimp simp: restrict-s-def*)
apply (*drule-tac x=a in fun-cong*)
apply (*fastforce simp: size-of-def intro: s-footprintI split: if-split-asm*)
apply (*drule-tac x=(ptr-val p + of-nat y) in fun-cong*)
apply (*fastforce simp: size-of-def restrict-s-def intro: s-footprintI2*)
done

lemma *lift-state-wf-heap-val* [*simp*]:

$$\text{wf-heap-val } (\text{lift-state } (h, d))$$
unfolding *wf-heap-val-def*
by (*auto simp: lift-state-def split: option.splits*)

lemma *wf-hs-proj-d*:

$$\text{fst } (\text{proj-d } s \ x) \implies s \ (x, \text{SIndexVal}) \neq \text{None}$$
by (*auto simp: proj-d-def*)

lemma (**in** *c-type*) *s-valid-g*:

$$s, g \models_s p \implies g \ p$$
by (*simp add: s-valid-def h-t-valid-def*)

lemma (**in** *c-type*) *lift-typ-heap-if*:

lift-typ-heap $g\ s = (\lambda(p::'a\ ptr). \text{if } s, g \models_s p \text{ then Some (from-bytes (heap-list-s } s \text{ (size-of TYPE('a)) (ptr-val } p))) \text{ else None})$
by (*force simp: lift-typ-heap-def*)

lemma (*in c-type*) *lift-typ-heap-s-valid*:
lift-typ-heap $g\ s\ p = \text{Some } x \implies s, g \models_s p$
by (*simp add: lift-typ-heap-if split: if-split-asm*)

lemma (*in c-type*) *lift-typ-heap-g*:
lift-typ-heap $g\ s\ p = \text{Some } x \implies g\ p$
by (*fast dest: lift-typ-heap-s-valid s-valid-g*)

lemma *lift-state-empty* [*simp*]:
lift-state $(h, \text{empty-htd}) = \text{Map.empty}$
by (*auto simp: lift-state-def empty-htd-def split: s-heap-index.splits*)

lemma *lift-state-eqI*:
 $\llbracket h\ x = h'\ x; d\ x = d'\ x \rrbracket \implies \text{lift-state } (h, d)\ (x, k) = \text{lift-state } (h', d')\ (x, k)$
by (*clarsimp simp: lift-state-def split: s-heap-index.splits*)

lemma *proj-h-lift-state*:
 $\text{fst } (d\ x) \implies \text{proj-h } (\text{lift-state } (h, d))\ x = h\ x$
by (*clarsimp simp: proj-h-def lift-state-def*)

lemma *lift-state-proj-simp* [*simp*]:
lift-state $(\text{proj-h } (\text{lift-state } (h, d)), d) = \text{lift-state } (h, d)$
by (*auto simp: lift-state-def proj-h-def split: s-heap-index.splits option.splits*)

lemma *f2l-length* [*simp*]:
 $\text{length } (\text{fun2list } f\ n) = n$
by (*simp add: fun2list-def*)

lemma *GREATEST-lt* [*simp*]:
 $0 < n \implies (\text{GREATEST } x. x < n) = n - (1::\text{nat})$
by (*rule Greatest-equality; simp*)

lemma *fun2list-nth* [*simp*]:
 $x < n \implies \text{fun2list } f\ n\ !\ x = f\ x$
by (*clarsimp simp: fun2list-def*)

lemma *proj-d-lift-state*:
 $\text{proj-d } (\text{lift-state } (h, d)) = d$
apply(*rule ext*)
subgoal for x
apply(*cases d x*)
apply(*auto simp: proj-d-def lift-state-def Let-def split: option.splits*)
done
done

lemma *lift-state-proj* [*simp*]:
 $wf\text{-heap}\text{-val } s \implies \text{lift}\text{-state } (\text{proj}\text{-h } s, \text{proj}\text{-d } s) = s$
apply (*clarsimp simp: proj-h-def proj-d-def lift-state-def fun-eq-iff*
split: if-split-asm s-heap-index.splits option.splits)
apply *safe*
apply (*metis s-heap-tag.simps s-heap-value.exhaust wf-heap-val-SIndexTyp-SValue-simp*)
apply (*metis id-apply s-heap-value.exhaust s-heap-value.simps(5) wf-heap-val-SIndexVal-STyp-simp*)
apply (*metis s-heap-tag.simps s-heap-value.exhaust wf-heap-val-SIndexTyp-SValue-simp*)
done

lemma *lift-state-Some*:
 $\text{lift}\text{-state } (h, d) (p, SIndexTyp\ n) = \text{Some } t \implies \text{snd } (d\ p)\ n = \text{Some } (s\text{-heap}\text{-tag } t)$
apply (*simp add: lift-state-def split: option.splits split: if-split-asm*)
apply (*cases t; simp*)
done

lemma *lift-state-Some2*:
 $\text{snd } (d\ p)\ n = \text{Some } t \implies$
 $\exists v. \text{lift}\text{-state } (h, d) (p, SIndexTyp\ n) = \text{Some } (STyp\ t)$
by (*simp add: lift-state-def split: option.split*)

lemma (**in** *c-type*) *h-t-s-valid*:
 $\text{lift}\text{-state } (h, d), g \models_s p = d, g \models_t p$
by (*simp add: s-valid-def proj-d-lift-state*)

lemma (**in** *c-type*) *lift-t*:
 $\text{lift}\text{-typ}\text{-heap } g (\text{lift}\text{-state } s) = \text{lift}\text{-t } g\ s$
by (*simp add: lift-t-def*)

lemma (**in** *c-type*) *lift-t-h-t-valid*:
 $\text{lift}\text{-t } g (h, d)\ p = \text{Some } x \implies d, g \models_t p$
by (*force simp: lift-t-def h-t-s-valid dest: lift-typ-heap-s-valid*)

lemma (**in** *c-type*) *lift-t-g*:
 $\text{lift}\text{-t } g\ s\ p = \text{Some } x \implies g\ p$
by (*force simp: lift-t-def dest: lift-typ-heap-g*)

lemma (**in** *c-type*) *lift-t-proj* [*simp*]:
 $wf\text{-heap}\text{-val } s \implies \text{lift}\text{-t } g (\text{proj}\text{-h } s, \text{proj}\text{-d } s) = \text{lift}\text{-typ}\text{-heap } g\ s$
by (*simp add: lift-t-def*)

lemma *valid-footprint-Some*:
assumes *valid: valid-footprint d p t and size: x < size-td t*
shows *fst (d (p + of-nat x))*
proof (*cases of-nat x=(0::addr)*)
case *True*
with *valid show ?thesis* **by** (*force simp add: valid-footprint-def Let-def*)
next
case *False*

with *size valid* **show** *?thesis* **by** (*force simp: valid-footprint-def Let-def*)
qed

lemma (**in** *c-type*) *h-t-valid-Some*:
 $\llbracket d, g \models_t (p::'a \text{ ptr}); x < \text{size-of } \text{TYPE}('a) \rrbracket \implies$
 $\text{fst } (d (\text{ptr-val } p + \text{of-nat } x))$
by (*force simp: h-t-valid-def size-of-def dest: valid-footprint-Some*)

lemma (**in** *c-type*) *h-t-valid-ptr-safe*:
 $d, g \models_t (p::'a \text{ ptr}) \implies \text{ptr-safe } p \ d$
apply(*clarsimp simp: ptr-safe-def h-t-valid-def valid-footprint-def s-footprint-def*
s-footprint-untyped-def dom-s-def size-of-def Let-def)
by (*metis (mono-tags, opaque-lifting) domIff list-map map-le-def option.simps(3)*
surj-pair)

lemma (**in** *c-type*) *lift-t-ptr-safe*:
 $\text{lift-t } g \ (h, d) \ (p::'a \text{ ptr}) = \text{Some } x \implies \text{ptr-safe } p \ d$
by (*fast dest: lift-t-h-t-valid h-t-valid-ptr-safe*)

lemma (**in** *c-type*) *s-valid-Some*:
 $\llbracket d, g \models_s (p::'a \text{ ptr}); x < \text{size-of } \text{TYPE}('a) \rrbracket \implies$
 $d (\text{ptr-val } p + \text{of-nat } x, \text{SIndexVal}) \neq \text{None}$
by (*auto simp: s-valid-def dest!: h-t-valid-Some wf-hs-proj-d split: option.splits*)

lemma *heap-list-s-heap-list-dom*:
 $\bigwedge n. (\lambda x. (x, \text{SIndexVal})) \ ' \{n..+k\} \subseteq \text{dom-s } d \implies$
 $\text{heap-list-s } (\text{lift-state } (h, d)) \ k \ n = \text{heap-list } h \ k \ n$

proof (*induct k*)
case 0 **show** *?case* **by** (*simp add: heap-list-s-def*)
next
case (*Suc k*)
hence $(\lambda x. (x, \text{SIndexVal})) \ ' \{n + 1..+k\} \subseteq \text{dom-s } d$
by (*force intro: intvl-plus-sub-Suc subset-trans simp: image-def*)
with *Suc* **have** $\text{heap-list-s } (\text{lift-state } (h, d)) \ k \ (n + 1) =$
 $\text{heap-list } h \ k \ (n + 1)$ **by** *simp*
moreover from *this Suc* **have** $(n, \text{SIndexVal}) \in \text{dom-s } d$
by (*force simp: dom-s-def image-def intro: intvl-self*)
ultimately show *?case*
by (*auto simp add: heap-list-s-def proj-h-lift-state dom-s-def*)
qed

lemma (**in** *c-type*) *heap-list-s-heap-list*:
 $d, (\lambda x. \text{True}) \models_t (p::'a \text{ ptr}) \implies$
 $\text{heap-list-s } (\text{lift-state } (h, d)) \ (\text{size-of } \text{TYPE}('a)) \ (\text{ptr-val } p)$
 $= \text{heap-list } h \ (\text{size-of } \text{TYPE}('a)) \ (\text{ptr-val } p)$
apply(*drule h-t-valid-ptr-safe*)
apply(*clarsimp simp: ptr-safe-def*)
apply(*subst heap-list-s-heap-list-dom*)
apply(*clarsimp simp: dom-s-def*)

```

apply(drule-tac c=(x,SIndexVal) in subsetD)
apply(clarsimp simp: intvl-def)
apply(erule s-footprintI2)
apply clarsimp+
done

```

```

lemma (in c-type) lift-t-if:
  lift-t g (h,d) = ( $\lambda p$ . if d,g  $\models_t$  p then Some (h-val h (p::'a ptr)) else None)
by (force simp: lift-t-def lift-tyr-heap-if h-t-s-valid h-val-def
      heap-list-s-heap-list h-t-valid-taut)

```

```

lemma (in c-type) lift-lift-t:
  d,g  $\models_t$  (p::'a ptr)  $\implies$  lift h p = the (lift-t g (h,d) p)
by (simp add: lift-t-if lift-def)

```

```

lemma (in c-type) lift-t-lift:
  lift-t g (h,d) (p::'a ptr) = Some v  $\implies$  lift h p = v
by (simp add: lift-t-if lift-def split: if-split-asm)

```

```

lemma heap-update-list-same:
   $\bigwedge h p k. \llbracket 0 < k; k \leq \text{addr-card} - \text{length } v \rrbracket \implies \text{heap-update-list } (p + \text{of-nat } k)
  v h p = h p$ 
```

proof (induct v)

```

  case Nil show ?case by simp
next
  case (Cons x xs)
  have heap-update-list (p + of-nat k) (x # xs) h p =
    heap-update-list (p + of-nat (k + 1)) xs (h(p + of-nat k := x)) p
  by (simp add: ac-simps)
  also have ... = (h(p + of-nat k := x)) p
  proof -
    from Cons have k + 1  $\leq$  addr-card - length xs by simp
    with Cons show ?thesis by (simp only:)
  qed
  also have ... = h p
  proof -
    from Cons have of-nat k  $\neq$  (0::addr)
    by - (erule of-nat-neq-0, simp add: addr-card)
    thus ?thesis by clarsimp
  qed
  finally show ?case .
qed

```

```

lemma heap-list-update:
   $\bigwedge h p. \text{length } v \leq \text{addr-card} \implies
  \text{heap-list } (\text{heap-update-list } p v h) (\text{length } v) p = v$ 
```

proof (induct v)

```

  case Nil thus ?case by simp
next

```


case (*Cons x xs*)
hence *heap-update-list* ($p + \text{of-nat } 1$) *xs* ($h(p := x)$) $p = (h(p := x)) p$
by $-$ (*rule heap-update-list-same, auto*)
with *Cons show ?case by simp*
qed

lemma (**in** *mem-type*) *heap-list-update-to-bytes*:
heap-list (*heap-update-list* p (*to-bytes* ($v::'a$) (*heap-list* h (*size-of* *TYPE*('a)) p))
 h)
 $(\text{size-of } \text{TYPE}('a)) p = \text{to-bytes } v$ (*heap-list* h (*size-of* *TYPE*('a)) p)
by (*metis (mono-tags) heap-list-length heap-list-update len less-imp-le max-size*)

lemma (**in** *mem-type*) *heap-list-update-to-bytes-padding*:
 $\text{length } bs = \text{size-of } \text{TYPE}('a) \implies \text{heap-list}$ (*heap-update-list* p (*to-bytes* ($v::'a$)
 bs) h)
 $(\text{size-of } \text{TYPE}('a)) p = \text{to-bytes } v$ bs
by (*metis heap-list-update local.len local.max-size nless-le*)

lemma (**in** *mem-type*) *h-val-heap-update[simp]*:
h-val (*heap-update* p v h) $p = (v::'a)$
by (*simp add: h-val-def heap-update-def heap-list-update-to-bytes inv*)

lemma (**in** *mem-type*) *h-val-heap-update-padding*:
fixes $p::'a$ *ptr*
shows $\text{length } bs = \text{size-of } \text{TYPE}('a) \implies \text{h-val}$ (*heap-update-padding* p v bs h) p
 $= v$
apply (*simp add: heap-update-padding-def h-val-def*)
by (*metis heap-list-update inv le-eq-less-or-eq len max-size*)

lemma *heap-list-update-disjoint-same*:
shows $\bigwedge q. \{p..+\text{length } v\} \cap \{q..+k\} = \{\} \implies$
 heap-list (*heap-update-list* p v h) k $q = \text{heap-list } h$ k q

proof (*induct k*)
case 0 **show** *?case by simp*

next
case (*Suc n*)
hence $\{p..+\text{length } v\} \cap \{q + 1..+n\} = \{\}$
by (*force intro: intvl-plus-sub-Suc*)
with *Suc have* *heap-list* (*heap-update-list* p v h) n ($q + 1$) =
 $\text{heap-list } h$ n ($q + 1$) **by** *simp*
moreover have *heap-update-list* ($q + \text{of-nat } (\text{unat } (p - q)))$ v h $q = h$ q
proof (*cases v*)
case *Nil thus ?thesis by simp*

next
case (*Cons y ys*)
with *Suc have* $0 < \text{unat } (p - q)$
by (*cases p=q*)
 $(\text{simp add: intvl-start-inter unat-gt-0})+$
moreover have $\text{unat } (p - q) \leq \text{addr-card} - \text{length } v$ (**is** *?G*)

```

proof (rule ccontr)
  assume  $\neg ?G$ 
  moreover from Suc have  $q \notin \{p..+length\ v\}$ 
    by (fast intro: intvl-self)
  ultimately show False
    by (simp only: linorder-not-le len-of-addr-card [symmetric])
      (frule-tac p=q in intvl-self-offset, force+)
  qed
  ultimately show ?thesis by (rule heap-update-list-same)
qed
ultimately show ?case by simp
qed

```

```

lemma heap-update-nmem-same:
  assumes nmem:  $q \notin \{p..+length\ v\}$ 
  shows heap-update-list p v h q = h q
proof -
  from nmem have heap-list (heap-update-list p v h) 1 q = heap-list h 1 q
    by - (rule heap-list-update-disjoint-same, force dest: intvl-Suc)
  thus ?thesis by simp
qed

```

```

lemma heap-update-mem-same:
   $\llbracket q \in \{p..+length\ v\}; length\ v < addr-card \rrbracket \implies$ 
    heap-update-list p v h q = heap-update-list p v h' q
  by (induct v arbitrary: p h h'; simp)
  (fastforce dest: intvl-neq-start simp: heap-update-list-same [where k=1, simplified])

```

```

lemma sub-tag-proper-TypScalar [simp]:
   $\neg t < TypDesc\ alg\ n' (TypScalar\ n\ alg\ d)\ nm$ 
  by (simp add: typ-tag-lt-def typ-tag-le-def)

```

```

lemma tag-disj-com [simp]:
   $f \perp_t g = g \perp_t f$ 
  by (force simp: tag-disj-def)

```

```

lemma typ-slice-set':
   $\forall m\ n. fst\ 'set\ (typ-slice-t\ s\ n) \subseteq fst\ 'td-set\ s\ m$ 
   $\forall m\ n. fst\ 'set\ (typ-slice-struct\ st\ n) \subseteq fst\ 'td-set-struct\ st\ m$ 
   $\forall m\ n. fst\ 'set\ (typ-slice-list\ xs\ n) \subseteq fst\ 'td-set-list\ xs\ m$ 
   $\forall m\ n. fst\ 'set\ (typ-slice-tuple\ x\ n) \subseteq fst\ 'td-set-tuple\ x\ m$ 
  apply(induct s and st and xs and x, all <clarsimp simp: ladder-set-def>)
  apply auto[1]
  apply (rule conjI; clarsimp)
  apply force
  apply(thin-tac All P for P)
  apply force
done

```

lemma *typ-slice-set*:
 $\text{fst } ' \text{ set } (\text{typ-slice-t } s \ n) \subseteq \text{fst } ' \text{ td-set } s \ m$
using *typ-slice-set'(1)* [of *s*] **by** *clarsimp*

lemma *typ-slice-struct-set*:
 $(s, t) \in \text{set } (\text{typ-slice-struct } st \ n) \implies \exists k. (s, k) \in \text{td-set-struct } st \ m$
using *typ-slice-set'(2)* [of *st*] **by** *force*

lemma *typ-slice-set-sub*:
 $\text{typ-slice-t } s \ m \leq \text{typ-slice-t } t \ n \implies$
 $\text{fst } ' \text{ set } (\text{typ-slice-t } s \ m) \subseteq \text{fst } ' \text{ set } (\text{typ-slice-t } t \ n)$
by (*force simp: image-def prefix-def less-eq-list-def*)

lemma *ladder-set-self*:
 $s \in \text{fst } ' \text{ set } (\text{typ-slice-t } s \ n)$
by (*cases s*) (*auto simp: ladder-set-def*)

lemma *typ-slice-sub*:
 $\text{typ-slice-t } s \ m \leq \text{typ-slice-t } t \ n \implies s \leq t$
apply (*drule typ-slice-set-sub*)
using *ladder-set-self* [of *s m*] *typ-slice-set* [of *t n 0*]
apply (*force simp: typ-tag-le-def*)
done

lemma *typ-slice-self*:
 $(s, \text{True}) \in \text{set } (\text{typ-slice-t } s \ 0)$
by (*cases s*) *simp*

lemma *typ-slice-struct-nmem*:
 $(\text{TypDesc } \text{algn } st \ nm, n) \notin \text{set } (\text{typ-slice-struct } st \ k)$
by (*fastforce dest: typ-slice-struct-set td-set-struct-size-lte*)

lemma *typ-slice-0-prefix*:
 $0 < n \implies \neg \text{typ-slice-t } t \ 0 \leq \text{typ-slice-t } t \ n \wedge \neg \text{typ-slice-t } t \ n \leq \text{typ-slice-t } t \ 0$
by (*cases t*) (*fastforce simp: less-eq-list-def typ-slice-struct-nmem dest: set-mono-prefix*)

lemma *prefix-eq-nth*:
 $xs \leq ys = ((\forall i. i < \text{length } xs \longrightarrow xs ! i = ys ! i) \wedge \text{length } xs \leq \text{length } ys)$
apply (*rule iffI; clarsimp simp: less-eq-list-def prefix-def nth-append*)
by (*metis append-take-drop-id nth-take-lemma order-refl take-all*)

lemma *map-prefix-same-cases*:
 $\llbracket \text{list-map } xs \subseteq_m f; \text{list-map } ys \subseteq_m f \rrbracket \implies xs \leq ys \vee ys \leq xs$
using *linorder-linear* [**where** $x = \text{length } xs$ **and** $y = \text{length } ys$]
apply (*clarsimp simp: prefix-eq-nth map-le-def prefix-def*)
apply (*erule disjE*)
apply *clarsimp*
subgoal for *i*

```

  by ((drule-tac x=i in bspec, simp)+, force dest: sym)
apply clarsimp
subgoal for i
  by ((drule-tac x=i in bspec, simp)+, force dest: sym)
done

```

lemma *list-map-mono*:

```

xs ≤ ys ⇒ list-map xs ⊆m list-map ys
by (auto simp: map-le-def prefix-def nth-append less-eq-list-def)

```

lemma *map-list-map-trans*:

```

[[ xs ≤ ys; list-map ys ⊆m f ]] ⇒ list-map xs ⊆m f
apply(drule list-map-mono)
apply(erule (1) map-le-trans)
done

```

lemma *valid-footprint-le*:

```

valid-footprint d x t ⇒ size-td t ≤ addr-card
apply(clarsimp simp: valid-footprint-def Let-def)
apply(rule ccontr)
apply(frule-tac x=addr-card in spec)
apply(drule-tac x=0 in spec)
apply clarsimp
apply(drule (1) map-prefix-same-cases)
apply(simp add: typ-slice-0-prefix addr-card)
done

```

lemma *typ-slice-True-set'*:

```

∀ s k m. (s, True) ∈ set (typ-slice-t t k) → (s, k+m) ∈ td-set t m
∀ s k m. (s, True) ∈ set (typ-slice-struct st k) → (s, k+m) ∈ td-set-struct st m
∀ s k m. (s, True) ∈ set (typ-slice-list xs k) → (s, k+m) ∈ td-set-list xs m
∀ s k m. (s, True) ∈ set (typ-slice-tuple x k) → (s, k+m) ∈ td-set-tuple x m

```

proof (induct t and st and xs and x)

```

  case (TypDesc nat typ-struct list)
  then show ?case by auto

```

next

```

  case (TypScalar nat1 nat2 a)
  then show ?case by auto

```

next

```

  case (TypAggregate list)
  then show ?case by auto

```

next

```

  case Nil-typ-desc
  then show ?case by auto

```

next

```

  case (Cons-typ-desc dt-tuple list)
  then show ?case
  apply clarsimp
  apply(cases dt-tuple, clarsimp)

```

```

  subgoal for  $s\ k\ m\ a\ b$ 
  apply(thin-tac All P for P)
  apply(drule spec [where  $x=s$ ])
  apply(drule spec [where  $x=k - \text{size-td } a$ ])
  apply clarsimp
  apply(drule spec [where  $x=m + \text{size-td } a$ ])
  apply simp
  done
done
next
case (DTuple-typ-desc typ-desc list b)
then show ?case by auto
qed

lemma typ-slice-True-set:
   $(s, \text{True}) \in \text{set } (\text{typ-slice-t } t\ k) \implies (s, k+m) \in \text{td-set } t\ m$ 
  by (simp add: typ-slice-True-set')

lemma typ-slice-True-prefix:
   $\text{typ-slice-t } s\ 0 \leq \text{typ-slice-t } t\ k \implies (s, k) \in \text{td-set } t\ 0$ 
  using typ-slice-self [of  $s$ ] typ-slice-True-set [of  $s\ t\ k\ 0$ ]
  by (force simp: less-eq-list-def dest: set-mono-prefix)

lemma tag-sub-prefix [simp]:
   $t < s \implies \neg \text{typ-slice-t } s\ m \leq \text{typ-slice-t } t\ n$ 
  by (fastforce dest: typ-slice-sub)

lemma tag-disj-prefix [simp]:
   $s \perp_t t \implies \neg \text{typ-slice-t } s\ m \leq \text{typ-slice-t } t\ n$ 
  by (auto dest: typ-slice-sub simp: tag-disj-def typ-slice-sub)

lemma typ-slice-0-True':
   $\forall x. x \in \text{set } (\text{typ-slice-t } t\ 0) \longrightarrow \text{snd } x = \text{True}$ 
   $\forall x. x \in \text{set } (\text{typ-slice-struct } st\ 0) \longrightarrow \text{snd } x = \text{True}$ 
   $\forall x. x \in \text{set } (\text{typ-slice-list } xs\ 0) \longrightarrow \text{snd } x = \text{True}$ 
   $\forall x. x \in \text{set } (\text{typ-slice-tuple } y\ 0) \longrightarrow \text{snd } x = \text{True}$ 
  by (induct  $t$  and  $st$  and  $xs$  and  $y$ ) auto

lemma typ-slice-0-True:
   $x \in \text{set } (\text{typ-slice-t } t\ 0) \implies \text{snd } x = \text{True}$ 
  by (simp add: typ-slice-0-True')

lemma typ-slice-False-self:
   $k \neq 0 \implies (t, \text{False}) \in \text{set } (\text{typ-slice-t } t\ k)$ 
  by (cases  $t$ ) simp

lemma tag-prefix-True:
   $\text{typ-slice-t } s\ k \leq \text{typ-slice-t } t\ 0 \implies k = 0$ 
  using typ-slice-0-True [of  $(s, \text{False})\ t$ ]

```

```

apply(clarsimp simp: less-eq-list-def dest!: set-mono-prefix)
apply(rule ccontr)
apply(fastforce dest!: typ-slice-False-self[where t=s and k=k])
done

```

lemma *valid-footprint-neq-nmem:*

```

assumes valid-p: valid-footprint d p f and
          valid-q: valid-footprint d q g and
          neq: p ≠ q and
          disj: f ⊥t g ∨ f=g

```

```

shows p ∉ {q..+size-td g} (is ?G)

```

proof –

```

from assms show ?thesis

```

```

apply(clarsimp simp: valid-footprint-def intvl-def Let-def)

```

```

apply(erule disjE)

```

```

apply(drule-tac x=0 in spec)

```

```

apply (fastforce dest: map-prefix-same-cases)

```

```

apply(drule-tac x=0 in spec)

```

```

apply(drule-tac x=k in spec)

```

```

apply clarsimp

```

```

by (metis add.comm-neutral map-prefix-same-cases neq semiring-1-class.of-nat-0)

```

```

          typ-slice-0-prefix zero-less-iff-neq-zero)

```

qed

lemma *valid-footprint-sub:*

```

assumes valid-p: valid-footprint d p s

```

```

assumes valid-q: valid-footprint d q t

```

```

assumes sub: ¬ t < s

```

```

shows p ∉ {q..+size-td t} ∨ field-of (p - q) (s) (t) (is ?G)

```

proof –

```

from assms show ?thesis

```

```

apply clarsimp

```

```

apply(insert valid-footprint-le[OF valid-q])

```

```

apply(clarsimp simp: valid-footprint-def Let-def)

```

```

apply(drule-tac x=0 in spec)

```

```

apply clarsimp

```

```

apply(drule intvlD)

```

```

apply clarsimp

```

```

apply(drule-tac x=k in spec)

```

```

apply clarsimp

```

```

apply(drule (1) map-prefix-same-cases)

```

```

apply(erule disjE)

```

```

prefer 2

```

```

apply(frule typ-slice-sub)

```

```

apply(subgoal-tac k = 0)

```

```

prefer 2

```

```

apply(rule ccontr, simp)

```

```

apply(simp add: typ-slice-0-prefix)

```

```

    apply simp

    apply (drule typ-slice-True-prefix)
    apply (clarsimp simp: field-of-def)
    apply (simp only: unat-simps)
    done
qed

lemma valid-footprint-sub2:
  [| valid-footprint d p s; valid-footprint d q t; ¬ t < s |] ⇒
    q ∉ {p..+size-td s} ∨ p=q
  apply (clarsimp simp: valid-footprint-def Let-def)
  apply (drule intvlD)
  apply clarsimp
  subgoal for k
    apply (drule spec [where x=k])
    apply clarsimp
    apply (drule spec [where x=0])
    apply clarsimp
    apply (drule (1) map-prefix-same-cases)
    apply (cases k=0)
    apply simp
    apply (erule disjE)
    prefer 2
    apply (frule typ-slice-sub)
    apply (drule order-le-imp-less-or-eq[where x=t])
    apply clarsimp
    apply (simp add: typ-slice-0-prefix)
    apply (drule tag-prefix-True)
    apply simp
    done
  done

lemma valid-footprint-neq-disjoint:
  [| valid-footprint d p s; valid-footprint d q t; ¬ t < s;
    ¬ field-of (p - q) (s) (t) |] ⇒ {p..+size-td s} ∩ {q..+size-td t} = {}
  apply (rule ccontr)
  apply (drule intvl-inter)
  apply (erule disjE)
  apply (drule (2) valid-footprint-sub)
  apply clarsimp
  apply (frule (1) valid-footprint-sub2, assumption)
  apply (frule (1) valid-footprint-sub2)
  apply simp
  apply simp
  apply (clarsimp simp: field-of-def)
  apply (clarsimp simp: valid-footprint-def Let-def)
  apply (drule-tac x=0 in spec)+
  apply clarsimp

```

```

apply(drule (1) map-prefix-same-cases [where xs=typ-slice-t s 0])
apply(erule disjE)
apply(drule typ-slice-True-prefix)
apply simp
apply(drule typ-slice-sub)
apply(drule order-le-imp-less-or-eq)
apply simp
done

```

```

lemma sub-typ-proper-not-same [simp]:
   $\neg t <_{\tau} t$ 
  by (simp add: sub-typ-proper-def)

```

```

lemma sub-typ-proper-not-simple [simp]:
   $\neg \text{TYPE}('a::c\text{-type}) <_{\tau} \text{TYPE}('b::\text{simple-mem-type})$ 
  apply(cases typ-uinfo-t TYPE('b))
  subgoal for x1 typ-struct
    by(cases typ-struct, auto simp: sub-typ-proper-def)
  done

```

```

lemma field-of-sub:
  field-of p s t  $\implies$  s  $\leq$  t
  by (auto simp: field-of-def typ-tag-lt-def typ-tag-le-def)

```

```

lemma h-t-valid-neq-disjoint:
  [ d,g  $\models_t$  (p::'a::c-type ptr); d,g'  $\models_t$  (q::'b::c-type ptr);
     $\neg \text{TYPE}('b) <_{\tau} \text{TYPE}('a)$ ;  $\neg$  field-of-t p q ]  $\implies$  {ptr-val p..+size-of TYPE('a)}
   $\cap$ 
    {ptr-val q..+size-of TYPE('b)} = {}
  by (fastforce dest: valid-footprint-neq-disjoint
      simp: size-of-def h-t-valid-def sub-typ-proper-def field-of-t-def)

```

```

lemma field-ti-sub-typ:
  [ field-ti (TYPE('b::mem-type)) f = Some t; export-uinfo t = (typ-uinfo-t TYPE('a::c-type)) ]
   $\implies$ 
    TYPE('a)  $\leq_{\tau}$  TYPE('b)
  by (auto simp: field-ti-def sub-typ-def typ-tag-le-def typ-uinfo-t-def
      dest!: td-set-export-uinfoD td-set-field-lookupD
      split: option.splits)

```

```

lemma h-t-valid-neq-disjoint-simple:
  [ d,g  $\models_t$  (p::'a::simple-mem-type ptr); d,g'  $\models_t$  (q::'b::simple-mem-type ptr) ]
   $\implies$  ptr-val p  $\neq$  ptr-val q  $\vee$  typ-uinfo-t TYPE('a) = typ-uinfo-t TYPE('b)
  apply(clarsimp simp: h-t-valid-def valid-footprint-def Let-def)
  apply(drule-tac x=0 in spec)+
  applyclarsimp
  apply(drule (1) map-prefix-same-cases[where xs=typ-slice-t (typ-uinfo-t TYPE('a))
  0])
  apply(erule disjE; drule typ-slice-sub)

```



```

apply(cases typ-info-t TYPE('b), rename-tac typ-struct xs)
subgoal for x1 typ-struct xs
  apply(cases typ-struct; fastforce simp: typ-tag-le-def typ-uinfo-t-def)
  done
apply(cases typ-info-t TYPE('a), rename-tac typ-struct xs)
subgoal for x1 typ-struct xs
  apply(cases typ-struct; simp add: typ-tag-le-def typ-uinfo-t-def)
  done
done

```

lemma *h-val-heap-same*:

```

fixes p::'a::mem-type ptr and q::'b::c-type ptr
assumes val-p: d,g ⊨t p and val-q: d,g' ⊨t q and
  subt: ¬ TYPE('a) <τ TYPE('b) and nf: ¬ field-of-t q p
shows h-val (heap-update p v h) q = h-val h q
proof –
  from val-p val-q subt nf
  have {ptr-val p..+length (to-bytes v (heap-list h (size-of TYPE('a)) (ptr-val p)))}
  ⊆
  {ptr-val q..+size-of TYPE('b)} = {}
  by (force dest: h-t-valid-neq-disjoint)
  hence heap-list (heap-update-list (ptr-val p) (to-bytes v (heap-list h (size-of TYPE('a))
  (ptr-val p))) h)
  (size-of TYPE('b)) (ptr-val q) = heap-list h (size-of TYPE('b))
  (ptr-val q)
  by – (erule heap-list-update-disjoint-same)
  thus ?thesis by (simp add: h-val-def heap-update-def)
qed

```

lemma *h-val-heap-same-padding*:

```

fixes p::'a::mem-type ptr and q::'b::c-type ptr
assumes val-p: d,g ⊨t p and val-q: d,g' ⊨t q and
  subt: ¬ TYPE('a) <τ TYPE('b) and nf: ¬ field-of-t q p
assumes lbs: length bs = size-of TYPE('a)
shows h-val (heap-update-padding p v bs h) q = h-val h q
proof –
  from val-p val-q subt nf
  have {ptr-val p..+length (to-bytes v bs)} ∩
  {ptr-val q..+size-of TYPE('b)} = {}
  using lbs
  by (force dest: h-t-valid-neq-disjoint)
  hence heap-list (heap-update-list (ptr-val p) (to-bytes v bs) h)
  (size-of TYPE('b)) (ptr-val q) = heap-list h (size-of TYPE('b))
  (ptr-val q)
  by – (erule heap-list-update-disjoint-same)
  thus ?thesis by (simp add: h-val-def heap-update-padding-def)
qed

```

lemma *peer-typI*:
 $\text{typ-uinfo-t } \text{TYPE}('a) \perp_t \text{typ-uinfo-t } \text{TYPE}('b) \implies \text{peer-typ } (a::'a::\text{c-type itself})$
 $(b::'b::\text{c-type itself})$
by (*simp add: peer-typ-def*)

lemma *peer-typD*:
 $\text{peer-typ } \text{TYPE}('a::\text{c-type}) \text{TYPE}('b::\text{c-type}) \implies \neg \text{TYPE}('a) <_{\tau} \text{TYPE}('b)$
by (*clarsimp simp: peer-typ-def tag-disj-def sub-typ-proper-def order-less-imp-le*)

lemma *peer-typ-refl* [*simp*]:
 $\text{peer-typ } t \ t$
by (*simp add: peer-typ-def*)

lemma *peer-typ-simple* [*simp*]:
 $\text{peer-typ } \text{TYPE}('a::\text{simple-mem-type}) \text{TYPE}('b::\text{simple-mem-type})$
apply(*clarsimp simp: peer-typ-def tag-disj-def typ-tag-le-def typ-uinfo-t-def*)
apply(*erule disjE*)
apply(*cases typ-uinfo-t TYPE('b)*)
subgoal for $x1$ *typ-struct xs*
by(*cases typ-struct; simp*)
apply(*cases typ-uinfo-t TYPE('a)*)
subgoal for $x1$ *typ-struct xs*
by(*cases typ-struct; simp*)
done

lemmas *peer-typ-nlt = peer-typD*

lemma *peer-typ-not-field-of*:
 $\llbracket \text{peer-typ } \text{TYPE}('a::\text{c-type}) \text{TYPE}('b::\text{c-type}); \text{ptr-val } p \neq \text{ptr-val } q \rrbracket \implies$
 $\neg \text{field-of-t } (q::'b \text{ ptr}) (p::'a \text{ ptr})$
by (*fastforce simp: peer-typ-def field-of-t-def field-of-def tag-disj-def typ-tag-le-def*
unat-eq-zero
dest: td-set-size-lte)

lemma *h-val-heap-same-peer*:
 $\llbracket d, g \models_t (p::'a::\text{mem-type ptr}); d, g' \models_t (q::'b::\text{c-type ptr});$
 $\text{ptr-val } p \neq \text{ptr-val } q; \text{peer-typ } \text{TYPE}('a) \text{TYPE}('b) \rrbracket \implies$
 $\text{h-val } (\text{heap-update } p \ v \ h) \ q = \text{h-val } h \ q$
apply(*erule (1) h-val-heap-same*)
apply(*erule peer-typ-nlt*)
apply(*erule (1) peer-typ-not-field-of*)
done

lemma *h-val-heap-same-peer-padding*:
 $\llbracket d, g \models_t (p::'a::\text{mem-type ptr}); d, g' \models_t (q::'b::\text{c-type ptr});$
 $\text{ptr-val } p \neq \text{ptr-val } q; \text{peer-typ } \text{TYPE}('a) \text{TYPE}('b); \text{length } bs = \text{size-of}$
 $\text{TYPE}('a) \rrbracket \implies$
 $\text{h-val } (\text{heap-update-padding } p \ v \ bs \ h) \ q = \text{h-val } h \ q$

```

apply(erule (1) h-val-heap-same-padding)
apply(erule peer-typ-nlt)
apply(erule (1) peer-typ-not-field-of)
apply assumption
done

```

lemma *field-offset-footprint-cons-simp* [simp]:
 $field_offset_footprint\ p\ (x\#\!xs) = \{Ptr\ \&(p\to x)\} \cup field_offset_footprint\ p\ xs$
unfolding *field-offset-footprint-def* **by** (cases x) auto

lemma *heap-list-update-list*:
 $\llbracket n + x \leq length\ v; length\ v < addr_card \rrbracket \implies$
 $heap_list\ (heap_update_list\ p\ v\ h)\ n\ (p + of_nat\ x) = take\ n\ (drop\ x\ v)$
apply(induct v arbitrary: n x p h; clarsimp)
subgoal for a list n x p h
apply(cases x; clarsimp)
apply(cases n; clarsimp)
apply (rule conjI)
apply(subgoal-tac heap-update-list (p + of-nat 1) list (h(p := a)) p = a, simp)
apply(subst heap-update-list-same; simp)
apply(metis add.right-neutral drop0 semiring-1-class.of-nat-0)
subgoal for nat
apply(drule meta-spec [where x=n])
apply(drule meta-spec [where x=nat])
apply(drule meta-spec [where x=p+1])
apply (simp add: add.assoc)
done
done
done

lemma *typ-slice-td-set'*:
 $\forall s\ m\ n\ k. (s, m + n) \in td_set\ t\ m \wedge k < size_td\ s \longrightarrow$
 $typ_slice_t\ s\ k \leq typ_slice_t\ t\ (n + k)$
 $\forall s\ m\ n\ k. (s, m + n) \in td_set_struct\ st\ m \wedge k < size_td\ s \longrightarrow$
 $typ_slice_t\ s\ k \leq typ_slice_struct\ st\ (n + k)$
 $\forall s\ m\ n\ k. (s, m + n) \in td_set_list\ ts\ m \wedge k < size_td\ s \longrightarrow$
 $typ_slice_t\ s\ k \leq typ_slice_list\ ts\ (n + k)$
 $\forall s\ m\ n\ k. (s, m + n) \in td_set_tuple\ x\ m \wedge k < size_td\ s \longrightarrow$
 $typ_slice_t\ s\ k \leq typ_slice_tuple\ x\ (n + k)$
apply(induct t and st and ts and x, all <clarsimp simp: split-DTuple-all>)
apply(safe; clarsimp)
apply(drule-tac x=s in spec)
apply(drule-tac x=m in spec)
apply(drule-tac x=0 in spec)
apply fastforce
apply (safe; clarsimp?)
apply(fastforce dest: td-set-list-offset-le)
apply(fastforce dest: td-set-offset-size-m)
apply(rotate-tac)

```

apply(drule-tac  $x=s$  in spec)
apply(rename-tac a list s m n k)
apply(drule-tac  $x=m + \text{size-td } a$  in spec)
apply(drule-tac  $x=n - \text{size-td } a$  in spec)
apply(drule-tac  $x=k$  in spec)
apply(erule impE)
apply(subgoal-tac  $m + \text{size-td } a + (n - \text{size-td } a) = m + n$ , simp)
apply(fastforce dest: td-set-list-offset-le)
apply(fastforce dest: td-set-list-offset-le)
done

```

lemma *typ-slice-td-set*:

```

 $\llbracket (s,n) \in \text{td-set } t \ 0; k < \text{size-td } s \rrbracket \implies$ 
   $\text{typ-slice-t } s \ k \leq \text{typ-slice-t } t \ (n + k)$ 
using typ-slice-td-set'(1) [of t]
apply(drule-tac  $x=s$  in spec)
apply(drule-tac  $x=0$  in spec)
apply clarsimp
done

```

lemma *typ-slice-td-set-list*:

```

 $\llbracket (s,n) \in \text{td-set-list } ts \ 0; k < \text{size-td } s \rrbracket \implies$ 
   $\text{typ-slice-t } s \ k \leq \text{typ-slice-list } ts \ (n + k)$ 
using typ-slice-td-set'(3) [of ts]
apply(drule-tac  $x=s$  in spec)
apply(drule-tac  $x=0$  in spec)
apply clarsimp
done

```

lemma *h-t-valid-sub*:

```

 $\llbracket d,g \models_t (p::'b::\text{mem-type ptr});$ 
   $\text{field-ti } \text{TYPE}('b) \ f = \text{Some } t; \text{export-uinfo } t = (\text{typ-uinfo-t } \text{TYPE}('a)) \rrbracket \implies$ 
   $d,(\lambda x. \text{True}) \models_t ((\text{Ptr } \&(p \rightarrow f))::'a::\text{mem-type ptr})$ 
apply(clarsimp simp: h-t-valid-def field-ti-def valid-footprint-def Let-def split: option.splits)
apply(rule conjI)
apply(simp add: typ-uinfo-t-def export-uinfo-def size-of-def [symmetric, where
 $t=\text{TYPE}('a)]$ )
apply clarsimp
apply(drule-tac  $x=\text{field-offset } \text{TYPE}('b) \ f + y$  in spec)
apply(erule impE)
apply(fastforce simp: field-offset-def field-offset-untyped-def typ-uinfo-t-def size-of-def
  dest: td-set-offset-size td-set-field-lookupD field-lookup-export-uinfo-Some)
apply clarsimp
apply(frule td-set-field-lookupD)
apply(clarsimp simp: field-lvalue-def ac-simps)
apply(drule td-set-export-uinfoD)
apply(drule typ-slice-td-set)
apply(simp add: size-of-def typ-uinfo-t-def)

```

```

apply(drule field-lookup-export-uinfo-Some)
apply(simp add: field-offset-def ac-simps field-offset-untyped-def typ-uinfo-t-def
export-uinfo-def)
apply(erule (1) map-list-map-trans)
done

```

```

lemma size-of-tag:
  size-td (typ-uinfo-t t) = size-of t
by (simp add: size-of-def typ-uinfo-t-def)

```

```

lemma size-of-neq-implies-typ-uinfo-t-neq [simp]:
  size-of TYPE('a::c-type) ≠ size-of TYPE('b::c-type) ⇒ typ-uinfo-t TYPE('a)
≠ typ-uinfo-t TYPE('b)
by (metis size-of-tag)

```

```

lemma guard-mono-self [simp]:
  guard-mono g g
by (fastforce dest: td-set-field-lookupD td-set-size-lte simp: guard-mono-def)

```

```

lemma (in c-type) field-lookup-offset-size:
  field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (t,n) ⇒
  size-td t + n ≤ size-td (typ-uinfo-t TYPE('a))
by (fastforce dest: td-set-field-lookupD td-set-offset-size)

```

```

lemma (in mem-type) sub-h-t-valid':
  [| d,g ⊨t (p::'a ptr);
   field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (typ-uinfo-t TYPE('b),n);
   guard-mono g (g'::'b::mem-type ptr-guard) |] ⇒
  d,g' ⊨t ((Ptr (ptr-val p + of-nat n))::'b::mem-type ptr)
apply(clarsimp simp: h-t-valid-def c-type-class.h-t-valid-def guard-mono-def valid-footprint-def
Let-def size-of-tag)
apply(drule-tac x=n+y in spec, erule impE;
  fastforce simp: ac-simps size-of-def c-type-class.size-of-def typ-uinfo-t-def
c-type-class.typ-uinfo-t-def
  dest: td-set-field-lookupD typ-slice-td-set map-list-map-trans
td-set-offset-size)
done

```

```

lemma (in mem-type) sub-h-t-valid:
  [| d,g ⊨t (p::'a ptr); (typ-uinfo-t TYPE('b),n) ∈ td-set (typ-uinfo-t TYPE('a)) 0
|] ⇒
  d,(λx. True) ⊨t ((Ptr (ptr-val p + of-nat n))::'b::mem-type ptr)
apply(subst (asm) td-set-field-lookup)
apply(simp add: typ-uinfo-t-def export-uinfo-def wf-desc-map)
apply(fastforce intro: sub-h-t-valid' simp: guard-mono-def)
done

```

```

lemma (in mem-type) h-t-valid-mono:
  [| field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (s,n);

```

```

    export-uinfo s = typ-uinfo-t TYPE('b); guard-mono g g' ] ==>
    d,g ⊢t (p::'a ptr) → d,g' ⊢t (Ptr (&(p→f))::'b::mem-type ptr)
  apply (clarsimp simp: field-lvalue-def)
  apply (drule field-lookup-export-uinfo-Some)
  apply (clarsimp simp: field-offset-def c-type-class.field-offset-def typ-uinfo-t-def
c-type-class.typ-uinfo-t-def field-offset-untyped-def)
  apply (rule sub-h-t-valid'; fast?)
  apply (fastforce simp: c-type-class.typ-uinfo-t-def typ-uinfo-t-def)
  done

```

lemma (in mem-type) s-valid-mono:

```

[ field-lookup (typ-info-t TYPE('a)) f 0 = Some (s,n);
  export-uinfo s = typ-uinfo-t TYPE('b); guard-mono g g' ] ==>
  d,g ⊢s (p::'a ptr) → d,g' ⊢s (Ptr (&(p→f))::'b::mem-type ptr)
  unfolding s-valid-def c-type-class.s-valid-def by (rule h-t-valid-mono)

```

lemma take-heap-list-le:

```

k ≤ n ==> take k (heap-list h n x) = heap-list h k x
  apply (induct n arbitrary: k x; clarsimp)
  subgoal for n k x by (cases k; simp)
  done

```

lemma drop-heap-list-le:

```

k ≤ n ==> drop k (heap-list h n x) = heap-list h (n - k) (x + of-nat k)
  apply (induct n arbitrary: k x; clarsimp)
  subgoal for n k x by (cases k; simp add: ac-simps)
  done

```

lemma (in mem-type) h-val-field-from-bytes':

```

[ field-ti TYPE('a) f = Some t;
  export-uinfo t = export-uinfo (typ-info-t TYPE('b::{mem-type})) ] ==>
  h-val h (Ptr &(pa→f)) :: 'b ptr = from-bytes (access-ti0 t (h-val h pa))
  apply (clarsimp simp: field-ti-def h-val-def c-type-class.h-val-def split: option.splits)
  apply (frule field-lookup-export-uinfo-Some)
  apply (frule-tac bs=heap-list h (size-of TYPE('a)) (ptr-val pa) in fi-fa-consistentD)
  apply simp
  apply (clarsimp simp: field-lvalue-def field-offset-def field-offset-untyped-def typ-uinfo-t-def
    field-names-def access-ti0-def)
  apply (subst drop-heap-list-le)
  apply (simp add: size-of-def)
  apply (drule td-set-field-lookupD)
  apply (drule td-set-offset-size)
  apply simp
  apply (subst take-heap-list-le)
  apply (simp add: size-of-def)
  apply (drule td-set-field-lookupD)
  apply (drule td-set-offset-size)
  apply simp
  apply (fold c-type-class.norm-bytes-def)

```

```

apply (subgoal-tac size-td t = size-of TYPE('b))
apply (clarsimp simp: wf-type-class.norm)
apply(clarsimp simp: c-type-class.size-of-def)
apply(subst c-type-class.typ-uinfo-size [symmetric])
apply(unfold c-type-class.typ-uinfo-t-def)
apply(drule sym)
apply simp
done

```

```

lemma (in mem-type) h-val-field-from-bytes:
  [| field-ti TYPE('a) f = Some t;
    export-uinfo t = export-uinfo (typ-info-t TYPE('b::{mem-type})) |] ==>
    h-val (hrs-mem h) (Ptr &(pa->f) :: 'b ptr) = from-bytes (access-ti0 t (h-val
(hrs-mem h) pa))
  by (simp add: h-val-field-from-bytes')

```

```

lemma (in mem-type) lift-tyr-heap-mono:
  [| field-lookup (typ-info-t TYPE('a)) f 0 = Some (t,n);
    lift-tyr-heap g s (p::'a ptr) = Some v;
    export-uinfo t = typ-uinfo-t TYPE('b); guard-mono g g' |] ==>
    lift-tyr-heap g' s (Ptr (&(p->f))::'b::mem-type ptr) = Some (from-bytes
(access-ti0 t v))
apply(clarsimp simp: c-type-class.lift-tyr-heap-if lift-tyr-heap-if split: if-split-asm)
apply(rule conjI; clarsimp?)
apply(clarsimp simp: heap-list-s-def)
apply(frule field-lookup-export-uinfo-Some)
apply(frule-tac bs=heap-list (proj-h s) (size-of TYPE('a)) (ptr-val p) in fi-fa-consistentD;
simp)
apply(simp add: c-type-class.field-lvalue-def
  field-lvalue-def c-type-class.field-offset-def field-offset-def field-offset-untyped-def

  typ-uinfo-t-def c-type-class.typ-uinfo-t-def
  field-names-def access-ti0-def)
apply(subst drop-heap-list-le)
apply(simp add: size-of-def)
apply(drule td-set-field-lookupD)
apply(drule td-set-offset-size)
apply simp
apply(subst take-heap-list-le)
apply(simp add: size-of-def)
apply(drule td-set-field-lookupD)
apply(drule td-set-offset-size)
apply simp
apply(fold c-type-class.norm-bytes-def)
apply(subgoal-tac size-td t = size-of TYPE('b))
apply(simp add: wf-type-class.norm)
apply(clarsimp simp: c-type-class.size-of-def size-of-def)
apply(subst c-type-class.typ-uinfo-size [symmetric])
apply(unfold c-type-class.typ-uinfo-t-def)[1]

```

```

apply(drule sym)
apply simp
apply (fastforce dest: s-valid-mono)
done

```

```

lemma (in mem-type) lift-t-mono:
  [| field-lookup (typ-info-t TYPE('a)) f 0 = Some (t,n);
    lift-t g s (p::'a ptr) = Some v;
    export-uinfo t = typ-uinfo-t TYPE('b); guard-mono g g' |] ==>
    lift-t g' s (Ptr (&(p->f))::'b::mem-type ptr) = Some (from-bytes (access-ti_0
t v))
by (clarsimp simp: lift-t-def c-type-class.lift-t-def lift-ty-heap-mono)

```

```

lemma align-td-wo-align-field-lookupD:
  field-lookup (t::('a,'b) typ-desc) f m = Some (s, n) ==> align-td-wo-align s ≤
align-td-wo-align t
by(simp add: align-td-wo-align-field-lookup)

```

```

lemma (in c-type) align-td-wo-align-uinfo:
  align-td-wo-align (typ-uinfo-t TYPE('a)) = align-td-wo-align (typ-info-t TYPE('a))
by (clarsimp simp: typ-uinfo-t-def)

```

```

lemma (in c-type) align-td-uinfo:
  align-td (typ-uinfo-t TYPE('a)) = align-td (typ-info-t TYPE('a))
by (clarsimp simp: typ-uinfo-t-def export-uinfo-def)

```

```

lemma align-td-export-uinfo: align-td (export-uinfo t) = align-td t
by (clarsimp simp add: export-uinfo-def)

```

```

lemma (in mem-type) align-field-uinfo:
  align-field (typ-uinfo-t TYPE('a)) = align-field (typ-info-t TYPE('a))
apply (standard)
apply (simp add: align-field)
apply (clarsimp simp add: typ-uinfo-t-def align-field-def align-field)
apply (drule field-lookup-export-uinfo-Some-rev)
apply (clarsimp simp add: align-td-export-uinfo)
done

```

```

lemma (in mem-type) ptr-aligned-mono':
  field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (typ-uinfo-t TYPE('b),n) ==>
  ptr-aligned (p::'a ptr) -> ptr-aligned (Ptr (&(p->f))::'b::mem-type ptr)
apply(clarsimp simp: ptr-aligned-def c-type-class.ptr-aligned-def
  align-of-def c-type-class.align-of-def
  field-lvalue-def c-type-class.field-lvalue-def)
apply(subgoal-tac align-field (typ-uinfo-t (TYPE('a))))
apply(subst (asm) align-field-def)
apply(drule-tac x=f in spec)
apply(drule-tac x=typ-uinfo-t TYPE('b) in spec)
apply(drule-tac x=n in spec)

```



```

apply clarsimp
apply(simp add: c-type-class.align-td-uinfo)
apply(clarsimp simp: field-offset-def field-offset-untyped-def)
apply(subst unat-word-ariths)
apply(rule dvd-mod)
apply(rule dvd-add)
subgoal
apply (drule field-lookup-uinfo-Some-rev)
apply clarsimp
by (metis ArraysMemInstance.align-td-export-uinfo c-type-class.typ-uinfo-t-def
align-td-field-lookup-mem-type power-le-dvd)
subgoal
apply(subst unat-of-nat)
apply(subst mod-less)
apply(frule td-set-field-lookupD)
apply(drule td-set-offset-size)
apply(subst len-of-addr-card)
using local.max-size local.size-of-def apply force
apply assumption
done
subgoal
apply(subst len-of-addr-card)
apply(subgoal-tac align-of TYPE('b) dvd addr-card)
apply(subst (asm) c-type-class.align-of-def)
apply simp
apply simp
done
apply(subst align-field-uinfo)
apply(rule align-field)
done

lemma (in mem-type) ptr-aligned-mono:
  guard-mono (ptr-aligned::'a ptr-guard) (ptr-aligned::'b::mem-type ptr-guard)
apply(clarsimp simp: guard-mono-def)
apply(frule ptr-aligned-mono')
apply(fastforce simp: field-lvalue-def field-offset-def field-offset-untyped-def)
done

lemma (in wf-type) wf-desc-tyt-tag [simp]:
  wf-desc (typ-uinfo-t TYPE('a))
by (simp add: typ-uinfo-t-def export-uinfo-def wf-desc-map)

lemma (in c-type) sft1':
  sub-field-update-t (f#fs) p (v::'a) s =
    (let s' = sub-field-update-t fs p (v::'a) s in
      s'(Ptr &(p→f) ↦ from-bytes (access-ti0 (field-tyt TYPE('a) f) v))) |'
      dom (s::'b::c-type typ-heap)
by (rule sft1)

```

lemma *size-map-td*:

size (map-td f g t) = *size* t
size (map-td-struct f g st) = *size* st
size-list (size-dt-tuple size (λ-. 0) (λ-. 0)) (map-td-list f g ts) = *size-list* (size-dt-tuple
size (λ-. 0) (λ-. 0)) ts
size-dt-tuple size (λ-. 0) (λ-. 0) (map-td-tuple f g x) = *size-dt-tuple* size (λ-. 0)
(λ-. 0) x
by (induct t and st and ts and x) auto

lemma *field-names-size'*:

field-names t s ≠ [] → *size* s ≤ *size* (t::('a,'b) typ-info)
field-names-struct st s ≠ [] → *size* s ≤ *size* (st::('a field-desc,'b) typ-struct)
field-names-list ts s ≠ [] → *size* s ≤ *size-list* (size-dt-tuple size (λ-. 0) (λ-. 0))
(ts::(('a,'b) typ-info,field-name,'b) dt-tuple list)
field-names-tuple x s ≠ [] → *size* s ≤ *size-dt-tuple* size (λ-. 0) (λ-. 0) (x::(('a,'b)
typ-info,field-name,'b) dt-tuple)
by (induct t and st and ts and x) (auto simp: size-map-td size-char-def)

lemma *field-names-size*:

f ∈ set (field-names t s) ⇒ *size* s ≤ *size* t
by (cases field-names t s; simp add: field-names-size')

lemma *field-names-size-struct*:

f ∈ set (field-names-struct st s) ⇒ *size* s ≤ *size* st
by (cases field-names-struct st s; simp add: field-names-size')

lemma *field-names-Some3*:

∀ f m s n. field-lookup (t::('a,'b) typ-info) f m = Some (s,n) →
f ∈ set (field-names t (export-uinfo s))
∀ f m s n. field-lookup-struct (st::('a field-desc,'b) typ-struct) f m = Some (s,n)
→
f ∈ set (field-names-struct st (export-uinfo s))
∀ f m s n. field-lookup-list (ts::(('a,'b) typ-info,field-name,'b) dt-tuple list) f m =
Some (s,n) →
f ∈ set (field-names-list ts (export-uinfo s))
∀ f m s n. field-lookup-tuple (x::(('a,'b) typ-info,field-name,'b) dt-tuple) f m =
Some (s,n) →
f ∈ set (field-names-tuple x (export-uinfo s))
apply(induct t and st and ts and x)
apply clarsimp
apply((erule allE)+, erule impE, fast)
apply simp
apply(drule field-names-size-struct)
apply(simp add: size-map-td)
apply (auto split: option.splits)[4]
apply clarsimp
by (metis image-eqI list.collapse)

lemma *field-names-SomeD3*:
 $field_lookup (t::('a,'b) \text{ typ-info}) f m = \text{Some } (s,n) \implies f \in \text{set } (field_names t (export_uinfo s))$
by (*simp add: field-names-Some3*)

lemma *empty-not-in-field-names* [*simp*]:
 $\square \notin \text{set } (field_names_tuple x s)$
by (*cases x, auto*)

lemma *empty-not-in-field-names-list* [*simp*]:
 $\square \notin \text{set } (field_names_list ts s)$
by (*induct ts, auto*)

lemma *empty-not-in-field-names-struct* [*simp*]:
 $\square \notin \text{set } (field_names_struct st s)$
by (*cases st, auto*)

lemma *field-names-Some*:
 $\forall m f. f \in \text{set } (field_names (t::('a,'b) \text{ typ-info}) s) \longrightarrow (field_lookup t f m \neq \text{None})$
 $\forall m f. f \in \text{set } (field_names_struct (st::('a \text{ field-desc}, 'b) \text{ typ-struct}) s) \longrightarrow (field_lookup_struct st f m \neq \text{None})$
 $\forall m f. f \in \text{set } (field_names_list (ts::(('a,'b) \text{ typ-info}, field_name, 'b) \text{ dt-tuple list}) s) \longrightarrow (field_lookup_list ts f m \neq \text{None})$
 $\forall m f. f \in \text{set } (field_names_tuple (x::(('a,'b) \text{ typ-info}, field_name, 'b) \text{ dt-tuple}) s) \longrightarrow (field_lookup_tuple x f m \neq \text{None})$
proof(*induct t and st and ts and x*)
 case (*TypDesc nat typ-struct list*)
 then show *?case by auto*
next
 case (*TypScalar nat1 nat2 a*)
 then show *?case by auto*
next
 case (*TypAggregate list*)
 then show *?case by auto*
next
 case *Nil-typ-desc*
 then show *?case by auto*
next
 case (*Cons-typ-desc dt-tuple list*)
 then show *?case*
 apply *clarsimp*
 apply (*safe; clarsimp?*)
 subgoal for *m f*
 apply(*drule spec [where x=m]*)
 apply(*drule spec [where x=f]*)
 apply *clarsimp*
 done

```

    apply (clarsimp split: option.splits)
  done

next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by force
qed

lemma dt-snd-field-names-list-simp [simp]:
  f ∉ dt-snd ' set xs ⟹ ¬ f#fs ∈ set (field-names-list xs s)
  by (induct xs; clarsimp) (auto simp: split-DTuple-all)

lemma field-names-Some2:
  ∀ m f. wf-desc t ⟶ f ∈ set (field-names (t::('a,'b) typ-info) s) ⟶
    (∃ n k. field-lookup t f m = Some (k,n) ∧ export-uinfo k = s)
  ∀ m f. wf-desc-struct st ⟶ f ∈ set (field-names-struct (st::('a field-desc,'b)
  typ-struct) s) ⟶
    (∃ n k. field-lookup-struct st f m = Some (k,n) ∧ export-uinfo k = s)
  ∀ m f. wf-desc-list ts ⟶ f ∈ set (field-names-list (ts::(('a,'b) typ-info,field-name,'b)
  dt-tuple list) s) ⟶
    (∃ n k. field-lookup-list ts f m = Some (k,n) ∧ export-uinfo k = s)
  ∀ m f. wf-desc-tuple x ⟶ f ∈ set (field-names-tuple (x::(('a,'b) typ-info,field-name,'b)
  dt-tuple) s) ⟶
    (∃ n k. field-lookup-tuple x f m = Some (k,n) ∧ export-uinfo k = s)
proof (induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case by auto
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply (clarsimp simp: export-uinfo-def)
    subgoal for m f
      apply (safe; clarsimp?)
      apply (drule spec [where x=m])
      apply (fastforce simp: split: option.split)
      apply (cases dt-tuple)
      apply (clarsimp)
      apply (cases f; fastforce)
    done
  done

```

```

next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
    apply (clarsimp simp: export-uinfo-def)
    subgoal for m f
      apply(cases f; fastforce)
    done
  done
qed

lemma field-names-SomeD2:
   $\llbracket f \in \text{set } (\text{field-names } (t::('a,'b) \text{ typ-info } s); \text{wf-desc } t) \rrbracket \implies$ 
   $(\exists n k. \text{field-lookup } t f m = \text{Some } (k,n) \wedge \text{export-uinfo } k = s)$ 
  by (simp add: field-names-Some2)

lemma field-names-SomeD:
   $f \in \text{set } (\text{field-names } (t::('a,'b) \text{ typ-info } s) \implies (\text{field-lookup } t f m \neq \text{None})$ 
  by (simp add: field-names-Some)

lemma lift-t-sub-field-update' [rule-format]:
   $\llbracket d, g' \models_t p; \neg (\text{TYPE}('a) <_\tau \text{TYPE}('b)) \rrbracket \implies \text{fs-consistent } \text{fs } \text{TYPE}('a)$ 
 $\text{TYPE}('b) \longrightarrow$ 
   $(\forall K. K = \text{UNIV} - (((\text{field-offset-footprint } p (\text{field-names } (\text{typ-info-t } \text{TYPE}('a))$ 
 $(\text{typ-info-t } \text{TYPE}('b)))))) - (\text{field-offset-footprint } p \text{ fs}) \longrightarrow$ 
   $\text{lift-t } g (\text{heap-update } p (v::'a::\text{mem-type } h, d) \mid 'K =$ 
   $\text{sub-field-update-t } \text{fs } p v ((\text{lift-t } g (h, d))::'b::\text{mem-type } \text{typ-heap}) \mid 'K)$ 
  apply(induct fs)
  apply clarsimp
  apply(rule ext)
  subgoal for x
    apply(clarsimp simp: lift-t-if restrict-map-def)
    apply(erule (2) h-val-heap-same)
    apply(clarsimp simp: field-of-t-def field-offset-footprint-def field-of-def)
    apply(cases x)
  apply(clarsimp simp: field-lvalue-def td-set-field-lookup field-offset-def field-offset-untyped-def)
  subgoal for xa f
    apply(drule-tac x=f in spec)
    apply(fastforce simp: typ-uinfo-t-def dest: field-lookup-export-uinfo-Some-rev
field-names-SomeD3)
  done
  done
  subgoal for a list
    apply clarify
    apply(clarsimp simp: fs-consistent-def)
    apply (rule conjI; clarsimp)
  subgoal for y
    apply(rule ext)
  subgoal for x
    apply(cases x  $\neq$  Ptr &x(p→a))

```

```

apply(clarsimp simp: restrict-map-def)
apply(drule fun-cong [where x=x])
apply clarsimp
apply(fastforce simp: lift-t-if split: if-split-asm)
apply(clarsimp simp: lift-t-if)
apply (rule conjI)
apply clarsimp
apply(clarsimp simp: h-val-def heap-update-def field-lvalue-def)
apply(subst heap-list-update-list; simp?)
apply(simp add: size-of-def)

apply(subst typ-uinfo-size [symmetric])
apply(subst typ-uinfo-size [symmetric])
apply(drule field-names-SomeD2 [where m=0]; clarsimp)
apply(frule td-set-field-lookupD [where m=0])
apply(clarsimp simp: field-offset-def field-offset-untyped-def typ-uinfo-t-def)
apply(drule field-lookup-export-uinfo-Some)
apply simp
apply(drule td-set-export-uinfoD)
apply(simp add: export-uinfo-def)
apply(drule td-set-offset-size)
apply fastforce

apply(drule field-names-SomeD2 [where m=0], clarsimp+)
subgoal for n k
  apply(frule fi-fa-consistentD [where bs=to-bytes v (heap-list h (size-of
TYPE('a)) (ptr-val p))])
  apply simp
  apply(simp add: size-of-def)
  apply(frule field-lookup-export-uinfo-Some)
  apply(simp add: typ-uinfo-t-def)
  apply(subgoal-tac size-td k = size-td (typ-info-t TYPE('b)))
  prefer 2
  apply(simp flip: export-uinfo-size)
  apply simp
  apply(rule sym)
  apply(fold norm-bytes-def)
  apply(subst typ-uinfo-size [symmetric])
  apply(drule sym [where t=Some (k,n)])
  apply(simp only: typ-uinfo-t-def)
  apply(simp)
  apply(clarsimp simp: access-ti0-def field-typ-def field-typ-untyped-def)
  apply(drule sym [where s=Some (k,n)])
  apply simp
  apply(rule norm)
  apply(simp add: size-of-def)
  apply(subst typ-uinfo-size [symmetric])+
  apply(drule td-set-field-lookupD, drule td-set-offset-size)
  apply(fastforce simp: min-def split: if-split-asm)

```

```

    done
  apply(clarsimp split: if-split-asm)
  done
done
apply(rule ccontr, clarsimp)
apply(erule notE, rule ext)
subgoal for x
  apply(cases x ≠ Ptr &(p→a))
  apply(clarsimp simp: restrict-map-def)
  apply(drule fun-cong [where x=x])
  apply clarsimp
  apply(rule ccontr, clarsimp)
  apply(erule impE [where P=x ∈ dom (lift-t g (h, d))])
  apply(clarsimp simp: lift-t-if h-t-valid-def split: if-split-asm)
  apply clarsimp
  apply(clarsimp simp: restrict-map-def)
  apply(rule conjI, clarsimp)
  apply(clarsimp simp: lift-t-if)
done
done
done

```

lemma *lift-t-sub-field-update*:

```

[[ d,g' ⊨t p; ¬ (TYPE('a) <τ TYPE('b)) ]] ⇒
  lift-t g (heap-update p (v::'a::mem-type) h,d) =
    sub-field-update-t (field-names (typ-info-t TYPE('a)) (typ-uinfo-t TYPE('b)))
p v
  ((lift-t g (h,d))::'b::mem-type typ-heap)
  apply(drule lift-t-sub-field-update' [where fs=(field-names (typ-info-t TYPE('a))
(typ-uinfo-t TYPE('b)))], assumption+)
  apply(clarsimp simp: fs-consistent-def)
  apply fast
  apply(simp add: restrict-map-def)
done

```

lemma *lift-t-field-ind*:

```

[[ d,g' ⊨t (p::'b::mem-type ptr); d,ga ⊨t (q::'b ptr);
  field-lookup (typ-info-t TYPE('b::mem-type)) f 0 = Some (a,ba);
  field-lookup (typ-info-t TYPE('b::mem-type)) z 0 = Some (c,da);
  size-td a = size-of TYPE('a); size-td c = size-of TYPE('c);
  ¬ f ≤ z; ¬ z ≤ f ]] ⇒
  lift-t g (heap-update (Ptr (&(p→f))) (v::'a::mem-type) h,d) (Ptr (&(q→z)))
=
  ((lift-t g (h,d) (Ptr (&(q→z))))::'c::c-type option)
  apply(clarsimp simp: lift-t-if h-val-def heap-update-def)
  apply(subgoal-tac (heap-list (heap-update-list &(p→f) (to-bytes v (heap-list h

```

```

(size-of TYPE('a)) &(p→f))) h)
      (size-of TYPE('c)) &(q→z)) =
      (heap-list h (size-of TYPE('c)) &(q→z)))
  apply (drule arg-cong [where f=from-bytes])
  apply simp
  apply (rule heap-list-update-disjoint-same)
  apply simp
  apply (simp add: field-lvalue-def field-offset-def field-offset-untyped-def)
  apply (simp add: typ-uinfo-t-def field-lookup-export-uinfo-Some)
  apply (frule field-lookup-export-uinfo-Some [where s=c])
  apply (cases ptr-val p = ptr-val q)
  apply clarsimp
  apply (subst intvl-disj-offset)
  apply (drule fa-fu-lookup-disj-interD)
    apply fast
    apply (simp add: disj-fn-def)
    apply simp
  apply (subst size-of-def [symmetric, where t=TYPE('b)])
  apply simp
  apply simp
  applyclarsimp
  apply (drule (1) h-t-valid-neq-disjoint, simp)
  apply (rule peer-typ-not-field-of; clarsimp)
  apply (subgoal-tac {ptr-val p + of-nat ba..+size-td a} ⊆ {ptr-val p..+size-of
TYPE('b)})
    apply (subgoal-tac {ptr-val q + of-nat da..+size-td c} ⊆ {ptr-val q..+size-of
TYPE('b)})
      apply fastforce
      apply (rule intvl-sub-offset)
      apply (simp add: size-of-def)
      apply (drule td-set-field-lookupD [where k=(c,da)])
      apply (drule td-set-offset-size)
  apply (smt (verit, ccfv-threshold) add commute add-le-cancel-right le-trans le-unat-voi
nat-le-linear)
  apply (rule intvl-sub-offset)
  apply (simp add: size-of-def)
  apply (drule td-set-field-lookupD)
  apply (drule td-set-offset-size)
  by (smt (verit, ccfv-threshold) add commute add-le-imp-le-right le-trans le-unat-voi
nat-le-linear)

```

lemma (in *c-type*) *wvt1'*:

update-value-t (f#fs) v (w::'b) x = (if x=field-offset TYPE('b) f then
update-ti-t (field-typ TYPE('b) f) (to-bytes-p (v::'a)) (w::'b::c-type) else up-
date-value-t fs v w x)

by *simp*

lemma (in *c-type*) *field-typ-self* [*simp*]:
field-typ *TYPE('a)* [] = *typ-info-t* *TYPE('a)*
by (*simp* add: *field-typ-def* *field-typ-untyped-def*)

lemma (in *mem-type*) *field-of-t-less-size*:
field-of-t (*p::'a ptr*) (*x::'b::c-type ptr*) \implies
 $\text{unat } (\text{ptr-val } p - \text{ptr-val } x) < \text{size-of } \text{TYPE}('b)$
 apply(*simp* add: *field-of-t-def* *field-of-def*)
 apply(*drule* *td-set-offset-size*)
 apply(*subgoal-tac* $0 < \text{size-td } (\text{typ-info-t } \text{TYPE}('a))$)
 apply(*simp* add: *c-type-class.size-of-def*)
 apply(*simp* add: *size-of-def* [*symmetric*, **where** $t = \text{TYPE}('a)$])
 done

lemma *unat-minus*:
 $x \neq 0 \implies \text{unat } (- (x::\text{addr})) = \text{addr-card} - \text{unat } x$
by (*metis* *word-bits-def* *word-bits-size* *len-of-addr-card* *unat-minus*)

lemma (in *mem-type*) *field-of-t-nmem*:
[*field-of-t* *p q*; *ptr-val* *p* \neq *ptr-val* (*q::'b::mem-type ptr*)] \implies
 $\text{ptr-val } q \notin \{ \text{ptr-val } (p::'a \text{ ptr}) .. + \text{size-of } \text{TYPE}('a) \}$
 supply *unsigned-of-nat* [*simp* *del*] *unsigned-of-int* [*simp* *del*]
 apply(*clarsimp* *simp*: *field-of-t-def* *field-of-def* *intvl-def*)
 apply(*drule* *td-set-offset-size*)
 apply(*simp* add: *unat-minus* *size-of-def*)
 apply(*subgoal-tac* $\text{size-td } (\text{typ-info-t } \text{TYPE}('b)) < \text{addr-card}$)
 apply(*simp* *only*: *unat-simps*)
 apply(*smt* add: *commute* *add.left-neutral* *diff-less-mono2* *gr-implies-not0* *le-eq-less-or-eq*
 less-diff-conv2 *less-trans* *max-size* *nat-mod-eq'* *size-of-def* *c-type-class.size-of-def*)
 by (*metis* *c-type-class.size-of-def* *mem-type-class.max-size*)

lemma (in *mem-type*) *field-of-t-init-neq-disjoint*:
field-of-t *p* (*x::'b::mem-type ptr*) \implies
 $\{ \text{ptr-val } (p::'a \text{ ptr}) .. + \text{size-of } \text{TYPE}('a) \} \cap$
 $\{ \text{ptr-val } x .. + \text{unat } (\text{ptr-val } p - \text{ptr-val } x) \} = \{ \}$
 apply(*cases* *ptr-val* $p = \text{ptr-val } x$, *simp*)
 apply(*rule* *ccontr*)
 apply(*drule* *intvl-inter*)
 apply(*auto* *simp*: *field-of-t-nmem* *le-unat-uo* *nat-less-le* *dest!*: *intvlD*)
 done

lemma (in *mem-type*) *field-of-t-final-neq-disjoint*:
field-of-t (*p::'a ptr*) (*x::'b ptr*) $\implies \{ \text{ptr-val } p .. + \text{size-of } \text{TYPE}('a) \} \cap$
 $\{ \text{ptr-val } p + \text{of-nat } (\text{size-of } \text{TYPE}('a)) .. + \text{size-of } \text{TYPE}('b::\text{mem-type}) -$
 $(\text{unat } (\text{ptr-val } p - \text{ptr-val } x) + \text{size-of } \text{TYPE}('a)) \} = \{ \}$
 supply *unsigned-of-nat* [*simp* *del*] *unsigned-of-int* [*simp* *del*]

```

apply(rule ccontr)
apply(drule intvl-inter)
apply(erule disjE)
apply(subgoal-tac ptr-val p  $\notin$  {ptr-val p + of-nat (size-of TYPE('a)) ..+
                                size-of TYPE('b) - (unat (ptr-val p - ptr-val x) +
                                size-of TYPE('a))})

  apply simp
apply(rule intvl-offset-nmem)
apply(rule intvl-self)
apply(subst unat-of-nat)
apply(subst mod-less)
  apply(subst len-of-addr-card)
  apply(rule max-size)
apply(rule sz-nzero)
apply(subst len-of-addr-card)
apply(thin-tac x  $\in$  S for x S)
apply(simp add: field-of-t-def field-of-def)
apply(drule td-set-offset-size)
apply(simp add: size-of-def)
subgoal

proof –
  have size-td (typ-info-t (TYPE('b)::'b itself))  $\leq$  addr-card
    by (metis c-type-class.size-of-def less-imp-le-nat mem-type-class.max-size)
  then show ?thesis
    by (smt (verit, ccfv-threshold) c-type-class.size-of-def diff-diff-left diff-le-mono2
diff-le-self
        le-diff-conv le-trans le-unat-uoI nat-le-linear)
qed

apply(drule intvlD, clarsimp)
apply(subst (asm) word-unat.norm-eq-iff [symmetric])
apply(subst (asm) mod-less)
  apply(subst len-of-addr-card)
  apply(rule max-size)
apply(subst (asm) mod-less)
  apply(subst len-of-addr-card)
  apply(erule less-trans)
  apply(rule max-size)
apply simp
done

lemma (in mem-type) h-val-super-update-bs:
  field-of-t p x  $\implies$  h-val (heap-update p (v::'a) h) (x::'b::mem-type ptr) =
    from-bytes (super-update-bs (to-bytes v (heap-list h (size-of TYPE('a)) (ptr-val
p)) ) (heap-list h (size-of TYPE('b)) (ptr-val x)) (unat (ptr-val p - ptr-val x)))
  apply(simp add: h-val-def c-type-class.h-val-def)
  apply (rule arg-cong [where f = c-type-class.from-bytes::byte list  $\Rightarrow$  'b])
  apply(simp add: heap-update-def c-type-class.heap-update-def super-update-bs-def)

```

```

apply(subst heap-list-split [of unat (ptr-val p - ptr-val x) size-of TYPE('b)])
apply(drule field-of-t-less-size)
apply simp
apply simp
apply(subst heap-list-update-disjoint-same)
apply(drule field-of-t-init-neq-disjoint)
apply (simp add: len)
apply(subst take-heap-list-le)
apply(drule field-of-t-less-size)
apply simp
apply simp
apply(subst heap-list-split [of size-of TYPE('a)
                             size-of TYPE('b) - unat (ptr-val p - ptr-val x)])
apply(frule field-of-t-less-size)
apply(simp add: field-of-t-def field-of-def)
apply(drule td-set-offset-size)
apply(simp add: size-of-def c-type-class.size-of-def)
apply clarsimp
apply (rule conjI)
apply(simp add: heap-list-update-to-bytes)
apply(subst heap-list-update-disjoint-same)
apply(drule field-of-t-final-neq-disjoint)
apply(simp)
apply(subst drop-heap-list-le)
apply(simp add: field-of-t-def field-of-def)
apply(drule td-set-offset-size)
apply(simp add: size-of-def c-type-class.size-of-def)
apply simp
done

```

```

lemma (in mem-type) h-val-super-update-bs-padding:
  field-of-t p x  $\implies$  length bs = size-of TYPE('a)  $\implies$  h-val (heap-update-padding
  p (v::'a) bs h) (x::'b::mem-type ptr) =
    from-bytes (super-update-bs (to-bytes v bs) (heap-list h (size-of TYPE('b))
  (ptr-val x)) (unat (ptr-val p - ptr-val x)))
apply(simp add: h-val-def c-type-class.h-val-def)
apply (rule arg-cong [where f = c-type-class.from-bytes::byte list  $\Rightarrow$  'b])
apply(simp add: heap-update-padding-def c-type-class.heap-update-padding-def su-
  per-update-bs-def)
apply(subst heap-list-split [of unat (ptr-val p - ptr-val x) size-of TYPE('b)])
apply(drule field-of-t-less-size)
apply simp
apply simp

```

```

apply(subst heap-list-update-disjoint-same)
apply(drule field-of-t-init-neq-disjoint)
apply (simp add: len)
apply(subst take-heap-list-le)
apply(drule field-of-t-less-size)

```

```

apply simp
apply simp
apply(subst heap-list-split [of size-of TYPE('a)
                               size-of TYPE('b) - unat (ptr-val p - ptr-val x)])
apply(frule field-of-t-less-size)
apply(simp add: field-of-t-def field-of-def)
apply(drule td-set-offset-size)
apply(simp add: size-of-def c-type-class.size-of-def)
apply clarsimp
apply (rule conjI)
apply(simp add: heap-list-update-to-bytes-padding)
apply(subst heap-list-update-disjoint-same)
apply(drule field-of-t-final-neq-disjoint)
apply(simp)
apply(subst drop-heap-list-le)
apply(simp add: field-of-t-def field-of-def)
apply(drule td-set-offset-size)
apply(simp add: size-of-def c-type-class.size-of-def)
apply simp
done

```

```

lemma (in c-type) update-field-update':
  n ∈ (λf. field-offset TYPE('b) f) ' set fs ⇒
    (∃f. update-value-t fs (v::'a) (v'::'b::c-type) n =
      field-update (field-desc (field-typ TYPE('b) f)) (to-bytes-p v) v' ∧ f ∈ set
fs ∧ n = field-offset TYPE('b) f)
  by (induct fs) auto

```

```

lemma (in c-type) update-field-update:
  field-of-t (p::'a ptr) (x::'b ptr) ⇒
    ∃f. update-value-t (field-names (typ-info-t TYPE('b)) (typ-uinfo-t TYPE('a)))
(v::'a)
  (v'::'b::mem-type) (unat (ptr-val p - ptr-val x)) =
    field-update (field-desc (field-typ TYPE('b) f)) (to-bytes-p v) v' ∧
  f ∈ set (field-names (typ-info-t TYPE('b)) (typ-uinfo-t TYPE('a))) ∧
  unat (ptr-val p - ptr-val x) = field-offset TYPE('b) f
apply(rule update-field-update')
apply(clarsimp simp: image-def field-offset-def c-type-class.field-offset-def field-of-t-def
field-of-def field-offset-untyped-def)
apply(simp add: td-set-field-lookup)
apply clarsimp
subgoal for f
  apply(simp add: typ-uinfo-t-def c-type-class.typ-uinfo-t-def)
  apply(rule bexI [where x=f], simp)
  apply(drule field-lookup-export-uinfo-Some-rev)
  apply clarsimp
  apply(drule field-names-SomeD3)
  apply clarsimp
done

```

done

lemma *length-fa-ti*:

$\llbracket wf\text{-}fd\ s; length\ bs = size\text{-}td\ s \rrbracket \implies length\ (access\text{-}ti\ s\ w\ bs) = size\text{-}td\ s$
apply(*drule wf-fd-consD*)
apply(*clarsimp simp: fd-cons-def fd-cons-length-def fd-cons-desc-def*)
done

lemma *fa-fu-v*:

$\llbracket wf\text{-}fd\ s; length\ bs = size\text{-}td\ s; length\ bs' = size\text{-}td\ s \rrbracket \implies$
 $access\text{-}ti\ s\ (update\text{-}ti\text{-}t\ s\ bs\ v)\ bs' = access\text{-}ti\ s\ (update\text{-}ti\text{-}t\ s\ bs\ w)\ bs'$
apply(*drule wf-fd-consD*)
apply(*clarsimp simp: fd-cons-def fd-cons-access-update-def fd-cons-desc-def*)
done

lemma *fu-fa*:

$\llbracket wf\text{-}fd\ s; length\ bs = size\text{-}td\ s \rrbracket \implies$
 $update\text{-}ti\text{-}t\ s\ (access\text{-}ti\ s\ v\ bs)\ v = v$
apply(*drule wf-fd-consD*)
apply(*clarsimp simp: fd-cons-def fd-cons-update-access-def fd-cons-desc-def*)
done

lemma *fu-fu*:

$\llbracket wf\text{-}fd\ s; length\ bs = length\ bs' \rrbracket \implies$
 $update\text{-}ti\text{-}t\ s\ bs\ (update\text{-}ti\text{-}t\ s\ bs'\ v) = update\text{-}ti\text{-}t\ s\ bs\ v$
apply(*drule wf-fd-consD*)
apply(*clarsimp simp: fd-cons-def fd-cons-double-update-def fd-cons-desc-def*)
done

lemma *fu-fa'-rpbs*:

$\llbracket export\text{-}uinfo\ s = export\text{-}uinfo\ t; length\ bs = size\text{-}td\ s; wf\text{-}fd\ s; wf\text{-}fd\ t \rrbracket \implies$
 $update\text{-}ti\text{-}t\ s\ (access\text{-}ti\ t\ v\ bs)\ w = update\text{-}ti\text{-}t\ s\ (access\text{-}ti_0\ t\ v)\ w$
apply(*clarsimp simp: access-ti₀-def*)
apply(*subgoal-tac size-td s = size-td t*)
apply(*frule-tac bs=access-ti t v bs in wf-fd-norm-tuD[where t=t]*)
apply(*subst length-fa-ti; simp*)
apply(*frule-tac bs=access-ti t v bs in wf-fd-norm-tuD*)
apply(*subst length-fa-ti; simp*)
apply(*clarsimp simp: access-ti₀-def*)
apply(*thin-tac norm-tu X Y = Z for X Y Z*)
apply(*subst (asm) fa-fu-v [where w=v]; simp?*)
apply(*simp add: length-fa-ti*)
apply(*subst (asm) fa-fu-v [where w=w]; simp?*)
apply(*simp add: length-fa-ti; simp*)
apply(*subst (asm) fu-fa; (solves <simp>)?*)
apply(*drule-tac f=update-ti-t s in arg-cong*)
apply(*drule-tac x=update-ti-t s (access-ti t v bs) w in fun-cong*)
apply(*subst (asm) fu-fa; (solves <simp>)?*)

```

apply(fastforce simp: fu-fu length-fa-ti)
apply(drule-tac f=size-td in arg-cong)
apply(simp add: export-uinfo-def)
done

```

lemma *lift-t-super-field-update*:

```

[[ d,g' ⊨t p; TYPE('a) ≤τ TYPE('b) ]] ⇒
  lift-t g (heap-update p (v::'a::mem-type) h,d) =
    super-field-update-t p v ((lift-t g (h,d))::'b::mem-type typ-heap)
apply(rule ext)
apply(clarsimp simp: super-field-update-t-def split: option.splits)
apply(rule conjI; clarsimp)
apply(simp add: lift-t-if split: if-split-asm)
apply(rule conjI; clarsimp)
apply(simp add: lift-t-if h-val-super-update-bs split: if-split-asm)
apply(drule sym)
apply(rename-tac x1 x2)
apply(drule-tac v=v and v'=x2 in update-field-update)
apply(clarsimp simp: h-val-def)
apply(frule-tac m=0 in field-names-SomeD)
apply clarsimp
apply(subgoal-tac export-uinfo a = typ-uinfo-t TYPE('a))
  prefer 2
apply(drule-tac m=0 in field-names-SomeD2; clarsimp)
apply(simp add: from-bytes-def)
apply(frule-tac bs=heap-list h (size-of TYPE('b)) (ptr-val x1) and
  v=to-bytes v (heap-list h (size-of TYPE('a)) (ptr-val p)) and
  w=undefined in fi-fu-consistentD)
  apply simp
  apply(simp add: size-of-def)
apply(clarsimp simp: size-of-def sub-typ-def typ-tag-le-def simp flip: export-uinfo-size)
apply(simp add: field-offset-def field-offset-untyped-def typ-uinfo-t-def)
apply(frule field-lookup-export-uinfo-Some)
apply(simp add: to-bytes-def to-bytes-p-def)
apply(subst fu-fa'-rpbs; simp?)
  apply(simp add: size-of-def flip: export-uinfo-size)
  apply(fastforce intro: wf-fd-field-lookupD)
apply(unfold access-ti0-def)[1]
apply(simp flip: export-uinfo-size)
apply(simp add: size-of-def access-ti0-def field-typ-def field-typ-untyped-def)
apply(frule lift-t-h-t-valid)
apply(simp add: lift-t-if)
apply(cases typ-uinfo-t TYPE('a) = typ-uinfo-t TYPE('b))
apply(subst h-val-heap-same-peer; assumption?)
  apply(clarsimp simp: field-of-t-def)
apply(simp add: peer-typ-def)
apply(drule (1) h-t-valid-neq-disjoint)
apply(simp add: sub-typ-proper-def)
apply(rule order-less-not-sym)

```

```

  apply(simp add: sub-typ-def)
  apply assumption
  apply(simp add: h-val-def heap-update-def heap-list-update-disjoint-same)
done

```

lemma *lift-t-super-field-update-padding*:

```

[[ d, g' ⊨t p; TYPE('a) ≤τ TYPE('b); length bs = size-of TYPE('a) ]] ⇒
  lift-t g (heap-update-padding p (v::'a::mem-type) bs h, d) =
    super-field-update-t p v ((lift-t g (h, d))::'b::mem-type typ-heap)
  apply(rule ext)
  apply(clarsimp simp: super-field-update-t-def split: option.splits)
  apply(rule conjI; clarsimp)
  apply(simp add: lift-t-if split: if-split-asm)
  apply(rule conjI; clarsimp)
  apply(simp add: lift-t-if h-val-super-update-bs-padding split: if-split-asm)
  apply(drule sym)
  apply(rename-tac x1 x2)
  apply(drule-tac v=v and v'=x2 in update-field-update)
  apply(clarsimp simp: h-val-def)
  apply(frute-tac m=0 in field-names-SomeD)
  apply clarsimp
  apply(subgoal-tac export-uinfo a = typ-uinfo-t TYPE('a))
  prefer 2
  apply(drule-tac m=0 in field-names-SomeD2; clarsimp)
  apply(simp add: from-bytes-def)
  apply(frute-tac bs=heap-list h (size-of TYPE('b)) (ptr-val x1) and
    v=to-bytes v bs and
    w=undefined in fi-fu-consistentD)
  apply simp
  apply(simp add: size-of-def)
  apply(clarsimp simp: size-of-def sub-typ-def typ-tag-le-def simp flip: export-uinfo-size)
  apply(simp add: field-offset-def field-offset-untyped-def typ-uinfo-t-def)
  apply(frute field-lookup-export-uinfo-Some)
  apply(simp add: to-bytes-def to-bytes-p-def)
  apply(subst fu-fa'-rpbs; simp?)
  apply(simp add: size-of-def flip: export-uinfo-size)
  apply(fastforce intro: wf-fd-field-lookupD)
  apply(unfold access-ti0-def)[1]
  apply(simp flip: export-uinfo-size)
  apply(simp add: size-of-def access-ti0-def field-typ-def field-typ-untyped-def)
  apply(frute lift-t-h-t-valid)
  apply(simp add: lift-t-if)
  apply(cases typ-uinfo-t TYPE('a) = typ-uinfo-t TYPE('b))
  apply(subst h-val-heap-same-peer-padding; assumption?)
  apply(clarsimp simp: field-of-t-def)
  apply(simp add: peer-typ-def)
  apply(drule (1) h-t-valid-neq-disjoint)
  apply(simp add: sub-typ-proper-def)
  apply(rule order-less-not-sym)

```

```

  apply(simp add: sub-typ-def)
  apply assumption
  apply(simp add: h-val-def heap-update-padding-def heap-list-update-disjoint-same)
  done

```

lemma *field-names-same*:

```

  k = export-uinfo ti  $\implies$  field-names ti k = []
  by (cases ti, clarsimp)

```

lemma *lift-t-heap-update*:

```

  d, g  $\models_t$  p  $\implies$  lift-t g (heap-update p v h, d) =
    ((lift-t g (h, d)) (p  $\mapsto$  (v::'a::mem-type)))
  apply(subst lift-t-sub-field-update)
  apply fast
  apply(simp add: sub-typ-proper-def)
  apply(simp add: typ-uinfo-t-def Let-def)
  apply(subgoal-tac access-ti_0 (typ-info-t TYPE('a)) = to-bytes-p)
  apply(simp add: field-names-same)
  apply(clarsimp simp: Let-def to-bytes-p-def)
  apply(rule conjI, clarsimp)
  apply(rule ext, clarsimp simp: restrict-self-UNIV)
  apply clarsimp
  apply(clarsimp simp: h-t-valid-def lift-t-if)
  apply(rule ext)
  apply(simp add: to-bytes-p-def to-bytes-def size-of-def access-ti_0-def)
  done

```

lemma *field-names-disj*:

```

  typ-uinfo-t TYPE('a::c-type)  $\perp_t$  typ-uinfo-t TYPE('b::mem-type)  $\implies$ 
  field-names (typ-info-t TYPE('b)) (typ-uinfo-t TYPE('a)) = []
  apply(rule ccontr)
  apply(subgoal-tac ( $\exists k. k \in \text{set (field-names (typ-info-t TYPE('b)) (typ-uinfo-t TYPE('a))))$ ))
  apply clarsimp
  apply(frule field-names-SomeD2 [where m=0], clarsimp+)
  apply(drule field-lookup-export-uinfo-Some)
  apply(drule td-set-field-lookupD)
  apply(fastforce simp: tag-disj-def typ-tag-le-def typ-uinfo-t-def)
  apply(cases field-names (typ-info-t TYPE('b)) (typ-uinfo-t TYPE('a)); simp)
  apply fast
  done

```

lemma *lift-t-heap-update-same*:

```

  [ $d, g' \models_t (p::'b::mem-type \text{ ptr}); \text{typ-uinfo-t TYPE('a)} \perp_t \text{typ-uinfo-t TYPE('b)}$ ]
 $\implies$ 
  lift-t g (heap-update p v h, d) = (lift-t g (h, d)::'a::mem-type typ-heap)
  apply(subst lift-t-sub-field-update, simp)
  apply(fastforce dest: order-less-imp-le simp: sub-typ-proper-def tag-disj-def)
  apply(simp add: field-names-disj)

```


done

lemma *lift-heap-update*:

$\llbracket d, g \models_t (p :: 'a \text{ ptr}); d, g' \models_t q \rrbracket \implies \text{lift } (\text{heap-update } p \ v \ h) \ q =$
 $((\text{lift } h)(p := (v :: 'a :: \text{mem-type}))) \ q$
by (*simp add: lift-def h-val-heap-update h-val-heap-same-peer*)

lemma (*in mem-type*) *lift-heap-update-p* [*simp*]:

lift (*heap-update* *p v h*) *p* = (*v :: 'a*)
by (*simp add: lift-def heap-update-def h-val-def heap-list-update-to-bytes*)

lemma *lift-heap-update-same*:

$\llbracket \text{ptr-val } p \neq \text{ptr-val } q; d, g \models_t (p :: 'a :: \text{mem-type } \text{ptr});$
 $d, g' \models_t (q :: 'b :: \text{mem-type } \text{ptr}); \text{peer-typ } \text{TYPE}('a) \ \text{TYPE}('b) \rrbracket \implies$
lift (*heap-update* *p v h*) *q* = *lift h q*
by (*simp add: lift-def h-val-heap-same-peer*)

lemma *lift-heap-update-same-type*:

fixes *p :: 'a :: mem-type ptr* **and** *q :: 'b :: mem-type ptr*
assumes *valid*: *d, g* \models_t *p* *d, g'* \models_t *q*
assumes *type*: *typ-uinfo-t* *TYPE('a)* \perp_t *typ-uinfo-t* *TYPE('b)*
shows *lift* (*heap-update* *p v h*) *q* = *lift h q*

proof –

from *valid type* **have** *ptr-val* *p* \neq *ptr-val* *q*
apply(*clarsimp simp: h-t-valid-def*)
apply(*drule* (1) *valid-footprint-sub*)
apply(*clarsimp simp: tag-disj-def order-less-imp-le*)
apply(*fastforce intro: intvl-self*
simp: field-of-def tag-disj-def typ-tag-le-def
simp flip: size-of-def [where t=TYPE('b)])

done

thus *?thesis* **using** *valid type*

by (*fastforce elim: lift-heap-update-same peer-typl*)

qed

lemma *lift-heap-update-same-ptr-coerce*:

$\llbracket \text{ptr-val } q \neq \text{ptr-val } p;$
 $d, g \models_t (\text{ptr-coerce } (q :: 'b :: \text{mem-type } \text{ptr}) :: 'c :: \text{mem-type } \text{ptr});$
 $d, g' \models_t (p :: 'a :: \text{mem-type } \text{ptr});$
 $\text{size-of } \text{TYPE}('b) = \text{size-of } \text{TYPE}('c); \text{peer-typ } \text{TYPE}('a) \ \text{TYPE}('c) \rrbracket \implies$
lift (*heap-update* *q v h*) *p* = *lift h p*
apply(*clarsimp simp: lift-def h-val-def heap-update-def size-of-def*)
apply(*subst heap-list-update-disjoint-same*)
apply(*drule* (1) *h-t-valid-neq-disjoint*)
apply(*erule peer-typl*)
apply(*erule peer-typl-not-field-of, simp*)
apply(*simp add: size-of-def*)
apply *simp*
done

lemma *heap-footprintI*:

$\llbracket \text{valid-footprint } d \ y \ t; x \in \{y..+size-td \ t\} \rrbracket \implies x \in \text{heap-footprint } d \ t$
by (*force simp: heap-footprint-def*)

lemma *valid-heap-footprint*:

$\text{valid-footprint } d \ x \ t \implies x \in \text{heap-footprint } d \ t$
by (*force simp: heap-footprint-def*)

lemma *heap-valid-footprint*:

$x \in \text{heap-footprint } d \ t \implies \exists y. \text{valid-footprint } d \ y \ t \wedge x \in \{y\} \cup \{y..+size-td \ t\}$
by (*simp add: heap-footprint-def*)

lemma *heap-footprint-Some*:

$x \in \text{heap-footprint } d \ t \implies d \ x \neq (\text{False}, \text{Map.empty})$
by (*auto simp: heap-footprint-def intvl-def valid-footprint-def Let-def*)

lemma *dom-tll-empty [simp]*:

$\text{dom-tll } p \ [] = \{\}$
by (*clarsimp simp: dom-tll-def*)

lemma *dom-s-upd [simp]*:

$\text{dom-s } (\lambda b. \text{if } b = p \text{ then } (\text{True}, a) \text{ else } d \ b) =$
 $(\text{dom-s } d - \{(p, y) \mid y. \text{True}\}) \cup \{(p, \text{SIndexVal})\} \cup \{(p, \text{SIndexTyp } n) \mid n. a$
 $n \neq \text{None}\}$
unfolding *dom-s-def* **by** (*cases d p*) *auto*

lemma *dom-tll-cons [simp]*:

$\text{dom-tll } p \ (x\#\text{xs}) = \text{dom-tll } (p + 1) \ \text{xs} \cup \{(p, \text{SIndexVal})\} \cup \{(p, \text{SIndexTyp } n) \mid$
 $n. x \ n \neq \text{None}\}$
unfolding *dom-tll-def*
apply (*rule equalityI;clarsimp*)
apply (*rule conjI;clarsimp*)

subgoal using *less-Suc-eq-0-disj* **by** *auto[1]*

subgoal using *less-Suc-eq-0-disj* **by** *force*

subgoal using *less-Suc-eq-0-disj Suc-mono of-nat-Suc*

by (*smt (verit) Groups.add-ac(2) Groups.add-ac(3) UnCI add.right-neutral*
mem-Collect-eq nth-Cons' nth-Cons-Suc semiring-1-class.of-nat-simps(1)

subsetI)

done

lemma *one-plus-x-zero*:

```

(1::addr) + of-nat x = 0  $\implies$  x  $\geq$  addr-card - 1
apply(simp add: addr-card)
apply(subst (asm) of-nat-1 [symmetric])
apply(subst (asm) Abs-fnat-homs)
apply(subst (asm) Word.of-nat-0)
apply(erule exE)
subgoal for q
  apply(cases q; simp)
  done
done

lemma htd-update-list-dom [rule-format, simp]:
  length xs < addr-card  $\longrightarrow$  ( $\forall p d$ . dom-s (htd-update-list p xs d) = dom-s d  $\cup$ 
  dom-tll p xs)
  by (induct xs; clarsimp) (auto simp: dom-s-def)

lemma htd-update-list-same:
  shows  $\bigwedge h p k$ .  $\llbracket 0 < k; k \leq \text{addr-card} - \text{length } v \rrbracket \implies$ 
    (htd-update-list (p + of-nat k) v) h p = h p
proof (induct v)
  case Nil show ?case by simp
next
  case (Cons x xs)
  have htd-update-list (p + of-nat k) (x # xs) h p =
    htd-update-list (p + of-nat (k + 1)) xs (h(p + of-nat k := (True,snd (h (p +
  of-nat k)) ++ x))) p
  by (simp add: ac-simps)
  also have ... = (h(p + of-nat k := (True,snd (h (p + of-nat k)) ++ x))) p
  proof -
    from Cons have k + 1  $\leq$  addr-card - length xs by simp
    with Cons show ?thesis by (simp only:)
  qed
  also have ... = h p
  proof -
    from Cons have of-nat k  $\neq$  (0::addr)
    by - (erule of-nat-neq-0, simp add: addr-card)
    thus ?thesis by clarsimp
  qed
finally show ?case .
qed

lemma htd-update-list-index [rule-format]:
   $\forall p d$ . length xs < addr-card  $\longrightarrow$  x  $\in$  {p..+length xs}  $\longrightarrow$ 
    htd-update-list p xs d x = (True, snd (d x) ++ (xs ! (unat (x - p))))
  apply(induct xs; clarsimp)
  subgoal for x1 xs p d
  apply(cases p=x)
  apply simp

```

```

apply(subst of-nat-1 [symmetric])
apply(subst htd-update-list-same; simp)
apply(drule spec [where x=p+1])
apply(erule impE)
apply(drule intvl-neq-start; simp)
apply(subgoal-tac unat (x - p) = unat (x - (p + 1)) + 1)
apply simp
apply (clarsimp simp: unatSuc[symmetric] field-simps)
done
done

```

```

lemma (in c-type) typ-slices-length [simp]:
  length (typ-slices TYPE('a)) = size-of TYPE('a)
by (simp add: typ-slices-def)

```

```

lemma (in c-type) typ-slices-index [simp]:
  n < size-of TYPE('a)  $\implies$  typ-slices TYPE('a) ! n =
    list-map (typ-slice-t (typ-uinfo-t TYPE('a)) n)
by (simp add: typ-slices-def)

```

```

lemma (in c-type) empty-not-in-typ-slices [simp]:
  Map.empty  $\notin$  set (typ-slices TYPE('a))
by (auto simp: typ-slices-def dest: sym)

```

```

lemma (in c-type) dom-tll-s-footprint [simp]:
  dom-tll (ptr-val p) (typ-slices TYPE('a)) = s-footprint (p::'a ptr)
by (fastforce simp: list-map-eq typ-slices-def s-footprint-def s-footprint-untyped-def
  dom-tll-def size-of-def
  split: if-split-asm)

```

```

lemma (in mem-type) ptr-retyp-dom [simp]:
  dom-s (ptr-retyp (p::'a ptr) d) = dom-s d  $\cup$  s-footprint p
by (simp add: ptr-retyp-def)

```

```

lemma dom-s-empty-htd [simp]:
  dom-s empty-htd = {}
by (clarsimp simp: empty-htd-def dom-s-def)

```

```

lemma dom-s-nempty:
  d x  $\neq$  (False, Map.empty)  $\implies$   $\exists k. (x,k) \in$  dom-s d
apply(clarsimp simp: dom-s-def)
apply(cases d x, clarsimp)
using None-not-eq by fastforce

```

```

lemma (in mem-type) ptr-retyp-None:
  x  $\notin$  {ptr-val p..+size-of TYPE('a)}  $\implies$ 
  ptr-retyp (p::'a ptr) empty-htd x = (False, Map.empty)
using ptr-retyp-dom [of p empty-htd]
by (fastforce dest: dom-s-nempty s-footprintD)

```

lemma (in mem-type) ptr-retyp-footprint:
 $x \in \{\text{ptr-val } p..+\text{size-of } \text{TYPE}('a)\} \implies$
 $\text{ptr-retyp } (p::'a \text{ ptr}) \text{ } d \text{ } x =$
 $(\text{True}, \text{snd } (d \text{ } x) ++ \text{list-map } (\text{typ-slice-t } (\text{typ-uinfo-t } \text{TYPE}('a))) (\text{unat } (x -$
 $\text{ptr-val } p))))$
apply(clarsimp simp: ptr-retyp-def)
apply(subst htd-update-list-index; simp)
apply(subst typ-slices-index; simp?)
apply(drule intvlD, clarsimp)
by (metis le-unat-uoi linorder-not-less nat-less-le)

lemma (in mem-type) ptr-retyp-Some:
 $\text{ptr-retyp } (p::'a \text{ ptr}) \text{ } d \text{ } (\text{ptr-val } p) =$
 $(\text{True}, \text{snd } (d \text{ } (\text{ptr-val } p))) ++ \text{list-map } (\text{typ-slice-t } (\text{typ-uinfo-t } \text{TYPE}('a))) 0))$
by (simp add: ptr-retyp-footprint)

lemma (in mem-type) ptr-retyp-Some2:
 $x \in \{\text{ptr-val } (p::'a \text{ ptr})..+\text{size-of } \text{TYPE}('a)\} \implies \text{ptr-retyp } p \text{ } d \text{ } x \neq (\text{False}, \text{Map.empty})$
by (auto simp: ptr-retyp-Some ptr-retyp-footprint dest: intvl-neq-start)

lemma snd-empty-htd [simp]:
 $\text{snd } (\text{empty-htd } x) = \text{Map.empty}$
by (auto simp: empty-htd-def)

lemma (in mem-type) ptr-retyp-d-empty:
 $x \in \{\text{ptr-val } (p::'a \text{ ptr})..+\text{size-of } \text{TYPE}('a)\} \implies$
 $(\forall d. \text{fst } (\text{ptr-retyp } p \text{ } d \text{ } x)) \wedge$
 $\text{snd } (\text{ptr-retyp } p \text{ } d \text{ } x) = \text{snd } (d \text{ } x) ++ \text{snd } (\text{ptr-retyp } p \text{ } \text{empty-htd } x)$
by (auto simp: ptr-retyp-Some ptr-retyp-footprint dest: intvl-neq-start)

lemma unat-minus-abs:
 $x \neq y \implies \text{unat } ((x::\text{addr}) - y) = \text{addr-card} - \text{unat } (y - x)$
by (clarsimp simp: unat-sub-if-size)
(metis (no-types) Nat.diff-cancel Nat.diff-diff-right add commute len-of-addr-card
nat-le-linear not-le trans-le-add1 unat-lt2p wsst-TYs(3))

lemma (in mem-type) ptr-retyp-d:
 $x \notin \{\text{ptr-val } (p::'a \text{ ptr})..+\text{size-of } \text{TYPE}('a)\} \implies$
 $\text{ptr-retyp } p \text{ } d \text{ } x = d \text{ } x$
apply(simp add: ptr-retyp-def)
apply(subgoal-tac $\exists k. \text{ptr-val } p = x + \text{of-nat } k \wedge 0 < k \wedge k \leq \text{addr-card} -$
 $\text{size-of } \text{TYPE}('a))$
apply clarsimp
apply(subst htd-update-list-same; simp)
apply(rule exI [where $x = \text{unat } (\text{ptr-val } p - x)$])
apply clarsimp
apply(cases ptr-val p = x)
apply simp

```

apply(erule notE)
apply(rule intvl-self)
apply simp
apply(rule conjI)
apply(subst unat-gt-0)
apply clarsimp
apply(rule ccontr)
apply(erule notE)
apply(clarsimp simp: intvl-def)
apply(rule exI [where x=unat (x - ptr-val p)])
apply simp
apply(subst (asm) unat-minus-abs; simp)
done

```

lemma (in mem-type) ptr-retyp-valid-footprint-disjoint:

```

[[ valid-footprint d p s; {p..+size-td s} ∩ {ptr-val q..+size-of TYPE('a)} = {} ]]
  ⇒ valid-footprint (ptr-retyp (q::'a ptr) d) p s
apply(clarsimp simp: valid-footprint-def Let-def)
apply((subst ptr-retyp-d; clarsimp), fastforce intro: intvlI)+
done

```

lemma ptr-retyp-disjoint:

```

[[ d,g ⊨t (q::'b::mem-type ptr); {ptr-val p..+size-of TYPE('a)} ∩
  {ptr-val q..+size-of TYPE('b)} = {} ]] ⇒
  ptr-retyp (p::'a::mem-type ptr) d,g ⊨t q
apply(clarsimp simp: h-t-valid-def)
apply(erule ptr-retyp-valid-footprint-disjoint)
apply(fastforce simp: size-of-def)
done

```

lemma ptr-retyp-d-fst:

```

(x, SIndexVal) ∉ s-footprint (p::'a::mem-type ptr) ⇒ fst (ptr-retyp p d x) = fst
(d x)
apply(simp add: ptr-retyp-def)
apply(subgoal-tac ∃k. ptr-val p = x + of-nat k ∧ 0 < k ∧ k ≤ addr-card -
size-of TYPE('a))
apply(fastforce simp: htd-update-list-same)
apply(rule exI [where x=unat (ptr-val p - x)])
apply clarsimp
apply(cases ptr-val p = x)
apply(fastforce dest: sym)
apply(rule conjI)
apply(subst unat-gt-0, clarsimp)
apply(rule ccontr)
apply(erule notE)
apply(subst (asm) unat-minus-abs, simp)
apply(clarsimp simp: s-footprint-def s-footprint-untyped-def)
apply(rule exI [where x=unat (x - ptr-val p)])
apply(simp add: size-of-def)

```

done

lemma (in mem-type) ptr-retyp-d-eq-fst:

fst (ptr-retyp p d x) =
(if x ∈ {ptr-val (p::'a ptr)..+size-of TYPE('a)} then True else *fst* (d x))
by (auto dest!: ptr-retyp-d-empty[where d=d] ptr-retyp-d[where d=d])

lemma (in mem-type) ptr-retyp-d-eq-snd:

snd (ptr-retyp p d x) =
(if x ∈ {ptr-val (p::'a ptr)..+size-of TYPE('a)}
then *snd* (d x) ++ *snd* (ptr-retyp p empty-htd x)
else *snd* (d x))
by (auto dest!: ptr-retyp-d-empty[where d=d] ptr-retyp-d[where d=d])

lemma lift-state-ptr-retyp-d-empty:

x ∈ {ptr-val (p::'a::mem-type ptr)..+size-of TYPE('a)} ⇒
lift-state (h, ptr-retyp p d) (x, k) =
(*lift-state* (h, d) ++ *lift-state* (h, ptr-retyp p empty-htd)) (x, k)
apply (clarsimp simp: lift-state-def map-add-def split: s-heap-index.splits)
apply safe
apply (subst ptr-retyp-d-empty; simp)
apply (subst ptr-retyp-d-empty; simp)
apply (subst (asm) ptr-retyp-d-empty; simp)
apply (subst ptr-retyp-d-empty, simp)
apply (auto split: option.splits)
done

lemma lift-state-ptr-retyp-d:

x ∉ {ptr-val (p::'a::mem-type ptr)..+size-of TYPE('a)} ⇒
lift-state (h, ptr-retyp p d) (x, k) = *lift-state* (h, d) (x, k)
by (clarsimp simp: lift-state-def ptr-retyp-d split: s-heap-index.splits)

lemma (in mem-type) ptr-retyp-valid-footprint:

valid-footprint (ptr-retyp p d) (ptr-val (p::'a ptr)) (typ-uinfo-t TYPE('a))
apply (clarsimp simp: valid-footprint-def Let-def)
apply (subst size-of-def [symmetric, where t=TYPE('a)])
apply clarsimp
apply (subst ptr-retyp-footprint)
apply (rule intvlI)
apply (simp add: size-of-def)
apply (subst ptr-retyp-footprint)
apply (rule intvlI)
apply (simp add: size-of-def)
apply clarsimp
by (metis (mono-tags) id-apply len-of-addr-card less-trans map-le-map-add max-size
of-nat-eq-id
size-of-def take-bit-nat-eq-self unsigned-of-nat)

lemma (in mem-type) ptr-retyp-h-t-valid:

$g p \implies \text{ptr-retty } p \ d, g \models_t (p::'a \ \text{ptr})$
by (*simp add: h-t-valid-def ptr-retty-valid-footprint*)

lemma (*in mem-type*) *ptr-retty-s-valid*:
 $g p \implies \text{lift-state } (h, \text{ptr-retty } p \ d), g \models_s (p::'a \ \text{ptr})$
by (*simp add: s-valid-def proj-d-lift-state ptr-retty-h-t-valid*)

lemma (*in mem-type*) *lt-size-of-unat-simps*:
 $k < \text{size-of } \text{TYPE}('a) \implies \text{unat } ((\text{of-nat } k)::\text{addr}) < \text{size-of } \text{TYPE}('a)$
by (*metis le-def le-unat-uoi less-trans*)

lemma (*in mem-type*) *ptr-retty-h-t-valid-same*:
 $\llbracket d, g \models_t (p::'a \ \text{ptr}); x \in \{\text{ptr-val } p..+\text{size-of } \text{TYPE}('a)\} \rrbracket \implies$
 $\text{snd } (\text{ptr-retty } p \ d \ x) \subseteq_m \text{snd } (d \ x)$
apply(*clarsimp simp: h-t-valid-def valid-footprint-def Let-def*)
apply(*subst ptr-retty-footprint*)
apply *simp*
apply *clarsimp*
apply(*drule-tac x=unat (x - ptr-val p) in spec*)
apply *clarsimp*
apply(*erule impE*)
apply(*simp only: size-of-def [symmetric, where t=TYPE('a)]*)
apply(*drule intvlD, clarsimp*)
using *lt-size-of-unat-simps* **apply** *blast*
apply(*fastforce intro: map-add-le-mapI*)
done

lemma (*in mem-type*) *ptr-retty-ptr-safe [simp]*:
 $\text{ptr-safe } p \ (\text{ptr-retty } (p::'a \ \text{ptr}) \ d)$
by (*force intro: h-t-valid-ptr-safe ptr-retty-h-t-valid*)

lemma (*in mem-type*) *lift-state-pretty-restrict*:
 $(\text{lift-state } (h, \text{ptr-retty } p \ d) \mid' \{(x, k). x \in \{\text{ptr-val } p..+\text{size-of } \text{TYPE}('a)\}\}) =$
 $(\text{lift-state } (h, d) \mid' \{(x, k). x \in \{\text{ptr-val } p..+\text{size-of } \text{TYPE}('a)\}\}) ++$
 $\text{lift-state } (h, \text{ptr-retty } (p::'a \ \text{ptr}) \ \text{empty-htd}) \ \text{is } ?x = ?y$

proof (*rule ext, cases*)
fix $x::\text{addr} \times s\text{-heap-index}$
assume $\text{fst } x \in \{\text{ptr-val } p..+\text{size-of } \text{TYPE}('a)\}$
thus $?x \ x = ?y \ x$
apply(*cases x*)
apply(*clarsimp simp: lift-state-def map-add-def split: s-heap-index.splits*)
apply(*safe; clarsimp*)
apply(*drule ptr-retty-d-empty, fast*)
apply(*frule-tac d=d in ptr-retty-d-empty, clarsimp simp: map-add-def*)
apply(*clarsimp simp: restrict-map-def split: option.splits*)
apply(*drule ptr-retty-d-empty, fast*)
apply(*drule ptr-retty-d-empty, fast*)
apply(*frule-tac d=d in ptr-retty-d-empty, clarsimp simp: map-add-def*)


```

    apply(clarsimp simp: restrict-map-def split: option.splits)
  done
next
  fix x::addr × s-heap-index
  assume fst x ∉ {ptr-val p..+size-of TYPE('a)}
  thus ?x x = ?y x
  by (cases x) (auto simp: lift-state-def ptr-retyp-None map-add-def split: s-heap-index.splits)
qed

```

```

lemmas typ-simps = lift-t-heap-update lift-t-heap-update-same lift-heap-update
      lift-t-h-t-valid h-t-valid-ptr-safe lift-t-ptr-safe lift-lift-t lift-t-lift
      tag-disj-def typ-tag-le-def typ-uinfo-t-def

```

```

declare field-desc-def [simp del]

```

```

lemma field-fd:
  field-fd (t::'a::c-type itself) n =
    (case field-lookup (typ-info-t TYPE('a)) n 0 of
      None ⇒ field-desc (fst (the (None::('a xtyp-info × nat) option)))
      | Some x ⇒ field-desc (fst x))
  by (auto simp: field-fd-def field-typ-def field-typ-untyped-def split: option.splits)

```

```

declare field-desc-def [simp add]

```

```

lemma (in mem-type) super-field-update-lookup:
  assumes field-lookup (typ-info-t TYPE('b)) f 0 = Some (s,n)
    and typ-uinfo-t TYPE('a) = export-uinfo s
    and lift-t g h p = Some v'
  shows super-field-update-t (Ptr (&(p→f))) (v::'a) (lift-t g h::'b::mem-type typ-heap)
  =
    (lift-t g h)(p ↦ field-update (field-desc s) (to-bytes-p v) v')

```

```

proof –
  note unsigned-of-nat [simp del]
  from assms have size-of TYPE('b) < addr-card by simp
  with assms have [simp]: unat (of-nat n :: addr-bitsize word) = n
  apply(subst unat-of-nat)
  apply(subst mod-less)
  apply(drule td-set-field-lookupD)+
  apply(drule td-set-offset-size)+
  apply(subst len-of-addr-card)
  apply(subst (asm) c-type-class.size-of-def [symmetric, where t=TYPE('b)])+
  apply arith
  apply simp
  done
from assms show ?thesis
  apply(clarsimp simp: super-field-update-t-def c-type-class.super-field-update-t-def)
  apply(rule ext)
  apply(clarsimp simp: field-lvalue-def c-type-class.field-lvalue-def split: option.splits)

```

```

apply(safe; clarsimp?)
  apply(frule-tac v=v and v'=v' in update-field-update)
  apply clarsimp
  apply(thin-tac P = update-ti-t x y z for P x y z)
  apply(clarsimp simp: field-of-t-def c-type-class.field-of-t-def field-of-def c-type-class.typ-uinfo-t-def
typ-uinfo-t-def)
  apply(frule-tac m=0 in field-names-SomeD2; clarsimp)
  apply(simp add: field-typ-def c-type-class.field-typ-def field-typ-untyped-def)
  apply(frule field-lookup-export-uinfo-Some)
  apply(frule-tac s=k in field-lookup-export-uinfo-Some)
  apply simp

apply(drule (1) field-lookup-inject)
  apply(subst c-type-class.typ-uinfo-t-def [symmetric, where t=TYPE('b)])
  apply simp
  apply simp
apply(drule field-of-t-mem)+
apply(cases h)
apply(clarsimp simp: lift-t-if c-type-class.lift-t-if split: if-split-asm)
apply(drule (1) h-t-valid-neq-disjoint)
  apply simp
  apply(fastforce simp: unat-eq-zero field-of-t-def c-type-class.field-of-t-def
field-of-def dest!: td-set-size-lte)
  apply fast
  apply(clarsimp simp: field-of-t-def c-type-class.field-of-t-def field-of-def td-set-field-lookup)
  apply(fastforce simp: typ-uinfo-t-def c-type-class.typ-uinfo-t-def dest: field-lookup-export-uinfo-Some)
  apply(clarsimp simp: field-of-t-def c-type-class.field-of-t-def field-of-def td-set-field-lookup)
  apply(fastforce simp: typ-uinfo-t-def c-type-class.typ-uinfo-t-def dest: field-lookup-export-uinfo-Some)
done
qed

```

```

lemmas typ-rewrs =
  lift-lift-t
  lift-t-heap-update
  lift-t-heap-update-same
  lift-t-heap-update [OF lift-t-h-t-valid]
  lift-t-heap-update-same [OF lift-t-h-t-valid]
  lift-lift-t [OF lift-t-h-t-valid]

```

end

```

theory Separation-UMM
imports TypHeap

```

begin

type-synonym (*'a','b*) *map-assert* = (*'a* \rightarrow *'b*) \Rightarrow *bool*
type-synonym *heap-assert* = (*addr* \times *s-heap-index,s-heap-value*) *map-assert*

definition *sep-emp* :: (*'a','b*) *map-assert* (\square) **where**
sep-emp \equiv (=) *Map.empty*

definition *sep-true* :: (*'a','b*) *map-assert* **where**
sep-true \equiv $\lambda s.$ *True*

definition *sep-false* :: (*'a','b*) *map-assert* **where**
sep-false \equiv $\lambda s.$ *False*

definition
sep-conj :: (*'a','b*) *map-assert* \Rightarrow (*'a','b*) *map-assert* \Rightarrow (*'a','b*) *map-assert* (**infixr**
 \wedge^* 35)
where
P \wedge^* *Q* \equiv $\lambda s.$ $\exists s_0 s_1. s_0 \perp s_1 \wedge s = s_1 ++ s_0 \wedge P s_0 \wedge Q s_1$

definition
sep-impl :: (*'a','b*) *map-assert* \Rightarrow (*'a','b*) *map-assert* \Rightarrow (*'a','b*) *map-assert* (**infixr**
 \longrightarrow^* 25)
where
x \longrightarrow^* *y* \equiv $\lambda s.$ $\forall s'. s \perp s' \wedge x s' \longrightarrow y (s ++ s')$

definition
singleton :: *'a::c-type ptr* \Rightarrow *'a* \Rightarrow *heap-mem* \Rightarrow *heap-typ-desc* \Rightarrow *heap-state*
where
singleton p v h d \equiv *lift-state (heap-update p v h,d) |' s-footprint p*

Like in Separation.thy, these arrows are defined using *bsub* and *esub* but have an *input* syntax abbreviation with just *sub*. Why? Because if *sub* is the only way, people write things like $p \mapsto^i (f x y) v$ instead of $p \mapsto^i (f x y) v$. We preserve the *sub* syntax though, because *esub* and *bsub* are a pain to type.

definition
sep-map :: *'a::c-type ptr* \Rightarrow *'a ptr-guard* \Rightarrow *'a* \Rightarrow *heap-assert* ($- \mapsto -$ [56,0,51]
56)
where
p $\mapsto_g v \equiv$ $\lambda s.$ *lift-typ-heap g s p = Some v* \wedge *dom s = s-footprint p* \wedge *wf-heap-val s*

notation (*input*)
sep-map ($- \mapsto -$ [56,1000,51] 56)

definition

$sep\text{-}map\text{-}any :: 'a :: c\text{-}type\ ptr \Rightarrow 'a\ ptr\text{-}guard \Rightarrow heap\text{-}assert\ (- \mapsto - \text{ - } [56,0] 56)$
where

$p \mapsto_g - \equiv \lambda s. \exists v. (p \mapsto_g v) s$

notation (*input*)

$sep\text{-}map\text{-}any\ (- \mapsto - \text{ - } [56,0] 56)$

definition

$sep\text{-}map' :: 'a :: c\text{-}type\ ptr \Rightarrow 'a\ ptr\text{-}guard \Rightarrow 'a \Rightarrow heap\text{-}assert\ (- \hookrightarrow - \text{ - } [56,0,51] 56)$

where

$p \hookrightarrow_g v \equiv (p \mapsto_g v) \wedge^* sep\text{-}true$

notation (*input*)

$sep\text{-}map'\ (- \hookrightarrow - \text{ - } [56,1000,51] 56)$

definition

$sep\text{-}map'\text{-}any :: 'a :: c\text{-}type\ ptr \Rightarrow 'a\ ptr\text{-}guard \Rightarrow heap\text{-}assert\ (- \hookrightarrow - \text{ - } [56,0] 56)$

where

$p \hookrightarrow_g - \equiv \lambda s. \exists x. (p \hookrightarrow_g x) s$

notation (*input*)

$sep\text{-}map'\text{-}any\ (- \hookrightarrow - \text{ - } [56,0] 56)$

syntax

$\text{-}sep\text{-}assert :: bool \Rightarrow heap\text{-}state \Rightarrow bool\ ('(-)^{sep} [0] 100)$

—

lemma *sep-empD*:

$\square s \Longrightarrow s = Map.empty$

by (*simp add: sep-emp-def*)

lemma *sep-emp-empty* [*simp*]:

$\square Map.empty$

by (*simp add: sep-emp-def*)

lemma *sep-true* [*simp*]:

$sep\text{-}true\ s$

by (*simp add: sep-true-def*)

lemma *sep-false* [*simp*]:

$\neg sep\text{-}false\ s$

by (*simp add: sep-false-def*)

declare *sep-false-def* [*symmetric, simp add*]

lemma *singleton-dom'*:

$dom\ (singleton\ p\ (v :: 'a :: mem\text{-}type)\ h\ d) = dom\ (lift\text{-}state\ (h, d)) \cap s\text{-}footprint\ p$

by (*auto simp: singleton-def lift-state-def*)

split: if-split-asm s-heap-index.splits)

lemma *lift-state-dom*:

$d, g \models_t p \implies s\text{-footprint } (p::'a::\text{mem-type ptr}) \subseteq \text{dom } (\text{lift-state } (h, d))$
supply *unsigned-of-nat* [*simp del*]
apply(*clarsimp simp: h-t-valid-def valid-footprint-def Let-def*)
apply(*clarsimp simp: lift-state-def split: s-heap-index.splits option.splits*)
apply(*rule conjI; clarsimp*)
apply(*fastforce dest: s-footprintD intvlD simp: size-of-def*)
apply(*frule s-footprintD2*)
apply(*drule s-footprintD*)
apply(*drule intvlD, clarsimp*)
apply(*rename-tac k*)
apply(*drule-tac x=k in spec*)
apply(*erule impE*)
apply(*simp add: size-of-def*)
apply(*subst (asm) word-unat.eq-norm*)
apply(*subst (asm) mod-less*)
apply(*subst len-of-addr-card*)
apply(*erule less-trans*)
apply(*rule max-size*)
apply(*force simp: map-le-def*)
done

lemma *singleton-dom*:

$d, g \models_t p \implies \text{dom } (\text{singleton } p (v::'a::\text{mem-type}) h d) = s\text{-footprint } p$
apply(*subst singleton-dom'*)
apply(*fastforce dest: lift-state-dom*)
done

lemma *wf-heap-val-restrict* [*simp*]:

$wf\text{-heap-}val\ s \implies wf\text{-heap-}val\ (s \mid X)$
unfolding *wf-heap-val-def* **by** (*auto simp: restrict-map-def*)

lemma *singleton-wf-heap-val* [*simp*]:

$wf\text{-heap-}val\ (\text{singleton } p\ v\ h\ d)$
unfolding *singleton-def* **by** *simp*

lemma *h-t-valid-restrict-proj-d*:

$\llbracket \text{proj-d } s, g \models_t p; \forall x. x \in s\text{-footprint } p \implies s\ x = s'\ x \rrbracket \implies$
 $\text{proj-d } s', g \models_t p$
apply(*clarsimp simp: h-t-valid-def valid-footprint-def Let-def*)
apply(*rule conjI*)
apply(*drule-tac x=y in spec*)
apply *simp*
apply(*clarsimp simp: proj-d-def map-le-def*)
apply(*drule-tac x=ptr-val p + of-nat y in spec*)
apply(*drule-tac x=SIndexTyp a in spec*)
apply(*erule impE*)

```

  apply(erule s-footprintI)
  apply(simp add: size-of-def)
  apply simp
  apply(clarsimp simp: proj-d-def)
  apply(drule-tac x=y in spec)
  apply clarsimp
  apply(drule-tac x=ptr-val p + of-nat y in spec)
  apply(drule-tac x=SIndexVal in spec)
  apply(erule impE)
  apply(rule s-footprintI2)
  apply(simp add: size-of-def)
  apply force
done

```

lemma *s-valid-restrict* [simp]:
 $s \mid' s\text{-footprint } p, g \models_s p = s, g \models_s p$
 by (fastforce simp: s-valid-def elim: h-t-valid-restrict-proj-d)

lemma *proj-h-restrict*:
 $(x, SIndexVal) \in X \implies \text{proj-h } (s \mid' X) x = \text{proj-h } s x$
 by (auto simp: proj-h-def)

lemma *heap-list-s-restrict*:
 $(\lambda x. (x, SIndexVal)) \mid' \{p..+n\} \subseteq X \implies \text{heap-list-s } (s \mid' X) n p = \text{heap-list-s } s n p$
 by (induct n arbitrary: p)
 apply(simp add: heap-list-s-def)
 apply(clarsimp simp: heap-list-s-def)
 apply(rule conjI)
 apply(fastforce intro: proj-h-restrict intvl-self)
 apply(fastforce intro: intvl-plus-sub-Suc)
done

lemma *lift-tyt-heap-restrict* [simp]:
 $\text{lift-tyt-heap } g (s \mid' s\text{-footprint } p) p = \text{lift-tyt-heap } g s p$
 apply(clarsimp simp: lift-tyt-heap-if)
 apply(subst heap-list-s-restrict)
 apply clarsimp
 apply(drule intvlD, clarsimp)
 apply(erule s-footprintI2)
 apply simp
done

lemma *singleton-s-valid*:
 $d, g \models_t p \implies \text{singleton } p (v::'a::\text{mem-type}) h d, g \models_s p$
 by (simp add: singleton-def h-t-s-valid)

lemma *singleton-lift-tyt-heap-Some*:
 $d, g \models_t p \implies \text{lift-tyt-heap } g (\text{singleton } p v h d) p = \text{Some } (v::'a::\text{mem-type})$

by (simp add: singleton-def lift-t lift-t-heap-update h-t-valid-restrict)

lemma *sep-map-g*:

$(p \mapsto_g v) s \implies g p$

by (force simp: sep-map-def dest: lift-tyr-heap-g)

lemma *sep-map-g-sep-false*:

$\neg g p \implies (p \mapsto_g v) = \text{sep-false}$

by (simp add: sep-map-def lift-tyr-heap-if s-valid-def h-t-valid-def)

lemma *sep-map-singleton*:

$d, g \models_t p \implies ((p::'a::\text{mem-type ptr}) \mapsto_g v) (\text{singleton } p \ v \ h \ d)$

by (simp add: sep-map-def singleton-lift-tyr-heap-Some singleton-dom)

lemma *sep-mapD*:

$(p \mapsto_g v) s \implies \text{lift-tyr-heap } g \ s \ p = \text{Some } v \wedge$

$\text{dom } s = \text{s-footprint } p \wedge \text{wf-heap-val } s$

by (simp add: sep-map-def)

lemma *sep-map-lift-tyr-heapD*:

$(p \mapsto_g v) s \implies \text{lift-tyr-heap } g \ s \ p = \text{Some } (v::'a::\text{c-type})$

by (simp add: sep-map-def)

lemma *sep-map-dom-exc*:

$(p \mapsto_g (v::'a::\text{c-type})) s \implies \text{dom } s = \text{s-footprint } p$

by (simp add: sep-map-def)

lemma *sep-map-inj*:

$\llbracket (p \mapsto_g (v::'a::\text{c-type})) s; (p \mapsto_h v') s \rrbracket \implies v = v'$

by (clarsimp simp: sep-map-def lift-tyr-heap-if split: if-split-asm)

lemma *sep-map-anyI-exc* [simp]:

$(p \mapsto_g v) s \implies (p \mapsto_g -) s$

by (force simp: sep-map-any-def)

lemma *sep-map-anyD-exc*:

$(p \mapsto_g -) s \implies \exists v. (p \mapsto_g v) s$

by (force simp: sep-map-any-def)

lemma *sep-map-any-singleton*:

$d, g \models_t i \implies (i \mapsto_g -) (\text{singleton } i \ (v::'a::\text{mem-type}) \ h \ d)$

by (unfold sep-map-any-def, rule exI [where x=v], erule sep-map-singleton)

lemma *proj-h-heap-merge*:

$\text{proj-h } (s \ ++ \ t) = (\lambda x. \text{if } (x, \text{SIndexVal}) \in \text{dom } t \text{ then } \text{proj-h } t \ x \ \text{else } \text{proj-h } s \ x)$

by (force simp: proj-h-def split: option.splits)

lemma *s-valid-heap-merge-right*:

$s_{1,g} \models_s p \implies s_0 \ ++ \ s_{1,g} \models_s p$

```

apply (clarsimp simp: s-valid-def h-t-valid-def valid-footprint-def Let-def)
apply (rule conjI)
apply(drule-tac x=y in spec, simp)
apply clarsimp
apply(erule map-le-trans)
apply(clarsimp simp: proj-d-def map-le-def split: option.splits)
apply(drule-tac x=y in spec, simp)
apply(clarsimp simp: proj-d-def map-le-def split: option.splits)
done

```

lemma *proj-d-map-add-fst*:
 $\text{fst } (\text{proj-d } (s \text{ ++ } t) x) = (\text{if } (x, SIndexVal) \in \text{dom } t \text{ then } \text{fst } (\text{proj-d } t x) \text{ else } \text{fst } (\text{proj-d } s x))$
by (auto simp: proj-d-def split: option.splits)

lemma *proj-d-map-add-snd*:
 $\text{snd } (\text{proj-d } (s \text{ ++ } t) x) n = (\text{if } (x, SIndexTyp n) \in \text{dom } t \text{ then } \text{snd } (\text{proj-d } t x) n \text{ else } \text{snd } (\text{proj-d } s x) n)$
by (auto simp: proj-d-def split: option.splits)

lemma *proj-d-restrict-map-fst*:
 $(x, SIndexVal) \in X \implies \text{fst } (\text{proj-d } (s \text{ |' } X) x) = \text{fst } (\text{proj-d } s x)$
by (auto simp: proj-d-def)

lemma *proj-d-restrict-map-snd*:
 $(x, SIndexTyp n) \in X \implies \text{snd } (\text{proj-d } (s \text{ |' } X) x) n = \text{snd } (\text{proj-d } s x) n$
by (auto simp: proj-d-def)

lemma *s-valid-heap-merge-right2*:
 $\llbracket s_0 \text{ ++ } s_{1,g} \models_s p; \text{s-footprint } p \subseteq \text{dom } s_1 \rrbracket \implies s_{1,g} \models_s p$
apply(clarsimp simp: s-valid-def h-t-valid-def valid-footprint-def Let-def)
apply(rule conjI)
apply(clarsimp simp: map-le-def)
apply(subst proj-d-map-add-snd)
apply(clarsimp split: if-split-asm)
apply(subgoal-tac (ptr-val p + of-nat y, SIndexTyp a) \in s-footprint p)
apply fast
apply(erule s-footprintI)
apply(simp add: size-of-def)
apply(subgoal-tac (ptr-val p + of-nat y, SIndexVal) \in s-footprint p)
apply(drule (1) subsetD)
apply clarsimp
apply(subst (asm) proj-d-map-add-fst)
apply(drule-tac x=y **in** spec)
apply(clarsimp split: if-split-asm)
apply(rule s-footprintI2)
apply(simp add: size-of-def)
done

lemma *heap-list-s-heap-merge-right'*:

$\llbracket s_1, g \models_s (p :: 'a :: c\text{-type ptr}); n \leq \text{size-of } TYPE('a) \rrbracket \implies$
 $\text{heap-list-s } (s_0 ++ s_1) n (\text{ptr-val } p + \text{of-nat } (\text{size-of } TYPE('a) - n))$
 $= \text{heap-list-s } s_1 n (\text{ptr-val } p + \text{of-nat } (\text{size-of } TYPE('a) - n))$

proof (*induct n*)

case 0 thus ?*case* **by** (*simp add: heap-list-s-def*)

next

case (*Suc n*)

hence (*ptr-val p + (of-nat (size-of TYPE('a) - Suc n)), SIndexVal*) $\in \text{dom } s_1$

by - (*drule-tac x=size-of TYPE('a) - Suc n in s-valid-Some, auto*)

with *Suc* **show** ?*case*

by (*simp add: heap-list-s-def proj-h-heap-merge algebra-simps*)

qed

lemma *heap-list-s-heap-merge-right*:

$s_1, g \models_s ((Ptr\ p) :: 'a :: c\text{-type ptr}) \implies$
 $\text{heap-list-s } (s_0 ++ s_1) (\text{size-of } TYPE('a))\ p = \text{heap-list-s } s_1 (\text{size-of } TYPE('a))$

p

by (*force dest: heap-list-s-heap-merge-right'*)

lemma *lift-typ-heap-heap-merge-right*:

$\text{lift-typ-heap } g\ s_1\ p = \text{Some } v \implies \text{lift-typ-heap } g\ (s_0 ++ s_1)\ (p :: 'a :: c\text{-type ptr}) =$
 $\text{Some } v$

by (*force simp: lift-typ-heap-if-s-valid-heap-merge-right heap-list-s-heap-merge-right split: if-split-asm*)

lemma *lift-typ-heap-heap-merge-sep-map*:

$(p \mapsto_g v)\ s_1 \implies \text{lift-typ-heap } g\ (s_0 ++ s_1)\ p = \text{Some } (v :: 'a :: c\text{-type})$

by (*drule sep-map-lift-typ-heapD, erule lift-typ-heap-heap-merge-right*)

lemma *sep-conjI*:

$\llbracket P\ s_0; Q\ s_1; s_0 \perp s_1; s = s_1 ++ s_0 \rrbracket \implies (P \wedge^* Q)\ s$

by (*force simp: sep-conj-def*)

lemma *sep-conjD*:

$(P \wedge^* Q)\ s \implies \exists s_0\ s_1. s_0 \perp s_1 \wedge s = s_1 ++ s_0 \wedge P\ s_0 \wedge Q\ s_1$

by (*force simp: sep-conj-def*)

lemma *sep-map'I*:

$((p \mapsto_g v) \wedge^* \text{sep-true})\ s \implies (p \hookrightarrow_g v)\ s$

by (*simp add: sep-map'-def*)

lemma *sep-map'D*:

$(p \hookrightarrow_g v)\ s \implies ((p \mapsto_g v) \wedge^* \text{sep-true})\ s$

by (*simp add: sep-map'-def*)

lemma *sep-map'-g-exc*:

$(p \hookrightarrow_g v)\ s \implies g\ p$

by (force simp add: sep-map'-def dest: sep-conjD sep-map-g)

lemma *sep-conj-sep-true*:

$P s \implies (P \wedge^* \text{sep-true}) s$

by (erule-tac $s_1 = \text{Map.empty}$ in *sep-conjI*, simp+)

lemma *sep-map-sep-map'-exc* [simp]:

$(p \mapsto_g v) s \implies (p \hookrightarrow_g v) s$

by (unfold *sep-map'-def*, erule *sep-conj-sep-true*)

lemma *sep-conj-true* [simp]:

$(\text{sep-true} \wedge^* \text{sep-true}) = \text{sep-true}$

apply (rule ext)

subgoal for x

 apply simp

 apply (rule *sep-conjI* [where $s_0 = x$ and $s_1 = \text{Map.empty}$])

 apply auto

 done

done

lemma *sep-conj-assocD*:

assumes $l: ((P \wedge^* Q) \wedge^* R) s$

shows $(P \wedge^* (Q \wedge^* R)) s$

proof –

from l obtain $s' s_2$ where $\text{disj-o}: s' \perp s_2$ and $\text{merge-o}: s = s_2 ++ s'$ and
 $l\text{-o}: (P \wedge^* Q) s'$ and $r\text{-o}: R s_2$ by (force dest: *sep-conjD*)

then obtain $s_0 s_1$ where $\text{disj-i}: s_0 \perp s_1$ and $\text{merge-i}: s' = s_1 ++ s_0$ and
 $l\text{-i}: P s_0$ and $r\text{-i}: Q s_1$ by (force dest: *sep-conjD*)

from disj-o disj-i merge-i have $\text{disj-i}': s_1 \perp s_2$

 by (force simp: *map-ac-simps*)

with $r\text{-i}$ and $r\text{-o}$ have $r\text{-o}': (Q \wedge^* R) (s_2 ++ s_1)$ by (fast intro: *sep-conjI*)

from disj-o merge-i disj-i $\text{disj-i}'$ have $s_0 \perp s_2 ++ s_1$

 by (force simp: *map-ac-simps*)

with $r\text{-o}'$ $l\text{-i}$ have $(P \wedge^* (Q \wedge^* R)) ((s_2 ++ s_1) ++ s_0)$

 by (force intro: *sep-conjI*)

moreover from merge-o merge-i disj-i $\text{disj-i}'$ have $s = ((s_2 ++ s_1) ++ s_0)$

 by (simp add: *map-add-assoc* [symmetric])

ultimately show ?thesis by simp

qed

lemma *sep-conj-com*:

$(P \wedge^* Q) = (Q \wedge^* P)$

by (rule ext) (auto simp: *map-ac-simps* *sep-conjI* dest!: *sep-conjD*)

lemma *sep-conj-false-right* [simp]:

$(P \wedge^* \text{sep-false}) = \text{sep-false}$

by (force dest: *sep-conjD*)

lemma *sep-conj-false-left* [*simp*]:

$(\text{sep-false} \wedge^* P) = \text{sep-false}$

by (*simp add: sep-conj-com*)

lemma *sep-conj-comD*:

$(P \wedge^* Q) s \implies (Q \wedge^* P) s$

by (*simp add: sep-conj-com*)

lemma *exists-left*:

$(Q \wedge^* (\lambda s. \exists x. P x s)) = ((\lambda s. \exists x. P x s) \wedge^* Q)$ **by** (*simp add: sep-conj-com*)

lemma *sep-conj-assoc*:

$((P \wedge^* Q) \wedge^* R) = (P \wedge^* (Q \wedge^* R))$ (**is** $?x = ?y$)

proof (*rule ext, standard*)

fix s

assume $?x s$

thus $?y s$ **by** $-$ (*erule sep-conj-assocD*)

next

fix s

assume $?y s$

hence $((R \wedge^* Q) \wedge^* P) s$ **by** (*simp add: sep-conj-com*)

hence $(R \wedge^* (Q \wedge^* P)) s$ **by** $-$ (*erule sep-conj-assocD*)

thus $?x s$ **by** (*simp add: sep-conj-com*)

qed

lemma *sep-conj-left-com*:

$(P \wedge^* (Q \wedge^* R)) = (Q \wedge^* (P \wedge^* R))$ (**is** $?x = ?y$)

proof $-$

have $?x = ((Q \wedge^* R) \wedge^* P)$ **by** (*simp add: sep-conj-com*)

also have $\dots = (Q \wedge^* (R \wedge^* P))$ **by** (*subst sep-conj-assoc, simp*)

finally show $?thesis$ **by** (*simp add: sep-conj-com*)

qed

lemmas *sep-conj-ac = sep-conj-com sep-conj-assoc sep-conj-left-com*

lemma *sep-conj-empty*:

$(P \wedge^* \square) = P$

proof (*rule ext, standard*)

fix x

assume $(P \wedge^* \square) x$

thus $P x$ **by** (*force simp: sep-emp-def dest: sep-conjD*)

next

fix x

assume $P x$

moreover have $\square \text{ Map.empty}$ **by** *simp*

ultimately show $(P \wedge^* \square) x$ **by** $-$ (*erule (1) sep-conjI, auto*)

qed

lemma *sep-conj-empty'* [simp]:

$$(\Box \wedge^* P) = P$$

by (simp add: sep-conj-empty sep-conj-ac)

lemma *sep-conj-true-P* [simp]:

$$(\text{sep-true} \wedge^* (\text{sep-true} \wedge^* P)) = (\text{sep-true} \wedge^* P)$$

by (simp add: sep-conj-ac)

lemma *sep-map'-unfold-exc*:

$$(p \hookrightarrow_g v) = ((p \hookrightarrow_g v) \wedge^* \text{sep-true})$$

by (simp add: sep-map'-def sep-conj-ac)

lemma *sep-conj-disj*:

$$((\lambda s. P s \vee Q s) \wedge^* R) s = ((P \wedge^* R) s \vee (Q \wedge^* R) s) \text{ (is } ?x = (?y \vee ?z))$$

proof

assume ?x

then obtain $s_0 s_1$ where $s_0 \perp s_1$ and $s = s_1 ++ s_0$ and $P s_0 \vee Q s_0$ and $R s_1$

by - (drule sep-conjD, auto)

moreover from this have $\neg ?z \implies \neg Q s_0$

by - (clarsimp, erule notE, erule (2) sep-conjI, simp)

ultimately show $?y \vee ?z$ by (force intro: sep-conjI)

next

have $?y \implies ?x$

by (force simp: map-ac-simps intro: sep-conjI dest: sep-conjD)

moreover have $?z \implies ?x$

by (force simp: map-ac-simps intro: sep-conjI dest: sep-conjD)

moreover assume $?y \vee ?z$

ultimately show $?x$ by fast

qed

lemma *sep-conj-conj*:

$$((\lambda s. P s \wedge Q s) \wedge^* R) s \implies (P \wedge^* R) s \wedge (Q \wedge^* R) s$$

by (force intro: sep-conjI dest!: sep-conjD)

lemma *sep-conj-exists1*:

$$((\lambda s. \exists x. P x s) \wedge^* Q) = (\lambda s. (\exists x. (P x \wedge^* Q) s))$$

by (force intro: sep-conjI dest: sep-conjD)

lemma *sep-conj-exists2*:

$$(P \wedge^* (\lambda s. \exists x. Q x s)) = (\lambda s. (\exists x. (P \wedge^* Q x) s))$$

by (force intro: sep-conjI dest: sep-conjD)

lemmas *sep-conj-exists* = *sep-conj-exists1 sep-conj-exists2*

lemma *sep-conj-forall*:

$$((\lambda s. \forall x. P x s) \wedge^* Q) s \implies (P \wedge^* Q) s$$

by (force intro: sep-conjI dest: sep-conjD)

lemma sep-conj-impl:

$\llbracket (P \wedge^* Q) s; \bigwedge s. P s \implies P' s; \bigwedge s. Q s \implies Q' s \rrbracket \implies (P' \wedge^* Q') s$
by (force intro: sep-conjI dest: sep-conjD)

lemma sep-conj-sep-true-left:

$(P \wedge^* Q) s \implies (sep\text{-true} \wedge^* Q) s$
by (erule sep-conj-impl, simp+)

lemma sep-conj-sep-true-right:

$(P \wedge^* Q) s \implies (P \wedge^* sep\text{-true}) s$
by (subst (asm) sep-conj-com, drule sep-conj-sep-true-left,
simp add: sep-conj-ac)

lemma sep-globalise:

$\llbracket (P \wedge^* R) s; (\bigwedge s. P s \implies Q s) \rrbracket \implies (Q \wedge^* R) s$
by (fast elim: sep-conj-impl)

lemma sep-implI:

$\forall s'. s \perp s' \wedge x s' \longrightarrow y (s ++ s') \implies (x \longrightarrow^* y) s$
by (force simp: sep-impl-def)

lemma sep-implI':

assumes $x: \bigwedge h'. \llbracket h \perp h'; x h' \rrbracket \implies y (h ++ h')$
shows $(x \longrightarrow^* y) h$
apply (simp add: sep-impl-def)
apply (intro allI impI)
apply (erule conjE)
apply (erule (1) x)
done

lemma sep-implD:

$(x \longrightarrow^* y) s \implies \forall s'. s \perp s' \wedge x s' \longrightarrow y (s ++ s')$
by (force simp: sep-impl-def)

lemma sep-emp-sep-impl [simp]:

$(\square \longrightarrow^* P) = P$
apply (rule ext)
apply (clarsimp simp: sep-impl-def)
apply (rule iffI; clarsimp?)
apply (drule-tac x=Map.empty in spec)
apply fastforce
apply (fastforce dest: sep-empD)
done

lemma sep-impl-sep-true [simp]:

$(P \longrightarrow^* sep\text{-true}) = sep\text{-true}$
by (force intro: sep-implI)

```

lemma sep-impl-sep-false [simp]:
  (sep-false  $\longrightarrow^*$  P) = sep-true
  by (force intro: sep-implI)

lemma sep-impl-sep-true-P:
  (sep-true  $\longrightarrow^*$  P) s  $\implies$  P s
  by (auto dest!: sep-implD, drule-tac x=Map.empty in spec, simp)

lemma sep-impl-sep-true-false [simp]:
  (sep-true  $\longrightarrow^*$  sep-false) = sep-false
  by (force dest: sep-impl-sep-true-P)

lemma sep-impl-impl:
  (P  $\longrightarrow^*$  Q  $\longrightarrow^*$  R) = (P  $\wedge^*$  Q  $\longrightarrow^*$  R)
  apply(rule ext)
  apply(rule iffI)
  apply(rule sep-implI)
  apply clarsimp
  apply(drule sep-conjD, clarsimp)
  apply(drule sep-implD)
  apply(drule-tac x=s0 in spec)
  apply(erule impE)
  apply(clarsimp simp: map-disj-def, fast)
  apply(drule sep-implD)
  apply(drule-tac x=s1 in spec)
  apply(erule impE)
  apply(clarsimp simp: map-disj-def, fast)
  apply(subst map-add-com [where h0=s1])
  apply(clarsimp simp: map-disj-def, fast)
  apply(subst map-add-assoc)
  apply simp
  apply(rule sep-implI, clarsimp)
  apply(rule sep-implI, clarsimp)
  apply(drule sep-implD)
  apply(drule-tac x=s' ++ s'a in spec)
  apply(erule impE)
  apply(rule conjI)
  apply(clarsimp simp: map-disj-def, fast)
  apply(erule (1) sep-conjI)
  apply(clarsimp simp: map-disj-def, fast)
  apply(subst map-add-com)
  apply(clarsimp simp: map-disj-def, fast)
  apply simp
  apply(simp add: map-add-assoc)
  done

```

```

lemma sep-conj-sep-impl:
  [P s;  $\bigwedge s. (P \wedge^* Q) s \implies R s$ ]  $\implies (Q \longrightarrow^* R) s$ 

```

apply (*rule sep-implI, clarsimp*)
proof –
fix s'
assume $P\ s$ **and** $s \perp s'$ **and** $Q\ s'$
hence $(P \wedge^* Q)\ (s ++ s')$ **by** (*force simp: map-ac-simps intro: sep-conjI*)
moreover assume $\bigwedge s. (P \wedge^* Q)\ s \implies R\ s$
ultimately show $R\ (s ++ s')$ **by** *simp*
qed

lemma *sep-conj-sep-impl2*:
 $\llbracket (P \wedge^* Q)\ s; \bigwedge s. P\ s \implies (Q \longrightarrow^* R)\ s \rrbracket \implies R\ s$
by (*force simp: map-ac-simps dest: sep-implD sep-conjD*)

lemma *sep-map'-anyI-exc* [*simp*]:
 $(p \hookrightarrow_g v)\ s \implies (p \hookrightarrow_g -)\ s$
by (*force simp: sep-map'-any-def*)

lemma *sep-map'-anyD-exc*:
 $(p \hookrightarrow_g -)\ s \implies \exists v. (p \hookrightarrow_g v)\ s$
by (*force simp: sep-map'-any-def*)

lemma *sep-map'-any-unfold-exc*:
 $(i \hookrightarrow_g -) = ((i \hookrightarrow_g -) \wedge^* \text{sep-true})$
by (*rule ext, simp add: sep-map'-any-def*)
(subst sep-map'-unfold-exc, subst sep-conj-com, subst sep-conj-exists,
simp add: sep-conj-ac)

lemma *sep-map'-inj-exc*:
assumes $pv: (p \hookrightarrow_g (v::'a::c\text{-type}))\ s$ **and** $pv': (p \hookrightarrow_h v')\ s$
shows $v = v'$

proof –
from $pv\ pv'$ **obtain** $s_0\ s_1\ s_0'\ s_1'$ **where** $pv\text{-}m: (p \mapsto_g v)\ s_1$ **and**
 $pv'\text{-}m: (p \mapsto_h v')\ s_1'$ **and** $s_0 ++ s_1 = s_0' ++ s_1'$
by (*force simp: sep-map'-def map-ac-simps dest!: sep-conjD*)
hence $s_1 = s_1'$ **by** (*force dest!: map-add-right-dom-eq sep-map-dom-exc*)
with $pv\text{-}m\ pv'\text{-}m$ **show** *?thesis* **by** (*force dest: sep-map-inj*)
qed

lemma *sep-map'-any-dom-exc*:
 $((p::'a::\text{mem-type}\ \text{ptr}) \hookrightarrow_g -)\ s \implies (\text{ptr-val}\ p, S\text{IndexVal}) \in \text{dom}\ s$
by (*clarsimp simp: sep-map'-def sep-map'-any-def sep-conj-ac*
dest!: sep-conjD)
(subgoal-tac s₁ (ptr-val p, SIndexVal) \neq None, force simp: map-ac-simps,
force dest: sep-map-dom-exc)

lemma *sep-map'-dom-exc*:
 $(p \hookrightarrow_g (v::'a::\text{mem-type}))\ s \implies (\text{ptr-val}\ p, S\text{IndexVal}) \in \text{dom}\ s$
apply (*clarsimp simp: sep-map'-def sep-conj-ac dest!: sep-conjD*)
apply (*subgoal-tac s₁ (ptr-val p, SIndexVal) \neq None*)

```

apply(force simp: map-ac-simps)
apply(drule sep-map-dom-exc)
apply(subgoal-tac (ptr-val p, SIndexVal) ∈ s-footprint p)
apply fast
apply(rule s-footprintI2 [where x=0, simplified])
apply simp
done

```

lemma *sep-map'-lift-tyr-heapD*:

```

(p ↦g v) s ⇒
  lift-tyr-heap g s p = Some (v::'a::c-type)
by (force simp: sep-map'-def map-ac-simps dest: sep-conjD
      lift-tyr-heap-heap-merge-sep-map)

```

lemma *sep-map'-merge*:

```

assumes map'-v: (p ↦g v) s0 ∨ (p ↦g v) s1 and disj: s0 ⊥ s1
shows (p ↦g v) (s0 ++ s1) (is ?x)

```

proof cases

```

assume (p ↦g v) s0
with disj show ?x
  apply (clarsimp simp: sep-map'-def sep-conj-ac dest!: sep-conjD)
  subgoal for s0' s1'
    apply (rule sep-conjI [where s1=s1' and s0=s0' ++ s1'])
    apply (auto simp: map-ac-simps)
  done
done

```

next

```

assume ¬ (p ↦g v) s0
with map'-v have (p ↦g v) s1 by simp
with disj show ?x
  apply (clarsimp simp: sep-map'-def sep-conj-ac dest!: sep-conjD)
  subgoal for s0' s1'
    apply (rule sep-conjI [where s1=s1' and s0=s0' ++ s0'])
    apply (auto simp: map-ac-simps)
  done
done

```

qed

lemma *sep-conj-overlapD*:

```

[[ (P ∧* Q) s; ∧s. P s ⇒ ((p::'a::mem-type ptr) ↦g -) s;
  ∧s. Q s ⇒ (p ↦h -) s ]] ⇒ False

```

```

apply(drule sep-conjD,clarsimp simp: map-disj-def)

```

```

apply(subgoal-tac (ptr-val p,SIndexVal) ∈ dom s0 ∧ (ptr-val p,SIndexVal) ∈ dom
s1)

```

```

apply fast
apply(fast intro!: sep-map'-any-dom-exc)
done

```

lemma *sep-no-skew*:


```

( $\lambda s. (p \hookrightarrow_g v) s \wedge (q \hookrightarrow_h w) s) s \implies$ 
   $p=q \vee \{ptr\text{-val } (p::'a::c\text{-type ptr})..+size\text{-of } TYPE('a)\} \cap$ 
   $\{ptr\text{-val } q..+size\text{-of } TYPE('a)\} = \{\}$ 
apply clarsimp
apply(drule sep-map'-lift-tyr-heapD)+
apply(clarsimp simp: lift-tyr-heap-if s-valid-def split: if-split-asm)
apply(rule ccontr)
apply(drule (1) h-t-valid-neq-disjoint; simp?)
apply(rule peer-tyr-not-field-of; simp)
done

```

lemma *sep-no-skew2*:

```

 $\llbracket (\lambda s. (p \hookrightarrow_g v) s \wedge (q \hookrightarrow_h w) s) s; typ\text{-uinfo-t } TYPE('a) \perp_t typ\text{-uinfo-t } TYPE('b) \rrbracket$ 
 $\implies \{ptr\text{-val } (p::'a::c\text{-type ptr})..+size\text{-of } TYPE('a)\} \cap$ 
 $\{ptr\text{-val } (q::'b::c\text{-type ptr})..+size\text{-of } TYPE('b)\} = \{\}$ 
apply clarsimp
apply(drule sep-map'-lift-tyr-heapD)+
apply(clarsimp simp: lift-tyr-heap-if s-valid-def split: if-split-asm)
apply(frule (1) h-t-valid-neq-disjoint[where q=q])
  apply(clarsimp simp: tag-disj-def sub-tyr-proper-def)
  apply(simp add: tyr-tag-ll-def)
  apply(clarsimp simp: tag-disj-def tyr-tag-le-def field-of-t-def field-of-def)
apply assumption
done

```

lemma *sep-conj-impl-same*:

```

( $P \wedge^* (P \longrightarrow^* Q)$ )  $s \implies Q s$ 
apply(drule sep-conjD, clarsimp)
apply(drule sep-implD)
apply(drule-tac x=s0 in spec)
apply(clarsimp simp: map-disj-com)
done

```

definition *pure* :: ($'a, 'b$) *map-assert* \Rightarrow *bool* **where**

```

pure  $P \equiv \forall s s'. P s = P s'$ 

```

lemma *pure-sep-true*:

```

pure sep-true
by (simp add: pure-def)

```

lemma *pure-sep-false*:

```

pure sep-true
by (simp add: pure-def)

```

lemma *pure-split*:

$pure\ P = (P = sep\ true \vee P = sep\ false)$
by (*force simp: pure-def*)

lemma *pure-sep-conj*:
 $\llbracket pure\ P; pure\ Q \rrbracket \implies pure\ (P \wedge^* Q)$
by (*force simp: pure-split*)

lemma *pure-sep-impl*:
 $\llbracket pure\ P; pure\ Q \rrbracket \implies pure\ (P \longrightarrow^* Q)$
by (*force simp: pure-split*)

lemma *pure-conj-sep-conj*:
 $\llbracket (\lambda s. P\ s \wedge Q\ s)\ s; pure\ P \vee pure\ Q \rrbracket \implies (P \wedge^* Q)\ s$
by (*force simp: pure-split sep-conj-ac intro: sep-conj-sep-true*)

lemma *pure-sep-conj-conj*:
 $\llbracket (P \wedge^* Q)\ s; pure\ P; pure\ Q \rrbracket \implies (\lambda s. P\ s \wedge Q\ s)\ s$
by (*force simp: pure-split*)

lemma *pure-conj-sep-conj-assoc*:
 $pure\ P \implies ((\lambda s. P\ s \wedge Q\ s) \wedge^* R) = (\lambda s. P\ s \wedge (Q \wedge^* R)\ s)$
by (*force simp: pure-split*)

lemma *pure-sep-impl-impl*:
 $\llbracket (P \longrightarrow^* Q)\ s; pure\ P \rrbracket \implies P\ s \longrightarrow Q\ s$
by (*force simp: pure-split dest: sep-impl-sep-true-P*)

lemma *pure-impl-sep-impl*:
 $\llbracket P\ s \longrightarrow Q\ s; pure\ P; pure\ Q \rrbracket \implies (P \longrightarrow^* Q)\ s$
by (*force simp: pure-split*)

definition

intuitionistic :: ('a,'b) map-assert \Rightarrow bool

where

intuitionistic $P \equiv \forall s\ s'. P\ s \wedge s \subseteq_m s' \longrightarrow P\ s'$

lemma *intuitionisticI*:
 $(\bigwedge s\ s'. \llbracket P\ s; s \subseteq_m s' \rrbracket \implies P\ s') \implies intuitionistic\ P$
by (*unfold intuitionistic-def, fast*)

lemma *intuitionisticD*:
 $\llbracket intuitionistic\ P; P\ s; s \subseteq_m s' \rrbracket \implies P\ s'$
by (*unfold intuitionistic-def, fast*)

lemma *pure-intuitionistic*:
 $pure\ P \implies intuitionistic\ P$
by (*clarsimp simp: intuitionistic-def pure-def, fast*)

lemma *intuitionistic-sep-map'*:
intuitionistic ($p \hookrightarrow_g v$)
proof (*rule intuitionisticI*)
fix $s s'$
assume ($p \hookrightarrow_g v$) s
then obtain $s_0 s_1$ **where** ($p \mapsto_g v$) s_0 **and** $s: s = s_1 ++ s_0$
by (*force dest!: sep-conjD sep-map'D*)
moreover assume $s \subseteq_m s'$
with s **have** $s' = s' \upharpoonright' (UNIV - \text{dom } s_0) ++ s_0$
apply *simp*
apply (*drule map-add-le-mapE*)
by (*subst map-le-restrict*) *force+*
ultimately show ($p \hookrightarrow_g v$) s' **by** (*force intro: sep-map'I sep-conjI*)
qed

lemma *intuitionistic-sep-conj-sep-true*:
intuitionistic ($P \wedge^* \text{sep-true}$)
by (*rule intuitionisticI, drule sep-conjD, clarsimp*)
(*erule-tac s₁=s'₁'(dom s' - dom s₀) in sep-conjI, simp,*
force simp: map-disj-def,
force intro: map-le-dom-restrict-sub-add map-add-le-mapE sym)

lemma *intuitionistic-sep-impl-sep-true*:
intuitionistic ($\text{sep-true} \longrightarrow^* P$)
apply (*rule intuitionisticI, rule sep-implI, clarsimp*)
proof –
fix $s s' s'a$
assume ($\text{sep-true} \longrightarrow^* P$) s **and** $le: s \subseteq_m s'$ **and** $s' \perp s'a$
moreover from this have $P (s ++ (s' \upharpoonright' (\text{dom } s' - \text{dom } s) ++ s'a))$
by – (*drule sep-implD,*
drule-tac x=s'₁'(dom s' - dom s) ++ s'a in spec,
force simp: map-disj-def dest: map-disj-map-le)
moreover have $\text{dom } s \cap \text{dom } (s' \upharpoonright' (\text{dom } s' - \text{dom } s)) = \{\}$ **by** *force*
ultimately show $P (s' ++ s'a)$
by (*force simp: map-le-dom-restrict-sub-add map-add-assoc dest!: map-add-comm*)
qed

lemma *intuitionistic-conj*:
 $\llbracket \text{intuitionistic } P; \text{intuitionistic } Q \rrbracket \Longrightarrow \text{intuitionistic } (\lambda s. P s \wedge Q s)$
by (*force intro: intuitionisticI dest: intuitionisticD*)

lemma *intuitionistic-disj*:
 $\llbracket \text{intuitionistic } P; \text{intuitionistic } Q \rrbracket \Longrightarrow \text{intuitionistic } (\lambda s. P s \vee Q s)$
by (*force intro: intuitionisticI dest: intuitionisticD*)

lemma *intuitionistic-forall*:
 $(\bigwedge x. \text{intuitionistic } (P x)) \Longrightarrow \text{intuitionistic } (\lambda s. \forall x. P x s)$
by (*force intro: intuitionisticI dest: intuitionisticD*)

lemma intuitionistic-exists:

$(\bigwedge x. \text{intuitionistic } (P x)) \implies \text{intuitionistic } (\lambda s. \exists x. P x s)$
by (force intro: intuitionisticI dest: intuitionisticD)

lemma intuitionistic-sep-conj:

$\text{intuitionistic } (P::('a,'b) \text{ map-assert}) \implies \text{intuitionistic } (P \wedge^* Q)$
apply (rule intuitionisticI, drule sep-conjD, clarsimp)

proof –

fix $s' s_0 s_1$

assume $le: s_1 ++ s_0 \subseteq_m s'$ **and** $disj: s_0 \perp (s_1::'a \rightarrow 'b)$

hence $le\text{-restrict}: s_0 \subseteq_m s' \mid' (dom s' - dom s_1)$

by – (rule map-le-dom-subset-restrict, erule map-add-le-mapE,
force simp: map-disj-def dest: map-le-implies-dom-le
map-add-le-mapE)

moreover assume $\text{intuitionistic } P$ **and** $P s_0$

ultimately have $P (s' \mid' (dom s' - dom s_1))$

by – (erule (2) intuitionisticD)

moreover from $le\text{-restrict}$ **have** $s' \mid' (dom s' - dom s_1) \perp s_1$

by (force simp: map-disj-def dest: map-le-implies-dom-le)

moreover from $le\ disj$ **have** $s_1 \subseteq_m s'$

by (subst (asm) map-add-comm, force simp: map-disj-def)
(erule map-add-le-mapE)

hence $s' = s_1 ++ s' \mid' (dom s' - dom s_1)$

by (subst map-add-comm, force simp: map-disj-def,

force simp: map-le-dom-restrict-sub-add map-add-comm map-disj-def)

moreover assume $Q s_1$

ultimately show $(P \wedge^* Q) s'$

by – (erule (3) sep-conjI)

qed

lemma intuitionistic-sep-impl:

$\text{intuitionistic } (Q::('a,'b) \text{ map-assert}) \implies \text{intuitionistic } (P \longrightarrow^* Q)$
apply (rule intuitionisticI, rule sep-implI, clarsimp)

proof –

fix $s s' s'a$

assume $le: s \subseteq_m s'$ **and** $disj: s' \perp (s'a::'a \rightarrow 'b)$

moreover from $this$ **have** $s ++ s'a \subseteq_m s' ++ s'a$

proof –

from $le\ disj$ **have** $s \subseteq_m s ++ s'a$

by (subst map-add-com)

(force simp: map-disj-def dest: map-le-implies-dom-le, simp)

with $le\ disj$ **show** $?thesis$

by – (rule map-add-le-mapI, subst map-add-com,
auto elim: map-le-trans)

qed

moreover assume $(P \longrightarrow^* Q) s$ **and** $\text{intuitionistic } Q$ **and** $P s'a$

ultimately show $Q (s' ++ s'a)$

by (fast elim: map-disj-map-le intuitionisticD dest: sep-implD)

qed

lemma *strongest-intuitionistic*:

$\neg (\exists Q. (\forall s. (Q\ s \longrightarrow (P \wedge^* \text{sep-true})\ s)) \wedge \text{intuitionistic}\ Q \wedge$
 $Q \neq (P \wedge^* \text{sep-true}) \wedge (\forall s. P\ s \longrightarrow Q\ s))$
by (*clarsimp*, *rule ext*) (*force dest!*: *sep-conjD intuitionisticD*)

lemma *weakest-intuitionistic*:

$\neg (\exists Q. (\forall s. ((\text{sep-true} \longrightarrow^* P)\ s \longrightarrow Q\ s)) \wedge \text{intuitionistic}\ Q \wedge$
 $Q \neq (\text{sep-true} \longrightarrow^* P) \wedge (\forall s. Q\ s \longrightarrow P\ s))$
apply (*clarsimp*, *rule ext*)
apply (*rename-tac Q x*)
apply (*rule iffI*; *clarsimp?*)
apply (*rule sep-implI'*)
apply (*rename-tac h'*)
apply (*drule-tac s=x and s'=x ++ h' in intuitionisticD; clarsimp simp: map-ac-simps*)
done

lemma *intuitionistic-sep-conj-sep-true-P*:

$\llbracket (P \wedge^* \text{sep-true})\ s; \text{intuitionistic}\ P \rrbracket \Longrightarrow P\ s$
by (*force dest: intuitionisticD sep-conjD*)

lemma *intuitionistic-sep-conj-sep-true-simp*:

$\text{intuitionistic}\ P \Longrightarrow (P \wedge^* \text{sep-true}) = P$
by (*fast intro: sep-conj-sep-true elim: intuitionistic-sep-conj-sep-true-P*)

lemma *intuitionistic-sep-impl-sep-true-P*:

$\llbracket P\ s; \text{intuitionistic}\ P \rrbracket \Longrightarrow (\text{sep-true} \longrightarrow^* P)\ s$
by (*force simp: map-add-com intro: sep-implI dest: intuitionisticD*)

lemma *intuitionistic-sep-impl-sep-true-simp*:

$\text{intuitionistic}\ P \Longrightarrow (\text{sep-true} \longrightarrow^* P) = P$
by (*fast elim: sep-impl-sep-true-P intuitionistic-sep-impl-sep-true-P*)

definition

$\text{dom-exact} :: ('a, 'b)\ \text{map-assert} \Rightarrow \text{bool}$

where

$\text{dom-exact}\ P \equiv \forall s\ s'. P\ s \wedge P\ s' \longrightarrow \text{dom}\ s = \text{dom}\ s'$

lemma *dom-exactI*:

$(\bigwedge s\ s'. \llbracket P\ s; P\ s' \rrbracket \Longrightarrow \text{dom}\ s = \text{dom}\ s') \Longrightarrow \text{dom-exact}\ P$
by (*unfold dom-exact-def, fast*)

lemma *dom-exactD*:

$\llbracket \text{dom-exact}\ P; P\ s_0; P\ s_1 \rrbracket \Longrightarrow \text{dom}\ s_0 = \text{dom}\ s_1$
by (*unfold dom-exact-def, fast*)

lemma *dom-exact-sep-conj*:

$\llbracket \text{dom-exact } P; \text{dom-exact } Q \rrbracket \Longrightarrow \text{dom-exact } (P \wedge^* Q)$
by (*rule dom-exactI*, (*drule sep-conjD*)₊, *clarsimp*)
(*drule* (2) *dom-exactD*, *drule* (2) *dom-exactD*, *simp*)

lemma *dom-exact-sep-conj-conj*:

$\llbracket (P \wedge^* R) s; (Q \wedge^* R) s; \text{dom-exact } R \rrbracket \Longrightarrow ((\lambda s. P s \wedge Q s) \wedge^* R) s$
apply (*drule sep-conjD*)₊
by (*clarsimp simp: sep-conj-ac*,
rule sep-conjI, *fast*, *rule conjI*)
(*fast*, *drule* (2) *dom-exactD*, *drule* (1) *map-disj-add-eq-dom-right-eq*,
auto simp: map-ac-simps)

lemma *sep-conj-conj-simp*:

$\text{dom-exact } R \Longrightarrow ((\lambda s. P s \wedge Q s) \wedge^* R) = (\lambda s. (P \wedge^* R) s \wedge (Q \wedge^* R) s)$
by (*fast intro!*: *sep-conj-conj dom-exact-sep-conj-conj*)

definition *dom-eps* :: ('a,'b) *map-assert* \Rightarrow 'a *set* **where**

dom-eps *P* \equiv *THE* *x*. $\forall s. P s \longrightarrow x = \text{dom } s$

lemma *dom-epsI*:

$\llbracket \text{dom-exact } P; P s; x \in \text{dom } s \rrbracket \Longrightarrow x \in \text{dom-eps } P$
by (*unfold dom-eps-def*, *rule theI2* [**where** *a=dom s*])
(*fastforce simp: dom-exact-def*, *clarsimp*)₊)

lemma *dom-epsD* [*rule-format*]:

$\llbracket \text{dom-exact } P; P s \rrbracket \Longrightarrow x \in \text{dom-eps } P \longrightarrow x \in \text{dom } s$
by (*unfold dom-eps-def*, *rule theI2* [**where** *a=dom s*])
(*fastforce simp: dom-exact-def*, *clarsimp*)₊)

lemma *dom-eps*:

$\llbracket \text{dom-exact } P; P s \rrbracket \Longrightarrow \text{dom } s = \text{dom-eps } P$
by (*force intro: dom-epsI dest: dom-epsD*)

lemma *map-restrict-dom-exact*:

$\llbracket \text{dom-exact } P; P s \rrbracket \Longrightarrow s \mid' \text{dom-eps } P = s$
by (*force simp: restrict-map-def None-com intro: dom-epsI*)

lemma *map-restrict-dom-exact2*:

$\llbracket \text{dom-exact } P; P s_0; s_0 \perp s_1 \rrbracket \Longrightarrow (s_1 \mid' \text{dom-eps } P) = \text{Map.empty}$
by (*force simp: restrict-map-def map-disj-def dest: dom-epsD*)

lemma *map-restrict-dom-exact3*:

$\llbracket \text{dom-exact } P; P s \rrbracket \Longrightarrow s \mid' (\text{UNIV} - \text{dom-eps } P) = \text{Map.empty}$
by (*force simp: restrict-map-def dest: dom-epsI*)

lemma *map-add-restrict-dom-exact*:

$\llbracket \text{dom-exact } P; s_0 \perp s_1; P s_1 \rrbracket \Longrightarrow (s_1 ++ s_0) \mid' (\text{dom-eps } P) = s_1$
by (*simp add: map-add-restrict map-restrict-dom-exact*)

(*subst map-restrict-dom-exact2, auto simp: map-disj-def*)

lemma *map-add-restrict-dom-exact2*:

$\llbracket \text{dom-exact } P; s_0 \perp s_1; P s_0 \rrbracket \implies (s_1 ++ s_0) \mid' (UNIV - \text{dom-eps } P) = s_1$
by (*force simp: map-add-restrict map-disj-def dom-eps*
intro: restrict-map-subdom dest: map-restrict-dom-exact3)

lemma *dom-exact-sep-conj-forall*:

assumes *sc*: $\forall x. (P x \wedge^* Q) s$ **and** *de*: *dom-exact* *Q*

shows $((\lambda s. \forall x. P x s) \wedge^* Q) s$

proof (*rule sep-conjI [where s₀=s |' (UNIV - dom-eps Q) and s₁=s |' dom-eps Q]*)

from *sc de show* $\forall x. P x (s \mid' (UNIV - \text{dom-eps } Q))$

by (*force simp: map-add-restrict-dom-exact2 sep-conj-ac dest: sep-conjD*)

next

from *sc de show* $Q (s \mid' \text{dom-eps } Q)$

by (*force simp: map-add-restrict-dom-exact dest!: sep-conjD spec*)

next

from *sc de show* $s \mid' (UNIV - \text{dom-eps } Q) \perp s \mid' \text{dom-eps } Q$

by (*force simp: map-add-restrict-dom-exact2 map-ac-simps*

dest: map-add-restrict-dom-exact dest!: sep-conjD spec)

next

show $s = s \mid' \text{dom-eps } Q ++ s \mid' (UNIV - \text{dom-eps } Q)$ **by** *simp*

qed

lemma *sep-conj-forall-simp*:

dom-exact *Q* $\implies ((\lambda s. \forall x. P x s) \wedge^* Q) = (\lambda s. \forall x. (P x \wedge^* Q) s)$

by (*fast dest: sep-conj-forall dom-exact-sep-conj-forall*)

lemma *dom-exact-sep-map*:

dom-exact $(i \mapsto_g x)$

by (*clarsimp simp: dom-exact-def sep-map-def*)

lemma *dom-exact-P-emp*:

$\llbracket \text{dom-exact } P; P \text{ Map.empty}; P s \rrbracket \implies s = \text{Map.empty}$

by (*auto simp: dom-exact-def*)

definition

strictly-exact :: $('a, 'b) \text{map-assert} \Rightarrow \text{bool}$

where

strictly-exact $P \equiv \forall s s'. P s \wedge P s' \longrightarrow s = s'$

lemma *strictly-exactD*:

$\llbracket \text{strictly-exact } P; P s_0; P s_1 \rrbracket \implies s_0 = s_1$

by (*unfold strictly-exact-def, fast*)

lemma *strictly-exactI*:
 $(\bigwedge s s'. \llbracket P\ s; P\ s' \rrbracket \implies s = s') \implies \text{strictly-exact } P$
by (*unfold strictly-exact-def, fast*)

lemma *strictly-exact-dom-exact*:
 $\text{strictly-exact } P \implies \text{dom-exact } P$
by (*force simp: strictly-exact-def dom-exact-def*)

lemma *strictly-exact-sep-emp*:
 $\text{strictly-exact } \square$
by (*force simp: strictly-exact-def dest: sep-empD*)

lemma *strictly-exact-sep-conj*:
 $\llbracket \text{strictly-exact } P; \text{strictly-exact } Q \rrbracket \implies \text{strictly-exact } (P \wedge^* Q)$
by (*force intro!: strictly-exactI dest: sep-conjD strictly-exactD*)

lemma *strictly-exact-conj-impl*:
 $\llbracket (Q \wedge^* \text{sep-true})\ s; P\ s; \text{strictly-exact } Q \rrbracket \implies (Q \wedge^* (Q \longrightarrow^* P))\ s$
by (*force intro: sep-conjI sep-implI dest: strictly-exactD dest!: sep-conjD*)

lemma *dom-eps-sep-emp [simp]*:
 $\text{dom-eps } \square = \{\}$
apply (*subst dom-eps [symmetric]*)
apply (*rule strictly-exact-dom-exact*)
apply (*rule strictly-exact-sep-emp*)
apply (*rule sep-emp-empty*)
apply *simp*
done

lemma *dom-eps-sep-map*:
 $g\ p \implies \text{dom-eps } (p \mapsto_g (v::'a::\text{mem-type})) = \text{s-footprint } p$
apply (*subst dom-eps [symmetric]*)
apply (*rule dom-exact-sep-map*)
apply (*rule sep-map-singleton*)
apply (*erule ptr-retyp-h-t-valid*)
apply (*subst singleton-dom*)
apply (*erule ptr-retyp-h-t-valid*)
apply *simp*
done

definition *non-empty* :: ('a,'b) *map-assert* \Rightarrow *bool* **where**
 $\text{non-empty } P \equiv \exists s. P\ s$

lemma *non-emptyI*:
 $P\ s \implies \text{non-empty } P$
by (*force simp: non-empty-def*)


```

lemma non-emptyD:
  non-empty P  $\implies \exists s. P s$ 
  by (force simp: non-empty-def)

lemma non-empty-sep-true:
  non-empty sep-true
  by (simp add: non-empty-def)

lemma non-empty-sep-false:
   $\neg$  non-empty sep-false
  by (simp add: non-empty-def)

lemma non-empty-sep-emp:
  non-empty  $\square$ 
  unfolding non-empty-def by (rule exI, rule sep-emp-empty)

lemma non-empty-sep-map:
   $g p \implies \text{non-empty } (p \mapsto_g (v::'a::\text{mem-type}))$ 
  apply(unfold non-empty-def)
  apply(rule exI, rule sep-map-singleton)
  apply(erule ptr-retyp-h-t-valid)
  done

lemma non-empty-sep-conj:
  [non-empty P; non-empty Q; dom-exact P; dom-exact Q;
   dom-eps P  $\cap$  dom-eps Q =  $\{\}$ ]  $\implies \text{non-empty } (P \wedge^* Q)$ 
  apply(clarsimp simp: non-empty-def)
  subgoal for s s'
    apply(rule exI [where x=s+++s'])
    apply(rule sep-conjI, assumption+)
    apply(clarsimp simp: map-disj-def dom-eps)
    apply(subst map-add-com)
    apply(clarsimp simp: map-disj-def dom-eps)
    apply simp
  done
done

lemma non-empty-sep-map':
   $g p \implies \text{non-empty } (p \mapsto_g (v::'a::\text{mem-type}))$ 
  apply(unfold sep-map'-def)
  apply(clarsimp simp: non-empty-def sep-conj-ac)
  apply(rule exI [where x=singleton p v h (ptr-retyp p d)])
  apply(rule sep-conjI [where s0=Map.empty])
    apply simp
    apply(rule sep-map-singleton)
    apply(erule ptr-retyp-h-t-valid)
  apply(simp add: map-disj-def)
  apply simp
  done

```

lemma *non-empty-sep-impl*:

```
¬ P Map.empty ⇒ non-empty (P →* Q)
apply (clarsimp simp: non-empty-def)
apply (rule exI [where x = λs. Some undefined])
apply (rule sep-implI)
apply (clarsimp simp: map-disj-def)
done
```

lemma *pure-conj-right*: $(Q \wedge^* (\lambda s. P' \wedge Q' s)) = (\lambda s. P' \wedge (Q \wedge^* Q') s)$
by (*rule ext, standard, standard, clarsimp dest!: sep-conjD*)
(*erule sep-conj-impl, auto*)

lemma *pure-conj-right'*: $(Q \wedge^* (\lambda s. P' s \wedge Q')) = (\lambda s. Q' \wedge (Q \wedge^* P') s)$
by (*simp add: conj-comms pure-conj-right*)

lemma *pure-conj-left*: $((\lambda s. P' \wedge Q' s) \wedge^* Q) = (\lambda s. P' \wedge (Q' \wedge^* Q) s)$
by (*simp add: pure-conj-right sep-conj-ac*)

lemma *pure-conj-left'*: $((\lambda s. P' s \wedge Q') \wedge^* Q) = (\lambda s. Q' \wedge (P' \wedge^* Q) s)$
by (*subst conj-comms, subst pure-conj-left, simp*)

lemmas *pure-conj* = *pure-conj-right pure-conj-right' pure-conj-left pure-conj-left'*

declare *pure-conj* [*simp add*]

lemma *sep-conj-sep-conj-sep-impl-sep-conj*:

```
(P ∧* R) s ⇒ (P ∧* (Q →* (Q ∧* R))) s
by (erule (1) sep-conj-impl, erule sep-conj-sep-impl, simp add: sep-conj-ac)
```

lemma *sep-map'-conjE1-exc*:

```
[[ (P ∧* Q) s; ∧s. P s ⇒ (i ↦g v) s ]] ⇒ (i ↦g v) s
by (subst sep-map'-unfold-exc, erule sep-conj-impl, simp+)
```

lemma *sep-map'-conjE2-exc*:

```
[[ (P ∧* Q) s; ∧s. Q s ⇒ (i ↦g v) s ]] ⇒ (i ↦g v) s
by (subst (asm) sep-conj-com, erule sep-map'-conjE1-exc, simp)
```

lemma *sep-map'-any-conjE1-exc*:

```
[[ (P ∧* Q) s; ∧s. P s ⇒ (i ↦g -) s ]] ⇒ (i ↦g -) s
by (subst sep-map'-any-unfold-exc, erule sep-conj-impl, simp+)
```

lemma *sep-map'-any-conjE2-exc*:

```
[[ (P ∧* Q) s; ∧s. Q s ⇒ (i ↦g -) s ]] ⇒ (i ↦g -) s
by (subst (asm) sep-conj-com, erule sep-map'-any-conjE1-exc, simp)
```

lemma *sep-conj-mapD-exc*:

$((i \mapsto_g v) \wedge^* P) s \implies (i \hookrightarrow_g v) s \wedge ((i \mapsto_g -) \wedge^* P) s$
by (*force simp: sep-conj-ac intro: sep-conj-impl sep-map'-conjE2-exc*)

lemma *sep-impl-conj-sameD*:

$\llbracket (P \longrightarrow^* P \wedge^* Q) s; \text{dom-exact } P; \text{non-empty } P; \text{dom } s \subseteq \text{UNIV} - \text{dom-eps } P$
 \rrbracket

$\implies Q s$
apply(*drule sep-implD*)
apply(*clarsimp simp: non-empty-def*)
apply(*rename-tac s'*)
apply(*drule-tac x=s' in spec*)
apply(*erule impE*)
apply(*fastforce simp: map-disj-def dom-eps*)
apply(*drule sep-conjD, clarsimp*)
apply(*clarsimp simp: map-disj-def*)
apply(*subst (asm) map-add-comm*)
apply(*clarsimp simp: dom-eps*)
apply fast
apply(*subst (asm) map-add-comm, fast*)
apply(*drule map-disj-add-eq-dom-right-eq*)
apply(*simp add: dom-eps*)
apply(*clarsimp simp: dom-eps map-disj-def*)
apply fast
apply(*simp add: map-disj-def*)
apply clarsimp
done

lemma *sep-impl-conj-sameI*:

$Q s \implies (P \longrightarrow^* P \wedge^* Q) s$
by (*fastforce intro: sep-implI sep-conjI simp: map-disj-com*)

end

theory *SepCode*

imports

Separation-UMM

Simpl.Vcg

begin

definition

singleton-t :: *'a::c-type ptr* \Rightarrow *'a* \Rightarrow *heap-state*

where

singleton-t p v \equiv *lift-state (heap-update p v ($\lambda x. 0$), (ptr-retyp p empty-htd))*

definition

tagd :: *'a ptr-guard* \Rightarrow *'a::c-type ptr* \Rightarrow *heap-assert (infix \vdash_s 100)*

where

$$g \vdash_s p \equiv \lambda s. s, g \models_s p \wedge \text{dom } s = \text{s-footprint } p$$

definition

$$\text{field-footprint} :: 'a::\text{c-type ptr} \Rightarrow \text{qualified-field-name} \Rightarrow (\text{addr} \times \text{s-heap-index}) \text{ set}$$

where

$$\begin{aligned} \text{field-footprint } p f &\equiv \\ &\text{s-footprint-untyped } (\text{ptr-val } p + \text{of-nat } (\text{field-offset } \text{TYPE}('a) f)) \\ &\quad (\text{export-uinfo } (\text{field-ty } \text{TYPE}('a) f)) \end{aligned}$$

definition

$$\text{fs-footprint} :: 'a::\text{c-type ptr} \Rightarrow \text{qualified-field-name set} \Rightarrow (\text{addr} \times \text{s-heap-index}) \text{ set}$$

where

$$\text{fs-footprint } p F \equiv \bigcup \{ \text{field-footprint } p f \mid f. f \in F \}$$

definition $\text{fields} :: 'a::\text{c-type itself} \Rightarrow \text{qualified-field-name set}$ **where**

$$\text{fields } t \equiv \{ f. \text{field-lookup } (\text{typ-info-t } \text{TYPE}('a)) f 0 \neq \text{None} \}$$

definition

$$\begin{aligned} \text{mfs-sep-map} :: 'a::\text{c-type ptr} \Rightarrow 'a \text{ ptr-guard} \Rightarrow \text{qualified-field-name set} \Rightarrow 'a \Rightarrow \\ \text{heap-assert} \\ (- \mapsto - \text{ } - [56,0,0,51] 56) \end{aligned}$$

where

$$\begin{aligned} p \mapsto_g^F v &\equiv \lambda s. \text{lift-ty-heap } g (\text{singleton-t } p v ++ s) p = \text{Some } v \wedge \\ &F \subseteq \text{fields } \text{TYPE}('a) \wedge \\ &\text{dom } s = \text{s-footprint } p - \text{fs-footprint } p F \wedge \text{wf-heap-val } s \end{aligned}$$

notation (*input*)

$$\text{mfs-sep-map } (- \mapsto - \text{ } - [56,0,1000,51] 56)$$

definition

$$\text{disjoint-fn} :: \text{qualified-field-name} \Rightarrow \text{qualified-field-name set} \Rightarrow \text{bool}$$

where

$$\text{disjoint-fn } f F \equiv \forall f' \in F. \neg f \leq f' \wedge \neg f' \leq f$$

definition

$$\text{sep-cut}' :: \text{addr} \Rightarrow \text{nat} \Rightarrow (\text{s-addr}, 'b) \text{ map-assert}$$

where

$$\text{sep-cut}' p n \equiv \lambda s. \text{dom } s = \{ (x, y). x \in \{ p..+n \} \}$$

definition

$$\text{sep-cut} :: \text{addr} \Rightarrow \text{addr-bitsize word} \Rightarrow (\text{s-addr}, 'b) \text{ map-assert}$$

where

$$\text{sep-cut } x y \equiv \text{sep-cut}' x (\text{unat } y)$$

—

lemma *heap-list-h-eq*:

```

[[ x ∈ {p..+q}; q < addr-card; heap-list h q p = heap-list h' q p ]] ⇒ h x = h' x
proof (induct q arbitrary: p)
  case 0 thus ?case by simp
next
  case (Suc n) thus ?case by (force dest: intvl-neq-start)
qed

```

```

lemma s-footprint-intvl:
  (a, SIndexVal) ∈ s-footprint p = (a ∈ {ptr-val (p::'a::c-type ptr)..+size-of TYPE('a)})
apply(clarsimp simp: s-footprint-def s-footprint-untyped-def)
apply(rule iffI, clarsimp)
apply(rule intvlI)
apply(simp add: size-of-def)
apply(drule intvlD, clarsimp)
apply(simp add: size-of-def)
apply fast
done

```

```

lemma singleton-t-dom [simp]:
  dom (singleton-t p (v::'a::mem-type)) = s-footprint p
supply unsigned-of-nat [simp del]
apply(rule equalityI; clarsimp simp: singleton-t-def lift-state-def s-footprint-intvl
  split: s-heap-index.splits if-split-asm option.splits)
  apply(rule ccontr)
  apply(simp add: ptr-retyp-None)
subgoal for a b y x2 x2a
apply(cases a ∈ {ptr-val p..+size-of TYPE('a)})
apply(simp add: ptr-retyp-footprint list-map-eq split: if-split-asm)
apply(drule intvlD, clarsimp)
apply(rule s-footprintI)
apply(subst (asm) word-unat.eq-norm)
apply(subst (asm) mod-less)
  apply(subst len-of-addr-card)
  apply(erule less-trans)
  apply(rule max-size)
apply(simp add: map-le-def)
apply assumption
apply(simp add: ptr-retyp-None)
done
apply(rule conjI; clarsimp)
apply (simp add: ptr-retyp-d-empty s-footprintD)
apply(frule s-footprintD2)
apply(frule s-footprintD)
apply(simp add: ptr-retyp-footprint)
done

```

```

lemma heap-update-merge:
  assumes val: d, g ⊢t p
  shows lift-state ((heap-update p (v::'a::mem-type) h), d)

```

$= \text{lift-state } (h, d) ++ \text{singleton } p \ v \ h \ d \ (\text{is } ?x = ?y)$

proof (*rule ext, cases*)

fix x

assume $c: x \in \text{dom } (\text{singleton } p \ v \ h \ d)$

with *val*

have $\text{lift-state } (\text{heap-update-list } (\text{ptr-val } p)$

$(\text{to-bytes } v \ (\text{heap-list } h \ (\text{size-of } \text{TYPE}'(a)) \ (\text{ptr-val } p))) \ h,$

$d) \ x =$

$\text{singleton } p \ v \ h \ d \ x$

by (*auto simp: heap-list-update-to-bytes singleton-def lift-state-def heap-update-def singleton-dom*

$\text{split: option.splits s-heap-index.splits}$)

with c **show** $?x \ x = ?y \ x$ **by** (*force simp: heap-update-def dest: domD*)

next

fix x

assume $nc: x \notin \text{dom } (\text{singleton } p \ v \ h \ d)$

with *val show* $?x \ x = ?y \ x$

apply(*cases x*)

apply(*clarsimp simp: lift-state-def heap-update-def map-add-def*

$\text{split: option.splits s-heap-index.splits}$)

apply(*safe; clarsimp*)

by (*metis heap-list-length heap-update-nmem-same len nc s-footprint-intvl singleton-dom*)

qed

lemma *tagd-dom-exc:*

$(g \vdash_s p) \ s \implies \text{dom } s = \text{s-footprint } p$

by (*clarsimp simp: tagd-def*)

lemma *tagd-dom-p-exc:*

$(g \vdash_s p) \ s \implies (\text{ptr-val } (p::'a::\text{mem-type } ptr), SIndexVal) \in \text{dom } s$

by (*drule tagd-dom-exc clarsimp*)

lemma *tagd-g-exc:*

$(g \vdash_s p \wedge^* P) \ s \implies g \ p$

by (*drule sep-conjD, force simp: tagd-def elim: s-valid-g*)

lemma *sep-map-tagd-exc:*

$(p \mapsto_g (v::'a::\text{mem-type})) \ s \implies (g \vdash_s p) \ s$

by (*clarsimp simp: sep-map-def tagd-def lift-tyr-heap-s-valid*)

lemma *sep-map-any-tagd-exc:*

$(p \mapsto_g -) \ s \implies (g \vdash_s (p::'a::\text{mem-type } ptr)) \ s$

by (*clarsimp dest!: sep-map-anyD-exc, erule sep-map-tagd-exc*)

lemma *ptr-retyp-tagd-exc:*

$g \ (p::'a::\text{mem-type } ptr) \implies$

$(g \vdash_s p) \ (\text{lift-state } (h, \text{ptr-retyp } p \ \text{empty-htd}))$

```

supply unsigned-of-nat [simp del]
apply(simp add: tagd-def ptr-retyp-s-valid lift-state-dom)
apply(rule equalityI;
  clarsimp simp: lift-state-def split: s-heap-index.splits if-split-asm option.splits)
subgoal for a
  apply(cases a ∈ {ptr-val p..+size-of TYPE('a)})
  apply(drule intvlD, clarsimp)
  apply(rule s-footprintI2, simp)
  apply(subst (asm) ptr-retyp-None; simp)
done
subgoal for a b y x2 x2a
  apply(cases a ∈ {ptr-val p..+size-of TYPE('a)})
  apply(subst (asm) ptr-retyp-footprint)
  apply simp
  apply(drule intvlD, clarsimp)
  apply(subst (asm) word-unat.eq-norm)
  apply(subst (asm) mod-less)
  apply(subst len-of-addr-card)
  apply(erule less-trans)
  apply simp
  apply(subst (asm) list-map-eq)
  apply(clarsimp split: if-split-asm)
  apply(erule (1) s-footprintI)
  apply(simp add: ptr-retyp-None)
done
subgoal for a b
  apply(rule conjI; clarsimp)
  apply(cases a ∈ {ptr-val p..+size-of TYPE('a)})
  apply(simp add: ptr-retyp-footprint)
  apply(drule s-footprintD)
  apply simp
  apply(cases a ∈ {ptr-val p..+size-of TYPE('a)})
  apply(subst (asm) ptr-retyp-footprint)
  apply simp
  apply(drule intvlD, clarsimp)
  apply(subst (asm) word-unat.eq-norm)
  apply(subst (asm) mod-less)
  apply(subst len-of-addr-card)
  apply(erule less-trans)
  apply simp
  apply(subst (asm) list-map-eq)
  apply(clarsimp split: if-split-asm)
  apply(drule s-footprintD2)
  apply simp
  apply(subst (asm) unat-of-nat)
  apply(subst (asm) mod-less)
  apply(subst len-of-addr-card)
  apply(erule less-trans, simp)
  apply simp

```

```

  apply(fastforce dest: s-footprintD)
done
done

```

lemma *singleton-dom-proj-d* [*simp*]:
 $(g \vdash_s p) s \implies \text{dom} (\text{singleton } p (v::'a::\text{mem-type}) h (\text{proj-d } s)) = \text{dom } s$
by (*clarsimp simp: tagd-def singleton-dom s-valid-def*)

lemma *singleton-d-restrict-eq*:
 $\text{restrict-s } d (s\text{-footprint } p) = \text{restrict-s } d' (s\text{-footprint } p)$
 $\implies \text{singleton } p v h d = \text{singleton } p (v::'a::\text{mem-type}) h d'$
apply(*clarsimp simp: singleton-def*)
apply(*rule ext*)
subgoal for *x*
apply(*cases x ∈ s-footprint p; simp*)
apply(*cases x, clarsimp*)
subgoal for *a b*
apply(*drule fun-cong [where x=a]*)
apply(*clarsimp simp: s-footprint-restrict lift-state-def*
split: s-heap-index.splits if-split-asm option.splits)
apply(*rule conjI, clarsimp simp: restrict-s-def*)
apply(*clarsimp simp: restrict-s-def*)
subgoal for *x'*
apply(*drule fun-cong [where x=x']*)
apply *auto*
done
done
done
done

lemma *sep-heap-update'-exc*:
assumes *sep: (g ⊢_s p ∧* (p ↦_g v ⟶* P)) (lift-state (h,d))*
shows *P (lift-state (heap-update p (v::'a::mem-type) h,d))*
proof –
from *sep* **obtain** *s₀ s₁* **where** *disj: s₀ ⊥ s₁* **and**
merge: lift-state (h,d) = s₁ ++ s₀ **and**
l: (g ⊢_s p) s₀ **and** *r: (p ↦_g v ⟶* P) s₁* **by** (*force dest: sep-conjD*)
moreover from *this* **have** *s₁ ⊥ singleton p v h (proj-d s₀)*
by (*fastforce simp: map-disj-def*)
moreover from *l* **have** *g p* **by** (*force simp: tagd-def elim: s-valid-g*)
moreover from *merge l* **have** *lift-state (h,d),g ⊢_s p*
by (*force simp: tagd-def intro: s-valid-heap-merge-right*)
hence *d,g ⊢_t p* **by** (*simp add: h-t-s-valid*)
moreover from *l* **have** *s₀ ++ singleton p v h (proj-d s₀) = singleton p v h*
(proj-d s₀)
by (*force simp: map-add-dom-eq singleton-dom dest: tagd-dom-exc*)
moreover from *l merge* **have** *s₁ ++ singleton p v h (proj-d s₀) = s₁ ++ s₀ ++*
singleton p v h d
apply(*clarsimp simp: tagd-def*)


```

apply(rule ext)
subgoal for x
  apply(cases x, clarsimp simp: restrict-map-def)
  subgoal for a b
    apply(simp add: s-valid-def)
    apply(drule singleton-dom [where v=v and h=h])
    apply(drule fun-cong [where x=(a,b)])
    apply(cases (a,b) ∈ s-footprint p)
    subgoal premises prems
    proof –
      have (s1 ++ singleton p v h (proj-d s0)) (a, b) = singleton p v h d (a, b)
      using prems
      by(force simp: lift-state-def map-add-def singleton-def proj-d-def
        split: option.splits s-heap-index.splits if-split-asm)
      then show ?thesis using prems
      apply(clarsimp simp: map-add-def s-valid-def split: option.splits)
      apply force
      apply (fastforce intro: sym)
      done
    qed
  apply(auto simp: map-add-def singleton-def split: option.splits)
  done
done
done
ultimately show ?thesis
  by (fastforce dest: sep-implD simp: sep-map-singleton tagd-def s-valid-def heap-update-merge)
qed

```

```

lemma sep-heap-update-exc:
   $\llbracket (p \mapsto_g - \wedge^* (p \mapsto_g v \longrightarrow^* P)) (\text{lift-state } (h,d)) \rrbracket \implies$ 
   $P (\text{lift-state } (\text{heap-update } p (v::'a::\text{mem-type}) h,d))$ 
  by (force intro: sep-heap-update'-exc dest: sep-map-anyD-exc sep-map-tagd-exc
    elim: sep-conj-impl)

```

```

lemma sep-heap-update-global'-exc:
   $(g \vdash_s p \wedge^* R) (\text{lift-state } (h,d)) \implies$ 
   $((p \mapsto_g v) \wedge^* R) (\text{lift-state } (\text{heap-update } p (v::'a::\text{mem-type}) h,d))$ 
  by (rule sep-heap-update'-exc, erule sep-conj-sep-conj-sep-impl-sep-conj)

```

```

lemma sep-heap-update-global-exc:
   $(p \mapsto_g - \wedge^* R) (\text{lift-state } (h,d)) \implies$ 
   $((p \mapsto_g v) \wedge^* R) (\text{lift-state } (\text{heap-update } p (v::'a::\text{mem-type}) h,d))$ 
  by (fastforce intro: sep-heap-update-global'-exc sep-conj-impl sep-map-any-tagd-exc)

```

```

lemma sep-heap-update-global-exc2:
   $(p \mapsto_g u \wedge^* R) (\text{lift-state } (h,d)) \implies$ 
   $((p \mapsto_g v) \wedge^* R) (\text{lift-state } (\text{heap-update } p (v::'a::\text{mem-type}) h,d))$ 
  by (fastforce intro: sep-heap-update-global-exc simp: sep-map-any-def sep-conj-exists)

```

lemma *heap-update-mem-same-point*:
 $\llbracket q \in \{p..+length\ v\}; length\ v < addr_card \rrbracket \implies$
 $heap_update_list\ p\ v\ h\ q = v ! unat\ (q - p)$
apply(*induct* *v arbitrary: p h; clarsimp*)
subgoal for *x1 v p h*
apply(*cases p=q*)
apply (*simp add: heap-update-list-same [where k=1, simplified]*)
apply(*drule meta-spec [where x=p+1]*)
apply(*drule meta-spec, drule meta-mp*)
apply(*fastforce dest: intvl-neq-start*)
by (*smt (verit, ccfv-SIG) Suc-lessD add-diff-cancel-left' diff-add-cancel diff-diff-eq*
 $diff_zero\ heap_update_mem_same\ intvl_neq_start\ nth_Cons'\ plus-1-eq-Suc$
 $unat-eq-zero$
 $unat-minus-one$)
done

lemma *heap-update-list-value*:
 $length\ v < addr_card \implies$
 $heap_update_list\ p\ v\ h\ q = (if\ q \in \{p..+length\ v\}\ then\ v!unat\ (q-p)\ else\ h\ q)$
by (*auto simp: heap-update-nmem-same heap-update-mem-same-point*
split: if-split)

lemma *heap-update-list-value'*:
 $length\ xs < addr_card \implies$
 $heap_update_list\ ptr\ xs\ hp\ x = (if\ unat\ (x - ptr) < length\ xs\ then\ xs ! unat\ (x - ptr)\ else\ hp\ x)$
supply *unsigned-of-nat [simp del]*
apply (*simp only: heap-update-list-value addr-card-def card-word*)
apply (*rule if-cong; simp*)
apply (*rule iffI*)
apply (*drule intvlD, clarsimp simp add: unat-of-nat*)
apply (*simp add: intvl-def unat-arith-simps(4) unat-of-nat split: if-split-asm*)
apply (*rule exI [where x=unat x - unat ptr], simp*)
apply (*rule exI [where x=unat x + 2^addr-bitsize - unat ptr]*)
using *unat-lt2p [of ptr]*
apply (*simp add: unat-arith-simps unat-of-nat*)
done

lemma *heap-list-h-eq2*:
 $(\bigwedge x. x \in \{p..+n\} \implies h\ x = h'\ x) \implies heap_list\ h\ n\ p = heap_list\ h'\ n\ p$
apply(*induct n arbitrary: p; clarsimp*)
apply(*rule conjI*)
apply(*fastforce intro: intvl-self*)
apply(*fastforce intro: intvl-plus-sub-Suc*)
done

lemma *map-td-f-eq'*:
 $(f=g) \longrightarrow (map_td\ f\ h\ t = map_td\ g\ h\ t)$

$(f=g) \longrightarrow (\text{map-td-struct } f \ h \ st = \text{map-td-struct } g \ h \ st)$
 $(f=g) \longrightarrow (\text{map-td-list } f \ h \ ts = \text{map-td-list } g \ h \ ts)$
 $(f=g) \longrightarrow (\text{map-td-tuple } f \ h \ x = \text{map-td-tuple } g \ h \ x)$
by (*induct t and st and ts and x*) *auto*

lemma *map-td-f-eq*:
 $f=g \implies \text{map-td } f \ t = \text{map-td } g \ t$
by (*erule arg-cong*)

lemma *sep-map'-lift-exc*:
 $(p \hookrightarrow_g (v::'a::\text{mem-type})) (\text{lift-state } (h,d)) \implies \text{lift } h \ p = v$
by (*frule sep-map'-lift-typ-heapD, simp add: lift-t lift-t-lift*)

lemma *sep-map-lift-wp-exc*:
 $\exists v. (p \mapsto_g v \wedge^* (p \mapsto_g v \longrightarrow^* P \ v)) (\text{lift-state } (h,d))$
 $\implies P (\text{lift } h \ (p::'a::\text{mem-type ptr})) (\text{lift-state } (h,d))$
apply *clarsimp*
apply (*subst sep-map'-lift-exc*)
apply (*fastforce simp: sep-map'-def elim: sep-conj-impl*)
subgoal for *v*
apply (*rule sep-conj-impl-same [where P=p \mapsto_g v and Q=P v]*)
apply (*erule (2) sep-conj-impl*)
done
done

lemma *sep-map-lift-exc*:
 $((p::'a::\text{mem-type ptr}) \mapsto_g -) (\text{lift-state } (h,d)) \implies$
 $(p \mapsto_g \text{lift } h \ p) (\text{lift-state } (h,d))$
by (*clarsimp simp: sep-map-any-def*)
(frule sep-map-sep-map'-exc, drule sep-map'-lift-exc, simp)

lemma *sep-map'-lift-rev-exc*:
 $\llbracket \text{lift } h \ p = (v::'a::\text{mem-type}); (p \hookrightarrow_g -) (\text{lift-state } (h,d)) \rrbracket \implies$
 $(p \hookrightarrow_g v) (\text{lift-state } (h,d))$
by (*clarsimp simp: sep-map'-any-def*)
(frule sep-map'-lift-exc, simp)

lemma *sep-lift-exists-exc*:
fixes $p :: 'a::\text{mem-type ptr}$
assumes $ex: ((\lambda s. \exists v. (p \hookrightarrow_g v) \ s \wedge P \ v \ s) \wedge^* Q) (\text{lift-state } (h,d))$
shows $(P (\text{lift } h \ p) \wedge^* Q) (\text{lift-state } (h,d))$
proof –
from ex **obtain** v **where** $((\lambda s. (p \hookrightarrow_g v) \ s \wedge P \ v \ s) \wedge^* Q)$
 $(\text{lift-state } (h,d))$
by (*subst (asm) sep-conj-exists, clarsimp*)
thus *?thesis*
by (*force simp: sep-map'-lift-exc sep-conj-ac*
 $dest: \text{sep-map'-conjE2-exc } dest!: \text{sep-conj-conj}$)

qed

lemma *merge-dom*:

$x \in \text{dom } s \implies (t ++ s) x = s x$
by (*force simp: map-add-def*)

lemma *merge-dom2*:

$x \notin \text{dom } s \implies (t ++ s) x = t x$
by (*force simp: map-add-def split: option.splits*)

lemma *fs-footprint-empty* [*simp*]:

$\text{fs-footprint } p \ \{\} = \{\}$
by (*auto simp: fs-footprint-def*)

lemma *fs-footprint-un*:

$\text{fs-footprint } p \ (\text{insert } f \ F) = \text{fs-footprint } p \ \{f\} \cup \text{fs-footprint } p \ F$
by (*auto simp: fs-footprint-def*)

lemma *proj-d-restrict-map-le*:

$\text{snd } (\text{proj-d } (s \mid' X) x) \subseteq_m \text{snd } (\text{proj-d } s x)$
by (*clarsimp simp: map-le-def proj-d-def restrict-map-def split: option.splits if-split-asm*)

lemma *SIndexVal-conj-setcomp-simp* [*simp*]:

$\{x. \text{snd } x = \text{SIndexVal} \wedge x \notin \text{s-footprint-untyped } p \ t\}$
 $= \{(x, \text{SIndexVal}) \mid x. x \notin \{p..+ \text{size-td } t\}\}$
by (*force dest: intvlD intro: intvlI simp: s-footprint-untyped-def*)

lemma *heap-list-s-restrict-same*:

$\{(x, \text{SIndexVal}) \mid x. x \in \{p..+n\}\} \subseteq X \implies \text{heap-list-s } (s \mid' X) \ n \ p = \text{heap-list-s}$
 $s \ n \ p$
apply (*induct n arbitrary: p;clarsimp simp: heap-list-s-def*)
apply (*rule conjI*)
apply (*fastforce intro: intvl-self simp: proj-h-def restrict-map-def*)
apply (*fastforce intro: intvl-plus-sub-Suc*)
done

lemma *heap-list-s-restrict-fs-footprint*:

$\text{field-lookup } (\text{typ-info-t } \text{TYPE}('a::\text{c-type})) \ f \ 0 = \text{Some } (t, n) \implies$
 $\text{heap-list-s } (s \mid' \text{fs-footprint } p \ \{f\}) \ (\text{size-td } t) \ \&(p \rightarrow f)$
 $= \text{heap-list-s } s \ (\text{size-td } t) \ \&((p::'a \ \text{ptr}) \rightarrow f)$
apply (*simp add: fs-footprint-def field-footprint-def field-offset-def*)
apply (*subst heap-list-s-restrict-same*)
apply (*fastforce simp: s-footprint-untyped-def field-size-def field-lvalue-def field-offset-def field-ti-def field-typ-def field-typ-untyped-def dest: intvlD*)
apply *simp*
done

lemma *heap-list-proj-h-disj*:

```

 $\{(x, SIndexVal) \mid x. x \in \{p..+n\}\} \cap \text{dom } s_1 = \{\} \implies$ 
  heap-list (proj-h (s_0 ++ s_1)) n p = heap-list (proj-h s_0) n p
apply(induct n arbitrary: p; clarsimp)
apply(rule conjI)
  apply(fastforce simp: proj-h-def intro: intvl-self split: option.splits)
apply(fastforce intro: intvl-plus-sub-Suc)
done

```

lemma heap-list-proj-h-sub:

```

 $\{(x, SIndexVal) \mid x. x \in \{p..+n\}\} \subseteq \text{dom } s_1 \implies$ 
  heap-list (proj-h (s_0 ++ s_1)) n p = heap-list (proj-h s_1) n p
apply(induct n arbitrary: p; clarsimp)
apply(rule conjI, fastforce simp: proj-h-def intro: intvl-self split: option.splits)
subgoal premises prems for n p
proof -
  have  $\{p + 1..+n\} \subseteq \{p..+Suc\ n\}$  using prems by (fastforce intro: intvl-plus-sub-Suc)
  then show ?thesis using prems by fast
qed
done

```

lemma heap-list-s-map-add-super-update-bs:

```

 $\llbracket \{x. (x, SIndexVal) \in \text{dom } s_1\} = \{p+\text{of-nat } k..+z\}; k + z \leq n; n < \text{addr-card} \rrbracket$ 
 $\implies$  heap-list-s (s_0 ++ s_1) n p =
  super-update-bs (heap-list-s s_1 z (p+of-nat k)) (heap-list-s s_0 n p) k
apply(clarsimp simp: super-update-bs-def heap-list-s-def)
subgoal premises prems
proof -
  have heap-list (proj-h (s_0 ++ s_1)) (k + z + (n - (k+z))) p =
    take k (heap-list (proj-h s_0) n p) @
    heap-list (proj-h s_1) z (p + of-nat k) @
    drop (k + z) (heap-list (proj-h s_0) n p)
  using prems
  apply(subst heap-list-split2)
  apply(subst heap-list-split2)
  apply simp
  apply(rule conjI)
  apply(subst take-heap-list-le, simp)
  apply(subst heap-list-proj-h-disj)
  using init-intvl-disj [of k z p]
  apply fastforce
  apply simp
  apply(rule conjI)
  apply(subst heap-list-proj-h-sub; fast)
  apply(simp add: drop-heap-list-le)
  apply(subst heap-list-proj-h-disj)
  using final-intvl-disj [of k z n p]
  apply fast

```

```

    apply simp
  done
  then show ?thesis using prems by simp
qed
done

```

lemma *s-footprint-untyped-dom-SIndexVal*:
 $dom\ s = s\text{-footprint-untyped}\ p\ t \implies$
 $\{x. (x, SIndexVal) \in dom\ s\} = \{p..+size\text{-td}\ t\}$
by (*auto simp: s-footprint-untyped-def intro: intvlI dest: intvlD*)

lemma *field-ti-s-sub*:
 $field\text{-lookup}\ (export\text{-uinfo}\ (typ\text{-info}\text{-t}\ TYPE('b::mem\text{-type})))\ f\ 0 = Some\ (a, b) \implies$
 $s\text{-footprint-untyped}\ \&(p \rightarrow f)\ a \subseteq s\text{-footprint}\ (p::'b\ ptr)$
apply (*clarsimp simp: field-ti-def s-footprint-def s-footprint-untyped-def split: option.splits*)
apply (*simp add: field-lvalue-def field-offset-def typ-uinfo-t-def*)
subgoal for $x\ k$
apply (*rule exI [where x=b+x]*)
apply *simp*
apply (*simp add: field-offset-untyped-def*)
apply (*drule td-set-field-lookupD*)
apply (*frule td-set-offset-size*)
apply (*drule typ-slice-td-set [where k=x]*)
apply *simp*
apply (*auto simp: prefix-def less-eq-list-def*)
done
done

lemma *wf-heap-val-map-add [simp]*:
 $\llbracket wf\text{-heap-val}\ s_0; wf\text{-heap-val}\ s_1 \rrbracket \implies wf\text{-heap-val}\ (s_0\ ++\ s_1)$
unfolding *wf-heap-val-def by auto*

lemma *of-nat-lt-size-of*:
 $\llbracket (of\text{-nat}\ x::addr) = of\text{-nat}\ y + of\text{-nat}\ z; x < size\text{-of}\ TYPE('a::mem\text{-type});$
 $y + z < size\text{-of}\ TYPE('a) \rrbracket \implies x = y+z$
by (*metis (mono-tags) len-of-addr-card less-trans max-size mod-less of-nat-add unat-of-nat*)

lemma *proj-d-map-add*:
 $snd\ (proj\text{-d}\ s_1\ p)\ n = Some\ k \implies snd\ (proj\text{-d}\ (s_0\ ++\ s_1)\ p)\ n = Some\ k$
by (*auto simp: proj-d-def split: option.splits*)

lemma *proj-d-map-add2*:
 $fst\ (proj\text{-d}\ s_1\ p) \implies fst\ (proj\text{-d}\ (s_0\ ++\ s_1)\ p)$
by (*auto simp: proj-d-def split: option.splits*)

lemma *heap-list-s-restrict-disj-same*:
 $dom\ s \cap (UNIV - X) = \{\} \implies heap\text{-list}\text{-s}\ (s\ |'\ X)\ n\ p = heap\text{-list}\text{-s}\ s\ n\ p$

apply(*induct n arbitrary: p; clarsimp simp: heap-list-s-def*)
apply(*fastforce simp: proj-h-def restrict-map-def split: option.splits*)
done

lemma *UNIV-minus-inter*:
 $(X - Y) \cap (X \cap (X - Y) - Z) = X - (Y \cup Z)$
by *fast*

lemma *sep-map-mfs-sep-map-empty*:
 $(p \mapsto_g (v::'a::mem-type)) = (p \mapsto_g (\{\}) v)$
by (*auto simp: sep-map-def mfs-sep-map-def map-add-dom-eq*)

lemma *fd-cons-double-update*:
 $\llbracket \text{fd-cons } t; \text{length } bs = \text{length } bs' \rrbracket \implies$
 $\text{update-ti-t } t \text{ } bs \text{ } (\text{update-ti-t } t \text{ } bs' \text{ } v) = \text{update-ti-t } t \text{ } bs \text{ } v$
by (*simp add: fd-cons-def Let-def fd-cons-double-update-def fd-cons-desc-def*)

lemma *fd-cons-update-access*:
 $\llbracket \text{fd-cons } t; \text{length } bs = \text{size-td } t \rrbracket \implies$
 $\text{update-ti-t } t \text{ } (\text{access-ti } t \text{ } v \text{ } bs) \text{ } v = v$
by (*simp add: fd-cons-def Let-def fd-cons-update-access-def fd-cons-desc-def*)

lemma *fd-cons-length*:
 $\llbracket \text{fd-cons } t; \text{length } bs = \text{size-td } t \rrbracket \implies$
 $\text{length } (\text{access-ti } t \text{ } v \text{ } bs) = \text{size-td } t$
by (*simp add: fd-cons-def Let-def fd-cons-desc-def fd-cons-length-def access-ti_0-def*)

lemma *fd-cons-length-p*:
 $\text{fd-cons } t \implies \text{length } (\text{access-ti}_0 \text{ } t \text{ } v) = \text{size-td } t$
by (*simp add: fd-cons-length access-ti_0-def*)

lemma *fd-cons-update-normalise*:
 $\llbracket \text{fd-cons } t; \text{length } bs = \text{size-td } t \rrbracket \implies$
 $\text{update-ti-t } t \text{ } ((\text{norm-desc } (\text{field-desc } t) \text{ } (\text{size-td } t)) \text{ } bs) \text{ } v = \text{update-ti-t } t \text{ } bs \text{ } v$
by (*fastforce simp: fd-cons-def Let-def fd-cons-desc-def fd-cons-update-normalise-def*
dest: fd-cons-update-normalise)

lemma *field-footprint-SIndexVal*:
 $\text{field-lookup } (\text{typ-info-t } \text{TYPE}('a::c-type)) \text{ } f \text{ } 0 = \text{Some } (t, n) \implies$
 $\{x. (x, \text{SIndexVal}) \in \text{field-footprint } (p::'a \text{ ptr}) \text{ } f\} =$
 $\{\text{ptr-val } p + \text{of-nat } n..+\text{size-td } t\}$
by (*auto simp: field-footprint-def s-footprint-untyped-def field-typ-def field-typ-untyped-def*
intro: intvlI dest: intvlD)

lemma *fs-footprint-subset*:
 $F \subseteq \text{fields } \text{TYPE}('a::mem-type) \implies \text{fs-footprint } (p::'a \text{ ptr}) \text{ } F \subseteq \text{s-footprint } p$
unfolding *fs-footprint-def field-footprint-def*
apply(*clarsimp simp: fields-def*)
apply(*drule (1) subsetD, clarsimp*)

```

apply(frule field-lookup-export-uinfo-Some)
apply(drule field-ti-s-sub)
apply(unfold field-lvalue-def)[1]
apply(subst (asm) field-lookup-offset-eq, assumption)
apply(fastforce simp: field-typ-def field-typ-untyped-def)
done

```

```

lemma length-heap-list-s [simp]:
  length (heap-list-s s n p) = n
by (clarsimp simp: heap-list-s-def)

```

```

lemma heap-list-proj-h-restrict:
  {p..+n}  $\subseteq$  {x. (x, SIndexVal)  $\in$  X}  $\implies$  heap-list (proj-h (s |' X)) n p = heap-list
  (proj-h s) n p
apply(induct n arbitrary: p; clarsimp)
apply(rule conjI)
apply(fastforce simp: proj-h-restrict intro: intvl-self)
apply(fastforce intro: intvl-plus-sub-Suc)
done

```

```

lemma heap-list-proj-h-lift-state:
  {p..+n}  $\subseteq$  {x. fst (d x)}  $\implies$  heap-list (proj-h (lift-state (h,d))) n p = heap-list h
  n p
by (fastforce intro: heap-list-h-eq2 simp: proj-h-lift-state)

```

```

lemma heap-list-rpbs:
  heap-list ( $\lambda$ x. 0) n p = replicate n 0
by (induct n arbitrary: p) auto

```

```

lemma field-access-take-drop:
fixes
  t::('a,'b) typ-info and
  st::('a,'b) typ-info-struct and
  ts::('a,'b) typ-info-tuple list and
  x::('a,'b) typ-info-tuple
shows
 $\forall$  s m n f. field-lookup t f m = Some (s,n)  $\longrightarrow$  wf-fd t  $\longrightarrow$ 
  take (size-td s) (drop (n - m) (access-ti0 t v)) =
  access-ti0 s v
 $\forall$  s m n f. field-lookup-struct st f m = Some (s,n)  $\longrightarrow$  wf-fd-struct st  $\longrightarrow$ 
  take (size-td s) (drop (n - m) (access-ti-struct st v (replicate (size-td-struct
  st) 0))) =
  access-ti0 s v
 $\forall$  s m n f. field-lookup-list ts f m = Some (s,n)  $\longrightarrow$  wf-fd-list ts  $\longrightarrow$ 
  take (size-td s) (drop (n - m) (access-ti-list ts v (replicate (size-td-list ts) 0)))
=
  access-ti0 s v
 $\forall$  s m n f. field-lookup-tuple x f m = Some (s,n)  $\longrightarrow$  wf-fd-tuple x  $\longrightarrow$ 
  take (size-td s) (drop (n - m) (access-ti-tuple x v (replicate (size-td-tuple x)

```



```

0))) =
  access-ti0 s v
  apply(induct t and st and ts and x, all ⟨clarsimp simp: access-ti0-def⟩)
  apply(fastforce dest: wf-fd-cons-structD
    simp: fd-cons-struct-def fd-cons-desc-def fd-cons-length-def)
  apply(drule wf-fd-cons-tupleD)
  apply(clarsimp simp: fd-cons-tuple-def fd-cons-desc-def fd-cons-length-def)
  apply(clarsimp split: option.splits)
  apply(subst drop-all)
  apply(fastforce dest: field-lookup-offset-le simp: fd-cons-length-def split-DTuple-all)
  apply simp
  apply (metis (no-types, opaque-lifting) add commute add-le-imp-le-diff diff-is-0-eq'

    diff-zero field-lookup-offset-le(2) fl2 size-td-tuple-dt-fst)
  apply (metis (no-types, lifting) Nat.diff-diff-right add-leD2 append.right-neutral

    diff-is-0-eq length-0-conv length-take nat-min-simps(1)
    td-set-offset-size' td-set-tuple-field-lookup-tupleD zero-le)
  apply fastforce
  done

```

lemma *field-access-take-dropD*:

```

[[ field-lookup t f 0 = Some (s,n); wf-lf (lf-set t []); wf-desc t ]] ==>
  take (size-td s) (drop n (access-ti0 t v)) = access-ti0 s v
  using field-access-take-drop(1) [of t v] by (fastforce dest: wf-fdp-fdD wf-lf-fdp)

```

lemma *singleton-t-field*:

```

field-lookup (typ-info-t TYPE('a::mem-type)) f 0 = Some (t, n) ==>
  heap-list-s (singleton-t p v |' fs-footprint p {f}) (size-td t) (ptr-val p + of-nat
n) =
  access-ti0 t v
  apply(clarsimp simp: heap-list-s-def singleton-def singleton-t-def)
  apply(subst heap-list-proj-h-restrict)
  apply(fastforce simp: fields-def fs-footprint-def
    intro!: fs-footprint-subset
    dest: field-footprint-SIndexVal)
  apply(subst heap-list-proj-h-lift-state)
  apply(frule field-tag-sub[where p=p] )
  apply(fastforce simp: field-lvalue-def dest: ptr-retyp-footprint[where d=empty-htd])
  apply(clarsimp simp: access-ti0-def heap-update-def)
  apply(subst heap-list-update-list; simp?)
  apply(simp add: size-of-def)
  apply(erule field-lookup-offset-size)
  apply(fastforce simp: access-ti0-def to-bytes-def heap-list-rpbs size-of-def
    dest: field-access-take-dropD
    elim: field-lookup-offset-size)
  done

```

lemma *field-lookup-fd-consD*:

field-lookup (*typ-info-t* *TYPE('a::mem-type)*) *f* 0 = *Some* (*t,n*) \implies *fd-cons* *t*
by (*erule* *fd-consistentD*) *simp*

lemma *s-valid-map-add*:

$\llbracket s, g \models_s p; t, g' \models_s p \rrbracket \implies (s ++ t \mid' X), g \models_s p$
by (*clarsimp* *simp*: *map-le-def* *s-valid-def* *h-t-valid-def* *valid-footprint-def* *Let-def*
proj-d-map-add-snd *proj-d-restrict-map-snd* *proj-d-map-add-fst*
proj-d-restrict-map-fst)

lemma *singleton-t-s-valid*:

$g \models p \implies \text{singleton-t } p (v::'a::\text{mem-type}), g \models_s p$
by (*fastforce* *simp*: *singleton-t-def* *h-t-s-valid* *elim*: *ptr-retyp-h-t-valid*)

lemma *sep-map-mfs-sep-map*:

$\llbracket (p \mapsto_g^F v) s; \text{field-lookup } (\text{typ-info-t } \text{TYPE}'a) f 0 = \text{Some } (t,n) \rrbracket \implies$
 $(p \mapsto_g (\{f\} \cup F) (v::'a::\text{mem-type})) (s \mid' (\text{dom } s - \text{fs-footprint } p \{f\}))$

apply(*clarsimp* *simp*: *mfs-sep-map-def*)

apply(*rule* *conjI*)

prefer 2

apply(*fastforce* *simp*: *fs-footprint-un*[**where** *F=F*] *fields-def*)

apply(*clarsimp* *simp*: *lift-ty-heap-if* *split*: *if-split-asm*)

apply(*rule* *conjI*, *clarsimp*)

subgoal **premises** *prems*

proof –

have (*singleton-t* *p* *v* ++

$s \mid' (s\text{-footprint } p - \text{fs-footprint } p F - \text{fs-footprint } p \{f\})) =$
 $(\text{singleton-t } p \ v \ ++ \ s) ++ (\text{singleton-t } p \ v \ \mid' \ \text{fs-footprint } p \ \{f\})$

using *prems*

apply(*simp* *add*: *map-add-restrict-sub*)

done

then show *?thesis*

using *prems*

apply *clarsimp*

apply(*subst* *heap-list-s-map-add-super-update-bs*)

apply *clarsimp*

apply(*subgoal-tac* *fs-footprint* *p* $\{f\} \subseteq s\text{-footprint } p$)

apply(*subgoal-tac* $\{x. (x, S\text{IndexVal}) \in \text{fs-footprint } p \{f\}\} =$
 $\{\text{ptr-val } p + \text{of-nat } n..+\text{size-td } t\}$)

apply *fast*

apply(*clarsimp* *simp*: *fs-footprint-def*)

apply(*erule* *field-footprint-SIndexVal*)

apply(*rule* *fs-footprint-subset*)

apply(*clarsimp* *simp*: *fields-def*)

apply(*clarsimp* *simp*: *size-of-def*)

apply(*subst* *ac-simps*)

apply(*rule* *td-set-offset-size*)

apply(*erule* *td-set-field-lookupD*)

apply *simp*

apply(*clarsimp* *simp*: *from-bytes-def*)

```

apply(frule-tac v=heap-list-s (singleton-t p v | ' fs-footprint p {f}) (size-td t)
      (ptr-val p + of-nat n) and
      bs=heap-list-s (singleton-t p v ++ s) (size-of TYPE('a)) (ptr-val p) and
      w=undefined in fi-fu-consistentD; simp)
apply(simp add: size-of-def)
apply(simp add: singleton-t-field)
apply(fastforce simp: access-ti0-def fd-cons-update-access dest!: field-lookup-fd-consD)
done
qed
subgoal premises prems
proof –
  have (singleton-t p v ++
        s | ' (s-footprint p – fs-footprint p F – fs-footprint p {f})) =
        (singleton-t p v ++ s) ++ (singleton-t p v | ' fs-footprint p {f})
  using prems by (simp add: map-add-restrict-sub)
  then show ?thesis using prems
  by(fastforce intro: s-valid-map-add singleton-t-s-valid ptr-retyp-h-t-valid)
qed
done

```

```

lemma disjoint-fn-disjoint:
  [ disjoint-fn f F; F ⊆ fields TYPE('a::mem-type);
    field-lookup (typ-info-t TYPE('a)) f 0 = Some (t,n) ] ⇒
    fs-footprint (p::'a ptr) F ∩ field-footprint p f = {}
  apply(clarsimp simp: fs-footprint-def field-footprint-def s-footprint-untyped-def
field-typ-def
      field-typ-untyped-def fields-def)
  apply safe
  apply (drule (1) subsetD, clarsimp,
          drule (1) fa-fu-lookup-disj-interD;
          force intro: intvlI simp: max-size[unfolded size-of-def] disj-fn-def dis-
joint-fn-def)
  apply (drule (1) subsetD, clarsimp,
          drule (1) fa-fu-lookup-disj-interD;
          force intro: intvlI simp: max-size[unfolded size-of-def] disj-fn-def dis-
joint-fn-def)
  done

```

```

lemma sep-map-mfs-sep-map2:
  [ field-lookup (typ-info-t TYPE('a::mem-type)) f 0 = Some (s, n);
    disjoint-fn f F; guard-mono g g';
    export-uinfo s = typ-uinfo-t TYPE('b); ((p::'a ptr) ↦gF v) x]
  ⇒ (Ptr &( p→f) ) ↦g' ((from-bytes (access-ti0 s v))::'b::mem-type)
    (x | ' field-footprint p f)
  apply(clarsimp simp: mfs-sep-map-def sep-map-def)
  apply(rule conjI)
  subgoal premises prems
  proof –
    have field-footprint p f = s-footprint ((Ptr &( p→f ))::'b ptr)

```

```

    using prems
  by (clarsimp simp: field-footprint-def s-footprint-def field-lvalue-def typ-uinfo-t-def
      field-typ-def field-typ-untyped-def)
  then show ?thesis
    using prems
    apply simp
    apply (erule lift-typ-heap-mono, assumption+)
    apply (clarsimp simp: lift-typ-heap-if split: if-split-asm)
    apply (rule conjI, fastforce simp: heap-list-s-heap-merge-right [where p=&(p→f)])
    apply (erule s-valid-heap-merge-right2)
    apply simp
    apply (erule (2) disjoint-fn-disjoint [where p=p])
    apply (subgoal-tac fs-footprint p {f} ⊆ s-footprint p)
      apply (fastforce simp: fs-footprint-def)
    apply (rule fs-footprint-subset)
    apply (fastforce simp: fields-def)
    done
qed
apply (clarsimp simp: field-footprint-def field-lvalue-def s-footprint-def field-typ-def
      field-typ-untyped-def)
subgoal premises prems
proof -
  have {f} ⊆ fields TYPE('a)
    using prems by (clarsimp simp: fields-def)
  then show ?thesis
    using prems
    apply -
    apply (drule fs-footprint-subset [where F={f} and p=p])
    apply (subst (asm) (2) fs-footprint-def)
    apply (clarsimp simp: field-footprint-def s-footprint-def field-typ-def field-typ-untyped-def)
    apply (subgoal-tac fs-footprint p F ∩
      s-footprint-untyped (ptr-val p + of-nat n) (typ-uinfo-t TYPE('b)))
  = {}
    apply blast
    apply (drule disjoint-fn-disjoint [where p=p], assumption+)
    apply (simp add: field-footprint-def field-typ-def field-typ-untyped-def)
    done
qed
done

```

lemma *export-size-of*:

$export-uinfo\ t = typ-uinfo-t\ TYPE('a) \implies size-of\ TYPE('a::c-type) = size-td\ t$
by (fastforce simp: size-of-def simp flip: typ-uinfo-size dest: sym)

lemma *sep-map-field-unfold*:

$\llbracket field-lookup\ (typ-info-t\ TYPE('a))\ f\ 0 = Some\ (t,n);$
 $disjoint-fn\ f\ F; guard-mono\ g\ g';$
 $export-uinfo\ t = typ-uinfo-t\ TYPE('b) \rrbracket \implies$

```

    (p ↦gF v) = (p ↦g({f}∪F) (v::'a::mem-type) ∧*
      Ptr (&(p→f)) ↦g' ((from-bytes (access-ti0 t v))::'b::mem-type))
supply unsigned-of-nat [simp del]
apply(rule ext)
apply(rule iffI)
subgoal for x
  apply(rule sep-conjI [where s0=x |' (dom x - fs-footprint p {f}) and
    s1=x |' fs-footprint p {f}])
    apply(erule (1) sep-map-mfs-sep-map)
    apply(clarsimp simp: fs-footprint-def)
    apply(erule (4) sep-map-mfs-sep-map2)
    apply(clarsimp simp: map-disj-def)
    apply fast
    applyclarsimp
  done

apply(drule sep-conjD,clarsimp)
apply(clarsimp simp: mfs-sep-map-def sep-map-def)
apply(rule conjI)
apply(subst map-ac-simps)
apply(subst map-add-com, solves ‹simp add: map-ac-simps›)
apply(subst map-add-assoc)
apply(clarsimp simp: lift-tyr-heap-if split: if-split-asm)
apply(rule conjI,clarsimp)
apply(subst heap-list-s-map-add-super-update-bs)
  apply(subst s-footprint-untyped-dom-SIndex Val)
  apply(fastforce simp: s-footprint-def)
  apply(fastforce simp: field-lvalue-def)
apply(drule field-lookup-offset-size)
apply(drule export-size-of)
apply(simp add: size-of-def)
apply simp
apply(clarsimp simp: from-bytes-def)
subgoal for s0 s1
  apply(frule fi-fu-consistentD [where v=heap-list-s s1 (size-td (typ-info-t TYPE('b)))
    (ptr-val p + of-nat n) and
    bs=heap-list-s (singleton-t p v ++ s0) (size-of TYPE('a)) (ptr-val p) and
    w=undefined]; simp)
  apply(simp add: size-of-def)
  apply(drule export-size-of, simp add: size-of-def)
apply(subst fd-cons-update-normalise [symmetric])
  apply(erule field-lookup-fd-consD)
apply simp
apply(drule export-size-of, simp add: size-of-def)
apply(simp add: norm-desc-def)
apply(drule arg-cong [where f=access-ti0 (typ-info-t TYPE('b))])
apply(drule arg-cong [where f=λbs. update-ti-t t bs v])
apply(subst (asm) wf-fd-norm-tuD [symmetric]; simp?)
apply(simp add: size-of-def)

```

```

apply(subst (asm) wf-fd-norm-tuD [symmetric])
  apply simp
  apply(subst fd-cons-length-p)
  apply(erule field-lookup-fd-consD)
  apply(drule export-size-of, simp add: size-of-def)
apply(subgoal-tac export-uinfo (typ-info-t TYPE('b)) = typ-uinfo-t TYPE('b))
  prefer 2
  apply(simp add: typ-uinfo-t-def)
apply simp
apply(drule sym, simp)
apply(subst (asm) wf-fd-norm-tuD)
  apply(erule wf-fd-field-lookupD, simp)
  apply simp
  apply(drule sym, drule export-size-of)
  apply(simp add: size-of-def)
apply(simp add: access-ti0-def)
apply(clarsimp simp: field-lvalue-def)
apply(simp add: size-of-def)
apply(subst wf-fd-norm-tuD)
  apply(erule wf-fd-field-lookupD, simp)
  apply(subst fd-cons-length; simp?)
  apply(erule field-lookup-fd-consD)
apply(subgoal-tac update-ti-t t (norm-desc (field-desc t) (size-td t)
  (access-ti t v (replicate (size-td t) 0))) v = v)
  apply(simp add: norm-desc-def access-ti0-def)
apply(subst fd-cons-update-normalise)
  apply(erule field-lookup-fd-consD)
  apply(subst fd-cons-length; simp?)
  apply(erule field-lookup-fd-consD)
apply(subst fd-cons-update-access; simp?)
apply(erule field-lookup-fd-consD)
done

prefer 2
apply(subst fs-footprint-un)
apply(subst fs-footprint-def)
apply(clarsimp simp: field-footprint-def s-footprint-def field-lvalue-def field-typ-def)
apply(drule disjoint-fn-disjoint [where p=p]; assumption?)
apply(clarsimp simp: field-footprint-def s-footprint-def field-lvalue-def field-typ-def)
apply(subgoal-tac {f} ⊆ fields TYPE('a))
  apply(drule fs-footprint-subset[where F={f} and p = p])
  apply(clarsimp simp: s-footprint-def)
  apply(fastforce simp: fs-footprint-def field-footprint-def s-footprint-def
  field-typ-def field-typ-untyped-def field-lvalue-def)
apply(clarsimp simp: fields-def)
apply(clarsimp simp: s-valid-def h-t-valid-def valid-footprint-def Let-def)

subgoal for s0 s1 y
  apply(standard, clarsimp simp: map-le-def)

```

```

subgoal for a
  apply(subst proj-d-map-add-snd[where s=a ++ b for a b])
  apply(clarsimp split: if-split-asm)
  apply(frule s-footprintD2)
  apply(drule s-footprintD)
  apply(drule spec [where x=y])
  apply clarsimp
  apply(drule bspec [where x=a])
  apply clarsimp
  apply(drule intvlD, clarsimp simp: field-lvalue-def)
subgoal for k
  apply(drule spec [where x=k])
  apply(clarsimp simp add: size-of-def)
  apply(drule bspec [where x=a])
  apply clarsimp
  apply(subst (asm) unat-of-nat)
  apply(subst (asm) mod-less)
  apply(subst len-of-addr-card)
  apply(erule less-trans)
  apply(simp add: max-size[unfolded size-of-def])
  apply simp
  apply(simp add: ac-simps)
  apply(rotate-tac -1)
  apply(drule sym)
  apply simp
  apply(drule sym[where s=Some s for s])
  apply simp
  apply(drule field-lookup-export-uinfo-Some)
  apply(drule td-set-field-lookupD)
  apply(frule typ-slice-td-set [where k=k])
  apply simp
  apply simp
  apply(simp add: typ-uinfo-t-def)
  apply(subgoal-tac y=n+k)
  apply(simp add: strict-prefix-def)
  apply(subst (asm) unat-of-nat)
  apply(subst (asm) mod-less)
  apply(subst len-of-addr-card)
  apply(erule less-trans)
  apply(simp add: max-size[unfolded size-of-def])
  apply (clarsimp simp: prefix-eq-nth)
  apply(drule arg-cong [where f=unat])
  apply(rotate-tac -1)
  apply(subst (asm) unat-of-nat)
  apply(subst (asm) mod-less)
  apply(subst len-of-addr-card)
  apply(erule less-trans)
  apply(simp add: max-size[unfolded size-of-def])
  apply(subst (asm) Abs-fnat-hom-add)

```

```

apply(subst (asm) unat-of-nat)
apply(subst (asm) mod-less)
apply(subst len-of-addr-card)
apply(drule td-set-offset-size)
apply(rule le-less-trans [where y=size-td (typ-info-t TYPE('a))] , simp)
apply(simp add: max-size[unfolded size-of-def])
apply simp
done
done

```

```

apply(subst proj-d-map-add-fst)
apply(fastforce simp: size-of-def field-lvalue-def ac-simps
  dest: intvID s-footprintD)
done
done

```

lemma disjoint-fn-empty [simp]:
 disjoint-fn f {}
by (simp add: disjoint-fn-def)

lemma sep-map-field-map':
 [[(p::'a::mem-type ptr) ↦_g v);
 field-lookup (typ-info-t TYPE('a)) f 0 = Some (d,n); export-uinfo d = typ-uinfo-t
 TYPE('b);
 guard-mono g g'] ⇒
 ((Ptr (&(p→f))::'b::mem-type ptr) ↦_{g'} from-bytes (access-ti₀ d v)) s
by (fastforce dest: sep-map-g elim: sep-conj-impl
 simp: sep-map-mfs-sep-map-empty sep-map-field-unfold sep-map'-def
 sep-conj-ac)

lemma fd-cons-access-update-p:
 [[fd-cons t; length bs = size-td t] ⇒
 access-ti₀ t (update-ti-t t bs v) = access-ti₀ t (update-ti-t t bs w)
by (simp add: fd-cons-def Let-def fd-cons-access-update-def fd-cons-desc-def ac-
 cess-ti₀-def)

lemma length-to-bytes-p [simp]:
 length (to-bytes-p (v::'a)) = size-of TYPE('a::mem-type)
by (simp add: to-bytes-p-def)

lemma inv-p [simp]:
 from-bytes (to-bytes-p v) = (v::'a::mem-type)
by (simp add: to-bytes-p-def)

lemma singleton-SIndexVal:
 x ∈ {ptr-val p..+size-of TYPE('a)} ⇒
 singleton-t p (v::'a::mem-type) (x,SIndexVal) = Some (SValue (to-bytes-p v !
 unat (x - ptr-val p)))
by (clarsimp simp: singleton-def singleton-t-def lift-state-def heap-update-def

heap-update-mem-same-point to-bytes-p-def heap-list-rpbs

ptr-retyp-d-eq-fst)

lemma *access-ti₀*:

access-ti s v (replicate (size-td s) 0) = access-ti₀ s v
by (*simp add: access-ti₀-def*)

lemma *fd-cons-mem-type* [*simp*]:

fd-cons (typ-info-t TYPE('a::mem-type))
by (*rule wf-fd-consD simp*)

lemma *norm-tu-rpbs*:

wf-fd t \implies norm-tu (export-uinfo t) (access-ti₀ t v) = access-ti₀ t v
apply(*frule wf-fd-consD*)
apply(*simp add: wf-fd-norm-tuD fd-cons-length-p*)
apply(*subst fd-cons-access-update-p [where w=v]; (simp add: fd-cons-length-p)?*)
apply(*fastforce simp: access-ti₀-def fd-cons-update-access*)
done

lemma *heap-list-s-singleton-t-field-update*:

\llbracket *field-lookup (typ-info-t TYPE('a::mem-type)) f 0 = Some (s, n);*
export-uinfo s = typ-uinfo-t TYPE('b) $\rrbracket \implies$
heap-list-s (singleton-t p (update-ti-t s (to-bytes-p w) v)) (size-td s)
(ptr-val (p::'a::mem-type ptr) + of-nat n) =
to-bytes-p (w::'b::mem-type)
apply(*clarsimp simp: singleton-t-def singleton-def*)
apply(*subst heap-list-s-heap-list-dom*)
apply *clarsimp*
apply(*frule field-tag-sub [where p=p]*)
apply(*clarsimp simp: field-lvalue-def*)
apply(*drule (1) subsetD*)
apply(*drule intvlD [where n=size-of TYPE('a)], clarsimp*)
apply(*erule s-footprintI2*)
apply(*simp add: heap-update-def*)
apply(*subst heap-list-update-list; simp?*)
apply(*drule field-lookup-offset-size*)
apply(*simp add: size-of-def*)
apply(*frule field-access-take-dropD [where v=(update-ti-t s (to-bytes-p w) v)];*
simp?)
apply(*simp add: access-ti₀-def to-bytes-def heap-list-rpbs size-of-def to-bytes-p-def*)
apply(*simp add: access-ti₀*)
apply(*subst fd-cons-access-update-p [where w=undefined]*)
apply(*erule field-lookup-fd-consD*)
apply(*subst fd-cons-length-p*)
apply *simp*
apply(*drule export-size-of, simp add: size-of-def*)
apply(*subst wf-fd-norm-tuD [symmetric]*)
apply(*erule wf-fd-field-lookupD*)
apply *simp*

```

apply(fastforce simp: fd-cons-length-p size-of-def dest: export-size-of)
apply(simp add: typ-uinfo-t-def norm-tu-rpbs)
done

```

lemma *field-access-update-nth-disj*:

fixes

```

t::('a,'b) typ-info and
st::('a,'b) typ-info-struct and
ts::('a,'b) typ-info-tuple list and
y::('a,'b) typ-info-tuple

```

shows

```

 $\forall m f s n x bs bs'. \text{field-lookup } t f m = \text{Some } (s,n) \longrightarrow x < \text{size-td } t \longrightarrow$ 
 $(x < n - m \vee x \geq (n - m) + \text{size-td } s) \longrightarrow \text{wf-fd } t \longrightarrow \text{length } bs = \text{size-td}$ 
 $s \longrightarrow \text{length } bs' = \text{size-td } t \longrightarrow$ 
 $\text{access-ti } t (\text{update-ti-t } s bs v) bs' ! x$ 
 $= \text{access-ti } t v bs' ! x$ 

 $\forall m f s n x bs bs'. \text{field-lookup-struct } st f m = \text{Some } (s,n) \longrightarrow x < \text{size-td-struct}$ 
 $st \longrightarrow$ 
 $(x < n - m \vee x \geq (n - m) + \text{size-td } s) \longrightarrow \text{wf-fd-struct } st \longrightarrow \text{length } bs =$ 
 $\text{size-td } s \longrightarrow \text{length } bs' = \text{size-td-struct } st \longrightarrow$ 
 $\text{access-ti-struct } st (\text{update-ti-t } s bs v) bs' ! x$ 
 $= \text{access-ti-struct } st v bs' ! x$ 

 $\forall m f s n x bs bs'. \text{field-lookup-list } ts f m = \text{Some } (s,n) \longrightarrow x < \text{size-td-list } ts \longrightarrow$ 
 $(x < n - m \vee x \geq (n - m) + \text{size-td } s) \longrightarrow \text{wf-fd-list } ts \longrightarrow \text{length } bs =$ 
 $\text{size-td } s \longrightarrow \text{length } bs' = \text{size-td-list } ts \longrightarrow$ 
 $\text{access-ti-list } ts (\text{update-ti-t } s bs v) bs' ! x$ 
 $= \text{access-ti-list } ts v bs' ! x$ 

 $\forall m f s n x bs bs'. \text{field-lookup-tuple } y f m = \text{Some } (s,n) \longrightarrow x < \text{size-td-tuple } y$ 
 $\longrightarrow$ 
 $(x < n - m \vee x \geq (n - m) + \text{size-td } s) \longrightarrow \text{wf-fd-tuple } y \longrightarrow \text{length } bs =$ 
 $\text{size-td } s \longrightarrow \text{length } bs' = \text{size-td-tuple } y \longrightarrow$ 
 $\text{access-ti-tuple } y (\text{update-ti-t } s bs v) bs' ! x$ 
 $= \text{access-ti-tuple } y v bs' ! x$ 

```

proof(*induct t and st and ts and y*)

case (*TypDesc nat typ-struct list*)

then show ?*case by auto*

next

case (*TypScalar nat1 nat2 a*)

then show ?*case by auto*

next

case (*TypAggregate list*)

then show ?*case by auto*

next

case *Nil-typ-desc*

then show ?*case by auto*

next

case (*Cons-typ-desc dt-tuple list*)

then show ?*case*

apply *clarify*

```

subgoal for  $m f s n x bs bs'$ 
  apply(clarsimp split: if-split-asm)
  apply(clarsimp split: option.splits)

  apply(rotate-tac -3)
  apply(drule spec [where x=m + size-td (dtfst dt-tuple)])
  apply(drule spec [where x=f])
  apply(drule spec [where x=s])
  apply(rotate-tac -1)
  apply(drule spec [where x=n])

  apply clarsimp
  apply(rotate-tac -1)
  apply(drule-tac spec [where x=x - size-td-tuple dt-tuple])
  apply(frule field-lookup-fa-fu-rhs-listD)
    apply simp
    apply assumption
  apply(clarsimp simp: fa-fu-ind-def)
  apply(subgoal-tac access-ti-tuple dt-tuple (update-ti-t s bs v) (take (size-td-tuple
dt-tuple) bs') =
    access-ti-tuple dt-tuple v (take (size-td-tuple dt-tuple) bs'))
    prefer 2
    apply (fastforce simp: min-def)
  apply(clarsimp simp: nth-append)
  apply(subgoal-tac length
    (access-ti-tuple dt-tuple v
      (take (size-td-tuple dt-tuple) bs')) = size-td-tuple dt-tuple)
  apply simp
  prefer 2
  apply(drule wf-fd-cons-tupleD)
  apply(clarsimp simp: fd-cons-tuple-def fd-cons-desc-def fd-cons-length-def)
  apply(erule impE)
  apply simp
  apply(cases dt-tuple, simp+)
subgoal for  $a b c$ 

  apply(drule spec [where x=bs])
  apply(drule spec [where x=drop (size-td a) bs'])
  apply clarsimp
  apply(frule field-lookup-offset-le)
  apply clarsimp
  apply(drule td-set-list-field-lookup-listD)
  apply(drule td-set-list-offset-size-m)
  apply clarsimp
  apply(erule disjE)
  apply arith
  apply arith
  done
  apply(frule field-lookup-fa-fu-rhs-tupleD, simp)

```

```

    apply assumption
  apply (clarsimp simp: fa-fu-ind-def)
  apply (subgoal-tac length (access-ti-tuple dt-tuple (update-ti-t s bs v)
    (take (size-td-tuple dt-tuple) bs'))) = size-td-tuple dt-tuple)
  apply (subgoal-tac length (access-ti-tuple dt-tuple v
    (take (size-td-tuple dt-tuple) bs'))) = size-td-tuple dt-tuple)
  apply (clarsimp simp: nth-append)
  apply (drule spec [where x=m])
  apply (drule spec [where x=f])
  apply (drule spec [where x=s])
  apply (drule spec [where x=n])
  apply clarsimp
  apply (drule spec [where x=x])
  apply clarsimp
  apply (drule spec [where x=bs])
  apply (drule spec [where x=take (size-td-tuple dt-tuple) bs'])
  apply (clarsimp simp: min-def split: if-split-asm)
  apply (drule wf-fd-cons-tupleD)
  apply (clarsimp simp: fd-cons-tuple-def fd-cons-desc-def fd-cons-length-def)
  apply (drule wf-fd-cons-tupleD)
  apply (clarsimp simp: fd-cons-tuple-def fd-cons-desc-def fd-cons-length-def)
  done
done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by auto
qed

```

lemma *field-access-update-nth-disjD*:

```

[[ field-lookup t f m = Some (s,n); x < size-td t;
  (x < n - m ∨ x ≥ (n - m) + size-td s); wf-fd t;
  length bs = size-td s; length bs' = size-td t ]] ⇒
  access-ti t (update-ti-t s bs v) bs' ! x
  = access-ti t v bs' ! x

```

by (*simp add: field-access-update-nth-disj*)

lemma *intvl-cut*:

```

[[ (x::addr) ∈ {p..+m}; x ∉ {p+of-nat k..+n}; m < addr-card ]] ⇒
  unat (x - p) < k ∨ k + n ≤ unat (x - p)

```

supply *unsigned-of-nat [simp del]*

apply (*drule intvlD, clarsimp*)

apply (*subst unat-of-nat, subst mod-less, subst len-of-addr-card*)

apply (*erule (1) less-trans*)

apply (*subst (asm) unat-of-nat, subst (asm) mod-less, subst len-of-addr-card*)

apply (*erule (1) less-trans*)

apply (*rule ccontr*)

apply (*subgoal-tac ∃ z. ka = k + z*)

apply (*force simp flip: add.assoc intro: intvlI*)

apply *arith*
done

lemma *singleton-t-mask-out*:

$\llbracket \text{field-lookup } (typ\text{-info-t } TYPE('a::mem\text{-type})) f 0 = \text{Some } (s,n);$
 $\text{export-uinfo } s = \text{typ-uinfo-t } TYPE('b);$
 $K = (UNIV - s\text{-footprint-untyped } \&(p \rightarrow f) (\text{export-uinfo } s)) \rrbracket \implies$
 $\text{singleton-t } p (\text{update-ti-t } s (\text{to-bytes-p } (w::'b::mem\text{-type})) (v::'a)) \mid 'K =$
 $\text{singleton-t } p v \mid 'K$

supply *unsigned-of-nat* [*simp del*]

supply *max-size*[*unfolded size-of-def, simp*]

apply(*rule ext*)

apply(*clarsimp simp: restrict-map-def singleton-t-def singleton-def lift-state-def*
heap-update-def to-bytes-def access-ti₀ heap-list-rpbs size-of-def
split: s-heap-index.splits)

apply(*subst heap-update-mem-same-point*)

apply(*fastforce simp: fd-cons-length-p ptr-retyp-None size-of-def intro: ccontr*)

apply(*simp add: fd-cons-length-p*)

apply(*subst heap-update-mem-same-point*)

apply(*fastforce simp: fd-cons-length-p ptr-retyp-None size-of-def intro: ccontr*)

apply(*simp add: fd-cons-length-p*)

apply(*simp add: access-ti₀-def*)

apply(*rule field-access-update-nth-disjD; simp?*)

apply(*subst (asm) ptr-retyp-d-eq-fst*)

apply(*clarsimp simp: empty-htd-def split: if-split-asm*)

apply(*drule intvlD, clarsimp*)

apply(*subst unat-of-nat*)

apply(*subst mod-less*)

apply(*subst len-of-addr-card*)

apply(*erule less-trans*)

apply *simp*

apply(*simp add: size-of-def*)

apply(*clarsimp simp: empty-htd-def ptr-retyp-d-eq-fst split: if-split-asm*)

apply(*drule-tac k=of-nat n and n=size-td s in intvl-cut; simp?*)

apply(*fastforce dest: intvlD export-size-of*

simp: size-of-def s-footprint-untyped-def field-lvalue-def)

apply(*drule export-size-of, simp add: size-of-def*)

done

lemma *singleton-t-SIndexTyp*:

$\text{singleton-t } p v (x, SIndexTyp n) = \text{singleton-t } p \text{ undefined } (x, SIndexTyp n)$

by (*auto simp: singleton-t-def singleton-def restrict-map-def lift-state-def*)

lemma *proj-d-singleton-t*:

$\text{proj-d } (\text{singleton-t } p (v::'a::mem\text{-type}) ++ x) = \text{proj-d } (\text{singleton-t } p \text{ undefined}$
 $++ x)$

apply(*rule ext*)

apply(*clarsimp simp: proj-d-def*)

```

apply(safe)
  apply(subgoal-tac dom (singleton-t p undefined) = dom (singleton-t p v), blast,
simp)+
  apply(rule ext)
  apply(clarsimp split: option.splits)
  apply(safe; clarsimp?)
  apply(subgoal-tac dom (singleton-t p undefined) = dom (singleton-t p v), blast,
simp)+
  apply(subst (asm) singleton-t-SIndexTyp)
  apply simp
  done

```

lemma from-bytes-heap-list-s-update:

```

[[ field-lookup (typ-info-t TYPE('a::mem-type)) f 0 = Some (s, n);
  export-uinfo s = typ-uinfo-t TYPE('b::mem-type);
  dom x = s-footprint p - fs-footprint p F; f ∈ F ]] ==>
  from-bytes (heap-list-s (singleton-t p (update-ti-t s (to-bytes-p (w::'b)) (v::'a))
++ x)
              (size-of TYPE('a)) (ptr-val p)) =
  update-ti-t s (to-bytes-p w)
  (from-bytes (heap-list-s (singleton-t p v ++ x) (size-of TYPE('a))
(ptr-val p)))
  apply(subst map-add-restrict-UNIV [where X=s-footprint-untyped (&(p→f))
(export-uinfo s) and
              h=singleton-t p v])
  apply(force simp: fs-footprint-def field-footprint-def field-lvalue-def
              field-typ-def field-typ-untyped-def )
  apply simp
  apply(subst heap-list-s-map-add-super-update-bs [where k=n and z=size-td s];
simp?)
  apply(rule equalityI)
  apply(fastforce dest: s-footprintD export-size-of intro: intvI
              simp: s-footprint-untyped-def field-lvalue-def size-of-def)
  apply clarsimp
  apply(rule conjI)
  apply(frule field-tag-sub)
  apply(clarsimp simp: field-lvalue-def)
  apply(drule (1) subsetD)
  apply(fastforce elim: s-footprintI2 dest: intvD)
  apply(fastforce dest: intvD export-size-of
              simp: size-of-def s-footprint-untyped-def field-lvalue-def)
  apply(fastforce dest: field-lookup-offset-size simp: size-of-def)
  apply(clarsimp simp: from-bytes-def)
  apply(frule-tac v=heap-list-s (singleton-t p (update-ti-t s (to-bytes-p w) v) |'
              s-footprint-untyped &(p→f) (export-uinfo s))
              (size-td s) (ptr-val p + of-nat n) and
              bs=heap-list-s (singleton-t p (update-ti-t s (to-bytes-p w) v) |'
              (UNIV - s-footprint-untyped &(p→f) (export-uinfo
s)) ++

```

```

      singleton-t p v |'
      s-footprint-untyped &(p→f) (typ-uinfo-t TYPE('b))
++ x)
      (size-of TYPE('a)) (ptr-val p) and
      w=undefined in fi-fu-consistentD; simp add: size-of-def)
apply(subst heap-list-s-restrict)
apply(fastforce dest: intvD export-size-of
      simp: size-of-def field-lvalue-def s-footprint-untyped-def)
apply(simp add: heap-list-s-singleton-t-field-update)
apply(subst singleton-t-mask-out; assumption?)
apply simp
apply(subst map-add-restrict-comp-left)
apply simp
done

```

lemma *mfs-sep-map-field-update*:

```

[[ field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n); f ∈ F;
  export-uinfo s = typ-uinfo-t TYPE('b) ]] ⇒
  (p ↦gF update-ti-t s (to-bytes-p (w::'b::mem-type)) v) =
  (p ↦gF update-ti-t s (to-bytes-p (u::'b::mem-type)) (v::'a::mem-type))
apply(rule ext)
apply(clarsimp simp: mfs-sep-map-def lift-typ-heap-if s-valid-def)
apply safe
  apply(simp add: from-bytes-heap-list-s-update)
  apply(drule-tac f=update-ti-t s (to-bytes-p u) in arg-cong)
  apply(simp add: fd-cons-double-update field-lookup-fd-consD)
  apply(simp add: from-bytes-heap-list-s-update)
  apply(drule-tac f=update-ti-t s (to-bytes-p w) in arg-cong)
  apply(simp add: fd-cons-double-update field-lookup-fd-consD)
  apply(subst (asm) proj-d-singleton-t)
  apply(subst (asm) proj-d-singleton-t[where v=update-ti-t s (to-bytes-p u) v])
  apply simp
  apply(subst (asm) proj-d-singleton-t)
  apply(subst (asm) proj-d-singleton-t[where v=update-ti-t s (to-bytes-p u) v])
  apply simp
done

```

lemma *mfs-sep-map-field-update-v*:

```

[[field-lookup (typ-info-t TYPE('a)) f 0 = Some (t, n); f ∈ F;
  disjoint-fn f (F - {f}); guard-mono g g';
  export-uinfo t = typ-uinfo-t TYPE('b) ]] ⇒
  p ↦gF update-ti-t t (to-bytes-p (w::'b::mem-type)) (v::'a::mem-type) = p ↦gF v
apply(subst mfs-sep-map-field-update [where u=from-bytes (access-ti0 t v)];
simp?)
apply(simp add: to-bytes-p-def to-bytes-def from-bytes-def access-ti0 size-of-def)
apply(subst wf-fd-norm-tuD [symmetric], simp)
apply(fastforce dest: fd-cons-length-p export-size-of field-lookup-fd-consD simp:
size-of-def)
apply(rotate-tac -1)

```

```

apply(drule sym)
apply(simp add: typ-uinfo-t-def)
apply(subgoal-tac norm-tu (typ-uinfo-t TYPE('b)) = norm-bytes TYPE('b))
prefer 2
apply(simp add: norm-bytes-def typ-uinfo-t-def)
apply(clarsimp simp: norm-bytes-def
      wf-fd-norm-tuD wf-fd-field-lookupD fd-cons-length-p field-lookup-fd-consD)
apply(subst fd-cons-access-update-p [where w=v])
apply(erule field-lookup-fd-consD)
apply(simp add: fd-cons-length-p field-lookup-fd-consD)
apply(simp add: access-ti0-def fd-cons-update-access field-lookup-fd-consD)
done

```

lemma *sep-map-field-fold*:

```

[[ field-lookup (typ-info-t TYPE('a)) f 0 = Some (t,n);
   f ∈ F; disjoint-fn f (F - {f}); guard-mono g g';
   export-uinfo t = typ-uinfo-t TYPE('b) ]]  $\implies$ 
(p  $\mapsto_g^F$  (v::'a::mem-type)  $\wedge^*$ 
  Ptr &( p  $\rightarrow$  f)  $\mapsto_{g'}$  (w::'b::mem-type))
= p  $\mapsto_g$  (F - {f}) (update-ti-t t (to-bytes-p w) v)
apply(simp add: sep-map-field-unfold)
apply(subst fd-cons-access-update-p [where w=undefined])
apply(erule field-lookup-fd-consD)
apply(fastforce dest: export-size-of simp: size-of-def)
apply(subst wf-fd-norm-tuD [symmetric])
apply(simp add: wf-fd-field-lookupD)
apply(fastforce dest: export-size-of simp: size-of-def)
apply(subgoal-tac norm-tu (typ-uinfo-t TYPE('b)) = norm-bytes TYPE('b))
apply(simp add: sep-conj-ac norm mfs-sep-map-field-update-v insert-absorb)
apply(simp add: norm-bytes-def typ-uinfo-t-def)
done

```

lemma *norm-bytes*:

```

length bs = size-of TYPE('a)  $\implies$ 
to-bytes-p ((from-bytes bs)::'a) = norm-bytes TYPE('a::mem-type) bs
by (simp add: norm-bytes-def wf-fd-norm-tuD size-of-def to-bytes-p-def from-bytes-def
      to-bytes-def access-ti0-def)

```

lemma *sep-heap-update-global-super-fl*:

```

[[ (p  $\mapsto_g$  u  $\wedge^*$  R) (lift-state (h,d));
   field-lookup (typ-info-t TYPE('b::mem-type)) f 0 = Some (t,n);
   export-uinfo t = (typ-uinfo-t TYPE('a)) ]]  $\implies$ 
(p  $\mapsto_g$  update-ti-t t (to-bytes-p v) u)  $\wedge^*$  R
(lift-state (heap-update (Ptr &(p  $\rightarrow$  f)) (v::'a::mem-type) h,d))
apply(subst sep-map-mfs-sep-map-empty)
apply(simp add: sep-map-field-unfold [where g'=λx. True] guard-mono-def)
apply(subst fd-cons-access-update-p [where w=undefined])
apply(erule field-lookup-fd-consD)
apply simp

```



```

apply(simp add: export-size-of)
apply(subst wf-fd-norm-tuD [symmetric])
  apply(erule wf-fd-field-lookupD, simp)
  apply(simp add: export-size-of)
apply(subgoal-tac norm-tu (typ-uinfo-t TYPE('a)) = norm-bytes TYPE('a))
prefer 2
  apply(simp add: norm-bytes-def typ-uinfo-t-def)
apply(simp add: norm-sep-conj-ac)
apply(subst sep-conj-com)
apply(simp add: sep-conj-assoc)
apply(rule sep-heap-update-global-exc2 [where u=from-bytes (access-ti0 t u)])
apply(simp add: sep-conj-ac)
apply(subst sep-conj-com)
apply(simp add: sep-map-field-fold-guard-mono-def)
apply(subst sep-map-mfs-sep-map-empty [symmetric])
apply(simp add: fd-cons-double-update field-lookup-fd-consD)
apply(simp add: norm-bytes fd-cons-length-p field-lookup-fd-consD export-size-of)
apply(simp add: norm-bytes-def typ-uinfo-t-def)
apply(rotate-tac -1)
apply(drule sym)
apply(simp add: wf-fd-norm-tuD wf-fd-field-lookupD fd-cons-length-p field-lookup-fd-consD)
apply(subst fd-cons-access-update-p [where w=u])
  apply(erule field-lookup-fd-consD)
  apply(simp add: fd-cons-length-p field-lookup-fd-consD)
apply(simp add: access-ti0-def fd-cons-update-access field-lookup-fd-consD sep-conj-com)
done

```

lemma *sep-cut'-dom*:
 $sep\text{-cut}'\ x\ y\ s \implies dom\ s = \{(a,b). a \in \{x..+y\}\}$
by (simp add: sep-cut'-def)

lemma *dom-exact-sep-cut'*:
 $dom\text{-exact}\ (sep\text{-cut}'\ x\ y)$
by (force intro!: dom-exactI dest!: sep-cut'-dom)

lemma *dom-lift-state-dom-s* [simp]:
 $dom\ (lift\text{-state}\ (h,d)) = dom\text{-s}\ d$
by (force simp: lift-state-def dom-s-def split: s-heap-index.splits if-split-asm option.splits)

lemma *dom-ptr-retyp-empty-htd* [simp]:
 $dom\ (lift\text{-state}\ (h,ptr\text{-retyp}\ (p::'a::mem\text{-type}\ ptr)\ empty\text{-htd})) = s\text{-footprint}\ p$
by simp

lemma *ptr-retyp-sep-cut'-exc*:
fixes $p::'a::mem\text{-type}\ ptr$
assumes $sc: (sep\text{-cut}'\ (ptr\text{-val}\ p)\ (size\text{-of}\ TYPE('a)) \wedge^* P)\ (lift\text{-state}\ (h,d))$ **and**
 $g\ p$
shows $(g \vdash_s p \wedge^* sep\text{-true} \wedge^* P)\ (lift\text{-state}\ (h,(ptr\text{-retyp}\ p\ d)))$

proof –
from *sc*
obtain s_0 **and** s_1 **where** $s_0 \perp s_1$ **and** $\text{lift-state } (h, d) = s_1 ++ s_0$ **and**
 $P \ s_1$ **and** $d: \text{dom } s_0 = \{(a, b). a \in \{\text{ptr-val } p..+\text{size-of } \text{TYPE}'(a)\}\}$
by (*fast dest: sep-conjD sep-cut'-dom*)
moreover from *this*
have $\text{lift-state } (h, \text{ptr-retyp } p \ d) = s_1 ++ \text{lift-state } (h, \text{ptr-retyp } p \ d) \mid' \text{dom } s_0$
apply –
apply(*rule ext*)
subgoal for x
apply (*cases* $x \in \text{dom } s_0$)
apply(*cases* $x \in \text{dom } s_1$)
apply(*fastforce simp: map-disj-def*)
apply(*subst map-add-com*)
apply(*fastforce simp: map-disj-def*)
apply(*clarsimp simp: map-add-def split: option.splits*)
apply(*cases* x , *clarsimp*)
apply(*clarsimp simp: lift-state-ptr-retyp-d merge-dom2*)
done
done
moreover have $g \ p$ **by** *fact*
with d **have** ($g \vdash_s p \wedge^* \text{sep-true}$) ($\text{lift-state } (h, \text{ptr-retyp } p \ d) \mid' \text{dom } s_0$)
apply(*clarsimp simp: lift-state-ptr-retyp-restrict sep-conj-ac intro: ptr-retyp-tagd-exc*)
apply(*rule sep-conjI* [**where** $s_0 = \text{lift-state } (h, d) \mid' \{(a, b). a \in \{\text{ptr-val } p..+\text{size-of } \text{TYPE}'(a)\}\}$ – *s-footprint* p])
apply (*simp add: sep-conj-ac*)
apply(*erule ptr-retyp-tagd-exc* [**where** $h=h$])
apply(*fastforce simp: map-disj-def*)
apply(*subst map-add-com*[**where** $h_0 = \text{lift-state } (h, \text{ptr-retyp } p \ \text{empty-htd})$])
apply (*simp add: map-disj-def*)
apply *fast*
apply(*rule ext*)
apply(*clarsimp simp: map-add-def split: option.splits*)
by (*metis* (*mono-tags*) *Diff-iff dom-ptr-retyp-empty-htd non-dom-eval-eq re-strict-in-dom restrict-out*)
ultimately
show *?thesis*
by (*subst sep-conj-assoc* [*symmetric*])
(*rule sep-conjI* [**where** $s_0 = (\text{lift-state } (h, \text{ptr-retyp } p \ d)) \mid' \text{dom } s_0$ **and** $s_1 = s_1$],
auto simp: map-disj-def)

qed

lemma *sep-cut-dom*:

$\text{sep-cut } x \ y \ s \implies \text{dom } s = \{(a, b). a \in \{x..+\text{unat } y\}\}$
by (*force simp: sep-cut-def dest: sep-cut'-dom*)

lemma *sep-cut-0* [*simp*]:

$\text{sep-cut } p \ 0 = \square$
by (*auto simp: sep-cut'-def sep-cut-def sep-emp-def None-com split-def*)

lemma *heap-merge-restrict-dom-un*:

$dom\ s = P \cup Q \implies (s|'P) ++ (s|'Q) = s$

by (*force simp: map-add-def restrict-map-def split: option.splits*)

lemma *sep-cut-split*:

assumes *sc: sep-cut p y s* **and** *le: x ≤ y*

shows $(sep\text{-}cut\ p\ x \wedge^* sep\text{-}cut\ (p + x)\ (y - x))\ s$

proof (*rule sep-conjI* [**where** $s_0 = s|'\{(a,b). a \in \{p..+unat\ x\}\}$ **and**

$s_1 = s|'\{(a,b). a \in \{p..+unat\ y\}\} - \{(a,b). a \in \{p..+unat\ x\}\}$])

from *sc le* **show** $sep\text{-}cut\ p\ x\ (s|'\{(a,b). a \in \{p..+unat\ x\}\})$

by (*force simp: sep-cut-def sep-cut'-def word-le-nat-alt*

dest: intvl-start-le)

next

from *sc le*

show $sep\text{-}cut\ (p + x)\ (y - x)\ (s|'\{(a,b). a \in \{p..+unat\ y\}\} - \{(a,b). a \in \{p..+unat\ x\}\})$

by (*force simp: sep-cut-def sep-cut'-def intvl-sub-eq*)

next

show $s|'\{(a,b). a \in \{p..+unat\ x\}\} \perp s|'\{(a,b). a \in \{p..+unat\ y\}\} - \{(a,b). a \in \{p..+unat\ x\}\}$

by (*force simp: map-disj-def*)

next

from *sc le*

show $s = s|'\{(a,b). a \in \{p..+unat\ y\}\} - \{(a,b). a \in \{p..+unat\ x\}\} ++ s|'\{(a,b). a \in \{p..+unat\ x\}\}$

by (*simp add: sep-cut-def sep-cut'-def, subst heap-merge-restrict-dom-un*)

(*auto simp: word-le-nat-alt dest: intvl-start-le*)

qed

lemma *tagd-ptr-safe-exc*:

$(g \vdash_s p \wedge^* sep\text{-}true)\ (lift\text{-}state\ (h,d)) \implies ptr\text{-}safe\ p\ d$

apply (*clarsimp simp: ptr-safe-def sep-conj-ac sep-conj-def, drule tagd-dom-exc*)

apply (*drule-tac x=(a,b) in fun-cong*)

apply (*force simp: map-ac-simps lift-state-def sep-conj-ac dom-s-def merge-dom split: option.splits s-heap-index.splits if-split-asm*)

done

lemma *sep-map'-ptr-safe-exc*:

$(p \hookrightarrow_g (v::'a::mem\text{-}type))\ (lift\text{-}state\ (h,d)) \implies ptr\text{-}safe\ p\ d$

by (*force simp: sep-map'-def intro: sep-conj-impl tagd-ptr-safe-exc*

dest: sep-map-tagd-exc)

end

```

theory SepInv
imports SepCode
begin

```

definition *inv-footprint* :: 'a::c-type ptr \Rightarrow heap-assert **where**

```

inv-footprint p  $\equiv$ 
   $\lambda s. \text{dom } s = \{(x,y). x \in \{\text{ptr-val } p..+\text{size-of } \text{TYPE('a)}\}\} - s\text{-footprint } p$ 

```

Like in Separation.thy, these arrows are defined using bsub and esub but have an *input* syntax abbreviation with just sub. See original comment there for justification.

definition

```

sep-map-inv :: 'a::c-type ptr  $\Rightarrow$  'a ptr-guard  $\Rightarrow$  'a  $\Rightarrow$  heap-assert ( $- \mapsto^i \_ - [56,0,51]$ 
56)

```

where

```

p  $\mapsto_g^i v \equiv p \mapsto_g v \wedge^* \text{inv-footprint } p$ 

```

notation (*input*)

```

sep-map-inv ( $- \mapsto^i \_ - [56,1000,51]$  56)

```

definition

```

sep-map-any-inv :: 'a ::c-type ptr  $\Rightarrow$  'a ptr-guard  $\Rightarrow$  heap-assert ( $- \mapsto^i \_ - [56,0]$ 
56)

```

where

```

p  $\mapsto_g^i \_ - \equiv p \mapsto_g \_ - \wedge^* \text{inv-footprint } p$ 

```

notation (*input*)

```

sep-map-any-inv ( $- \mapsto^i \_ - [56,0]$  56)

```

definition

```

sep-map'-inv :: 'a::c-type ptr  $\Rightarrow$  'a ptr-guard  $\Rightarrow$  'a  $\Rightarrow$  heap-assert ( $- \hookrightarrow^i \_ - [56,0,51]$ 
56)

```

where

```

p  $\hookrightarrow_g^i v \equiv p \hookrightarrow_g v \wedge^* \text{inv-footprint } p$ 

```

notation (*input*)

```

sep-map'-inv ( $- \hookrightarrow^i \_ - [56,1000,51]$  56)

```

definition

```

sep-map'-any-inv :: 'a::c-type ptr  $\Rightarrow$  'a ptr-guard  $\Rightarrow$  heap-assert ( $- \hookrightarrow^i \_ - [56,0]$ 
56)

```

where

```

p  $\hookrightarrow_g^i \_ - \equiv p \hookrightarrow_g \_ - \wedge^* \text{inv-footprint } p$ 

```

notation (*input*)

```

sep-map'-any-inv ( $- \hookrightarrow^i \_ - [56,0]$  56)

```

definition

$tagd\text{-}inv :: 'a\ ptr\text{-}guard \Rightarrow 'a::c\text{-}type\ ptr \Rightarrow heap\text{-}assert\ (\mathbf{infix}\ \vdash_s^i\ 100)$

where

$g \vdash_s^i p \equiv g \vdash_s p \wedge^* inv\text{-}footprint\ p$

—

lemma $sep\text{-}map'\text{-}g$:

$(p \hookrightarrow_g^i v) s \Longrightarrow g\ p$

unfolding $sep\text{-}map'\text{-}inv\text{-}def$ **by** $(fastforce\ dest: sep\text{-}conjD\ sep\text{-}map'\text{-}g\text{-}exc)$

lemma $sep\text{-}map'\text{-}unfold$:

$(p \hookrightarrow_g^i v) = ((p \hookrightarrow_g^i v) \wedge^* sep\text{-}true)$

by $(simp\ add: sep\text{-}map'\text{-}inv\text{-}def\ sep\text{-}map'\text{-}def\ sep\text{-}conj\text{-}ac)$

lemma $sep\text{-}map'\text{-}any\text{-}unfold$:

$(i \hookrightarrow_g^i -) = ((i \hookrightarrow_g^i -) \wedge^* sep\text{-}true)$

apply $(rule\ ext, simp\ add: sep\text{-}map'\text{-}any\text{-}inv\text{-}def\ sep\text{-}map'\text{-}any\text{-}def\ sep\text{-}conj\text{-}ac)$

apply $(rule\ iffI)$

apply $(subst\ sep\text{-}conj\text{-}com)$

apply $(subst\ sep\text{-}conj\text{-}assoc)+$

apply $(erule\ (1)\ sep\text{-}conj\text{-}impl)$

apply $(clarsimp\ simp: sep\text{-}conj\text{-}ac)$

apply $(subst\ (asm)\ sep\text{-}map'\text{-}unfold\text{-}exc, subst\ sep\text{-}conj\text{-}com)$

apply $(subst\ sep\text{-}conj\text{-}exists, fast)$

apply $(subst\ (asm)\ sep\text{-}conj\text{-}com)$

apply $(subst\ (asm)\ sep\text{-}conj\text{-}assoc)+$

apply $(erule\ (1)\ sep\text{-}conj\text{-}impl)$

apply $(subst\ sep\text{-}map'\text{-}unfold\text{-}exc)$

apply $(subst\ (asm)\ sep\text{-}conj\text{-}exists, fast)$

done

lemma $sep\text{-}map'\text{-}conjE1$:

$\llbracket (P \wedge^* Q) s; \bigwedge s. P\ s \Longrightarrow (i \hookrightarrow_g^i v) s \rrbracket \Longrightarrow (i \hookrightarrow_g^i v) s$

by $(subst\ sep\text{-}map'\text{-}unfold, erule\ sep\text{-}conj\text{-}impl, simp+)$

lemma $sep\text{-}map'\text{-}conjE2$:

$\llbracket (P \wedge^* Q) s; \bigwedge s. Q\ s \Longrightarrow (i \hookrightarrow_g^i v) s \rrbracket \Longrightarrow (i \hookrightarrow_g^i v) s$

by $(subst\ (asm)\ sep\text{-}conj\text{-}com, erule\ sep\text{-}map'\text{-}conjE1, simp)$

lemma $sep\text{-}map'\text{-}any\text{-}conjE1$:

$\llbracket (P \wedge^* Q) s; \bigwedge s. P\ s \Longrightarrow (i \hookrightarrow_g^i -) s \rrbracket \Longrightarrow (i \hookrightarrow_g^i -) s$

by $(subst\ sep\text{-}map'\text{-}any\text{-}unfold, erule\ sep\text{-}conj\text{-}impl, simp+)$

lemma $sep\text{-}map'\text{-}any\text{-}conjE2$:

$\llbracket (P \wedge^* Q) s; \bigwedge s. Q\ s \Longrightarrow (i \hookrightarrow_g^i -) s \rrbracket \Longrightarrow (i \hookrightarrow_g^i -) s$

by $(subst\ (asm)\ sep\text{-}conj\text{-}com, erule\ sep\text{-}map'\text{-}any\text{-}conjE1, simp)$

lemma $sep\text{-}map\text{-}any\text{-}old$:

$(p \mapsto_g^i -) = (\lambda s. \exists v. (p \mapsto_g^i v) s)$

by (*simp add: sep-map-inv-def sep-map-any-inv-def sep-map-any-def sep-conj-ac sep-conj-exists*)

lemma *sep-map'-old*:

$(p \hookrightarrow_g^i v) = ((p \mapsto_g^i v) \wedge^* \text{sep-true})$

by (*simp add: sep-map'-inv-def sep-map-inv-def sep-map'-def sep-conj-ac*)

lemma *sep-map'-any-old*:

$(p \hookrightarrow_g^i -) = (\lambda s. \exists v. (p \hookrightarrow_g^i v) s)$

by (*simp add: sep-map'-inv-def sep-map'-any-inv-def sep-map'-any-def sep-conj-exists*)

lemma *sep-map-sep-map'* [*simp*]:

$(p \mapsto_g^i v) s \implies (p \hookrightarrow_g^i v) s$

unfolding *sep-map-inv-def sep-map'-inv-def sep-map'-def*

apply(*simp add: sep-conj-ac*)

apply(*subst sep-conj-com*)

apply(*simp add: sep-conj-assoc sep-conj-impl sep-conj-sep-true*)

done

lemmas *guardI = sep-map'-g[OF sep-map-sep-map']*

lemma *sep-map-anyI* [*simp*]:

$(p \mapsto_g^i v) s \implies (p \hookrightarrow_g^i -) s$

by (*fastforce simp: sep-map-any-inv-def sep-map-inv-def sep-map-any-def sep-conj-ac elim: sep-conj-impl*)

lemma *sep-map-anyD*:

$(p \mapsto_g^i -) s \implies \exists v. (p \mapsto_g^i v) s$

apply(*simp add: sep-map-any-def sep-map-any-inv-def sep-map-inv-def sep-conj-ac*)

apply(*subst (asm) sep-conj-com*)

apply(*clarsimp simp: sep-conj-exists sep-conj-ac*)

done

lemma *sep-conj-mapD*:

$((i \mapsto_g^i v) \wedge^* P) s \implies (i \hookrightarrow_g^i v) s \wedge ((i \mapsto_g^i -) \wedge^* P) s$

by (*simp add: sep-conj-impl sep-map'-conjE2 sep-conj-ac*)

lemma *sep-map'-ptr-safe*:

$(p \hookrightarrow_g^i (v::'a::\text{mem-type})) (\text{lift-state } (h,d)) \implies \text{ptr-safe } p \ d$

unfolding *sep-map'-inv-def*

apply(*rule sep-map'-ptr-safe-exc*)

apply(*subst sep-map'-unfold-exc*)

apply(*fastforce elim: sep-conj-impl*)

done

lemmas *sep-map-ptr-safe = sep-map'-ptr-safe[OF sep-map-sep-map']*

lemma *sep-map-any-ptr-safe*:

fixes *p::'a::mem-type ptr*

shows $(p \mapsto_g^i -) (\text{lift-state } (h, d)) \implies \text{ptr-safe } p \ d$
by (*blast dest: sep-map-anyD intro: sep-map-ptr-safe*)

lemma *sep-heap-update'*:

$(g \vdash_s^i p \wedge^* (p \mapsto_g^i v \longrightarrow^* P)) (\text{lift-state } (h, d)) \implies$
 $P (\text{lift-state } (\text{heap-update } p (v::'a::\text{mem-type}) h, d))$
apply(*rule sep-heap-update'-exc [where g=g]*)
apply(*unfold tagd-inv-def*)
apply(*subst (asm) sep-conj-assoc*)
apply(*erule (1) sep-conj-impl*)
apply(*subst (asm) sep-map-inv-def*)
apply(*simp add: sep-conj-ac*)
apply(*drule sep-conjD, clarsimp*)
apply(*rule sep-implI, clarsimp*)
apply(*drule sep-implD*)
apply(*drule-tac x=s₀ ++ s' in spec*)
apply(*simp add: map-disj-com map-add-disj*)
apply(*clarsimp simp: map-disj-com*)
apply(*erule notE*)
apply(*erule (1) sep-conjI*)
apply(*simp add: map-disj-com*)
apply(*subst map-add-com; simp*)
done

lemma *tagd-g*:

$(g \vdash_s^i p \wedge^* P) s \implies g \ p$
by (*auto simp: tagd-inv-def tagd-def dest!: sep-conjD elim: s-valid-g*)

lemma *tagd-ptr-safe*:

$(g \vdash_s^i p \wedge^* \text{sep-true}) (\text{lift-state } (h, d)) \implies \text{ptr-safe } p \ d$
apply(*rule tagd-ptr-safe-exc*)
apply(*unfold tagd-inv-def*)
apply(*subst (asm) sep-conj-assoc*)
apply(*erule (1) sep-conj-impl*)
apply *simp*
done

lemma *sep-map-tagd*:

$(p \mapsto_g^i (v::'a::\text{mem-type})) s \implies (g \vdash_s^i p) s$
apply(*unfold sep-map-inv-def tagd-inv-def*)
apply(*erule sep-conj-impl*)
apply(*erule sep-map-tagd-exc*)
apply *assumption*
done

lemma *sep-map-any-tagd*:

$(p \mapsto_g^i -) s \implies (g \vdash_s^i (p::'a::\text{mem-type ptr})) s$
by (*clarsimp dest!: sep-map-anyD, erule sep-map-tagd*)

lemma *sep-heap-update*:
 $\llbracket (p \mapsto_g^i - \wedge^* (p \mapsto_g^i v \longrightarrow^* P)) (\text{lift-state } (h,d)) \rrbracket \Longrightarrow$
 $P (\text{lift-state } (\text{heap-update } p (v::'a::\text{mem-type}) h,d))$
by (*force intro: sep-heap-update' dest: sep-map-anyD sep-map-tagd*
elim: sep-conj-impl)

lemma *sep-heap-update-global'*:
 $(g \vdash_s^i p \wedge^* R) (\text{lift-state } (h,d)) \Longrightarrow$
 $((p \mapsto_g^i v) \wedge^* R) (\text{lift-state } (\text{heap-update } p (v::'a::\text{mem-type}) h,d))$
by (*rule sep-heap-update', erule sep-conj-sep-conj-sep-impl-sep-conj*)

lemma *sep-heap-update-global*:
 $(p \mapsto_g^i - \wedge^* R) (\text{lift-state } (h,d)) \Longrightarrow$
 $((p \mapsto_g^i v) \wedge^* R) (\text{lift-state } (\text{heap-update } p (v::'a::\text{mem-type}) h,d))$
by (*fast intro: sep-heap-update-global' sep-conj-impl sep-map-any-tagd*)

lemma *sep-heap-update-global-super-fl-inv*:
 $\llbracket (p \mapsto_g^i u \wedge^* R) (\text{lift-state } (h,d));$
 $\text{field-lookup } (\text{typ-info-t } \text{TYPE}('b::\text{mem-type})) f 0 = \text{Some } (t,n);$
 $\text{export-uinfo } t = (\text{typ-uinfo-t } \text{TYPE}('a)) \rrbracket \Longrightarrow$
 $((p \mapsto_g^i \text{update-ti-t } t (\text{to-bytes-p } v) u) \wedge^* R)$
 $(\text{lift-state } (\text{heap-update } (\text{Ptr } \&(p \rightarrow f)) (v::'a::\text{mem-type}) h,d))$
apply(*unfold sep-map-inv-def*)
apply(*simp only: sep-conj-assoc*)
apply(*erule (2) sep-heap-update-global-super-fl*)
done

lemma *sep-map'-inv*:
 $(p \hookrightarrow_g^i v) s \Longrightarrow (p \hookrightarrow_g v) s$
apply(*unfold sep-map'-inv-def*)
apply(*subst sep-map'-unfold-exc*)
apply(*erule (1) sep-conj-impl, simp*)
done

lemma *sep-map'-lift*:
 $(p \hookrightarrow_g^i (v::'a::\text{mem-type})) (\text{lift-state } (h,d)) \Longrightarrow \text{lift } h p = v$
apply(*drule sep-map'-inv*)
apply(*erule sep-map'-lift-exc*)
done

lemma *sep-map-lift*:
 $((p::'a::\text{mem-type } \text{ptr}) \mapsto_g^i -) (\text{lift-state } (h,d)) \Longrightarrow$
 $(p \mapsto_g^i \text{lift } h p) (\text{lift-state } (h,d))$
apply(*frule sep-map-anyD*)
apply *clarsimp*
apply(*frule sep-map-sep-map'*)
apply(*drule sep-map'-lift*)
apply *simp*
done

lemma *sep-map-lift-up*:

$$\llbracket \exists v. (p \mapsto^i_g v \wedge^* (p \mapsto^i_g v \longrightarrow^* P v)) (lift\text{-}state (h,d)) \rrbracket$$

$$\implies P (lift\ h (p::'a::mem\text{-}type\ ptr)) (lift\text{-}state (h,d))$$
apply *clarsimp*
subgoal for *v*
apply(*subst sep-map'-lift [where g=g and d=d]*)
apply(*subst sep-map'-inv-def*)
apply(*subst sep-map'-def*)
apply(*subst sep-conj-assoc*)+
apply(*subst sep-conj-com [where P=sep-true]*)
apply(*subst sep-conj-assoc [symmetric]*)
apply(*erule sep-conj-impl*)
apply(*simp add: sep-map-inv-def*)
apply *simp*
apply(*rule sep-conj-impl-same [where P=p \mapsto^i_g v and Q=P v]*)
apply(*unfold sep-map-inv-def*)
apply(*erule (2) sep-conj-impl*)
done
done

lemma *sep-map'-anyI [simp]*:

$$(p \mapsto^i_g v) s \implies (p \mapsto^i_g -) s$$
apply(*unfold sep-map'-inv-def sep-map'-any-inv-def*)
apply(*erule sep-conj-impl*)
apply(*erule sep-map'-anyI-exe*)
apply *assumption*
done

lemma *sep-map'-anyD*:

$$(p \mapsto^i_g -) s \implies \exists v. (p \mapsto^i_g v) s$$
unfolding *sep-map'-inv-def sep-map'-any-inv-def sep-map'-any-def*
by (*clarsimp simp: sep-conj-exists*)

lemma *sep-map'-lift-rev*:

$$\llbracket lift\ h\ p = (v::'a::mem\text{-}type); (p \mapsto^i_g -) (lift\text{-}state (h,d)) \rrbracket \implies$$

$$(p \mapsto^i_g v) (lift\text{-}state (h,d))$$
by (*fastforce dest: sep-map'-anyD simp: sep-map'-lift*)

lemma *sep-map'-any-g*:

$$(p \mapsto^i_g -) s \implies g\ p$$
by (*blast dest: sep-map'-anyD intro: sep-map'-g*)

lemma *any-guardI*:

$$(p \mapsto^i_g -) s \implies g\ p$$
by (*drule sep-map-anyD (blast intro: guardI)*)

lemma *sep-map-sep-map-any*:

$$(p \mapsto^i_g v) s \implies (p \mapsto^i_g -) s$$

by (rule sep-map-anyI)

lemma *sep-lift-exists*:

fixes $p :: 'a::mem\text{-}type\ ptr$

assumes $ex: ((\lambda s. \exists v. (p \hookrightarrow_g^i v) s \wedge P v s) \wedge^* Q) (lift\text{-}state\ (h,d))$

shows $(P (lift\ h\ p) \wedge^* Q) (lift\text{-}state\ (h,d))$

proof –

from ex obtain v where $((\lambda s. (p \hookrightarrow_g^i v) s \wedge P v s) \wedge^* Q) (lift\text{-}state\ (h,d))$

by (subst (asm) sep-conj-exists, clarsimp)

thus ?thesis

by (force simp: sep-map'-lift sep-conj-ac
dest: sep-map'-conjE2 dest!: sep-conj-conj)

qed

lemma *sep-map-dom*:

$(p \hookrightarrow_g^i (v::'a::c\text{-}type)) s \implies dom\ s = \{(a,b). a \in \{ptr\text{-}val\ p..size\text{-}of\ TYPE('a)}\}$

unfolding sep-map-inv-def

by (drule sep-conjD, clarsimp)

(auto dest!: sep-map-dom-exc elim: s-footprintD simp: inv-footprint-def)

lemma *sep-map'-dom*:

$(p \hookrightarrow_g^i (v::'a::mem\text{-}type)) s \implies (ptr\text{-}val\ p, SIndexVal) \in dom\ s$

unfolding sep-map'-inv-def

by (drule sep-conjD, clarsimp) (drule sep-map'-dom-exc, clarsimp)

lemma *sep-map'-inj*:

$\llbracket (p \hookrightarrow_g^i (v::'a::c\text{-}type)) s; (p \hookrightarrow_h^i v') s \rrbracket \implies v=v'$

by (drule sep-map'-inv)+ (drule (2) sep-map'-inj-exc)

lemma *ptr-retyp-sep-cut'*:

fixes $p::'a::mem\text{-}type\ ptr$

assumes $sc: (sep\text{-}cut'\ (ptr\text{-}val\ p) (size\text{-}of\ TYPE('a)) \wedge^* P)$

$(lift\text{-}state\ (h,d))$ and $g\ p$

shows $(g \vdash_s^i p \wedge^* P) (lift\text{-}state\ (h,(ptr\text{-}retyp\ p\ d)))$

proof –

from sc

obtain s_0 and s_1

where $s_0 \perp s_1$ and $lift\text{-}state\ (h,d) = s_1 ++ s_0$

and $P\ s_1$ and $d: dom\ s_0 = \{(a,b). a \in \{ptr\text{-}val\ p..size\text{-}of\ TYPE('a)}\}$

and $k: dom\ s_0 \subseteq dom\text{-}s\ d$

by (auto dest!: sep-conjD sep-cut'-dom simp: dom-lift-state-dom-s [where $h=h, symmetric$])

moreover from this

have $lift\text{-}state\ (h, ptr\text{-}retyp\ p\ d) = s_1 ++ lift\text{-}state\ (h, ptr\text{-}retyp\ p\ d) \upharpoonright' (dom\ s_0)$

apply –

apply (rule ext)

subgoal for x

```

apply(cases  $x \in \text{dom } s_0$ )
apply(cases  $x \in \text{dom } s_1$ )
  apply(fastforce simp: map-disj-def)
  apply(subst map-add-com)
  apply(fastforce simp: map-disj-def)
  apply(clarsimp simp: map-add-def split: option.splits)
apply(cases  $x$ , clarsimp)
apply(clarsimp simp: lift-state-ptr-retyp-d merge-dom2)
done
done
moreover have  $g \ p$  by fact
with  $d \ k$  have  $(g \vdash_s^i \ p)$  (lift-state  $(h, \text{ptr-retyp } p \ d)$  |  $\text{dom } s_0$ )
  apply(clarsimp simp: lift-state-ptr-retyp-restrict sep-conj-ac tagd-inv-def)
  apply(rule sep-conjI [where  $s_0 = \text{lift-state } (h, d)$  |  $\{ (a, b). a \in \{\text{ptr-val } p.. + \text{size-of TYPE('a)}\} - s\text{-footprint } p\}$ ])
    apply(fastforce simp: inv-footprint-def)
    apply(erule ptr-retyp-tagd-exc[where  $h = h$ ])
    apply(fastforce simp: map-disj-def)
    apply(subst map-add-comm[of lift-state  $(h, \text{ptr-retyp } p \ \text{empty-htd})$ ])
    apply force
    apply(rule ext)
    apply(clarsimp simp: map-add-def split: option.splits)
    by (metis (mono-tags) Diff-iff dom-ptr-retyp-empty-htd non-dom-eval-eq restrict-in-dom restrict-out)
  ultimately show ?thesis
    by (metis restrict-map-on-disj sep-conjI)
qed

lemma ptr-retyp-sep-cut'-wp:
   $\llbracket (\text{sep-cut}' (\text{ptr-val } p) (\text{size-of TYPE('a)}) \wedge^* (g \vdash_s^i \ p \longrightarrow^* P))$ 
   $(\text{lift-state } (h, d)); g \ (p::'a::\text{mem-type } \text{ptr}) \rrbracket$ 
   $\implies P \ (\text{lift-state } (h, (\text{ptr-retyp } p \ d)))$ 
by (rule sep-conj-impl-same [where  $P = g \vdash_s^i \ p$  and  $Q = P$ ]) (simp add: ptr-retyp-sep-cut')

lemma tagd-dom:
   $(g \vdash_s^i \ p) \ s \implies \text{dom } s = \{(a, b). a \in \{\text{ptr-val } (p::'a::\text{c-type } \text{ptr}).. + \text{size-of TYPE('a)}\}\}$ 
  unfolding tagd-inv-def
  by (drule sep-conjD, clarsimp)
  (auto simp: inv-footprint-def dest!: tagd-dom-exc elim: s-footprintD)

lemma tagd-dom-p:
   $(g \vdash_s^i \ p) \ s \implies (\text{ptr-val } (p::'a::\text{mem-type } \text{ptr}), S\text{IndexVal}) \in \text{dom } s$ 
  by (drule tagd-dom) clarsimp

end

```

```

theory SepTactic
imports SepInv
begin

```

11.23 *sep-point-tac*

```

definition sep-conj-extract :: heap-assert  $\Rightarrow$  heap-assert where
  sep-conj-extract  $\equiv$  id

```

```

definition sep-points :: heap-assert  $\Rightarrow$  heap-assert where
  sep-points P  $\equiv$  P  $\wedge^*$  sep-true

```

```

lemma sep-conj-extract1D:
  (P  $\wedge^*$  Q) s  $\Longrightarrow$  sep-conj-extract (P  $\wedge^*$  Q) s
by (simp add: sep-conj-extract-def)

```

```

lemma sep-conj-extract2D:
  sep-conj-extract P s  $\Longrightarrow$  P s  $\wedge$  (sep-conj-extract P  $\wedge^*$  sep-true) s
by (auto simp: sep-conj-extract-def elim: sep-conj-sep-true)

```

```

lemma sep-conj-extract-assoc:
  sep-conj-extract ((P  $\wedge^*$  Q)  $\wedge^*$  R) = sep-conj-extract (P  $\wedge^*$  (Q  $\wedge^*$  R))
by (simp add: sep-conj-ac)

```

```

lemma sep-conj-extract-decomposeD:
  (sep-conj-extract (P  $\wedge^*$  Q)  $\wedge^*$  sep-true) s  $\Longrightarrow$  sep-points P s  $\wedge$ 
  (sep-conj-extract Q  $\wedge^*$  sep-true) s
apply (rule conjI)
apply (simp add: sep-conj-extract-def sep-points-def sep-conj-ac)
apply (erule (1) sep-conj-impl, simp)
apply (simp add: sep-conj-extract-def sep-conj-ac)
apply (subst (asm) sep-conj-assoc [symmetric])
apply (subst (asm) sep-conj-com)
apply (subst (asm) sep-conj-assoc)
apply (erule (1) sep-conj-impl)
apply simp
done

```

```

lemma sep-conj-extract-decomposeD2:
  (sep-conj-extract P  $\wedge^*$  sep-true) s  $\Longrightarrow$  sep-points P s
by (simp add: sep-conj-extract-def sep-points-def)

```

```

lemma sep-point-mapD:
  sep-points (p  $\mapsto_g^i$  v) s  $\Longrightarrow$  (p  $\hookrightarrow_g^i$  v) s
by (simp add: sep-points-def sep-map-inv-def sep-map'-inv-def sep-map'-def
  sep-conj-ac)

```

```

lemma sep-point-map-excD:

```

sep-points ($p \mapsto_g v$) $s \implies (p \hookrightarrow_g v) s$
by (*simp add: sep-points-def sep-map'-def*)

lemma *sep-point-otherD*:
sep-points $P s \implies \text{True}$
by (*rule TrueI*)

ML \langle

val sep-point-ds = [*@{thm sep-point-mapD}*], [*@{thm sep-point-map-excD}*]

fun sep-point-tacs ds ctxt = [
REPEAT1 (*dresolve-tac* *ctxt* [*@{thm sep-conj-extract1D}*] 1),
REPEAT1 (*dresolve-tac* *ctxt* [*@{thm sep-conj-extract2D}*] 1),
clarify-tac *ctxt* 1,
TRY (*full-simp-tac* (*put-simpset HOL-ss* *ctxt* *addsimps*
[*@{thm sep-conj-extract-assoc}*])) 1),
REPEAT (*dresolve-tac* *ctxt* [*@{thm sep-conj-extract-decomposeD}*],
[*@{thm sep-conj-extract-decomposeD2}*] 1) *THEN*
clarify-tac *ctxt* 1),
TRY (*full-simp-tac* *ctxt* 1),
REPEAT (*dresolve-tac* *ctxt* (*sep-point-ds@ds*) 1),
REPEAT (*dresolve-tac* *ctxt* [*@{thm sep-point-otherD}*] 1),
TRY (*full-simp-tac* *ctxt* 1)
]

fun sep-point-tac ds ctxt = *EVERY* (*sep-point-tacs ds* *ctxt*)
 \rangle

method-setup *sep-point-tac* =
Attrib.thms \gg (*fn thms* => *Method.SIMPLE-METHOD* *o sep-point-tac* *thms*)
extract points—to facts from separation assertions in assumptions

lemma ($P \wedge^* p \mapsto_g v \wedge^* Q$) $s \implies (p \hookrightarrow_g v) s$
by *sep-point-tac*

11.24 *sep-exists-tac*

ML \langle

fun sep-exists-tac ctxt = *full-simp-tac*
(*put-simpset HOL-ss* *ctxt* *addsimps* [*@{thm sep-conj-exists1}*], [*@{thm sep-conj-exists2}*])
 \rangle

method-setup *sep-exists-tac* =
Scan.succeed (*Method.SIMPLE-METHOD'* *o sep-exists-tac*)
push existentials inside separation assertions to the outside

lemma ($P \wedge^* (\lambda s. (\exists x. Q x s))$) $s \implies \exists x. (P \wedge^* Q x) s$
by *sep-exists-tac*

11.25 *sep-select-tac*

definition *sep-mark* :: *heap-assert* \Rightarrow *heap-assert* **where**
sep-mark \equiv *id*

definition *sep-mark2* :: *heap-assert* \Rightarrow *heap-assert* **where**
sep-mark2 \equiv *id*

lemma *sep-mark2-id*:
sep-mark2 *P* = *P*
by (*simp add: sep-mark2-def*)

lemma *sep-markI*:
 $(\Box \wedge^* \text{sep-mark } (P \wedge^* Q)) s \Longrightarrow (P \wedge^* Q) s$
by (*simp add: sep-mark-def sep-conj-empty*)

lemma *sep-mark-match*:
 $(R \wedge^* \text{sep-mark2 } P \wedge^* Q) s \Longrightarrow (R \wedge^* \text{sep-mark } (P \wedge^* Q)) s$
by (*simp add: sep-mark-def sep-mark2-def*)

lemma *sep-mark-match2*:
 $(R \wedge^* \text{sep-mark2 } P) s \Longrightarrow (R \wedge^* \text{sep-mark } P) s$
by (*simp add: sep-mark-def sep-mark2-def*)

lemma *sep-mark-mismatch*:
 $((R \wedge^* P) \wedge^* \text{sep-mark } Q) s \Longrightarrow (R \wedge^* \text{sep-mark } (P \wedge^* Q)) s$
by (*simp add: sep-mark-def sep-conj-ac*)

lemma *sep-mark-mismatch2*:
 $(R \wedge^* P) s \Longrightarrow (R \wedge^* \text{sep-mark } P) s$
by (*simp add: sep-mark-def*)

lemma *sep-emp-rem*:
 $P s \Longrightarrow (\Box \wedge^* P) s$
by (*simp add: sep-conj-empty*)

lemma *sep-mark2-left*:
 $(P \wedge^* (\text{sep-mark2 } Q \wedge^* R)) = (\text{sep-mark2 } Q \wedge^* (P \wedge^* R))$
by (*rule sep-conj-left-com*)

lemma *sep-mark2-left2*:
 $(P \wedge^* \text{sep-mark2 } Q) = (\text{sep-mark2 } Q \wedge^* P)$
by (*rule sep-conj-com*)

ML \langle

(* Replace all occurrences of an underscore that does not have an alphanumeric
on either side of it *)

local

```

val dummy-char = # ;
fun get-next (x::-) = x
  | get-next [] = dummy-char;
fun nth-id n = SEPTAC^(Int.toString n);

fun ok-char c = not (Char.isAlphaNum c);

fun can-replace (prev,cur,last) =
  (ok-char prev) andalso (cur = #-) andalso (ok-char last);

fun replace-usc n prev-char (x::xs) =
  let val (xs', ys) = replace-usc (n+1) x xs
  in
    if can-replace (prev-char,x,get-next xs)
    then (nth-id n::xs', nth-id n::ys)
    else (String.str x::xs', ys)
  end
  | replace-usc - - [] = ([,[]])
in
  val rusc = apfst String.concat o replace-usc 0 dummy-char o String.explode
end;

(* some tests *)
rusc (- +p -) ↦ -;
rusc - ↦- -;
rusc -;
rusc - O-o -
>

```

ML⋄

```

fun sep-select-tacs s ctxt =
  let val (str, vars) = rusc s
      val subst = [((Lexicon.read-indexname P, Position.none), str)]
      val fixes = map (fn v => (Binding.name v, NONE, Mixfix.NoSyn)) vars
  in
    [
      TRY (simp-tac (put-simpset HOL-ss ctxt addsimps [@{thm sep-conj-assoc}])
1),
      resolve-tac ctxt [@{thm sep-markI}] 1,
      REPEAT (Rule-Insts.res-inst-tac ctxt subst fixes @{thm sep-mark-match} 1
ORELSE
      (Rule-Insts.res-inst-tac ctxt subst fixes @{thm sep-mark-match2} 1 ORELSE
      resolve-tac ctxt [@{thm sep-mark-mismatch}] 1 ORELSE
      resolve-tac ctxt [@{thm sep-mark-mismatch2}] 1)),
      TRY (simp-tac (put-simpset HOL-ss ctxt addsimps [@{thm sep-conj-assoc}])
1),
      resolve-tac ctxt [@{thm sep-emp-rem}] 1,
      TRY (simp-tac (put-simpset HOL-ss ctxt addsimps [@{thm sep-mark2-left},

```

```

    @{thm sep-mark2-left2}}] 1),
    TRY (simp-tac (put-simpset HOL-ss ctxt addsimps [@{thm sep-mark2-id}}] 1)
  ]
end

fun sep-select-tac s ctxt = SELECT-GOAL (EVERY (sep-select-tacs s ctxt))

fun lift-parser p (ctxt,ts) =
  let val (r, ts') = p ts
  in (r, (ctxt,ts')) end
>

method-setup sep-select-tac =
  <lift-parser Args.name >> (fn s => Method.SIMPLE-METHOD' o sep-select-tac
s)>
  takes a target conjunct in the goal and reorders it to the front

lemma
   $\bigwedge R x f n. ((P::\text{heap-assert}) \wedge^* \text{fac } x \ n \wedge^* R (f \ x)) \ s$ 
  apply(sep-select-tac fac - -)
  apply(sep-select-tac R -)
  apply(sep-select-tac fac x -)
  apply(sep-select-tac R -)
  oops

lemma
   $((P::\text{heap-assert}) \wedge^* \text{fac } x \ n) \ s$ 
  apply(sep-select-tac fac - -)
  oops

lemma
   $((P::\text{heap-assert}) \wedge^* \text{long-name}) \ s$ 
  apply(sep-select-tac long-name)
  oops

lemma sep-heap-update'-hrs:
   $(c\text{-guard} \vdash_s^i p \wedge^* (p \mapsto_c^i \text{guard } v \longrightarrow^* P)) (lift\text{-state } s) \implies$ 
   $P (lift\text{-state } (\text{hrs-mem-update } (\text{heap-update } p (v::'a::\text{mem-type}))) s)$ 
  by (cases s)
  (simp add: hrs-mem-update-def, erule sep-heap-update')

lemma sep-map-lift-wp-hrs:
   $\llbracket \exists v. (p \mapsto_c^i \text{guard } v \wedge^* (p \mapsto_c^i \text{guard } v \longrightarrow^* P \ v)) (lift\text{-state } s) \rrbracket$ 
   $\implies P (lift (\text{hrs-mem } s) (p::'a::\text{mem-type } ptr)) (lift\text{-state } s)$ 
  by (cases s)
  (simp add: hrs-mem-def, erule sep-map-lift-wp)

```


lemma *ptr-retyp-sep-cut'-wp-hrs*:

$$\begin{aligned} & \llbracket (sep\text{-}cut' (ptr\text{-}val\ p) (size\text{-}of\ TYPE('a)) \wedge^* (c\text{-}guard \vdash_s^i p \longrightarrow^* P)) \\ & \quad (lift\text{-}state\ s); c\text{-}guard (p::'a::mem\text{-}type\ ptr) \rrbracket \\ & \implies P (lift\text{-}state (hrs\text{-}htd\text{-}update (ptr\text{-}retyp\ p) s)) \\ \text{by } & (cases\ s) \\ & (simp\ add: hrs\text{-}htd\text{-}update\text{-}def, erule (1) ptr\text{-}retyp\text{-}sep\text{-}cut'\text{-}wp) \end{aligned}$$

ML ‹

```
fun destruct bs (Bound n)      = Free (List.nth (bs,n))
  | destruct bs (Abs (s,ty,t)) = destruct ((s,ty)::bs) t
  | destruct bs (l$r)          = destruct bs l $ destruct bs r
  | destruct - x               = x
```

```
fun schems (Abs (-,ty,t)) = schems t
  | schems (l$r)          = schems l + schems r
  | schems (Var -)        = 1
  | schems -              = 0
```

```
fun hrs\text{-}mem\text{-}update (Const (@{const\text{-}name HeapRawState.hrs\text{-}mem\text{-}update},-)$s)
= SOME s
  | hrs\text{-}mem\text{-}update - = NONE
```

```
fun hrs\text{-}htd\text{-}update (Const (@{const\text{-}name HeapRawState.hrs\text{-}htd\text{-}update},-)$s) =
SOME s
  | hrs\text{-}htd\text{-}update - = NONE
```

```
fun lift\text{-}state\text{-}arg - - (Const (@{const\text{-}name TypHeap.lift\text{-}state},-)$s) = SOME s
  | lift\text{-}state\text{-}arg thy nc (l$r) =
    =
    (case lift\text{-}state\text{-}arg thy nc r of
      SOME s => if nc orelse hrs\text{-}mem\text{-}update s <> NONE orelse
                hrs\text{-}htd\text{-}update s <> NONE then SOME s
                else lift\text{-}state\text{-}arg thy nc l
      | NONE => lift\text{-}state\text{-}arg thy nc l)
  | lift\text{-}state\text{-}arg thy nc (Abs (-,ty,t)) =
    =
    lift\text{-}state\text{-}arg thy nc t
  | lift\text{-}state\text{-}arg - - - =
    NONE (*error (Sign.string\text{-}of\text{-}term thy s)*)
```

```
fun term\text{-}of\text{-}thm thm = Thm.term\text{-}of (Thm.cprop\text{-}of thm)
```

```
fun lift\text{-}tac ctxt s old\text{-}schems =
  Rule\text{-}Insts.res\text{-}inst\text{-}tac ctxt [((Lexicon.read\text{-}index\text{-}name s,Position.none),s)] [] @{thm
sep\text{-}map\text{-}lift\text{-}wp\text{-}hrs} 1 THEN
  (fn thm => if schems (term\text{-}of\text{-}thm thm) = old\text{-}schems then all\text{-}tac thm
            else no\text{-}tac thm)
```

```
fun heap\text{-}update\text{-}tac ctxt s =
  Rule\text{-}Insts.res\text{-}inst\text{-}tac ctxt [((Lexicon.read\text{-}index\text{-}name s,Position.none),s)] [] @{thm
```

```

sep-heap-update'-hrs} 1

fun ptr-retyp-tac ctxt s =
  Rule-Insts.res-inst-tac ctxt [((Lexicon.read-indexname s,Position.none),s)] [] @{thm
ptr-retyp-sep-cut'-wp-hrs} 1

fun sep-wp-step-tac' ctxt lift-only s old-schems = let
  fun prt s = (
    case s of
      Free (a, -) => a
    | - => raise TERM (Not a free variable,[s])
  val state = prt s
  val step = case hrs-mem-update s of
    SOME s => heap-update-tac ctxt (prt s)
  | NONE => case hrs-htd-update s of
    SOME s => ptr-retyp-tac ctxt (prt s)
  | NONE => no-tac
  in
    TRY (lift-tac ctxt state old-schems) THEN (if lift-only then all-tac else step)
  end

fun sep-wp-step-tac ctxt lift-only thm = let
  val t = term-of-thm thm
  val thy = Proof-Context.theory-of ctxt
  val old-schems = schems t
  val s = lift-state-arg thy lift-only (destruct [] t)
  in
    (case s of
      SOME state => sep-wp-step-tac' ctxt lift-only state old-schems
    | NONE => no-tac) thm
  end

fun sep-wp-tac ctxt = (REPEAT (sep-wp-step-tac ctxt false)) THEN sep-wp-step-tac
ctxt true

›

method-setup sep-wp-tac =
  Scan.succeed (Method.SIMPLE-METHOD o sep-wp-tac)
  apply WP separation logic rules to goal

lemma
  ((λz. lift (hrs-mem s) (p::(32 word) ptr) + 1 = 2) ∧* P) (lift-state s)
  apply sep-wp-tac
  oops

```

end

theory *SepFrame*
imports *SepTactic*
begin

class *heap-state-type'*

instance *heap-state-type' ⊆ type ..*

consts

hst-mem :: '*a*::*heap-state-type'* ⇒ *heap-mem*
hst-mem-update :: (*heap-mem* ⇒ *heap-mem*) ⇒ '*a*::*heap-state-type'* ⇒ '*a*
hst-htd :: '*a*::*heap-state-type'* ⇒ *heap-tyt-desc*
hst-htd-update :: (*heap-tyt-desc* ⇒ *heap-tyt-desc*) ⇒ '*a*::*heap-state-type'* ⇒ '*a*

class *heap-state-type = heap-state-type' +*
assumes *hst-htd-htd-update [simp]: hst-htd (hst-htd-update d s) = d (hst-htd s)*
assumes *hst-mem-mem-update [simp]: hst-mem (hst-mem-update h s) = h (hst-mem s)*
assumes *hst-htd-mem-update [simp]: hst-htd (hst-mem-update h s) = hst-htd s*
assumes *hst-mem-htd-update [simp]: hst-mem (hst-htd-update d s) = hst-mem s*

translations

s(*hst-mem := x*) ≤_{CONST} *hst-mem-update (K-record x) s*
s(*hst-htd := x*) ≤_{CONST} *hst-htd-update (K-record x) s*

definition *lift-hst* :: '*a*::*heap-state-type'* ⇒ *heap-state* **where**

lift-hst s ≡ *lift-state (hst-mem s, hst-htd s)*

definition *point-eq-mod* :: ('*a* ⇒ '*b*) ⇒ ('*a* ⇒ '*b*) ⇒ '*a* set ⇒ *bool* **where**

point-eq-mod f g X ≡ ∀ *x*. *x* ∉ *X* ⟶ *f x* = *g x*

definition *exec-fatal* :: ('*s*, '*b*, '*c*) *com* ⇒ ('*s*, '*b*, '*c*) *body* ⇒ '*s* ⇒ *bool* **where**

exec-fatal C Γ s ≡ (∃ *f*. Γ ⊢ ⟨*C*, *Normal s*⟩ ⇒ *Fault f*) ∨ (Γ ⊢ ⟨*C*, *Normal s*⟩ ⇒ *Stuck*)

definition *restrict-htd* :: '*s*::*heap-state-type'* ⇒ *s-addr set* ⇒ '*s* **where**

restrict-htd s X ≡ *s*(*hst-htd := restrict-s (hst-htd s) X*)

definition *restrict-safe-OK* ::

'*s* ⇒ '*s* ⇒ ('*s* ⇒ ('*s*, '*c*) *xstate*) ⇒ *s-addr set* ⇒ ('*s*::*heap-state-type'*, '*b*, '*c*) *com* ⇒ ('*s*, '*b*, '*c*) *body* ⇒ *bool* **where**

restrict-safe-OK s t f X C Γ ≡

Γ ⊢ ⟨*C*, (Normal (restrict-htd s X))⟩ ⇒ *f (restrict-htd t X)* ∧

point-eq-mod (lift-state (hst-mem t, hst-htd t)) (lift-state (hst-mem s, hst-htd s))

X

definition *restrict-safe* ::

$'s \Rightarrow ('s, 'c) \text{ xstate} \Rightarrow ('s::\text{heap-state-type}', 'b, 'c) \text{ com} \Rightarrow ('s, 'b, 'c) \text{ body} \Rightarrow \text{bool}$

where

$\text{restrict-safe } s \ t \ C \ \Gamma \equiv$
 $\forall X. (\text{case } t \text{ of}$
 $\quad \text{Normal } t' \Rightarrow \text{restrict-safe-OK } s \ t' \ \text{Normal } X \ C \ \Gamma$
 $\quad | \text{Abrupt } t' \Rightarrow \text{restrict-safe-OK } s \ t' \ \text{Abrupt } X \ C \ \Gamma$
 $\quad | - \Rightarrow \text{False}) \vee$
 $\text{exec-fatal } C \ \Gamma \ (\text{restrict-htd } s \ X)$

definition *mem-safe* :: $('s::\{\text{heap-state-type}', \text{type}\}, 'b, 'c) \text{ com} \Rightarrow ('s, 'b, 'c) \text{ body} \Rightarrow \text{bool}$ **where**

$\text{mem-safe } C \ \Gamma \equiv \forall s \ t. \ \Gamma \vdash \langle C, \text{Normal } s \rangle \Rightarrow t \longrightarrow \text{restrict-safe } s \ t \ C \ \Gamma$

definition *point-eq-mod-safe* ::

$'s::\{\text{heap-state-type}', \text{type}\} \text{ set} \Rightarrow ('s \Rightarrow 's) \Rightarrow ('s \Rightarrow s\text{-addr} \Rightarrow 'c) \Rightarrow \text{bool}$ **where**
 $\text{point-eq-mod-safe } P \ f \ g \equiv \forall s \ X. \ \text{restrict-htd } s \ X \in P \longrightarrow \text{point-eq-mod } (g \ (f \ s)) \ (g \ s) \ X$

definition *comm-restrict* :: $('s::\{\text{heap-state-type}', \text{type}\} \Rightarrow 's) \Rightarrow 's \Rightarrow s\text{-addr} \text{ set} \Rightarrow \text{bool}$ **where**

$\text{comm-restrict } f \ s \ X \equiv f \ (\text{restrict-htd } s \ X) = \text{restrict-htd } (f \ s) \ X$

definition *comm-restrict-safe* :: $'s \text{ set} \Rightarrow ('s::\{\text{heap-state-type}', \text{type}\} \Rightarrow 's) \Rightarrow \text{bool}$ **where**

$\text{comm-restrict-safe } P \ f \equiv \forall s \ X. \ \text{restrict-htd } s \ X \in P \longrightarrow \text{comm-restrict } f \ s \ X$

definition *htd-ind* :: $('a::\{\text{heap-state-type}', \text{type}\} \Rightarrow 'b) \Rightarrow \text{bool}$ **where**

$\text{htd-ind } f \equiv \forall x \ s. \ f \ s = f \ (\text{hst-htd-update } x \ s)$

definition *mono-guard* :: $'s::\{\text{heap-state-type}', \text{type}\} \text{ set} \Rightarrow \text{bool}$ **where**

$\text{mono-guard } P \equiv \forall s \ X. \ \text{restrict-htd } s \ X \in P \longrightarrow s \in P$

definition *expr-htd-ind* :: $'s::\{\text{heap-state-type}', \text{type}\} \text{ set} \Rightarrow \text{bool}$ **where**

$\text{expr-htd-ind } P \equiv \forall d \ s. \ s \langle \text{hst-htd} := d \rangle \in P = (s \in P)$

primrec *intra-safe* :: $('s::\text{heap-state-type}', 'b, 'c) \text{ com} \Rightarrow \text{bool}$

where

$\text{intra-safe } \text{Skip} = \text{True}$
 $| \text{intra-safe } (\text{Basic } f) = (\text{comm-restrict-safe } \text{UNIV } f \wedge$
 $\quad \text{point-eq-mod-safe } \text{UNIV } f \ (\lambda s. \ \text{lift-state } (\text{hst-mem } s, \text{hst-htd } s)))$
 $| \text{intra-safe } (\text{Spec } r) = (\forall \Gamma. \ \text{mem-safe } (\text{Spec } r) \ (\Gamma :: ('s, 'b, 'c) \text{ body}))$
 $| \text{intra-safe } (\text{Seq } C \ D) = (\text{intra-safe } C \wedge \text{intra-safe } D)$
 $| \text{intra-safe } (\text{Cond } P \ C \ D) = (\text{expr-htd-ind } P \wedge \text{intra-safe } C \wedge \text{intra-safe } D)$
 $| \text{intra-safe } (\text{While } P \ C) = (\text{expr-htd-ind } P \wedge \text{intra-safe } C)$
 $| \text{intra-safe } (\text{Call } p) = \text{True}$
 $| \text{intra-safe } (\text{DynCom } f) = (\text{htd-ind } f \wedge (\forall s. \ \text{intra-safe } (f \ s)))$

| *intra-safe* (*Guard* *f P C*) = (*mono-guard* *P* \wedge (*case* *C* of
 Basic *g* \Rightarrow *comm-restrict-safe* *P g* \wedge
 point-eq-mod-safe *P g* ($\lambda s.$ *lift-state* (*hst-mem* *s*, *hst-htd* *s*))
 | - \Rightarrow *intra-safe* *C*))
 | *intra-safe* *Throw* = *True*
 | *intra-safe* (*Catch* *C D*) = (*intra-safe* *C* \wedge *intra-safe* *D*)

instance *state-ext* :: (*heap-state-type'*, *type*) *heap-state-type'* ..

overloading

hs-mem-state \equiv *hst-mem*
hs-mem-update-state \equiv *hst-mem-update*
hs-htd-state \equiv *hst-htd*
hs-htd-update-state \equiv *hst-htd-update*

begin

definition *hs-mem-state* [*simp*]: *hs-mem-state* \equiv *hst-mem* \circ *globals*

definition *hs-mem-update-state* [*simp*]: *hs-mem-update-state* \equiv *globals-update* \circ
hst-mem-update

definition *hs-htd-state* [*simp*]: *hs-htd-state* \equiv *hst-htd* \circ *globals*

definition *hs-htd-update-state* [*simp*]: *hs-htd-update-state* \equiv *globals-update* \circ *hst-htd-update*

end

instance *state-ext* :: (*heap-state-type*, *type*) *heap-state-type*

by *intro-classes auto*

primrec

intra-deps :: ('*s'*, '*b*', '*c*') *com* \Rightarrow '*b* *set*

where

intra-deps *Skip* = {}
 | *intra-deps* (*Basic* *f*) = {}
 | *intra-deps* (*Spec* *r*) = {}
 | *intra-deps* (*Seq* *C D*) = (*intra-deps* *C* \cup *intra-deps* *D*)
 | *intra-deps* (*Cond* *P C D*) = (*intra-deps* *C* \cup *intra-deps* *D*)
 | *intra-deps* (*While* *P C*) = *intra-deps* *C*
 | *intra-deps* (*Call* *p*) = {*p*}
 | *intra-deps* (*DynCom* *f*) = $\bigcup \{ \textit{intra-deps} (f s) \mid s. \textit{True} \}$
 | *intra-deps* (*Guard* *f P C*) = *intra-deps* *C*
 | *intra-deps* *Throw* = {}
 | *intra-deps* (*Catch* *C D*) = (*intra-deps* *C* \cup *intra-deps* *D*)

inductive-set

proc-deps :: ('*s'*, '*b*', '*c*') *com* \Rightarrow ('*s'*, '*b*', '*c*') *body* \Rightarrow '*b* *set*

for *C* :: ('*s'*, '*b*', '*c*') *com*

and Γ :: ('*s'*, '*b*', '*c*') *body*

where

$x \in \textit{intra-deps} C \Longrightarrow x \in \textit{proc-deps} C \Gamma$
 | $\llbracket x \in \textit{proc-deps} C \Gamma; \Gamma x = \textit{Some} D; y \in \textit{intra-deps} D \rrbracket \Longrightarrow y \in \textit{proc-deps} C \Gamma$

—

lemma *point-eq-mod-refl* [*simp*]:

point-eq-mod f f X

by (*simp add: point-eq-mod-def*)

lemma *point-eq-mod-subst*:

$\llbracket \text{point-eq-mod } f g Y; Y \subseteq X \rrbracket \Longrightarrow \text{point-eq-mod } f g X$

by (*force simp: point-eq-mod-def*)

lemma *point-eq-mod-trans*:

$\llbracket \text{point-eq-mod } x y X; \text{point-eq-mod } y z X \rrbracket \Longrightarrow \text{point-eq-mod } x z X$

by (*force simp: point-eq-mod-def*)

lemma *mem-safe-NormalD*:

$\llbracket \Gamma \vdash \langle C, \text{Normal } s \rangle \Rightarrow \text{Normal } t; \text{mem-safe } C \Gamma;$

$\neg \text{exec-fatal } C \Gamma (\text{restrict-htd } s X) \rrbracket \Longrightarrow$

$(\Gamma \vdash \langle C, (\text{Normal } (\text{restrict-htd } s X)) \rangle \Rightarrow \text{Normal } (\text{restrict-htd } t X) \wedge$

$\text{point-eq-mod } (\text{lift-state } (\text{hst-mem } t, \text{hst-htd } t))$

$(\text{lift-state } (\text{hst-mem } s, \text{hst-htd } s)) X$

by (*force simp: mem-safe-def restrict-safe-def restrict-safe-OK-def*)

lemma *mem-safe-AbruptD*:

$\llbracket \Gamma \vdash \langle C, \text{Normal } s \rangle \Rightarrow \text{Abrupt } t; \text{mem-safe } C \Gamma;$

$\neg \text{exec-fatal } C \Gamma (\text{restrict-htd } s X) \rrbracket \Longrightarrow$

$(\Gamma \vdash \langle C, (\text{Normal } (\text{restrict-htd } s X)) \rangle \Rightarrow \text{Abrupt } (\text{restrict-htd } t X) \wedge$

$\text{point-eq-mod } (\text{lift-state } (\text{hst-mem } t, \text{hst-htd } t))$

$(\text{lift-state } (\text{hst-mem } s, \text{hst-htd } s)) X$

by (*force simp: mem-safe-def restrict-safe-def restrict-safe-OK-def*)

lemma *mem-safe-FaultD*:

$\llbracket \Gamma \vdash \langle C, \text{Normal } s \rangle \Rightarrow \text{Fault } f; \text{mem-safe } C \Gamma \rrbracket \Longrightarrow$

$\text{exec-fatal } C \Gamma (\text{restrict-htd } s X)$

by (*force simp: mem-safe-def restrict-safe-def*)

lemma *mem-safe-StuckD*:

$\llbracket \Gamma \vdash \langle C, \text{Normal } s \rangle \Rightarrow \text{Stuck}; \text{mem-safe } C \Gamma \rrbracket \Longrightarrow$

$\text{exec-fatal } C \Gamma (\text{restrict-htd } s X)$

by (*force simp: mem-safe-def restrict-safe-def*)

lemma *lift-state-d-restrict* [*simp*]:

$\text{lift-state } (h, (\text{restrict-s } d X)) = \text{lift-state } (h, d) \upharpoonright' X$

by (*auto simp: lift-state-def restrict-map-def restrict-s-def split: s-heap-index.splits*)

lemma *dom-merge-restrict* [*simp*]:

$(x ++ y) \upharpoonright' \text{dom } y = y$

by (*rule map-add-restrict-dom-right*)

lemma *dom-compl-restrict* [*simp*]:

$x \upharpoonright' (\text{UNIV} - \text{dom } x) = \text{Map.empty}$

by (*force simp: restrict-map-def*)

lemma *lift-state-point-eq-mod*:
 $\llbracket \text{point-eq-mod } (\text{lift-state } (h,d)) (\text{lift-state } (h',d')) X \rrbracket \Longrightarrow$
 $\text{lift-state } (h,d) \mid' (UNIV - X) =$
 $\text{lift-state } (h',d') \mid' (UNIV - X)$
by (*auto simp: point-eq-mod-def restrict-map-def*)

lemma *htd-'-update-ind* [*simp*]:
 $\text{htd-ind } f \Longrightarrow f (\text{hst-htd-update } x s) = f s$
by (*simp add: htd-ind-def*)

lemma *sep-frame'*:
assumes *orig-spec*: $\forall s. \Gamma \vdash \{s. P (f \text{' } (\lambda x. x)) (\text{lift-hst \text{' } } (\lambda x. x))\}$
 C
 $\{Q (g s \text{' } (\lambda x. x)) (\text{lift-hst \text{' } } (\lambda x. x))\}$
and *hi-f*: $\text{htd-ind } f$ **and** *hi-g*: $\text{htd-ind } g$
and *hi-g'*: $\forall s. \text{htd-ind } (g s)$
and *safe*: $\text{mem-safe } (C::('s::\text{heap-state-type}, 'b, 'c) \text{ com}) \Gamma$
shows $\forall s. \Gamma \vdash \{s. (P (f \text{' } (\lambda x. x)) \wedge^* R (h \text{' } (\lambda x. x))) (\text{lift-hst \text{' } } (\lambda x. x))\}$
 C
 $\{Q (g s \text{' } (\lambda x. x)) \wedge^* R (h s) (\text{lift-hst \text{' } } (\lambda x. x))\}$
apply (*standard, rule hoare-complete, simp only: valid-def, clarify*)

proof –
fix *ta x*
assume *ev*: $\Gamma \vdash \langle C, \text{Normal } x \rangle \Rightarrow ta$ **and**
 $\text{pre}: (P (f x) \wedge^* R (h x)) (\text{lift-hst } x)$
then obtain s_0 **and** s_1 **where** *pre-P*: $P (f x) s_0$ **and** *pre-R*: $R (h x) s_1$ **and**
 $\text{disj}: s_0 \perp s_1$ **and** *m*: $\text{lift-hst } x = s_1 ++ s_0$
by (*clarsimp simp: sep-conj-def map-ac-simps*)
with *orig-spec hi-f* **have** *nofault*: $\neg \text{exec-fatal } C \Gamma$
 $(\text{restrict-htd } x (\text{dom } s_0))$
by (*force simp: exec-fatal-def image-def lift-hst-def cvalid-def valid-def*
 restrict-htd-def
 dest: hoare-sound)
show $ta \in \text{Normal } \{t. (Q (g x t) \wedge^* R (h x)) (\text{lift-hst } t)\}$
proof (*cases ta*)
case (*Normal s*)
moreover from *this ev safe nofault* **have** *ev'*: $\Gamma \vdash$
 $\langle C, \text{Normal } (x \mid \text{hst-htd } := (\text{restrict-s } (\text{hst-htd } x) (\text{dom } s_0)) \mid) \rangle \Rightarrow$
 $\text{Normal } (s \mid \text{hst-htd } := (\text{restrict-s } (\text{hst-htd } s) (\text{dom } s_0)) \mid)$ **and**
 $\text{point-eq-mod } (\text{lift-state } (\text{hst-mem } s, \text{hst-htd } s))$
 $(\text{lift-state } (\text{hst-mem } x, \text{hst-htd } x)) (\text{dom } s_0)$
by (*auto simp: restrict-htd-def dest: mem-safe-NormalD*)
moreover from *this m disj* **have** $s_1 = \text{lift-hst } s \mid' (UNIV - \text{dom } s_0)$
apply (*clarsimp simp: lift-hst-def*)
apply (*subst lift-state-point-eq-mod*)
apply (*fastforce dest: sym*)
apply (*simp add: lift-hst-def lift-state-point-eq-mod map-add-restrict*)
apply (*subst restrict-map-subdom, auto dest: map-disjD*)

```

done
ultimately show ?thesis using orig-spec hi-f hi-g hi-g' pre-P pre-R m
  by (force simp: cvalid-def valid-def image-def lift-hst-def
      map-disj-def
      intro: sep-conjI dest: hoare-sound)
next
case (Abrupt s) with ev safe nofault orig-spec pre-P hi-f m show ?thesis
  apply simp
  apply (drule spec)
  apply (drule hoare-sound)
  apply (drule-tac X=dom s0 in mem-safe-AbruptD)
  by (assumption+, force simp: valid-def cvalid-def lift-hst-def restrict-htd-def)
next
case (Fault f) with ev safe nofault show ?thesis
  by (force dest: mem-safe-FaultD)
next
case Stuck with ev safe nofault show ?thesis
  by (force dest: mem-safe-StuckD)
qed
qed

```

lemma sep-frame:

```

[[ k = (λs. (hst-mem s, hst-htd s));
  ∀ s. Γ ⊢ {s. P (f '(λx. x)) (lift-state (k '(λx. x)))}
  C
  {Q (g s '(λx. x)) (lift-state (k '(λx. x)))};
  htd-ind f; htd-ind g; ∀ s. htd-ind (g s);
  mem-safe (C::('s::heap-state-type, 'b, 'c) com) Γ ]] ⇒
  ∀ s. Γ ⊢ {s. (P (f '(λx. x)) ∧* R (h '(λx. x))) (lift-state (k '(λx. x)))}
  C
  {(Q (g s '(λx. x)) ∧* R (h s)) (lift-state (k '(λx. x)))}

```

```

apply(simp only:)
apply(fold lift-hst-def)
apply(erule (4) sep-frame)
done

```

lemma point-eq-mod-safe [simp]:

```

[[ point-eq-mod-safe P f g; restrict-htd s X ∈ P; x ∉ X ]] ⇒
  g (f s) x = (g s) x
apply(simp add: point-eq-mod-safe-def point-eq-mod-def)
apply(cases x, force)
done

```

lemma comm-restrict-safe [simp]:

```

[[ comm-restrict-safe P f; restrict-htd s X ∈ P ]] ⇒
  restrict-htd (f s) X = f (restrict-htd s X)
by (simp add: comm-restrict-safe-def comm-restrict-def)

```


lemma *mono-guardD*:

$\llbracket \text{mono-guard } P; \text{ restrict-htd } s \ X \in P \rrbracket \implies s \in P$
by (*unfold mono-guard-def, fast*)

lemma *expr-htd-ind*:

$\text{expr-htd-ind } P \implies \text{restrict-htd } s \ X \in P = (s \in P)$
by (*simp add: expr-htd-ind-def restrict-htd-def*)

lemmas *exec-other-intros* = *exec.intros(1–3) exec.intros(5–14) exec.intros(16–17)*
exec.intros(19–)

lemma *exec-fatal-Seq*:

$\text{exec-fatal } C \ \Gamma \ s \implies \text{exec-fatal } (C;;D) \ \Gamma \ s$
by (*force simp: exec-fatal-def intro: exec-other-intros*)

lemma *exec-fatal-Seq2*:

$\llbracket \Gamma \vdash \langle C, \text{Normal } s \rangle \Rightarrow \text{Normal } t; \text{exec-fatal } D \ \Gamma \ t \rrbracket \implies \text{exec-fatal } (C;;D) \ \Gamma \ s$
by (*force simp: exec-fatal-def intro: exec-other-intros*)

lemma *exec-fatal-Cond*:

$\text{exec-fatal } (\text{Cond } P \ C \ D) \ \Gamma \ s = (\text{if } s \in P \ \text{then } \text{exec-fatal } C \ \Gamma \ s \ \text{else } \text{exec-fatal } D \ \Gamma \ s)$
by (*force simp: exec-fatal-def intro: exec-other-intros*
elim: exec-Normal-elim-cases)

lemma *exec-fatal-While*:

$\llbracket \text{exec-fatal } C \ \Gamma \ s; s \in P \rrbracket \implies \text{exec-fatal } (\text{While } P \ C) \ \Gamma \ s$
by (*force simp: exec-fatal-def intro: exec-other-intros*
elim: exec-Normal-elim-cases)

lemma *exec-fatal-While2*:

$\llbracket \text{exec-fatal } (\text{While } P \ C) \ \Gamma \ t; \Gamma \vdash \langle C, \text{Normal } s \rangle \Rightarrow \text{Normal } t; s \in P \rrbracket \implies$
 $\text{exec-fatal } (\text{While } P \ C) \ \Gamma \ s$
by (*force simp: exec-fatal-def intro: exec-other-intros*
elim: exec-Normal-elim-cases)

lemma *exec-fatal-Call*:

$\llbracket \Gamma \ p = \text{Some } C; \text{exec-fatal } C \ \Gamma \ s \rrbracket \implies \text{exec-fatal } (\text{Call } p) \ \Gamma \ s$
by (*force simp: exec-fatal-def intro: exec-other-intros*)

lemma *exec-fatal-DynCom*:

$\text{exec-fatal } (f \ s) \ \Gamma \ s \implies \text{exec-fatal } (\text{DynCom } f) \ \Gamma \ s$
by (*force simp: exec-fatal-def intro: exec-other-intros*)

lemma *exec-fatal-Guard*:

$\text{exec-fatal } (\text{Guard } f \ P \ C) \ \Gamma \ s = (s \in P \longrightarrow \text{exec-fatal } C \ \Gamma \ s)$

proof (*cases s ∈ P*)

case *True thus ?thesis*

by (force simp: exec-fatal-def intro: exec-other-intros
 elim: exec-Normal-elim-cases)

next
 case *False* **thus** ?thesis
 by (force simp: exec-fatal-def intro: exec-other-intros)

qed

lemma *restrict-safe-Guard*:
 assumes *restrict*: *restrict-safe s t C* Γ
 shows *restrict-safe s t (Guard f P C)* Γ
proof (*cases t*)
 case (*Normal s*) **with** *restrict* **show** ?thesis
 by (force simp: restrict-safe-def restrict-safe-OK-def exec-fatal-Guard
 intro: exec-other-intros)

next
 case (*Abrupt s*) **with** *restrict* **show** ?thesis
 by (force simp: restrict-safe-def restrict-safe-OK-def exec-fatal-Guard
 intro: exec-other-intros)

next
 case (*Fault f*) **with** *restrict* **show** ?thesis
 by (force simp: restrict-safe-def exec-fatal-Guard)

next
 case *Stuck* **with** *restrict* **show** ?thesis
 by (force simp: restrict-safe-def exec-fatal-Guard)

qed

lemma *restrict-safe-Guard2*:
 $\llbracket s \notin P; \text{mono-guard } P \rrbracket \implies \text{restrict-safe } s \text{ (Fault } f \text{) (Guard } f \text{ } P \text{ } C) \Gamma$
by (force simp: restrict-safe-def exec-fatal-def intro: exec-other-intros
 dest: mono-guardD)

lemma *exec-fatal-Catch*:
exec-fatal C Γ *s* \implies *exec-fatal (TRY C CATCH D END)* Γ *s*
by (force simp: exec-fatal-def intro: exec-other-intros)

lemma *exec-fatal-Catch2*:
 $\llbracket \Gamma \vdash \langle C, \text{Normal } s \rangle \Rightarrow \text{Abrupt } t; \text{exec-fatal } D \Gamma t \rrbracket \implies$
exec-fatal (TRY C CATCH D END) Γ *s*
by (force simp: exec-fatal-def intro: exec-other-intros)

lemma *intra-safe-restrict* [*rule-format*]:
 assumes *safe-env*: $\bigwedge n. C. \Gamma n = \text{Some } C \implies \text{intra-safe } C$ **and**
 exec: $\Gamma \vdash \langle C, s \rangle \Rightarrow t$
 shows $\forall s'. s = \text{Normal } s' \longrightarrow \text{intra-safe } C \longrightarrow \text{restrict-safe } s' t C \Gamma$
using *exec*
proof *induct*
 case (*Skip s*) **thus** ?case
 by (force simp: restrict-safe-def restrict-safe-OK-def intro: exec-other-intros)

next

```

case (Guard s' P C t f) show ?case
proof (cases  $\exists g. C = \text{Basic } g$ )
  case False with Guard show ?thesis
    by - (clarsimp, split com.splits, auto dest: restrict-safe-Guard)
next
  case True with Guard show ?thesis
    by (cases t) (force simp: restrict-safe-def restrict-safe-OK-def
      point-eq-mod-safe-def exec-fatal-Guard
      intro: exec-other-intros
      elim: exec-Normal-elim-cases,
      (fast elim: exec-Normal-elim-cases)+)
qed
next
  case (GuardFault C f P s) thus ?case
    by (force dest: restrict-safe-Guard2)
next
  case (FaultProp C f) thus ?case by simp
next
  case (Basic f s) thus ?case
    by (force simp: restrict-safe-def restrict-safe-OK-def point-eq-mod-safe-def
      intro: exec-other-intros)
next
  case (Spec r s t) thus ?case
    by (fastforce simp: mem-safe-def intro: exec.Spec)
next
  case (SpecStuck r s) thus ?case
    by (simp add: exec.SpecStuck mem-safe-StuckD restrict-safe-def)
next
  case (Seq C s sa D ta) show ?case
proof (cases sa)
  case (Normal s') with Seq show ?thesis
    by (cases ta)
      (clarsimp simp: restrict-safe-def restrict-safe-OK-def,
      (drule-tac x=X in spec)+, auto intro: exec-other-intros point-eq-mod-trans
      exec-fatal-Seq exec-fatal-Seq2)+
next
  case (Abrupt s') with Seq show ?thesis
    by (force simp: restrict-safe-def restrict-safe-OK-def
      intro: exec-other-intros dest: exec-fatal-Seq
      elim: exec-Normal-elim-cases)
next
  case (Fault f) with Seq show ?thesis
    by (force simp: restrict-safe-def dest: exec-fatal-Seq
      elim: exec-Normal-elim-cases)
next
  case Stuck with Seq show ?thesis
    by (force simp: restrict-safe-def dest: exec-fatal-Seq
      elim: exec-Normal-elim-cases)
qed

```

```

next
  case (CondTrue s P C t D) thus ?case
  by (cases t)
    (auto simp: restrict-safe-def restrict-safe-OK-def exec-fatal-Cond
      intro: exec-other-intros dest: expr-htd-ind split: if-split-asm)
next
  case (CondFalse s P C t D) thus ?case
  by (cases t)
    (auto simp: restrict-safe-def restrict-safe-OK-def exec-fatal-Cond
      intro: exec-other-intros dest: expr-htd-ind split: if-split-asm)
next
  case (WhileTrue P C s s' t) show ?case
  proof (cases s')
    case (Normal sa) with WhileTrue show ?thesis
    by (cases t)
      (clarsimp simp: restrict-safe-def restrict-safe-OK-def,
        (drule-tac x=X in spec)+, auto simp: expr-htd-ind intro: exec-other-intros
        point-eq-mod-trans exec-fatal-While exec-fatal-While2)+
  next
    case (Abrupt sa) with WhileTrue show ?thesis
    by (force simp: restrict-safe-def restrict-safe-OK-def expr-htd-ind
      intro: exec-other-intros exec-fatal-While
      elim: exec-Normal-elim-cases)
  next
    case (Fault f) with WhileTrue show ?thesis
    by (force simp: restrict-safe-def expr-htd-ind intro: exec-fatal-While)
  next
    case Stuck with WhileTrue show ?thesis
    by (force simp: restrict-safe-def expr-htd-ind intro: exec-fatal-While)
  qed
next
  case (WhileFalse P C s) thus ?case
  by (force simp: restrict-safe-def restrict-safe-OK-def expr-htd-ind
    intro: exec-other-intros)
next
  case (Call C p s t) with safe-env show ?case
  by (cases t)
    (auto simp: restrict-safe-def restrict-safe-OK-def
      intro: exec-fatal-Call exec-other-intros)
next
  case (CallUndefined p s) thus ?case
  by (force simp: restrict-safe-def exec-fatal-def intro: exec-other-intros)
next
  case (StuckProp C) thus ?case by simp
next
  case (DynCom f s t) thus ?case
  by (cases t)
    (auto simp: restrict-safe-def restrict-safe-OK-def

```

restrict-htd-def
intro!: exec-other-intros exec-fatal-DynCom)

next
case (*Throw s*) **thus** *?case*
by (*force simp: restrict-safe-def restrict-safe-OK-def intro: exec-other-intros*)

next
case (*AbruptProp C s*) **thus** *?case by simp*

next
case (*CatchMatch C D s s' t*) **thus** *?case*
by (*cases t*)
(*clarsimp simp: restrict-safe-def, drule-tac x=X in spec,*
auto simp: restrict-safe-OK-def intro: exec-other-intros point-eq-mod-trans
dest: exec-fatal-Catch exec-fatal-Catch2)**+**

next
case (*CatchMiss C s t D*) **thus** *?case*
by (*cases t*)
(*clarsimp simp: restrict-safe-def, drule-tac x=X in spec,*
auto simp: restrict-safe-OK-def intro: exec-other-intros
dest: exec-fatal-Catch)**+**

qed

lemma *intra-mem-safe*:
 $\llbracket \bigwedge n C. \Gamma n = \text{Some } C \implies \text{intra-safe } C; \text{intra-safe } C \rrbracket \implies \text{mem-safe } C \Gamma$
by (*force simp: mem-safe-def intro: intra-safe-restrict*)

lemma *point-eq-mod-safe-triv*:
 $(\bigwedge s. g (f s) = g s) \implies \text{point-eq-mod-safe } P f g$
by (*simp add: point-eq-mod-safe-def point-eq-mod-def*)

lemma *comm-restrict-safe-triv*:
 $(\bigwedge s X. f (s \llbracket \text{hst-htd} := \text{restrict-s } (\text{hst-htd } s) X \rrbracket)) =$
 $(f s) \llbracket \text{hst-htd} := \text{restrict-s } (\text{hst-htd } (f s)) X \rrbracket) \implies \text{comm-restrict-safe } P f$
by (*force simp: comm-restrict-safe-def comm-restrict-def restrict-htd-def*)

lemma *mono-guard-UNIV* [*simp*]:
mono-guard UNIV
by (*force simp: mono-guard-def*)

lemma *mono-guard-triv*:
 $(\bigwedge s X. s \llbracket \text{hst-htd} := X \rrbracket \in g \implies s \in g) \implies \text{mono-guard } g$
by (*unfold mono-guard-def, unfold restrict-htd-def, fast*)

lemma *mono-guard-triv2*:
 $(\bigwedge s X. s \llbracket \text{hst-htd} := X \rrbracket \in g = ((s::'\text{a}::\text{heap-state-type}') \in g)) \implies$
mono-guard g
by (*unfold mono-guard-def, unfold restrict-htd-def, fast*)

lemma *dom-restrict-s*:
 $x \in \text{dom-s } (\text{restrict-s } d X) \implies x \in \text{dom-s } d \wedge x \in X$

by (*auto simp: restrict-s-def dom-s-def split: if-split-asm*)

lemma *mono-guard-ptr-safe*:

$\llbracket \bigwedge s. d\ s = \text{hst-htd} :: 'a :: \text{heap-state-type}; \text{htd-ind } p \rrbracket \implies$
 $\text{mono-guard } \{s. \text{ptr-safe } (p\ s) (d\ s)\}$

by (*auto simp: mono-guard-def ptr-safe-def restrict-htd-def dest: subsetD dom-restrict-s*)

lemma *point-eq-mod-safe-ptr-safe-update*:

$\llbracket d = (\text{hst-htd} :: 'a :: \text{heap-state-type} \Rightarrow \text{heap-tyr-desc});$
 $m = (\lambda s. \text{hst-mem-update } (\text{heap-update } (p\ s) ((v\ s) :: 'b :: \text{mem-type}))\ s);$
 $h = \text{hst-mem}; k = (\lambda s. \text{lift-state } (h\ s, d\ s)); \text{htd-ind } p \rrbracket \implies$
 $\text{point-eq-mod-safe } \{s. \text{ptr-safe } (p\ s) (d\ s)\}\ m\ k$

apply (*clarsimp simp: point-eq-mod-safe-def point-eq-mod-def ptr-safe-def heap-update-def*
restrict-htd-def lift-state-def
intro!: heap-update-nmem-same
split: s-heap-index.splits)

apply(*subgoal-tac (a, SIndexVal) ∈ s-footprint (p s)*)

apply(*drule (1) subsetD*)

apply(*drule dom-restrict-s, clarsimp*)

apply(*drule introD, clarsimp*)

apply(*erule s-footprintI2*)

done

lemma *field-ti-s-sub-tyr*:

$\text{field-lookup } (\text{export-uinfo } (\text{typ-info-t } \text{TYPE}('b :: \text{mem-type})))\ f\ 0 = \text{Some } (\text{typ-uinfo-t}$
 $\text{TYPE}('a), b) \implies$

$s\text{-footprint } ((\text{Ptr } \&(p \rightarrow f)) :: 'a :: \text{mem-type } \text{ptr}) \subseteq s\text{-footprint } (p :: 'b\ \text{ptr})$

by (*drule field-ti-s-sub (simp add: s-footprint-def)*)

lemma *ptr-safe-mono*:

$\llbracket \text{ptr-safe } (p :: 'a :: \text{mem-type } \text{ptr})\ d; \text{field-lookup } (\text{typ-info-t } \text{TYPE}('a))\ f\ 0$
 $= \text{Some } (t, n); \text{export-uinfo } t = \text{typ-uinfo-t } \text{TYPE}('b) \rrbracket \implies$
 $\text{ptr-safe } ((\text{Ptr } \&(p \rightarrow f)) :: 'b :: \text{mem-type } \text{ptr})\ d$

unfolding *ptr-safe-def*

by (*drule field-lookup-export-uinfo-Some (auto dest: field-ti-s-sub-tyr)*)

lemma *point-eq-mod-safe-ptr-safe-update-fl*:

$\llbracket d = (\text{hst-htd} :: 'a :: \text{heap-state-type} \Rightarrow \text{heap-tyr-desc});$
 $m = (\lambda s. \text{hst-mem-update } (\text{heap-update } (\text{Ptr } \&((p\ s) \rightarrow f)) ((v\ s) :: 'b :: \text{mem-type}))$
 $s);$

$h = \text{hst-mem}; k = (\lambda s. \text{lift-state } (h\ s, d\ s)); \text{htd-ind } p;$

$\text{field-lookup } (\text{typ-info-t } \text{TYPE}('c))\ f\ 0 = \text{Some } (t, n);$

$\text{export-uinfo } t = \text{typ-uinfo-t } \text{TYPE}('b) \rrbracket \implies$

$\text{point-eq-mod-safe } \{s. \text{ptr-safe } ((p :: 'a \Rightarrow 'c :: \text{mem-type } \text{ptr})\ s) (d\ s)\}\ m\ k$

apply(*drule (3) point-eq-mod-safe-ptr-safe-update*)

apply(*fastforce simp: htd-ind-def*)

apply(*fastforce simp: point-eq-mod-safe-def intro!: ptr-safe-mono*)

done

context
begin

private method $m =$

(*clarsimp simp: ptr-retyp-d-eq-snd ptr-retyp-footprint list-map-eq,*
erule notE,
drule intvlD, clarsimp,
(rule s-footprintI; assumption?),
subst (asm) unat-of-nat,
(subst (asm) mod-less; assumption?),
subst len-of-addr-card,
erule less-trans,
simp)

lemma *point-eq-mod-safe-ptr-safe-tag:*

$\llbracket d = (\text{hst-htd}::'a::\text{heap-state-type} \Rightarrow \text{heap-typ-desc}); h = \text{hst-mem};$
 $m = (\lambda s. \text{hst-htd-update } (\text{ptr-retyp } (p \ s)) \ s);$
 $k = (\lambda s. \text{lift-state } (h \ s, d \ s));$
 $\text{htd-ind } p \rrbracket \Longrightarrow$

point-eq-mod-safe $\{s. \text{ptr-safe } ((p \ s)::'b::\text{mem-type } \text{ptr}) \ (d \ s)\} \ m \ k$

supply *if-split-asm*[*split*]

supply *unsigned-of-nat* [*simp del*]

apply(*clarsimp simp: point-eq-mod-safe-def point-eq-mod-def ptr-safe-def*)

apply(*subgoal-tac* ($a, b \notin s\text{-footprint } (p \ (\text{restrict-htd } s \ X))$))

prefer 2

apply(*fastforce simp: restrict-htd-def dest: dom-restrict-s*)

apply(*clarsimp simp: restrict-htd-def lift-state-def split: s-heap-index.split option.splits*)

apply (*safe; m?*)

apply(*fastforce simp: ptr-retyp-d-eq-fst dest!: intvlD dest: s-footprintI2*)

apply(*fastforce simp: ptr-retyp-d-eq-fst*)

apply(*subst (asm) ptr-retyp-d-eq-snd, clarsimp*)

done

end

lemma *comm-restrict-safe-ptr-safe-tag:*

fixes $d::'a::\text{heap-state-type} \Rightarrow \text{heap-typ-desc}$

assumes

fun-d: $d = \text{hst-htd}$ **and**

fun-upd: $m = (\lambda s. \text{hst-htd-update } (\text{ptr-retyp } (p \ s)) \ s)$ **and**

ind: $\text{htd-ind } p$ **and**

upd: $\bigwedge d \ d' \ (s::'a).$

$\text{hst-htd-update } (d \ s) \ (\text{hst-htd-update } (d' \ s) \ s) = \text{hst-htd-update } ((d \ s) \circ$

$(d' \ s)) \ s$

shows *comm-restrict-safe* $\{s. \text{ptr-safe } ((p \ s)::'b::\text{mem-type } \text{ptr}) \ (d \ s)\} \ m$

proof –

{

fix $s \ X$

```

assume ptr-safe (p (restrict-htd s X)) (d (restrict-htd s X))
moreover from ind
have p: p (restrict-htd s X) = p s
  by (simp add: restrict-htd-def)
ultimately
  have ptr-retyp (p s) (restrict-s (hst-htd s) X) = restrict-s (ptr-retyp (p s)
(hst-htd s)) X
  using fun-d
  supply unsigned-of-nat [simp del]
  apply –
  apply(rule ext)
  apply(clarsimp simp: point-eq-mod-safe-def point-eq-mod-def ptr-safe-def re-
strict-htd-def)
  subgoal for x
    apply(cases x ∉ {ptr-val (p s)..+size-of TYPE('b)})
    apply(clarsimp simp: ptr-retyp-d restrict-map-def restrict-s-def)
    apply(subst ptr-retyp-d; simp)
    apply(clarsimp simp: ptr-retyp-footprint restrict-map-def restrict-s-def)
    apply(subst ptr-retyp-footprint, simp)
    apply(rule conjI)
    apply(subgoal-tac (x,SIndexVal) ∈ s-footprint (p s))
    apply(fastforce simp: dom-s-def)
    apply(fastforce dest: intvlD elim: s-footprintI2)
    apply(rule ext)
    apply(clarsimp simp: map-add-def list-map-eq)
    apply(subgoal-tac (x,SIndexTyp y) ∈ s-footprint (p s))
    apply(fastforce simp: dom-s-def split: if-split-asm)
    apply(drule intvlD, clarsimp)
    apply(rule s-footprintI; assumption?)
    apply(metis len-of-addr-card less-trans max-size mod-less word-unat.eq-norm)
  done
done
hence ((ptr-retyp (p s) ∘ (λx -. x)) (restrict-s (hst-htd s) X))::heap-typ-desc ⇒
heap-typ-desc =
  (λx -. x) (restrict-s (ptr-retyp (p s) (hst-htd s)) X)
  by – (rule ext, simp)
moreover from upd have hst-htd-update (ptr-retyp (p s))
  (hst-htd-update ((λx -. x) (restrict-s (hst-htd s) X)) s) =
  hst-htd-update (((ptr-retyp (p s)) ∘ ((λx -. x) (restrict-s (hst-htd s) X)))) s .
moreover from upd
have
  hst-htd-update ((λx -. x) (restrict-s (ptr-retyp (p s) (hst-htd s)) X))
  (hst-htd-update (ptr-retyp (p s)) s) =
  hst-htd-update (((λx -. x) (restrict-s ((ptr-retyp (p s) (hst-htd s))) X)) ∘
(ptr-retyp (p s))) s .
  ultimately have m (restrict-htd s X) = restrict-htd (m s) X using fun-d
fun-upd upd p
  by (simp add: restrict-htd-def o-def)
}

```



```

thus ?thesis
  by (simp only: comm-restrict-safe-def comm-restrict-def, auto)
qed

lemmas intra-sc = hrs-comm comp-def hrs-htd-update-htd-update
  point-eq-mod-safe-triv comm-restrict-safe-triv mono-guard-triv2
  mono-guard-ptr-safe point-eq-mod-safe-ptr-safe-update
  point-eq-mod-safe-ptr-safe-tag comm-restrict-safe-ptr-safe-tag
  point-eq-mod-safe-ptr-safe-update-fl

declare expr-htd-ind-def [iff]
declare htd-ind-def [iff]

lemma proc-deps-Skip [simp]:
  proc-deps Skip  $\Gamma = \{\}$ 
  by (force elim: proc-deps.induct)

lemma proc-deps-Basic [simp]:
  proc-deps (Basic f)  $\Gamma = \{\}$ 
  by (force elim: proc-deps.induct)

lemma proc-deps-Spec [simp]:
  proc-deps (Spec r)  $\Gamma = \{\}$ 
  by (force elim: proc-deps.induct)

lemma proc-deps-Seq [simp]:
  proc-deps (Seq C D)  $\Gamma = \text{proc-deps } C \ \Gamma \cup \text{proc-deps } D \ \Gamma$ 
proof
  show proc-deps (C;; D)  $\Gamma \subseteq \text{proc-deps } C \ \Gamma \cup \text{proc-deps } D \ \Gamma$ 
    by - (standard, erule proc-deps.induct, auto intro: proc-deps.intros)
  next
  show proc-deps C  $\Gamma \cup \text{proc-deps } D \ \Gamma \subseteq \text{proc-deps } (C;; D) \ \Gamma$ 
    by auto (erule proc-deps.induct, auto intro: proc-deps.intros)+
qed

lemma proc-deps-Cond [simp]:
  proc-deps (Cond P C D)  $\Gamma = \text{proc-deps } C \ \Gamma \cup \text{proc-deps } D \ \Gamma$ 
proof
  show proc-deps (Cond P C D)  $\Gamma \subseteq \text{proc-deps } C \ \Gamma \cup \text{proc-deps } D \ \Gamma$ 
    by (standard, erule proc-deps.induct, auto intro: proc-deps.intros)
  next
  show proc-deps C  $\Gamma \cup \text{proc-deps } D \ \Gamma \subseteq \text{proc-deps } (Cond P C D) \ \Gamma$ 
    by auto (erule proc-deps.induct, auto intro: proc-deps.intros)+
qed

lemma proc-deps-While [simp]:
  proc-deps (While P C)  $\Gamma = \text{proc-deps } C \ \Gamma$ 
  by auto (erule proc-deps.induct, auto intro: proc-deps.intros)+

```

```

lemma proc-deps-Guard [simp]:
  proc-deps (Guard f P C)  $\Gamma = \text{proc-deps } C \Gamma$ 
  by auto (erule proc-deps.induct, auto intro: proc-deps.intros)+

lemma proc-deps-Throw [simp]:
  proc-deps Throw  $\Gamma = \{\}$ 
  by (force elim: proc-deps.induct)

lemma proc-deps-Catch [simp]:
  proc-deps (Catch C D)  $\Gamma = \text{proc-deps } C \Gamma \cup \text{proc-deps } D \Gamma$ 
proof
  show proc-deps (Catch C D)  $\Gamma \subseteq \text{proc-deps } C \Gamma \cup \text{proc-deps } D \Gamma$ 
    by (standard, erule proc-deps.induct, auto intro: proc-deps.intros)
next
  show proc-deps C  $\Gamma \cup \text{proc-deps } D \Gamma \subseteq \text{proc-deps } (\text{Catch } C D) \Gamma$ 
    by auto (erule proc-deps.induct, auto intro: proc-deps.intros)+
qed

lemma proc-deps-Call [simp]:
  proc-deps (Call p)  $\Gamma = \{p\} \cup (\text{case } \Gamma \text{ } p \text{ of Some } C \Rightarrow$ 
    proc-deps C ( $\Gamma(p := \text{None})$ ) |  $- \Rightarrow \{\}$ ) (is  $?X = ?Y \cup ?Z$ )
proof
  note proc-deps.intros[intro]
  show  $?X \subseteq ?Y \cup ?Z$ 
    apply (rule subsetI, erule proc-deps.induct, fastforce)
    subgoal for  $x' x D y$ 
      apply (cases x=p)
      apply (auto split: option.splits)
      done
    done
next
  show  $?Y \cup ?Z \subseteq ?X$ 
    apply (clarsimp, standard)
  proof  $-$ 
    show  $p \in ?X$  by (force intro: proc-deps.intros)
  next
  show  $?Z \subseteq ?X$ 
    by (split option.splits, standard, force intro: proc-deps.intros)
      (clarify, erule proc-deps.induct;
        force intro: proc-deps.intros split: if-split-asm)
qed
qed

lemma proc-deps-DynCom [simp]:
  proc-deps (DynCom f)  $\Gamma = \bigcup \{\text{proc-deps } (f s) \Gamma \mid s. \text{True}\}$ 
  by (rule equalityI;clarsimp;erule proc-deps.induct;force intro: proc-deps.intros)

lemma proc-deps-restrict:
  proc-deps C  $\Gamma \subseteq \text{proc-deps } C (\Gamma(p := \text{None})) \cup \text{proc-deps } (\text{Call } p) \Gamma$ 

```

```

proof
  fix  $xa$ 
  assume  $mem: xa \in proc-deps\ C\ \Gamma$ 
  hence  $\forall p. xa \in proc-deps\ C\ (\Gamma(p := None)) \cup proc-deps\ (Call\ p)\ \Gamma$  (is ?X)
  using  $mem$ 
  proof induct
    fix  $x$ 
    assume  $x \in intra-deps\ C$ 
    thus  $\forall p. x \in proc-deps\ C\ (\Gamma(p := None)) \cup proc-deps\ (Call\ p)\ \Gamma$ 
      by (force intro: proc-deps.intros)
    next
      fix  $D\ x\ y$ 
      assume  $x:$ 
         $x \in proc-deps\ C\ \Gamma$ 
         $x \in proc-deps\ C\ \Gamma \implies \forall p. x \in proc-deps\ C\ (\Gamma(p := None)) \cup proc-deps\ (Call$ 
  p)  $\Gamma$ 
         $\Gamma\ x = Some\ D$ 
         $y \in intra-deps\ D$ 
         $y \in proc-deps\ C\ \Gamma$ 
        show  $\forall p. y \in proc-deps\ C\ (\Gamma(p := None)) \cup proc-deps\ (Call\ p)\ \Gamma$ 
        proof clarify
          fix  $p$ 
          assume  $y: y \notin proc-deps\ (Call\ p)\ \Gamma$ 
          show  $y \in proc-deps\ C\ (\Gamma(p := None))$ 
          proof (cases x=p)
            case True with x y show ?thesis
              by (force intro: proc-deps.intros)
            next
              case False with x y show ?thesis
                by (clarsimp, drule-tac x=p in spec)
                  (auto intro: proc-deps.intros split: option.splits)
          qed
        qed
      thus  $xa \in proc-deps\ C\ (\Gamma(p := None)) \cup proc-deps\ (Call\ p)\ \Gamma$  by simp
  qed

```

```

lemma exec-restrict:
  assumes  $exec: \Gamma' \vdash \langle C, s \rangle \Rightarrow t$ 
  shows  $\bigwedge \Gamma\ X. \llbracket \Gamma' = \Gamma \mid X; proc-deps\ C\ \Gamma \subseteq X \rrbracket \implies \Gamma \vdash \langle C, s \rangle \Rightarrow t$ 
  using  $exec$ 
  proof induct
    case (Call p C s t)
    thus ?case
    using proc-deps-restrict [of C Γ p] by (force intro: exec-other-intros)
  qed (force intro: exec-other-intros)+

```

```

lemma exec-restrict2:
  assumes  $exec: \Gamma \vdash \langle C, s \rangle \Rightarrow t$ 

```

```

  shows  $\bigwedge X. \text{proc-deps } C \Gamma \subseteq X \implies \Gamma \mid \langle C, s \rangle \Rightarrow t$ 
using exec
proof induct
  case (Call p C s t) thus ?case
  using proc-deps-restrict [of C  $\Gamma$  p]
  by (auto intro!: exec-other-intros split: option.splits)
next
  case (DynCom f s t) thus ?case
  by - (rule exec.intros, simp, blast)
qed (auto intro: exec-other-intros)

```

```

lemma exec-restrict-eq:
 $\Gamma \mid \langle \text{proc-deps } C \Gamma \rangle \Rightarrow t = \Gamma \mid \langle C, s \rangle \Rightarrow t$ 
by (fast intro: exec-restrict exec-restrict2)

```

```

lemma mem-safe-restrict:
  mem-safe C  $\Gamma$  = mem-safe C ( $\Gamma \mid \langle \text{proc-deps } C \Gamma \rangle$ )
  by (auto simp: mem-safe-def restrict-safe-def restrict-safe-OK-def
    exec-restrict-eq exec-fatal-def
    split: xstate.splits)

```

end

```

theory StructSupport
imports SepCode SepInv
begin

```

```

lemma field-lookup-list-Some2:
   $fn \notin \text{dt-snd } \langle \text{set } ts \rangle \implies$ 
   $\text{field-lookup-list } (ts@[DTuple t fn d]) (fn \# fs) m = \text{field-lookup } t fs (m +$ 
   $\text{size-td-list } ts)$ 
  apply (induct ts arbitrary: m; clarsimp split: option.splits)
  subgoal for a list m
  apply (safe; cases a; clarsimp simp: ac-simps split: if-split-asm)
  done
done

```

```

lemma field-lookup-list-mismatch:
   $f \neq fn \implies \text{field-lookup-list } (ts@[DTuple t f d]) (fn \# fs) m =$ 
   $\text{field-lookup-list } ts (fn \# fs) m$ 
  by (induct ts arbitrary: m, auto split: option.split)

```

```

lemma fnl-set:
   $\text{set } (\text{CompoundCTypes.field-names-list } (\text{TypDesc } \text{algn } (\text{TypAggregate } xs) tn)) =$ 
   $\text{dt-snd } \langle \text{set } xs \rangle$ 
  by (auto simp: CompoundCTypes.field-names-list-def)

```

```

lemma fnl-extend-ti:

```

```

[[ fn ∉ set (CompoundCTypes.field-names-list tag); aggregate tag ]] ⇒
  field-lookup (extend-ti tag t algn fn d) (f # fs) m =
    (if f=fn then field-lookup t fs (size-td tag+m) else field-lookup tag (f # fs) m)
apply(cases tag)
subgoal for x1 typ-struct xs
  apply(cases typ-struct; simp)
  apply(simp add: ac-simps field-lookup-list-Some2 fnl-set)
  apply(clarsimp simp: field-lookup-list-append split: option.splits)
done
done

```

lemma *fnl-extend-ti-mismatch*:

```

f ≠ fn ⇒ field-lookup (extend-ti tag t algn fn d) (f # fs) m = field-lookup tag
(f # fs) m
apply (cases tag)
subgoal for x1 typ-struct xs
  apply (cases typ-struct)
  apply clarsimp
  apply (simp add: field-lookup-list-mismatch)
done
done

```

lemma *ft-ti-pad-combine*:

```

[[ hd f ≠ CHR "!" ; aggregate tag ]] ⇒
  field-lookup (ti-pad-combine n tag) (f#fs) m = field-lookup tag (f#fs) m
by (auto simp: ti-pad-combine-def Let-def fnl-extend-ti foldl-append-nmem)

```

lemma *ft-ti-typ-combine*:

```

[[ fn ∉ set (CompoundCTypes.field-names-list tag); aggregate tag ]] ⇒
  field-lookup (ti-typ-combine (t-b::'b::c-type itself) f-ab f-upd-ab algn fn tag)
(f#fs) m =
  (if f=fn
   then field-lookup (adjust-ti (typ-info-t TYPE('b)) f-ab f-upd-ab) fs (size-td
tag+m)
   else field-lookup tag (f # fs) m)
by (simp add: ti-typ-combine-def Let-def fnl-extend-ti)

```

lemma *ft-ti-typ-combine-match*:

```

[[ fn ∉ set (CompoundCTypes.field-names-list tag); aggregate tag ]] ⇒
  field-lookup (ti-typ-combine (t-b::'b::c-type itself) f-ab f-upd-ab algn fn tag)
(fn#fs) m =
  field-lookup (adjust-ti (typ-info-t TYPE('b)) f-ab f-upd-ab) fs (size-td tag+m)
by (simp add: ft-ti-typ-combine)

```

lemma *ft-ti-typ-combine-mismatch*:

```

[[ f ≠ fn ]] ⇒
  field-lookup (ti-typ-combine (t-b::'b::c-type itself) f-ab f-upd-ab algn fn tag) (f#fs)
m =

```

field-lookup tag (f # fs) m
unfolding *ti-typ-combine-def Let-def by (erule fnl-extend-ti-mismatch)*

lemma *fl-ti-typ-pad-combine:*

\llbracket *fn* \notin *set (CompoundCTypes.field-names-list tag); hd f* \neq *CHR "!"*; *hd fn* \neq *CHR "!"*;

aggregate tag $\rrbracket \implies$

field-lookup (ti-typ-pad-combine (t-b::'b::c-type itself) f-ab f-upd-ab algn fn tag)

(f#fs) m =

(if f=fn

then field-lookup (adjust-ti (typ-info-t TYPE('b)) f-ab f-upd-ab) fs

(padup (max (2 ^ algn) (align-of TYPE('b)))

(size-td tag) + size-td tag+m)

else field-lookup tag (f # fs) m)

unfolding *ti-typ-pad-combine-def Let-def*

by (*subst fl-ti-typ-combine; clarsimp*) (*simp add: fl-ti-typ-pad-combine size-td-ti-pad-combine*)

lemma *fl-ti-typ-pad-combine-match:*

\llbracket *fn* \notin *set (CompoundCTypes.field-names-list tag); hd fn* \neq *CHR "!"*;

aggregate tag $\rrbracket \implies$

field-lookup (ti-typ-pad-combine (t-b::'b::c-type itself) f-ab f-upd-ab algn fn tag)

(fn#fs) m =

field-lookup (adjust-ti (typ-info-t TYPE('b)) f-ab f-upd-ab) fs

(padup (max (2 ^ algn) (align-of TYPE('b)))

(size-td tag) + size-td tag+m)

by (*simp add: fl-ti-typ-pad-combine*)

lemma *fl-ti-typ-pad-combine-mismatch:*

\llbracket *hd f* \neq *CHR "!"*;

aggregate tag; f \neq *fn* $\rrbracket \implies$

field-lookup (ti-typ-pad-combine (t-b::'b::c-type itself) f-ab f-upd-ab algn fn tag)

(f#fs) m =

field-lookup tag (f # fs) m

apply(*unfold ti-typ-pad-combine-def Let-def*)

apply(*subst fl-ti-typ-combine-mismatch*)

apply *assumption*

apply(*simp add: fl-ti-typ-pad-combine size-td-ti-pad-combine*)

done

lemma *field-lookup-map-align-cons: field-lookup (map-align g t) (f#fs) m = field-lookup t (f#fs) m*

by (*cases t*) *clarsimp*

lemma *fl-final-pad:*

\llbracket *hd f* \neq *CHR "!"*; *aggregate tag* $\rrbracket \implies$

field-lookup (final-pad algn tag) (f#fs) m = field-lookup tag (f#fs) m

by (*clarsimp simp: final-pad-def Let-def fl-ti-typ-pad-combine field-lookup-map-align-cons*)

lemma *field-lookup-adjust-ti2' [rule-format]:*

$\forall fn\ m\ s\ n. \text{field-lookup}\ ti\ fn\ m = \text{Some}\ (s,n) \longrightarrow$
 $(\text{field-lookup}\ (\text{adjust-ti}\ ti\ f\ g)\ fn\ m = \text{Some}\ (\text{adjust-ti}\ s\ f\ g,n))$
 $\forall fn\ m\ s\ n. \text{field-lookup-struct}\ st\ fn\ m = \text{Some}\ (s,n) \longrightarrow$
 $\text{field-lookup-struct}\ (\text{map-td-struct}\ (\lambda n\ \text{algn}\ d. \text{update-desc}\ f\ g\ d)\ (\text{update-desc}\ f\ g)\ st)\ fn\ m = \text{Some}\ (\text{adjust-ti}\ s\ f\ g,n)$
 $\forall fn\ m\ s\ n. \text{field-lookup-list}\ ts\ fn\ m = \text{Some}\ (s,n) \longrightarrow$
 $\text{field-lookup-list}\ (\text{map-td-list}\ (\lambda n\ \text{algn}\ d. \text{update-desc}\ f\ g\ d)\ (\text{update-desc}\ f\ g)\ ts)\ fn\ m = \text{Some}\ (\text{adjust-ti}\ s\ f\ g,n)$
 $\forall fn\ m\ s\ n. \text{field-lookup-tuple}\ x\ fn\ m = \text{Some}\ (s,n) \longrightarrow$
 $\text{field-lookup-tuple}\ (\text{map-td-tuple}\ (\lambda n\ \text{algn}\ d. \text{update-desc}\ f\ g\ d)\ (\text{update-desc}\ f\ g)\ x)\ fn\ m = \text{Some}\ (\text{adjust-ti}\ s\ f\ g,n)$
apply(*induct ti and st and ts and x, all <clarsimp>*)
apply(*clarsimp simp: adjust-ti-def*)
apply(*clarsimp split: option.splits*)
apply(*fastforce simp: split-DTuple-all simp flip: adjust-ti-def split: if-split-asm*
dest: field-lookup-adjust-ti)
apply (*clarsimp simp flip: adjust-ti-def*)
done

lemma *field-lookup-adjust-ti2*:
 $\text{field-lookup}\ t\ fn\ m = \text{Some}\ (s,n) \implies$
 $\text{field-lookup}\ (\text{adjust-ti}\ t\ f\ g)\ fn\ m = \text{Some}\ (\text{adjust-ti}\ s\ f\ g,n)$
by (*simp add: field-lookup-adjust-ti2'*)

lemma *fl-update*:
 $\text{field-lookup}\ (\text{adjust-ti}\ ti\ f\ g)\ fs\ m =$
 $(\text{case-option}\ \text{None}\ (\lambda(t,n). \text{Some}\ (\text{adjust-ti}\ t\ f\ g,n))\ (\text{field-lookup}\ ti\ fs\ m))$
apply(*clarsimp split: option.splits*)
apply *safe*
apply(*rule ccontr, clarsimp*)
apply(*drule field-lookup-adjust-ti, clarsimp*)
apply(*erule field-lookup-adjust-ti2*)
done

lemmas *fl-simps = fl-final-pad fl-ti-pad-combine*
fl-ti-typ-combine-match fl-ti-typ-combine-mismatch
fl-ti-typ-pad-combine-match fl-ti-typ-pad-combine-mismatch

lemma *access-ti-props-simps [simp]*:
 $\forall g\ x. \text{access-ti}\ (\text{adjust-ti}\ (\text{tag}::'a\ \text{xtyp-info})\ (f::'b \Rightarrow 'a)\ g)\ x = \text{access-ti}\ \text{tag}\ (f\ x)$
 $\forall g\ x. \text{access-ti-struct}\ (\text{map-td-struct}\ (\lambda n\ \text{algn}\ d. \text{update-desc}\ f\ g\ d)\ (\text{update-desc}\ f\ g)\ (st::'a\ \text{xtyp-info-struct}))\ x = \text{access-ti-struct}\ st\ (f\ x)$
 $\forall g\ x. \text{access-ti-list}\ (\text{map-td-list}\ (\lambda n\ \text{algn}\ d. \text{update-desc}\ f\ g\ d)\ (\text{update-desc}\ f\ g)\ (ts::'a\ \text{xtyp-info-tuple}\ \text{list}))\ x = \text{access-ti-list}\ ts\ (f\ x)$
 $\forall g\ x. \text{access-ti-tuple}\ (\text{map-td-tuple}\ (\lambda n\ \text{algn}\ d. \text{update-desc}\ f\ g\ d)\ (\text{update-desc}\ f\ g)\ (k::'a\ \text{xtyp-info-tuple}))\ x = \text{access-ti-tuple}\ k\ (f\ x)$
unfolding *adjust-ti-def*
by (*induct tag and st and ts and k (auto simp: update-desc-def)*)

lemma *field-norm-blah*:

$\llbracket \forall u v. f (g u v) = u; \text{fd-cons-access-update } d n \rrbracket \implies$
 $\text{field-norm } n \text{ algn } (\text{update-desc } f g d) = \text{field-norm } n \text{ algn } d$
by (*auto simp: update-desc-def field-norm-def fd-cons-access-update-def*)

lemma *map-td-ext'*:

$\text{wf-fd } t \wedge (\forall n \text{ algn } d. \text{fd-cons-access-update } d n \longrightarrow (f n \text{ algn } d = g n \text{ algn } d))$
 $\longrightarrow \text{map-td } f h t = \text{map-td } g h t$
 $\text{wf-fd-struct } st \wedge (\forall n \text{ algn } d. \text{fd-cons-access-update } d n \longrightarrow (f n \text{ algn } d = g n \text{ algn } d))$
 $\longrightarrow \text{map-td-struct } f h st = \text{map-td-struct } g h st$
 $\text{wf-fd-list } ts \wedge (\forall n \text{ algn } d. \text{fd-cons-access-update } d n \longrightarrow (f n \text{ algn } d = g n \text{ algn } d))$
 $\longrightarrow \text{map-td-list } f h ts = \text{map-td-list } g h ts$
 $\text{wf-fd-tuple } x \wedge (\forall n \text{ algn } d. \text{fd-cons-access-update } d n \longrightarrow (f n \text{ algn } d = g n \text{ algn } d))$
 $\longrightarrow \text{map-td-tuple } f h x = \text{map-td-tuple } g h x$
by (*induct t and st and ts and x*)
(auto simp: fd-cons-struct-def fd-cons-access-update-def fd-cons-desc-def)

lemma *map-td-extI*:

$\llbracket \text{wf-fd } t; (\forall n \text{ algn } d. \text{fd-cons-access-update } d n \longrightarrow (f n \text{ algn } d = g n \text{ algn } d)) \rrbracket$
 $\implies \text{map-td } f h t = \text{map-td } g h t$
by (*simp add: map-td-ext'*)

lemma *comp-unit*: $(\lambda-. ()) \circ f = (\lambda-. ())$

by (*simp add: comp-def*)

lemma *export-tag-adjust-ti2*:

$\llbracket \forall u v. f (g u v) = u; \text{wf-lf } (\text{lf-set } t \llbracket \rrbracket); \text{wf-desc } t \rrbracket \implies$
 $\text{export-uinfo } (\text{adjust-ti } t f g) = (\text{export-uinfo } t)$
unfolding *export-uinfo-def adjust-ti-def map-td-map comp-unit*
by (*fastforce simp: field-norm-blah intro: wf-fdp-fdD map-td-extI elim: wf-lf-fdp*)

lemma *field-names-list*:

$\text{field-names-list } (xs@ys) t = \text{field-names-list } xs t @ \text{field-names-list } ys t$
by (*induct xs*) *auto*

lemma *field-names-extend-ti*:

$\text{typ-name } t \neq \text{typ-name } ti \implies$
 $\text{field-names } (\text{extend-ti } ti xi \text{ algn } fn d) t = \text{field-names } ti t @ (\text{map } (\lambda fs. fn \# fs)$
 $(\text{field-names } xi t))$
apply (*cases ti*)
subgoal for *x1 typ-struct xs*
by (*cases typ-struct; fastforce simp: field-names-list*)
done

lemma *field-names-ti-pad-combine*:

$\llbracket \text{typ-name } t \neq \text{typ-name } ti; \text{hd } (\text{typ-name } t) \neq \text{CHR } "!" \rrbracket \implies$
 $\text{field-names } (\text{ti-pad-combine } n ti) t = \text{field-names } ti t$

by (*clarsimp simp: ti-pad-combine-def Let-def field-names-extend-ti export-uinfo-def size-map-td*)

lemma *export-uinfo-map-align-commute*: $\text{export-uinfo} (\text{map-align } f \ t) = \text{map-align } f \ (\text{export-uinfo } t)$
by (*cases t*) *auto*

lemma *field-names-map-align*: $\text{typ-name } t \neq \text{typ-name } ti \implies \text{field-names} (\text{map-align } f \ ti) \ t = \text{field-names } ti \ t$
apply (*cases ti*)
apply *auto*
done

lemma *typ-name-map-align* [*simp*]: $\text{typ-name} (\text{map-align } f \ t) = \text{typ-name } t$
by (*cases t*) *auto*

lemma *typ-name-ti-pad-combine* [*simp*]:
 $\text{typ-name} (\text{ti-pad-combine } n \ ti) = \text{typ-name } ti$
by (*cases ti*) (*simp add: ti-pad-combine-def Let-def*)

lemma *field-names-final-pad*:
 $\llbracket \text{typ-name } t \neq \text{typ-name } ti; \text{hd} (\text{typ-name } t) \neq \text{CHR } "!" \rrbracket \implies$
 $\text{field-names} (\text{final-pad } a \ ti) \ t = \text{field-names } ti \ t$
by (*clarsimp simp: final-pad-def Let-def field-names-ti-pad-combine field-names-map-align*)

lemma *field-names-adjust-ti*:
assumes *fg-cons f g*
shows
 $\text{wf-fd } ti \longrightarrow$
 $\text{field-names} (\text{adjust-ti} (ti::'a \ xtyp\text{-info}) \ f \ g) \ t = \text{field-names } ti \ t$
 $\text{wf-fd-struct } st \longrightarrow$
 $\text{field-names-struct} ((\text{map-td-struct} (\lambda n \ \text{algn } \ d. \ \text{update-desc } f \ g \ d) (\text{update-desc } f \ g))$
 $\quad (\text{st}::'a \ xtyp\text{-info-struct})) \ t =$
 $\text{field-names-struct } st \ t$
 $\text{wf-fd-list } ts \longrightarrow$
 $\text{field-names-list} (\text{map-td-list} (\lambda n \ \text{algn } \ d. \ \text{update-desc } f \ g \ d) (\text{update-desc } f \ g)$
 $\quad (ts::'a \ xtyp\text{-info-tuple } list)) \ t =$
 $\text{field-names-list } ts \ t$
 $\text{wf-fd-tuple } x \longrightarrow$
 $\text{field-names-tuple} (\text{map-td-tuple} (\lambda n \ \text{algn } \ d. \ \text{update-desc } f \ g \ d) (\text{update-desc } f \ g)$
 $\quad (x::'a \ xtyp\text{-info-tuple})) \ t =$
 $\text{field-names-tuple } x \ t$ **using** *assms*
by (*induct ti and st and ts and x*) (*auto simp: adjust-ti-def export-tag-adjust-ti*)

lemma *field-names-ti-typ-combine*:
 $\llbracket \text{typ-name } t \neq \text{typ-name } ti; \text{fg-cons } f \ g \rrbracket \implies$
 $\text{field-names} (\text{ti-typ-combine} (t::'b::\text{mem-type } itself) \ f \ g \ \text{algn } \ \text{fn } \ ti) \ t =$

field-names ti t @ map ((#) fn) (field-names (typ-info-t TYPE('b)) t)
by (*clarsimp simp: ti-typ-combine-def Let-def field-names-adjust-ti field-names-extend-ti*
export-uinfo-def size-map-td)

lemma *size-empty-typ-info* [*simp*]:
size (empty-typ-info algn tn) = 2
by (*simp add: empty-typ-info-def*)

lemma *list-size-char*:
size-list (λc. 0) xs = length xs
by (*induct xs*) *auto*

lemma *size-ti-extend-ti* [*simp*]:
aggregate ti ⇒ size (extend-ti ti t algn fn d) = size ti + size t + 2
apply (*cases ti*)
subgoal for *x1 typ-struct xs*
by (*cases typ-struct, auto simp: list-size-char*)
done

lemma *typ-name-empty-typ-info* [*simp*]:
typ-name (empty-typ-info algn tn) = tn
by (*clarsimp simp: empty-typ-info-def*)

lemma *typ-name-extend-ti* [*simp*]:
typ-name (extend-ti ti t algn fn d) = typ-name ti
by (*cases ti, simp*)

lemma *typ-name-ti-typ-combine* [*simp*]:
typ-name (ti-typ-combine (t-b::'b::c-type itself) f g algn fn ti) = typ-name ti
by (*clarsimp simp: ti-typ-combine-def Let-def*)

lemma *typ-name-ti-typ-pad-combine* [*simp*]:
typ-name (ti-typ-pad-combine (t-b::'b::c-type itself) f g algn fn ti) = typ-name ti
by (*clarsimp simp: ti-typ-pad-combine-def Let-def*)

lemma *typ-name-ti-final-pad* [*simp*]:
typ-name (final-pad algn ti) = typ-name ti
by (*clarsimp simp: final-pad-def Let-def*)

lemma *typ-name-map-td* [*simp*]:
typ-name (map-td f g td) = typ-name td
by (*cases td, simp*)

lemma *field-names-ti-typ-pad-combine*:
 $\llbracket \text{typ-name } t \neq \text{typ-name } ti; fg\text{-cons } f\ g; \text{aggregate } ti; hd (\text{typ-name } t) \neq CHR$
 $\text{'!'} \rrbracket \implies$
field-names (ti-typ-pad-combine (t-b::'b::mem-type itself) f g algn fn ti) t =
field-names ti t @ map ((#) fn) (field-names (typ-info-t TYPE('b)) t)

by (*auto simp: ti-typ-pad-combine-def Let-def field-names-ti-typ-combine field-names-ti-pad-combine*)

lemma *field-names-empty-typ-info*:

$typ\text{-}name\ t \neq tn \implies field\text{-}names\ (empty\text{-}typ\text{-}info\ algn\ tn)\ t = []$
by (*clarsimp simp: empty-typ-info-def*)

lemma *sep-heap-update-global-super-fl'*:

$\llbracket (p \mapsto_g u \wedge^* R)\ (lift\text{-}state\ (h,d));$
 $field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('b::mem\text{-}type))\ f\ 0 = Some\ (t,n);$
 $export\text{-}uinfo\ t = (typ\text{-}uinfo\text{-}t\ TYPE('a));$
 $w = update\text{-}ti\text{-}t\ t\ (to\text{-}bytes\text{-}p\ v)\ u \rrbracket \implies$
 $((p \mapsto_g w) \wedge^* R)$
 $(lift\text{-}state\ (heap\text{-}update\ (Ptr\ \&(p \rightarrow f))\ (v::'a::mem\text{-}type)\ h,d))$
by (*auto dest: sep-heap-update-global-super-fl*)

lemma *sep-heap-update-global-super-fl'-inv*:

$\llbracket (p \mapsto_g^i u \wedge^* R)\ (lift\text{-}state\ (h,d));$
 $field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('b::mem\text{-}type))\ f\ 0 = Some\ (t,n);$
 $export\text{-}uinfo\ t = (typ\text{-}uinfo\text{-}t\ TYPE('a));$
 $w = update\text{-}ti\text{-}t\ t\ (to\text{-}bytes\text{-}p\ v)\ u \rrbracket \implies$
 $((p \mapsto_g^i w) \wedge^* R)\ (lift\text{-}state\ (heap\text{-}update\ (Ptr\ \&(p \rightarrow f))\ (v::'a::mem\text{-}type)\ h,d))$
unfolding *sep-map-inv-def*
by (*simp only: sep-conj-assoc*) (*erule* (2) *sep-heap-update-global-super-fl*)

lemma *sep-map'-field-map'*:

$\llbracket ((p::'b::mem\text{-}type\ ptr) \hookrightarrow_g v)\ s;\ field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('b))\ f\ 0$
 $= Some\ (d,n); export\text{-}uinfo\ d = typ\text{-}uinfo\text{-}t\ TYPE('a);$
 $guard\text{-}mono\ g\ h \rrbracket \implies$
 $((Ptr\ (\&(p \rightarrow f))::'a::mem\text{-}type\ ptr) \hookrightarrow_h from\text{-}bytes\ (access\text{-}ti_0\ d\ v))\ s$
by (*subst sep-map'-unfold-exc, subst (asm) sep-map'-def*)
(fastforce simp: sep-map'-def elim: sep-conj-impl sep-map-field-map')

lemma *sep-map'-field-map*:

$\llbracket ((p::'b::mem\text{-}type\ ptr) \hookrightarrow_g v)\ s;\ field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('b))\ f\ 0$
 $= Some\ (d,n); export\text{-}uinfo\ d = typ\text{-}uinfo\text{-}t\ TYPE('a);$
 $guard\text{-}mono\ g\ h;\ w = from\text{-}bytes\ (access\text{-}ti_0\ d\ v) \rrbracket \implies$
 $((Ptr\ (\&(p \rightarrow f))::'a::mem\text{-}type\ ptr) \hookrightarrow_h w)\ s$
by (*simp add: sep-map'-field-map'*)

lemma *inter-sub*:

$\llbracket Y \subseteq X;\ Y = Z \rrbracket \implies X \cap Y = Z$
by *fast*

lemma *sep-map'-field-map-inv*:

$\llbracket ((p::'b::mem\text{-}type\ ptr) \hookrightarrow_g^i v)\ s;\ field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('b))\ f\ 0 = Some$
 $(d,n);$
 $export\text{-}uinfo\ d = typ\text{-}uinfo\text{-}t\ TYPE('a); guard\text{-}mono\ g\ h;\ w = from\text{-}bytes\ (access\text{-}ti_0$
 $d\ v) \rrbracket \implies$
 $((Ptr\ \&(p \rightarrow f)::'a::mem\text{-}type\ ptr) \hookrightarrow_h^i w)\ s$

```

apply(unfold sep-map'-inv-def)
apply(drule sep-conjD, clarsimp simp: sep-conj-ac)
subgoal for  $s_0 s_1$ 
  apply(subst sep-map'-unfold-exc)
  apply(subst sep-conj-assoc [symmetric])
  apply(rule sep-conjI [where  $s_0=(s_1 ++ s_0) \mid \{(x,y) \mid x y. x \in \{\&(p \rightarrow f)..+size-td$ 
d\} and
   $s_1=(s_1 ++ s_0) \mid (dom (s_1 ++ s_0) - \{(x,y) \mid x y. x \in \{\&(p \rightarrow f)..+size-td$ 
d\})
   $\}$ )
  apply(subst sep-conj-com)
  apply(rule sep-conjI [where  $s_0=(s_1 ++ s_0) \mid s-footprint ((Ptr \&(p \rightarrow f))::'a::mem-type$ 
ptr) and
   $s_1=(s_1 ++ s_0) \mid (\{(x, y) \mid x y. x \in \{\&(p \rightarrow f)..+size-td d\} -$ 
   $s-footprint ((Ptr \&(p \rightarrow f))::'a::mem-type ptr) \}$ )
  apply(drule (3) sep-map'-field-map')
  apply(clarsimp simp: sep-conj-ac sep-map'-def sep-conj-def)
subgoal for  $s_0' s_1'$ 
apply(rule exI [where  $x=s_0' \mid s-footprint ((Ptr \&(p \rightarrow f))::'a::mem-type ptr)$ )
apply(rule exI [where  $x=s_1'$ ])
apply(clarsimp simp: sep-conj-ac)
apply(rule conjI)
  apply(fastforce simp: map-disj-def sep-conj-ac)
apply(subst map-add-com [where  $h_0=a ++ b$  for  $a b$ ])
  apply(fastforce simp: map-disj-def sep-conj-ac)
apply(subst map-add-assoc)
apply(simp add: map-add-restrict)
apply(frule sep-map-dom-exc)
apply(rotate-tac -1)
apply(drule sym)
apply(thin-tac  $s = x$  for  $x$ )
apply(fastforce simp: restrict-map-disj-dom-empty map-ac-simps sep-conj-ac)
done
  apply(clarsimp simp: inv-footprint-def sep-conj-ac)
  apply(rule inter-sub)
  apply(clarsimp simp: sep-conj-ac)
  apply(frule-tac  $p=p$  in field-tag-sub)
  apply(drule (1) subsetD)
  apply(clarsimp simp: sep-conj-ac)
  apply(clarsimp simp: sep-conj-ac sep-map'-def sep-conj-def)
  apply(drule sep-map-dom-exc)
subgoal for  $x y s_0' s_1'$ 
apply(subgoal-tac  $s_1' (x,y) \neq None$ )
apply(clarsimp simp: sep-conj-ac)
apply(subst map-add-comm)
  apply(fastforce simp: map-disj-def sep-conj-ac)
apply simp
apply(clarsimp simp: sep-conj-ac, fast)
done

```

```

    apply(fastforce dest: export-size-of)
    apply(fastforce simp: map-disj-def)
    apply clarsimp
    apply(subst subset-map-restrict-sub-add; simp?)
    apply(fastforce intro!: intvlI dest: export-size-of
      simp: size-of-def s-footprint-def s-footprint-untyped-def)
    apply simp
    apply(fastforce simp: map-disj-def)
    apply (metis (lifting) map-add-com map-add-restrict-comp-right-dom map-le-iff-map-add-commute
      restrict-map-sub-add restrict-map-sub-disj)
  done
done

```

```

lemma guard-mono-True [simp]:
  guard-mono f (λx. True)
  by (simp add: guard-mono-def)

```

```

lemma access-ti0-to-bytes [simp]:
  access-ti0 (typ-info-t TYPE('a::c-type)) = (to-bytes-p::'a ⇒ byte list)
  by (auto simp: to-bytes-p-def to-bytes-def access-ti0-def size-of-def)

```

```

lemma update-ti-s-from-bytes:
  length bs = size-of TYPE('a) ⇒
    update-ti-t (typ-info-t TYPE('a::mem-type)) bs x = from-bytes bs
  by (simp add: from-bytes-def upd)

```

```

lemma access-ti0-update-ti [simp]:
  access-ti0 (adjust-ti ti f g) = access-ti0 ti ∘ f
  by (auto simp: access-ti0-def)

```

```

lemma update-ti-s-adjust-ti:
  [ length bs = size-td ti; fg-cons f g ] ⇒
    update-ti-t (adjust-ti ti f g) bs v = g (update-ti-t ti bs (f v)) v
  by (rule update-ti-t-adjust-ti)

```

```

lemma update-ti-s-adjust-ti-to-bytes-p [simp]:
  fg-cons f g ⇒
    update-ti-t (adjust-ti (typ-info-t TYPE('a)) f g) (to-bytes-p (v::'a::mem-type)) w
  = g v w
  apply(simp add: update-ti-t-adjust-ti to-bytes-p-def to-bytes-def)
  apply(subst upd-rf; simp add: size-of-def fd-cons-length)
  apply(subst fd-cons-update-access; simp)
  done

```

```

definition
  pad-typ-name ≡ "!pad-typ"

```

primrec

$td\text{-names} :: ('a, 'b) \text{ typ-desc} \Rightarrow (\text{char list}) \text{ set}$ **and**
 $td\text{-names-struct} :: ('a, 'b) \text{ typ-struct} \Rightarrow (\text{char list}) \text{ set}$ **and**
 $td\text{-names-list} :: ('a, 'b) \text{ typ-tuple list} \Rightarrow (\text{char list}) \text{ set}$ **and**
 $td\text{-names-tuple} :: ('a, 'b) \text{ typ-tuple} \Rightarrow (\text{char list}) \text{ set}$

where

$tnm0: td\text{-names} (TypDesc \text{ algn } st \text{ nm}) = (\{nm\} \cup td\text{-names-struct } st) - \{pad\text{-typ-name}\}$

| $tnm1: td\text{-names-struct} (TypScalar \text{ n } \text{ algn } d) = \{\}$

| $tnm2: td\text{-names-struct} (TypAggregate \text{ xs}) = td\text{-names-list } xs$

| $tnm3: td\text{-names-list} [] = \{\}$

| $tnm4: td\text{-names-list} (x\#xs) = td\text{-names-tuple } x \cup td\text{-names-list } xs$

| $tnm5: td\text{-names-tuple} (DTuple \text{ t } \text{ nm } d) = td\text{-names } t$

lemma $td\text{-set-td-names}$:

$\bigwedge (tp :: ('a, 'b) \text{ typ-desc}) \text{ n m. } \llbracket (tp, n) \in td\text{-set } tp' \text{ m}; \text{ typ-name } tp \neq pad\text{-typ-name} \rrbracket \Longrightarrow$

$\text{typ-name } tp \in td\text{-names } tp' \text{ and}$

$\bigwedge (tp :: ('a, 'b) \text{ typ-desc}) \text{ n m. } \llbracket (tp, n) \in td\text{-set-struct } tps \text{ m}; \text{ typ-name } tp \neq pad\text{-typ-name} \rrbracket \Longrightarrow$

$\text{typ-name } tp \in td\text{-names-struct } tps \text{ and}$

$\bigwedge (tp :: ('a, 'b) \text{ typ-desc}) \text{ n m. } \llbracket (tp, n) \in td\text{-set-list } tpl \text{ m}; \text{ typ-name } tp \neq pad\text{-typ-name} \rrbracket \Longrightarrow$

$\text{typ-name } tp \in td\text{-names-list } tpl \text{ and}$

$\bigwedge (tp :: ('a, 'b) \text{ typ-desc}) \text{ n m. } \llbracket (tp, n) \in td\text{-set-tuple } tpr \text{ m}; \text{ typ-name } tp \neq pad\text{-typ-name} \rrbracket \Longrightarrow$

$\text{typ-name } tp \in td\text{-names-tuple } tpr$

by ($induct \text{ tp}' \text{ and } tps \text{ and } tpl \text{ and } tpr$) $auto$

lemma $td\text{-names-map-td}$ [$simp$]:

$td\text{-names} (map\text{-td } f \text{ g } tp) = td\text{-names } tp$

$td\text{-names-struct} (map\text{-td-struct } f \text{ g } tps) = td\text{-names-struct } tps$

$td\text{-names-list} (map\text{-td-list } f \text{ g } tpl) = td\text{-names-list } tpl$

$td\text{-names-tuple} (map\text{-td-tuple } f \text{ g } tpr) = td\text{-names-tuple } tpr$

by ($induct \text{ tp} \text{ and } tps \text{ and } tpl \text{ and } tpr$) $simp\text{-all}$

lemma $td\text{-names-list-append}$ [$simp$]:

$td\text{-names-list} (a @ b) = td\text{-names-list } a \cup td\text{-names-list } b$

by ($induct \text{ a}; \text{ simp } add: Un\text{-assoc}$)

lemma $pad\text{-typ-name-td-names}$:

$A - \{pad\text{-typ-name}\} \cup td\text{-names } tp = (A \cup td\text{-names } tp) - \{pad\text{-typ-name}\}$

by ($cases \text{ tp}$) $fastforce$

lemma $td\text{-names-extend-ti}$ [$simp$]:

shows $td\text{-names} (extend\text{-ti } tp \text{ tp}' \text{ algn } ls \text{ d}) = td\text{-names } tp \cup td\text{-names } tp' \text{ and}$

$td\text{-names-struct} (extend\text{-ti-struct } tps \text{ tp}' \text{ ls } d) = td\text{-names-struct } tps \cup td\text{-names}$

tp'
by (*induct* tp **and** tps) (*simp-all* *add: pad-typ-name-td-names*)

lemma *td-names-pad-combine* [*simp*]:
 $td_names (ti_pad_combine\ m\ td) = td_names\ td$
unfolding *ti-pad-combine-def*
by (*simp* *add: Let-def pad-typ-name-def*)

lemma *td-names-map-align* [*simp*]:
 $td_names (map_align\ f\ td) = td_names\ td$
by (*cases* td) *simp*

lemma *td-names-final-pad* [*simp*]:
 $td_names (final_pad\ algn\ td) = td_names\ td$
unfolding *final-pad-def*
by (*simp* *add: Let-def*)

lemma *td-names-adjust-ti* [*simp*]:
 $td_names (adjust_ti\ td\ f\ u) = td_names\ td$
unfolding *adjust-ti-def*
by (*simp* *add: Let-def*)

lemma *td-names-typ-combine* [*simp*]:
fixes $its :: ('a :: c_type)\ itself$
shows $td_names (ti_typ_combine\ its\ f\ u\ algn\ fn\ td) = (td_names (typ_info_t\ TYPE('a)) \cup td_names\ td)$
unfolding *ti-typ-combine-def*
by (*simp* *add: Let-def Un-ac*)

lemma *td-names-typ-pad-combine* [*simp*]:
fixes $its :: ('a :: c_type)\ itself$
shows $td_names (ti_typ_pad_combine\ its\ f\ u\ algn\ nm\ td) = td_names (typ_info_t\ TYPE('a)) \cup td_names\ td$
unfolding *ti-typ-pad-combine-def*
by (*simp* *add: Let-def*)

lemma *td-names-empty-typ-info* [*simp*]:
shows $td_names (empty_typ_info\ algn\ nm) = \{nm\} - \{pad_typ_name\}$
unfolding *empty-typ-info-def*
by (*simp* *add: Let-def*)

lemma *td-names-ptr* [*simp*]:
 $td_names (typ_info_t\ TYPE(('a :: c_type)\ ptr)) = \{typ_name_itself\ TYPE('a)\ @\ "+ptr"\}$
by (*simp* *add: pad-typ-name-def*)

lemma *td-names-word8* [*simp*]:
fixes $x :: byte\ itself$

shows $td\text{-names } (typ\text{-info-t } x) = \{\text{"word00010"}\}$
by (*simp add: pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-word8'* [*simp*]:
shows $td\text{-names } (typ\text{-info-t } TYPE(8 \text{ word})) = \{\text{"word00010"}\}$
by (*simp add: pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-signed-word8* [*simp*]:
shows $td\text{-names } (typ\text{-info-t } TYPE(8 \text{ signed word})) = \{\text{"word00010"}\}$
by (*simp add: pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-word16* [*simp*]:
shows $td\text{-names } (typ\text{-info-t } TYPE(16 \text{ word})) = \{\text{"word000010"}\}$
by (*simp add: pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-signed-word16* [*simp*]:
shows $td\text{-names } (typ\text{-info-t } TYPE(16 \text{ signed word})) = \{\text{"word000010"}\}$
by (*simp add: pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-word32* [*simp*]:
 $td\text{-names } (typ\text{-info-t } TYPE(32 \text{ word})) = \{\text{"word0000010"}\}$
by (*simp add: pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-signed-word32* [*simp*]:
 $td\text{-names } (typ\text{-info-t } TYPE(32 \text{ signed word})) = \{\text{"word0000010"}\}$
by (*simp add: pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-word64* [*simp*]:
 $td\text{-names } (typ\text{-info-t } TYPE(64 \text{ word})) = \{\text{"word00000010"}\}$
by (*simp add: pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-signed-word64* [*simp*]:
 $td\text{-names } (typ\text{-info-t } TYPE(64 \text{ signed word})) = \{\text{"word00000010"}\}$
by (*simp add: pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-export-uinfo* [*simp*]:
 $td\text{-names } (export\text{-uinfo } td) = td\text{-names } td$
unfolding *export-uinfo-def*
by *simp*

lemma *typ-name-export-uinfo* [*simp*]:
 $typ\text{-name } (export\text{-uinfo } td) = typ\text{-name } td$
unfolding *export-uinfo-def*
by *simp*

lemma *replicate-Suc-append*:
 $replicate (Suc \ n) \ x = replicate \ n \ x \ @ \ [x]$
by (*induct n; simp*)

lemma *list-eq-subset*:

$xs = ys \implies \text{set } ys \subseteq \text{set } xs$ **by** *simp*

lemma *td-names-array-tag-n*:

$\text{td-names } ((\text{array-tag-n } n) :: ('a :: \text{c-type}, 'b :: \text{finite}) \text{ array } x\text{typ-info})) =$
 $\{ \text{typ-name } (\text{typ-info-t } \text{TYPE}('a)) @ \text{"-array-"} @ \text{nat-to-bin-string } (\text{card } (\text{UNIV}$
 $:: 'b \text{ set})) \} \cup$
 $(\text{if } n = 0 \text{ then } \{ \} \text{ else } \text{td-names } (\text{typ-info-t } \text{TYPE}('a)))$
apply (*induct n*; *simp add: array-tag-n.simps pad-typ-name-def*)
apply (*subst Diff-triv*; *clarsimp simp: typ-uinfo-t-def*)
apply (*fastforce dest: list-eq-subset*)
done

lemma *td-names-array [simp]*:

$\text{td-names } (\text{typ-info-t } \text{TYPE}('a :: \text{c-type})['b :: \text{finite}])) =$
 $\{ \text{typ-name } (\text{typ-info-t } \text{TYPE}('a)) @ \text{"-array-"} @ \text{nat-to-bin-string } (\text{card } (\text{UNIV}$
 $:: 'b \text{ set})) \} \cup$
 $\text{td-names } (\text{typ-info-t } \text{TYPE}('a))$
by (*simp add: typ-info-array array-tag-def td-names-array-tag-n*)

lemma *tag-disj-via-td-name*:

assumes *ta*: $\text{typ-name } (\text{typ-info-t } \text{TYPE}('a :: \text{c-type})) \neq \text{pad-typ-name}$
and *tb*: $\text{typ-name } (\text{typ-info-t } \text{TYPE}('b :: \text{c-type})) \neq \text{pad-typ-name}$
and *tina*: $\text{typ-name } (\text{typ-info-t } \text{TYPE}('a :: \text{c-type})) \notin \text{td-names } (\text{typ-info-t}$
 $\text{TYPE}('b :: \text{c-type}))$
and *tinb*: $\text{typ-name } (\text{typ-info-t } \text{TYPE}('b :: \text{c-type})) \notin \text{td-names } (\text{typ-info-t}$
 $\text{TYPE}('a :: \text{c-type}))$
shows $\text{typ-uinfo-t } \text{TYPE}('a :: \text{c-type}) \perp_t \text{typ-uinfo-t } \text{TYPE}('b :: \text{c-type})$
by (*auto simp add: typ-simps dest: td-set-td-names simp: ta tb tina tinb*)

lemma *lift-t-hrs-mem-update-fld*:

fixes *val* :: $'b :: \text{mem-type}$ **and** *ptr* :: $'a :: \text{mem-type ptr}$
assumes *fl*: $\text{field-lookup } (\text{typ-info-t } \text{TYPE}('a)) f 0 \equiv$
 $\text{Some } (\text{adjust-ti } (\text{typ-info-t } \text{TYPE}('b)) xf (xfu \circ (\lambda x -. x)), m')$
and *xf-xfu*: $\text{fg-cons } xf (xfu \circ (\lambda x -. x))$
and *cl*: $\text{lift-t } g \text{ hp } ptr = \text{Some } z$
shows $(\text{lift-t } g (\text{hrs-mem-update } (\text{heap-update } (\text{Ptr } \&(\text{ptr} \rightarrow f)) \text{ val}) \text{ hp})) =$
 $(\text{lift-t } g \text{ hp})(\text{ptr} \mapsto \text{xfu } (\lambda -. \text{val}) z)$
(is ?LHS = ?RHS)

proof –

let *?ati* = $\text{adjust-ti } (\text{typ-info-t } \text{TYPE}('b)) xf (xfu \circ (\lambda x -. x))$
have *eui*: $\text{typ-uinfo-t } \text{TYPE}('b) = \text{export-uinfo } (?ati)$ **using** *xf-xfu*
by (*simp add: typ-uinfo-t-def export-tag-adjust-ti*)

have *cl'*: $\text{lift-t } g (\text{fst } \text{hp}, \text{snd } \text{hp}) \text{ ptr} = \text{Some } z$ **using** *cl* **by** *simp*

have *?LHS* = $\text{super-field-update-t } (\text{Ptr } \&(\text{ptr} \rightarrow f)) \text{ val } (\text{lift-t } g (\text{fst } \text{hp}, \text{snd } \text{hp}))$
unfolding *hrs-mem-update-def split-def*

```

proof (rule lift-t-super-field-update [OF h-t-valid-sub])
  from cl' show snd hp, g  $\models_t$  ptr by (rule lift-t-h-t-valid)

  show fti: field-ti TYPE('a) f = Some ?ati
    by (simp add: field-ti-def fl)

  moreover show export-uinfo (?ati) = typ-uinfo-t TYPE('b)
    by (rule eui [symmetric])

  ultimately show TYPE('b)  $\leq_\tau$  TYPE('a) by (rule field-ti-sub-ty)
qed

also
have ... = (lift-t g hp)(ptr  $\mapsto$  update-ti-t (adjust-ti (typ-info-t TYPE('b)) xf (xfu
 $\circ$  ( $\lambda x$  . x)))
              (to-bytes-p val) z)
    by (simp add: cl eui fl super-field-update-lookup)

also have ... = ?RHS using xf-xfu
    by (simp add: update-ti-t-adjust-ti update-ti-s-from-bytes)

finally show ?thesis .
qed

declare pad-tyt-name-def [simp]

lemma tyt-name-array-tag-n:
  tyt-name (array-tag-n n :: ('a :: c-type ['b :: finite]) xtyp-info) =
    tyt-name (typ-info-t TYPE('a) @ "-array-" @ nat-to-bin-string (card (UNIV
:: 'b set)))
  by (induct n; clarsimp simp: array-tag-n.simps typ-uinfo-t-def)

lemma tyt-name-array [simp]:
  tyt-name (typ-info-t TYPE('a :: c-type ['b :: finite])) =
    tyt-name (typ-info-t TYPE('a) @ "-array-" @ nat-to-bin-string (card (UNIV
:: 'b set)))
  by (simp add: typ-info-array array-tag-def tyt-name-array-tag-n)

definition
  size-align-td :: ('a, 'b) tyt-desc  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bool
  where
  size-align-td t s a  $\equiv$  size-td t = s  $\wedge$  align-td t = a

lemma size-align-td-size-td:
  size-align-td t s a  $\Longrightarrow$  size-td t = s
  unfolding size-align-td-def by simp

lemma size-align-td-align-td:

```

$size-align-td\ t\ s\ a \implies align-td\ t = a$
unfolding $size-align-td-def$ **by** $simp$

lemma $size-align-td-empty-typ-infoI$:
 $size-align-td\ (empty-typ-info\ algn\ l)\ 0\ algn$
unfolding $size-align-td-def$ **by** $simp$

lemma $size-align-td-ti-typ-pad-combineI$:
fixes $t :: ('a :: c-type)\ itself$
shows $\llbracket size-align-td\ t'\ s'\ a';\ aggregate\ t'$
 $; s' + size-td\ (typ-info-t\ TYPE('a)) + padup\ (2 \wedge (max\ algn\ (align-td$
 $(typ-info-t\ TYPE('a))))\ s' = s$
 $; max\ a'\ (max\ algn\ (align-td\ (typ-info-t\ TYPE('a)))) = a$
 \rrbracket
 $\implies size-align-td\ (ti-typ-pad-combine\ t\ fa\ fu\ algn\ nm\ t')\ s\ a$
unfolding $size-align-td-def$
by $(simp\ add:\ size-td-lt-ti-typ-pad-combine)$

lemma $size-align-td-ti-final-padI$:
fixes $t :: ('a :: c-type)\ xtyp-info$
shows $\llbracket size-align-td\ t'\ s'\ a';\ aggregate\ t';\ s' + padup\ (2 \wedge max\ algn\ a')\ s' = s;$
 $max\ algn\ a' = a \rrbracket$
 $\implies size-align-td\ (final-pad\ algn\ t')\ s\ a$
unfolding $size-align-td-def$
by $(simp\ add:\ size-td-lt-final-pad\ align-of-final-pad)$

lemma $field-lookup-empty-typ-infoI$:
 $field-lookup\ (empty-typ-info\ algn\ l)\ [fld]\ m = None$
unfolding $empty-typ-info-def$
by $simp$

lemma $field-lookup-ti-typ-pad-combine-matchI$:
 $\llbracket n \# nm \notin set\ (CompoundCTypes.field-names-list\ t');\ aggregate\ t';\ size-align-td$
 $t'\ s\ a$
 $; padup\ (max\ (2 \wedge algn)\ (align-of\ TYPE('b)))\ s + s = m';\ n \neq CHR\ '!' \rrbracket$
 $\implies field-lookup\ (ti-typ-pad-combine\ (t :: 'b\ itself)\ fa\ fu\ algn\ (n\#\ nm)\ t')\ [(n\#\ nm)]$
 $0 = Some\ (adjust-ti\ (typ-info-t\ TYPE('b :: c-type))\ fa\ fu,\ m')$
by $(simp\ add:\ ft-simps\ size-align-td-def)$

lemma $field-lookup-ti-typ-pad-combine-nomatchI$:
 $\llbracket aggregate\ t';\ f \neq CHR\ '!';\ f \# fld \neq nm;\ field-lookup\ t'\ [f \# fld]\ 0 = R \rrbracket$
 $\implies field-lookup\ (ti-typ-pad-combine\ t\ fa\ fu\ algn\ nm\ t')\ [f \# fld]\ 0 = R$
by $(simp\ add:\ ft-simps)$

lemma $field-lookup-final-padI$:
 $\llbracket aggregate\ t';\ f \neq CHR\ '!';\ field-lookup\ t'\ [f \# fld]\ 0 = R \rrbracket \implies field-lookup$
 $(final-pad\ algn\ t')\ [f \# fld]\ 0 = R$
by $(simp\ add:\ ft-simps)$

```

lemmas field-lookup-ti-intros =
  field-lookup-ti-typ-pad-combine-matchI field-lookup-ti-typ-pad-combine-nomatchI
  field-lookup-final-padI field-lookup-empty-typ-infoI

lemma notin-field-names-list-empty-typ-info:
  fld ∉ set (CompoundCTypes.field-names-list (empty-typ-info l))
  by simp

lemma notin-field-names-list-ti-typ-pad-combine:
  [| f#fld ∉ set (CompoundCTypes.field-names-list t'); f#fld ≠ nm; f ≠ CHR "!" |]
  ||
  ⇒ f#fld ∉ set (CompoundCTypes.field-names-list (ti-typ-pad-combine t fa fu
  algn nm t'))
  by simp

lemma fold-typ-uinfo-t: export-uinfo (typ-info-t TYPE('a::c-type)) = typ-uinfo-t
  TYPE('a)
  by (simp add: typ-uinfo-t-def)

end

```

```

theory ArrayAssertion

```

```

imports

```

```

  ArchArraysMemInstance
  StructSupport

```

```

begin

```

```

lemma array-tag-n-eq:

```

```

  (array-tag-n n :: ('a :: c-type['b :: finite]) xtyp-info) =
  TypDesc (align-td (typ-uinfo-t TYPE('a))) (TypAggregate
    (map (λn. DTuple (adjust-ti (typ-info-t TYPE('a)) (λx. index x n)
      (λx f. Arrays.update f n x)) (replicate n CHR "1"))
      (|field-access = xto-bytes ∘ (λx. index x n),
        field-update = (λx f. Arrays.update f n x) ∘ xfrom-bytes,
        field-sz = size-of TYPE('a::c-type)|) [0..<n]))
  (typ-name (typ-uinfo-t TYPE('a)) @ "-array-" @ nat-to-bin-string (card (UNIV
  :: 'b :: finite set)))
  apply (induct n)
  apply (simp add: typ-info-array array-tag-def eval-nat-numeral
    array-tag-n.simps empty-typ-info-def align-of-def align-td-uinfo)
  apply (simp add: typ-info-array array-tag-def eval-nat-numeral array-tag-n.simps
    empty-typ-info-def)

```

apply (*simp add: ti-typ-combine-def Let-def comp-def align-td-uinfo*)
done

lemma *typ-info-array'*:

typ-info-t TYPE ('a :: c-type['b :: finite]) =
 TypDesc (align-td (typ-uinfo-t TYPE('a))) (TypAggregate
 (map (λn. DTuple (adjust-ti (typ-info-t TYPE('a)) (λx. index x n)
 (λx f. Arrays.update f n x)) (replicate n CHR "1")
 (field-access = xto-bytes ∘ (λx. index x n),
 field-update = (λx f. Arrays.update f n x) ∘ xfrom-bytes,
 field-sz = size-of TYPE('a::c-type))) [0..<(card (UNIV :: 'b :: finite
 set))]))
 (typ-name (typ-uinfo-t TYPE('a)) @ "-array-" @ nat-to-bin-string (card (UNIV
 :: 'b :: finite set)))
by (*simp add: typ-info-array array-tag-def array-tag-n-eq*)

definition

uinfo-array-tag-n-m (v :: 'a itself) n m = TypDesc
 (align-td (typ-uinfo-t TYPE('a)))
 (TypAggregate (map (λi. DTuple (typ-uinfo-t TYPE('a)) (replicate i CHR "1")
 ())) [0 ..< n]))
 (typ-name (typ-uinfo-t TYPE('a :: c-type)) @ "-array-" @ nat-to-bin-string m)

lemma *map-td-list-map*:

map-td-list f g = map (map-td-tuple f g)
apply (*rule ext*)
subgoal for x **by** (*induct x simp-all*)
done

lemma *uinfo-array-tag-n-m-eq*:

n ≤ CARD('b)
 ⇒ export-uinfo (array-tag-n n :: (('a :: wf-type)['b :: finite]) xtyp-info)
 = uinfo-array-tag-n-m TYPE ('a) n (CARD('b))
apply (*clarsimp simp: uinfo-array-tag-n-m-def array-tag-n-eq map-td-list-map
 o-def adjust-ti-def map-td-map typ-uinfo-t-def export-uinfo-def*)
apply (*fastforce intro: map-td-extI simp: field-norm-blah*)
done

lemma *typ-uinfo-array-tag-n-m-eq*:

typ-uinfo-t TYPE (('a :: wf-type)['b :: finite])
 = uinfo-array-tag-n-m TYPE ('a) (CARD('b)) (CARD('b))
by (*simp add: typ-uinfo-t-def typ-info-array array-tag-def uinfo-array-tag-n-m-eq*)

Alternative to *h-t-valid* for arrays. This allows reasoning about arrays of variable width.

definition *h-t-array-valid* :: heap-typ-desc ⇒ ('a :: c-type) ptr ⇒ nat ⇒ bool
where

h-t-array-valid htd ptr n = valid-footprint htd (ptr-val ptr) (uinfo-array-tag-n-m

TYPE ('a) n n)

Assertion that pointer p is within an array that continues for at least n more elements.

definition

```
array-assertion (p :: ('a :: c-type) ptr) n htd
= (∃ q i j. h-t-array-valid htd q j
  ∧ p = CTypesDefs.ptr-add q (int i) ∧ i < j ∧ i + n ≤ j)
```

lemma *array-assertion-shrink-right*:

```
array-assertion p n htd ⇒ n' ≤ n ⇒ array-assertion p n' htd
by (fastforce simp: array-assertion-def)
```

lemma *array-assertion-shrink-leftD*:

```
array-assertion p n htd ⇒ j < n ⇒ array-assertion (CTypesDefs.ptr-add p (int
j)) (n - j) htd
apply (clarsimp simp: array-assertion-def)
subgoal for q i
  apply (rule exI, rule exI[where x=i + j], rule exI, erule conjI)
  apply (simp add: CTypesDefs.ptr-add-def field-simps)
done
done
```

lemma *array-assertion-shrink-leftI*:

```
array-assertion (CTypesDefs.ptr-add p (- (int j))) (n + j) htd
⇒ n ≠ 0 ⇒ array-assertion p n htd
apply (drule-tac j=j in array-assertion-shrink-leftD, simp)
apply (simp add: CTypesDefs.ptr-add-def)
done
```

lemma *h-t-array-valid*:

```
h-t-valid htd gd (p :: (('a :: wf-type)['b :: finite]) ptr)
⇒ h-t-array-valid htd (ptr-coerce p :: 'a ptr) (CARD('b))
by (clarsimp simp: h-t-valid-def h-t-array-valid-def typ-winfo-array-tag-n-m-eq)
```

lemma *array-ptr-valid-array-assertionD*:

```
h-t-valid htd gd (p :: (('a :: wf-type)['b :: finite]) ptr)
⇒ array-assertion (ptr-coerce p :: 'a ptr) (CARD('b)) htd
apply (clarsimp simp: array-assertion-def dest!: h-t-array-valid)
apply (fastforce intro: exI[where x=0])
done
```

lemma *array-ptr-valid-array-assertionI*:

```
h-t-valid htd gd (q :: (('a :: wf-type)['b :: finite]) ptr)
⇒ q = ptr-coerce p
⇒ n ≤ CARD('b)
⇒ array-assertion (p :: 'a ptr) n htd
by (auto dest: array-ptr-valid-array-assertionD
  simp: array-assertion-shrink-right)
```

Derived from *array-assertion*, an appropriate assertion for performing a pointer addition, or for dereferencing a pointer addition (the strong case).

In either case, there must be an array connecting the two ends of the pointer transition, with the caveat that the last address can be just out of the array if the pointer is not dereferenced, thus the strong/weak distinction.

If the pointer doesn't actually move, nothing is learned.

definition *ptr-add-assertion* :: ('a :: c-type) ptr ⇒ int ⇒ bool ⇒ heap-typ-desc ⇒ bool **where**

ptr-add-assertion ptr offs strong htd ≡
 offs = 0 ∨
 array-assertion (if offs < 0 then CTypesDefs.ptr-add ptr offs else ptr)
 (if offs < 0 then nat (− offs) else if strong then Suc (nat offs) else
 nat offs)
 htd

lemma *ptr-add-assertion-positive*:

offs ≥ 0 ⇒ ptr-add-assertion ptr offs strong htd
 = (offs = 0 ∨ array-assertion ptr (case strong of True ⇒ Suc (nat offs)
 | False ⇒ nat offs) htd)
by (simp add: ptr-add-assertion-def)

lemma *ptr-add-assertion-negative*:

offs < 0 ⇒ ptr-add-assertion ptr offs strong htd
 = array-assertion (CTypesDefs.ptr-add ptr offs) (nat (− offs)) htd
by (simp add: ptr-add-assertion-def)

lemma *ptr-add-assertion-uint*[simp]:

ptr-add-assertion ptr (uint offs) strong htd
 = (offs = 0 ∨ array-assertion ptr
 (case strong of True ⇒ Suc (unat offs) | False ⇒ unat offs) htd)
by (simp add: ptr-add-assertion-positive uint-0-iff
 split: bool.split)

Ignore char and void pointers. The C standard specifies that arithmetic on char and void pointers doesn't create any special checks.

definition *ptr-add-assertion'* :: ('a :: c-type) ptr ⇒ int ⇒ bool ⇒ heap-typ-desc ⇒ bool **where**

ptr-add-assertion' ptr offs strong htd =
 (typ-uinfo-t TYPE('a) = typ-uinfo-t TYPE(word8)
 ∨ typ-uinfo-t TYPE('a) = typ-uinfo-t TYPE(unit)
 ∨ ptr-add-assertion ptr offs strong htd)

lemma *td-diff-from-typ-name*:

typ-name td ≠ typ-name td' ⇒ td ≠ td'
by clarsimp

lemma *typ-name-void*:

typ-name (*typ-uinfo-t* *TYPE*(*unit*)) = "unit"
by (*simp add: typ-uinfo-t-def*)

lemmas *ptr-add-assertion'* = *ptr-add-assertion'-def* *td-diff-from-typ-name* *typ-name-void*

Mechanism for retyping a range of memory to a non-constant array size.

definition *ptr-arr-retyps* :: *nat* \Rightarrow (*'a* :: *c-type*) *ptr* \Rightarrow *heap-tyt-desc* \Rightarrow *heap-tyt-desc*
where

ptr-arr-retyps *n* *p* \equiv
htd-update-list (*ptr-val* *p*)
 (*map* ($\lambda i.$ *list-map* (*typ-slice-t* (*uinfo-array-tag-n-m* *TYPE*(*'a*) *n* *n*) *i*))
 [*0*..*n* * *size-of* *TYPE*(*'a*)])

lemma *ptr-arr-retyps-to-retyp*:

n = *CARD*(*'b* :: *finite*)
 \implies *ptr-arr-retyps* *n* (*p* :: (*'c* :: *wf-type*) *ptr*) = *ptr-retyp* (*ptr-coerce* *p* :: (*'c*[*'b*])
ptr)

by (*auto simp: ptr-arr-retyps-def ptr-retyp-def typ-slices-def typ-uinfo-array-tag-n-m-eq*)

definition

array-ptr-index :: ((*'a* :: *c-type*)[*'b* :: *finite*]) *ptr* \Rightarrow *bool* \Rightarrow *nat* \Rightarrow *'a* *ptr*

where

array-ptr-index *p* *coerce* *n* = *CTypesDefs.ptr-add* (*ptr-coerce* *p*)
 (*if* *coerce* \wedge *n* \geq *CARD* (*'b*) *then* 0 *else* *of-nat* *n*)

lemmas *array-ptr-index-simps*

= *array-ptr-index-def*[**where** *coerce*=*False*, *simplified*]
array-ptr-index-def[**where** *coerce*=*True*, *simplified*]

lemma *field-lookup-list-Some-again*:

dt-snd (*xs* ! *i*) = *f*
 \implies *i* < *length* *xs*
 \implies *f* \notin *dt-snd* ' *set* ((*take* *i* *xs*))
 \implies *field-lookup-list* *xs* [*f*] *n*
 = *Some* (*dt-fst* (*xs* ! *i*), *n* + *sum-list* (*map* (*size-td* *o* *dt-fst*) (*take* *i* *xs*)))

apply (*induct* *xs* *arbitrary: i n, simp-all*)

subgoal for *x1* *xs* *i*

apply (*cases* *x1*, *simp*)
apply (*cases* *i*, *auto split: if-split*)

done

done

lemma *field-lookup-array*:

n < *CARD*(*'b*) \implies *field-lookup* (*typ-info-t* *TYPE*((*'a* :: *c-type*)[*'b* :: *finite*]))
 [*replicate* *n* (*CHR* "1")] *i* = *Some* (*adjust-ti* (*typ-info-t* *TYPE*(*'a*))
 ($\lambda x.$ *x*[*n*]) ($\lambda x f.$ *Arrays.update* *f* *n* *x*), *i* + *n* * *size-of* *TYPE* (*'a*))

apply (*simp add: typ-info-array array-tag-def array-tag-n-eq*)

apply (*subst* *field-lookup-list-Some-again*[**where** *i*=*n*],


```

    auto simp add: take-map o-def sum-list-triv size-of-def)
done

lemma field-ti-array:
  n < CARD('b) ==>
  field-ti (TYPE(('a :: c-type)['b :: finite])) [replicate n (CHR "'1'")] =
  Some (adjust-ti (typ-info-t TYPE('a)) (λx. x.[n]) (λx f. Arrays.update f n x))
by (simp add: field-ti-def field-lookup-array)

lemma fg-cons-array [simp]:
  n < card (UNIV :: 'b :: finite set) ==>
  fg-cons (λx. index x n) (λx f. Arrays.update (f :: 'a['b]) n x)
unfolding fg-cons-def by simp

lemma ptr-val-array-field-lvalue-0:
  fixes p:: (('a :: c-type)['b :: finite]) ptr
  shows &(p->[[]]) = ptr-val p
proof -
  note fl = field-lookup-array [where n=0 and i=0 and 'a='a and 'b='b,
simplified]
  note fl-exp = field-lookup-export-uinfo-Some [OF fl]
  show ?thesis
  unfolding field-lvalue-def field-offset-def field-offset-untyped-def typ-uinfo-t-def
  apply (subst fl-exp)
  apply simp
  done
qed

end

theory ML-Infer-Instantiate
  imports Main
begin

```

11.26 Instantiation, inferring type instantiation from term instantiation.

```

ML-val
<
instantiate <'a=typ <nat> and x=term <1::nat> in term <x + x> for x::'a::plus>
>

```

This is a variant of of the instantiate antiquotation, e.g.: `\<^instantiate>\<open>'a=\<^typ>\<close> and x=\<^term>\<open>1::nat\<close> in term \<open>x + x\<close> for x::\<open>'a::plus\<close>\<close>.`

The instantiate antiquotation requires the explicit instantiation of type-

variables, including those that can be inferred from the term instantiation. This makes the code very explicit, predictable and efficient as no types have to be calculated at runtime or be 'guessed' by the reader. On the other hand it can make the resulting code quite verbose when the number of type variables increases.

In this theory we provide a variant, that allows to omit those type-instantiations that are already fixed by the given term instantiations. This follows the same idea as e.g. `Drule.infer_instantiate`.

Moreover, we provide a variant that also takes a morphism into account. The use-case here is that the instantiate expression is written within a locale context but later on is used in an interpretation of the locale (given by morphism phi). Before instantiation the term is exported (according to morphism phi).

```

ML <
  fun pretty-terms ctxt ts =
    Pretty.list [] (map (Syntax.pretty-term ctxt) ts)
  fun string-of-terms ctxt ts = pretty-terms ctxt ts |> Pretty.string-of
  fun pretty-thms ctxt thms =
    Pretty.list [] (map (Thm.pretty-thm ctxt) thms)
  fun pretty-big-list-thms name ctxt thms =
    Pretty.big-list name (map (Thm.pretty-thm-item ctxt) thms)

  fun string-of-thms ctxt thms = pretty-thms ctxt thms |> Pretty.string-of
  >

ML <
  signature ML-INFER-INSTANTIATE =
  sig
    val make-cterm: Proof.context -> term -> cterm
    type insts = ((indexname * sort) * typ) list * ((indexname * typ) * term) list
    type cinsts = ((indexname * sort) * ctyp) list * ((indexname * typ) * cterm) list
    val morph-instantiate-term: Position.T -> insts -> term -> Morphism.morphism
    -> Proof.context -> term
    val morph-instantiate-cterm: Position.T -> cinsts -> cterm -> Morphism.morphism
    -> Proof.context -> cterm
    val morph-instantiate-thm: Position.T -> cinsts -> thm -> Morphism.morphism
    -> Proof.context -> thm
    val morph-instantiate-thms: Position.T -> cinsts -> thm list -> Morphism.morphism
    -> Proof.context -> thm list
    val instantiate-term: Position.T -> insts -> term -> Proof.context -> term
    val instantiate-cterm: Position.T -> cinsts -> cterm -> Proof.context ->
    cterm
    val instantiate-thm: Position.T -> cinsts -> thm -> Proof.context -> thm
    val instantiate-thms: Position.T -> cinsts -> thm list -> Proof.context ->
    thm list
    val get-thms: Proof.context -> int -> thm list
    val get-thm: Proof.context -> int -> thm
  end

```

```

end;

structure ML-Infer-Instantiate:ML-INFER-INSTANTIATE =
struct

(* exported operations *)

fun make-cterm ctxt t = Thm.cterm-of ctxt t |> Thm.trim-context-cterm;

type insts = ((indexname * sort) * typ) list * ((indexname * typ) * term) list
type cinsts = ((indexname * sort) * ctyp) list * ((indexname * typ) * cterm) list

type 'a operations = {
  type-of: 'a -> typ,
  transfer: theory -> 'a -> 'a,
  maxidx-of: 'a -> int,
  term-of: 'a -> term}

val term-operations = {
  type-of = fastype-of,
  transfer = K I,
  maxidx-of = Term.maxidx-of-term,
  term-of = I}: term operations

val cterm-operations = {
  type-of = Thm.typ-of-cterm,
  transfer = Thm.transfer-cterm,
  maxidx-of = Thm.maxidx-of-cterm,
  term-of = Thm.term-of}: cterm operations

— cf. Drule.infer_instantiate_types
fun gen-infer (operations:'a operations) ctxt ((xi, T), cu) (tyenv, maxidx) =
  let
    val thy = Proof-Context.theory-of ctxt
    val - = Thm.ctyp-of ctxt T;
    val - = #transfer operations thy cu;
    val U = #type-of operations cu;
    val maxidx' = maxidx
      |> Integer.max (#2 xi)
      |> Term.maxidx-ty T
      |> Integer.max (#maxidx-of operations cu);
    val (tyenv', maxidx'') = Sign.typ-unify thy (T, U) (tyenv, maxidx')
    handle Type.TUNIFY =>
      let
        val t = Var (xi, T);
        val u = #term-of operations cu;
      in
        raise TERM (ML-Infer-Instantiate.gen-infer: type ^
          Syntax.string-of-ty ctxt (Envir.norm-type tyenv T) ^ of variable ^

```

```

    Syntax.string-of-term ctxt (Term.map-types (Envir.norm-type tyenv) t) ^
    \ncannot be unified with type ^
    Syntax.string-of-typ ctxt (Envir.norm-type tyenv U) ^ of term ^
    Syntax.string-of-term ctxt (Term.map-types (Envir.norm-type tyenv) u),
    [t, u]
  end;
in (tyenv', maxidx') end;

val infer = gen-infer term-operations
val cterm-infer = gen-infer cterm-operations

fun incr-instT k (((n, i), S), T) = (((n, i + k), S), T)
fun incr-inst k (((n, i), T), t) = (((n, i + k), Logic.incr-tvar k T), t)
val maxidx-ctyp = Term.maxidx-typ o Thm.typ-of
val maxidx-cterm = Integer.max o Thm.maxidx-of-cterm

fun mk-term ctxt P =
  if Thm.typ-of-cterm P = propT then
    P
  else
    instantiate ⟨'a=⟨Thm.ctyp-of-cterm P⟩ and P = ⟨Thm.transfer-cterm (Proof-Context.theory-of
    ctxt) P⟩
    in cterm ⟨TERM P⟩ for P::'a⟩

fun dest-term ct =
  if can Logic.dest-term (Thm.term-of ct) then
    Thm.dest-arg ct
  else
    ct

(* Simultaneous export on instantiations and term *)
type 'a morph-operations = {
  mk-typ: Proof.context -> (indexname * sort) -> 'a,
  dest-typ: 'a -> (indexname * sort),
  mk-term: Proof.context -> (indexname * typ) -> 'a,
  dest-term: 'a -> (indexname * typ),
  wrap: Proof.context -> 'a -> 'a,
  unwrap: 'a -> 'a,
  mk-conjunctions: 'a list -> 'a,
  dest-conjunctions: int -> 'a -> 'a list,
  morph: Morphism.morphism -> 'a -> 'a
}

val term-morph-operations: term morph-operations = {
  mk-typ = fn - => Logic.mk-term o Logic.mk-type o TVar,
  dest-typ = dest-TVar o Logic.dest-type o Logic.dest-term,
  mk-term = fn - => Logic.mk-term o Var,
  dest-term = dest-Var o Logic.dest-term,

```

```

wrap = K I,
unwrap = I,
mk-conjunctions = Logic.mk-conjunction-list,
dest-conjunctions = fn - => Logic.dest-conjunction-list,
morph = Morphism.term
}

val cterm-morph-operations: cterm morph-operations = {
mk-typ = fn ctxt => Thm.cterm-of ctxt o Logic.mk-term o Logic.mk-type o TVar,
dest-typ = dest-TVar o Logic.dest-type o Logic.dest-term o Thm.term-of,
mk-term = fn ctxt => Thm.cterm-of ctxt o Logic.mk-term o Var,
dest-term = dest-Var o Logic.dest-term o Thm.term-of,
wrap = mk-term,
unwrap = dest-term,
mk-conjunctions = Conjunction.mk-conjunction-balanced,
dest-conjunctions = fn - => Conjunction.dest-conjunctions,
morph = Morphism.cterm
}

val thm-morph-operations: thm morph-operations = {
mk-typ = fn ctxt => Thm.reflexive o Thm.cterm-of ctxt o Logic.mk-term o
Logic.mk-type o TVar,
dest-typ = dest-TVar o Logic.dest-type o Logic.dest-term o fst o Logic.dest-equals
o Thm.concl-of,
mk-term = fn ctxt => Thm.reflexive o Thm.cterm-of ctxt o Logic.mk-term o Var,
dest-term = dest-Var o Logic.dest-term o fst o Logic.dest-equals o Thm.concl-of,
wrap = K I,
unwrap = I,
mk-conjunctions = Conjunction.intr-balanced,
dest-conjunctions = Conjunction.elim-balanced,
morph = Morphism.thm
}

fun gen-morph-insts (ops:'a morph-operations) ctxt phi t (type-insts, term-insts) =
if Morphism.is-identity phi then (t, (type-insts, term-insts))
else
let
val Ts = map (#mk-typ ops ctxt o fst) type-insts
val ts = map (#mk-term ops ctxt o fst) term-insts
val n = length Ts + length ts + 1
val (Ts', t'::ts') = #mk-conjunctions ops (Ts @ #wrap ops ctxt t::ts)
|> #morph ops phi
|> #dest-conjunctions ops n
|> chop (length Ts)
val type-insts' = map (#dest-typ ops) Ts' ~ map snd type-insts
val term-insts' = map (#dest-term ops) ts' ~ map snd term-insts
val t' = #unwrap ops t'
in
(t', (type-insts', term-insts'))

```

```

end

val morph-insts-term = gen-morph-insts term-morph-operations
val morph-insts-cterm = gen-morph-insts cterm-morph-operations
val morph-insts-thm = gen-morph-insts thm-morph-operations

fun prep-insts (insts: insts) =
  let
    val maxidx = ~ 1
    |> fold Term.maxidx-typ (map snd (#1 insts))
    |> fold Term.maxidx-term (map snd (#2 insts))
    val insts = (map (incr-instT (maxidx + 1)) (#1 insts), map (incr-inst (maxidx
+ 1)) (#2 insts))
  in (insts, maxidx) end

fun init-tenv tinsts = Vartab.make (map (fn ((xi, S), T) => (xi, (S, T))) tinsts)
fun init-ctenv tinsts = Vartab.make (map (fn ((xi, S), cT) => (xi, (S, Thm.typ-of
cT))) tinsts)

fun morph-instantiate-term pos (insts: insts) t phi ctxt =
  let
    val (t, insts) = morph-insts-term ctxt phi t insts
    val (insts, maxidx) = prep-insts insts
    val t = t |> Logic.incr-indexes ([], maxidx + 1)
    val instT = TVars.make (#1 insts);
    val instantiateT = Term-Subst.instantiateT instT;
    val inst = (map o apfst o apsnd) instantiateT (#2 insts);
    val (tyenv, -) = fold (infer ctxt) inst (init-tenv (#1 insts), maxidx + 1)
    val instT = TVars.build (tyenv |> Vartab.fold (fn (xi, (S, T)) =>
      TVars.add ((xi, S), Envir.norm-type tyenv T)));
    val inst = Vars.build (#2 insts |> fold (fn ((xi, T), t) =>
      Vars.add ((xi, Envir.norm-type tyenv T),
        Term-Subst.instantiate (instT, Vars.empty) t)));
  in Term-Subst.instantiate-beta (instT, inst) t end
  handle TERM (msg, args) => Exn.reraise (TERM (msg ^ Position.here pos,
args));

fun instantiate-term pos (insts: insts) t = morph-instantiate-term pos (insts: insts)
t Morphism.identity

fun make-cinsts ctxt maxidx (cinsts: cinsts) =
  let
    val cinstT = TVars.make (#1 cinsts);
    val instantiateT = Term-Subst.instantiateT (TVars.map (K Thm.typ-of) cinst-
stT);
    val cinst = (map o apfst o apsnd) instantiateT (#2 cinsts);
    val (tyenv, -) = fold (cterm-infer ctxt) cinst (init-ctenv (#1 cinsts), maxidx +
1)
    val cinstT = TVars.build (tyenv |> Vartab.fold (fn (xi, (S, T)) =>

```

```

      TVars.add ((xi, S), Thm.ctyp-of ctxt (Envir.norm-type tyenv T));
    val thy = Proof-Context.theory-of ctxt
    val cinst =
      Vars.build (#2 cinsts |> fold (fn ((xi, T), cu) =>
        Vars.add ((xi, Envir.norm-type tyenv T),
          Thm.instantiate-cterm (cinstT, Vars.empty) (Thm.transfer-cterm thy
cu)))));
    in (cinstT, cinst) end;

```

```

fun prep-cinsts cinsts =
  let
    val maxidx = ~1
    |> fold maxidx-ctyp (map snd (#1 cinsts))
    |> fold maxidx-cterm (map snd (#2 cinsts))
    val cinsts = (map (incr-instT (maxidx + 1)) (#1 cinsts), map (incr-inst (maxidx
+ 1)) (#2 cinsts))
  in (cinsts, maxidx) end

```

```

fun morph-instantiate-cterm pos cinsts ct phi ctxt =
  let
    val (ct, cinsts) = morph-insts-cterm ctxt phi ct cinsts
    val (cinsts, maxidx) = prep-cinsts cinsts
    val ct = Thm.incr-indices-cterm (maxidx + 1) ct
  in
    Thm.instantiate-beta-cterm (make-cinsts ctxt maxidx cinsts) ct
  end
  handle CTERM (msg, args) => Exn.reraise (CTERM (msg ^ Position.here pos,
args))
  | TERM (msg, args) => Exn.reraise (TERM (msg ^ Position.here pos,
args));

```

```

fun instantiate-cterm pos cinsts ct = morph-instantiate-cterm pos cinsts ct Morphism.identity

```

```

fun morph-instantiate-thm pos cinsts th phi ctxt =
  let
    val (th, cinsts) = morph-insts-thm ctxt phi th cinsts
    val (cinsts, maxidx) = prep-cinsts cinsts
    val th = Thm.incr-indices (maxidx + 1) th
  in
    Thm.instantiate-beta (make-cinsts ctxt maxidx cinsts) th
  end
  handle THM (msg, i, args) => Exn.reraise (THM (msg ^ Position.here pos, i,
args))
  | TERM (msg, args) => Exn.reraise (TERM (msg ^ Position.here pos,
args));

```

```

fun instantiate-thm pos cinsts th = morph-instantiate-thm pos cinsts th Morphism.identity

```

```
fun morph-instantiate-thms pos cinsts ths phi ctxt = map (fn th => morph-instantiate-thm
pos cinsts th phi ctxt) ths;
```

```
fun instantiate-thms pos cinsts ths = morph-instantiate-thms pos cinsts ths Morphism.identity
```

```
(* context data *)
```

```
structure Data = Proof-Data
(
  type T = int * thm list Inttab.table;
  fun init - = (0, Inttab.empty);
);
```

```
fun put-thms ths ctxt =
  let
    val (i, thms) = Data.get ctxt;
    val ctxt' = ctxt |> Data.put (i + 1, Inttab.update (i, map Thm.trim-context
ths) thms);
  in (i, ctxt') end;
```

```
fun get-thms ctxt i = the (Inttab.lookup (#2 (Data.get ctxt)) i);
fun get-thm ctxt i = the-single (get-thms ctxt i);
```

```
(* ML antiquotation *)
```

```
local
```

```
val make-keywords =
  Thy-Header.get-keywords'
  #> Keyword.no-major-keywords
  #> Keyword.add-major-keywords [term, prop, cterm, cprop, lemma];
```

```
val parse-inst-name = Parse.position (Parse.type-ident >> pair true || Parse.name
>> pair false);
```

```
val parse-inst =
  (parse-inst-name -- (Parse.$$$ = |-- Parse.!!! Parse.embedded-ml) ||
  Scan.ahead parse-inst-name -- Parse.embedded-ml)
  >> (fn (((b, a), pos), ml) => (b, ((a, pos), ml)));
```

```
val parse-insts =
  Parse.and-list1 parse-inst >> (List.partition #1 #> apply2 (map #2));
```

```
val ml = ML-Lex.tokenize-no-range;
val ml-range = ML-Lex.tokenize-range;
fun ml-parens x = ml ( @ x @ ml );
```



```

fun ml-bracks x = ml [ @ x @ ml ];
fun ml-commas xs = flat (separate (ml , ) xs);
val ml-list = ml-bracks o ml-commas;
fun ml-pair (x, y) = ml-parens (ml-commas [x, y]);
val ml-list-pair = ml-list o ListPair.map ml-pair;
val ml-here = ML-Syntax.atomic o ML-Syntax.print-position;

fun get-tfree envT (a, pos) =
  (case AList.lookup (op =) envT a of
   SOME S => (a, S)
  | NONE => error (No occurrence of type variable ^ quote a ^ Position.here
  pos));

fun check-free ctxt env (x, pos) =
  (case AList.lookup (op =) env x of
   SOME T =>
    (Context-Position.reports ctxt (map (pair pos) (Syntax-Phases.markup-free
  ctxt x)); (x, T))
  | NONE => error (No occurrence of variable ^ quote x ^ Position.here pos));

fun missing-instT pos envT instT =
  (case filter-out (fn (a, -) => exists (fn (b, -) => a = b) instT) envT of
   [] => ()
  | bad =>
    error (No instantiation for free type variable(s) ^ commas-quote (map #1
  bad) ^
    Position.here pos))

fun missing-inst pos env inst =
  (case filter-out (fn (a, -) => exists (fn (b, -) => a = b) inst) env of
   [] => ()
  | bad =>
    error (No instantiation for free variable(s) ^ commas-quote (map #1 bad) ^
    Position.here pos));

fun make-instT (a, pos) T =
  (case try (Term.dest-TVar o Logic.dest-type) T of
   NONE => error (Not a free type variable ^ quote a ^ Position.here pos)
  | SOME v => ml (ML-Syntax.print-pair ML-Syntax.print-indexname ML-Syntax.print-sort
  v));

fun make-inst (a, pos) t =
  (case try Term.dest-Var t of
   NONE => error (Not a free variable ^ quote a ^ Position.here pos)
  | SOME v => ml (ML-Syntax.print-pair ML-Syntax.print-indexname ML-Syntax.print-ty
  v));

fun make-env ts =
  let
    val envT = fold Term.add-tfrees ts [];
  end

```

```

    val env = fold Term.add-frees ts [];
  in (envT, env) end;

fun prepare-insts morph pos {schematic} ctxt1 ctxt0 (instT, inst) ts =
  let
    val (envT, env) = make-env ts;
    val freesT = map (Logic.mk-type o TFree o get-tfree envT) instT;
    val frees = map (Free o check-free ctxt1 env) inst;
    val (ts', (varsT, vars)) =
      Variable.export-terms ctxt1 ctxt0 (ts @ freesT @ frees)
      |> chop (length ts) ||> chop (length freesT);
    val ml-insts = (map2 make-instT instT varsT, map2 make-inst inst vars);
  in
    if schematic then ()
    else
      let val (envT', env') = make-env ts'
          val holes = subtract (eq-fst op =) env' env
          val inferableTs = fold Term.add-tfreesT (map snd holes) []
          val holesT = subtract (eq-fst op =) envT' envT
          val - = if null holesT andalso not morph
              then warning (All type instances are already explicit and do not have to be
                ^
                ^
                inferred from the term instantiations.\n ^
                Use antiquotation 'instantiate' instead. ^ Position.here pos)
              else ()
          val holesT = subtract (eq-fst op =) inferableTs holesT
        in
          missing-instT pos holesT instT;
          missing-inst pos holes inst
        end;
      (ml-insts, ts')
    end;

fun prepare-ml range kind ml1 ml2 ml-val (ml-instT, ml-inst) ctxt =
  let
    val (ml-name, ctxt') = ML-Context.variant kind ctxt;
    val ml-env = ml (val ^ ml-name ^ =) @ ml ml1 @ ml-parens (ml ml-val) @
  ml ;\n;
    fun ml-body (ml-argsT, ml-args) =
      ml-parens (ml ml2 @
        ml-pair (ml-list-pair (ml-instT, ml-argsT), ml-list-pair (ml-inst, ml-args)) @
        ml-range range (ML-Context.struct-name ctxt ^ . ^ ml-name));
  in ((ml-env, ml-body), ctxt') end;

fun prepare-term morph read range (((kind, pos), ml1, ml2), schematic), (s, fixes))
  insts ctxt =
  let
    val ctxt' = #2 (Proof-Context.add-fixes-cmd fixes ctxt);
    val t = read ctxt' s;
  end;

```

```

    val ctxt1 = Proof-Context.augment t ctxt';
    val (ml-insts, t') = prepare-insts morph pos schematic ctxt1 ctxt insts [t] ||>
the-single;
    in prepare-ml range kind ml1 ml2 (ML-Syntax.print-term t') ml-insts ctxt end;

fun prepare-lemma morph range ((pos, schematic), make-lemma) insts ctxt =
let
    val (ths, (props, ctxt1)) = make-lemma ctxt
    val (i, thms-ctxt) = put-thms ths ctxt;
    val ml-insts = #1 (prepare-insts morph pos schematic ctxt1 ctxt insts props);
    val ml-instantiate-thm =
        if morph then
            ML-Infer-Instantiate.morph-instantiate-thm
        else
            ML-Infer-Instantiate.instantiate-thm
    val ml-instantiate-thms =
        if morph then
            ML-Infer-Instantiate.morph-instantiate-thms
        else
            ML-Infer-Instantiate.instantiate-thms
    val (ml1, ml2) =
        if length ths = 1
        then (ML-Infer-Instantiate.get-thm ML-context,
            ml-instantiate-thm ^ ml-here pos)
        else (ML-Infer-Instantiate.get-thms ML-context,
            ml-instantiate-thms ^ ml-here pos);
    in prepare-ml range lemma ml1 ml2 (ML-Syntax.print-int i) ml-insts thms-ctxt
end;

fun term-ml morph (kind, pos: Position.T) =
let
    val ml-instantiate-term =
        if morph then
            ML-Infer-Instantiate.morph-instantiate-term
        else
            ML-Infer-Instantiate.instantiate-term
    in
        ((kind, pos), , ml-instantiate-term ^ ml-here pos)
    end

fun cterm-ml morph (kind, pos) =
let
    val ml-instantiate-cterm =
        if morph then
            ML-Infer-Instantiate.morph-instantiate-cterm
        else
            ML-Infer-Instantiate.instantiate-cterm
    in
        ((kind, pos),

```

```

    ML-Infer-Instantiate.make-cterm ML-context,
    ml-instantiate-cterm ^ ml-here pos)
end

val command-name = Parse.position o Parse.command-name;

val parse-schematic = Args.mode schematic >> (fn b => {schematic = b});

fun parse-body morph range =
  (command-name term >> term-ml morph || command-name cterm >> cterm-ml
  morph) -- parse-schematic --
  Parse.!!! (Parse.term -- Parse.for-fixes) >> prepare-term morph Syntax.read-term
  range ||
  (command-name prop >> term-ml morph || command-name cprop >> cterm-ml
  morph) -- parse-schematic --
  Parse.!!! (Parse.term -- Parse.for-fixes) >> prepare-term morph Syntax.read-prop
  range ||
  (command-name lemma >> #2) -- parse-schematic -- ML-Thms.embedded-lemma
  >> prepare-lemma morph range;

fun antiquotation morph =
  (fn range => fn src => fn ctxt =>
    let
      val (insts, prepare-val) = src
      |> Parse.read-embedded ctxt (make-keywords ctxt)
      ((parse-insts --| Parse.!!! in) -- parse-body morph range);

      val (((ml-env, ml-body), decls), ctxt1) = ctxt
      |> prepare-val (apply2 (map #1) insts)
      ||>> ML-Context.expand-antiquotes-list (op @ (apply2 (map #2) insts));

      fun decl' ctxt' =
        let val (ml-args-env, ml-args) = split-list (decls ctxt')
            in (ml-env @ flat ml-args-env, ml-body (chop (length (#1 insts)) ml-args))
        end;
    end;
  in (decl', ctxt1) end)

val - = Theory.setup
  (ML-Context.add-antiquotation-embedded binding <infer-instantiate> (antiquotation
  false) #>
  ML-Context.add-antiquotation-embedded binding <morph-infer-instantiate> (antiquotation
  true))

in end;

end
>

```

Here an example with explicit instantiation.

```

ML-val <
val b = Free(n, typ <nat>)
val t1 = instantiate <'a = typ <nat> and a = <@{term 1::nat}> and b=<b>
    in term <a + b> for a b::'a::plus>
>

```

With the new variant the type instantiation can be omitted as it is already inferred from the term instantiations. Note that the result of the antiquotation is function that expects a `Proof.context`. This is used to perform the necessary type inference and provide context sensitive error messages.

```

ML-val <
val b = Free(n, typ <nat>)
val t1 = infer-instantiate <a = term <1::nat> and b=<b>
    in term <a + b> for a b::'a::plus> @<context>
>

```

This can be shortened:

```

ML-val <
val b = Free(n, typ <nat>)
val t1 = infer-instantiate <a = term <1::nat> and b=<b>
    in term <a + b>> @<context>
>

```

Here is an example where not all type variables can be inferred from the term instantiation. Those types can be explicitly instantiated.

```

ML-val <
val t1 = infer-instantiate <'b = typ <int> and a = <@{term 1::nat}>
    in term <Pair a> for a ::'a > @<context>
>

```

```

ML-val <
val t1 = morph-infer-instantiate <'b = typ <int> and a = <@{term 1::nat}>
    in term <Pair a> for a ::'a > Morphism.identity @<context>
>

```

```

ML-val <
val v = Proof-Context.read-term-pattern @<context> v
val t1 = infer-instantiate <'b = typ <int> and a = v
    in term <Pair a> for a ::'a > @<context>
>

```

```

ML-val <
val v = Proof-Context.read-term-pattern @<context> v::'a::type |> Thm.cterm-of
@<context>
val t1 = infer-instantiate <'b = ctyp <int> and a = v
    in cterm <Pair a> for a ::'a > @<context>
>

```

```

ML-val <
  val t1 = infer-instantiate <'b = ctyp <int> and a = <@{cterm 1::nat}>
    in cterm <Pair a> for a ::'a > @<context>
  >

```

```

context
  fixes x::'a::plus
begin
ML-val <
  val b = Free(n, typ <nat>)
  val t1 = infer-instantiate <a = <@{term 1::nat}> and b=<b>
    in term <a + b> for a b> @<context>
  >
end

```

Note that the antiquotation only makes some local type-inference on the term parameters alone. There might be further use cases to perform a type inference on the whole term instead. This might be further fine-tuned by inserting explicit dummy types to be inferred: -.

```

ML <
  fun infer-types-simple ctxt =
    singleton (Type-Infer-Context.infer-types ctxt) #>
    singleton (Type-Infer.fixate ctxt false);

  val b = Free(n, dummyT)
  val t1 = instantiate <'a = typ <-> and a = <@{term 1::nat}> and b=<b>
    in prop <a + b = a + b> for a b::'a::plus > |> infer-types-simple @<context>
  >

end

```

```

theory CProof
imports
  umm-heap/SepFrame
  Simpl.Vcg
  umm-heap/StructSupport
  umm-heap/ArrayAssertion
  AutoCorres-Utills
  Match-Cterm
  ML-Infer-Instantiate
begin

```

```

ML-file General.ML
ML-file SourcePos.ML
ML-file SourceFile.ML
ML-file Region.ML

```

ML-file *Binaryset.ML*
ML-file *Feedback.ML*
ML-file *basics.ML*
ML-file *MString.ML*

ML-file *TargetNumbers-sig.ML*
ML-file *./umm-heap/ARM/TargetNumbers-ARM.ML*
ML-file *./umm-heap/ARM64/TargetNumbers-ARM64.ML*
ML-file *./umm-heap/ARM-HYP/TargetNumbers-ARM-HYP.ML*
ML-file *./umm-heap/RISCV64/TargetNumbers-RISCV64.ML*
ML-file *./umm-heap/X64/TargetNumbers-X64.ML*
ML-file *./umm-heap/TargetNumbers.ML*

ML-file *RegionExtras.ML*
ML-file *Absyn-CType.ML*
ML-file *Absyn-Expr.ML*
ML-file *Absyn-StmtDecl.ML*
ML-file *Absyn.ML*
ML-file *Absyn-Serial.ML*
ML-file *../lib/ml-helpers/StringExtras.ML*
ML-file *name-generation.ML*

setup ‹
 Hoare.add-foldcongsimps [@ { *thm update-update* }, @ { *thm o-def* }]
›

syntax
-quote :: 'b => ('a => 'b) (([.[-.]) [0] 1000)

syntax
-heap :: 'b ⇒ ('a ⇒ 'b)
-heap-state :: 'a (ζ)
-heap-stateOld :: ('a ⇒ 'b) ⇒ 'b (¬ζ [100] 100)

-derefCur :: ('a ⇒ 'b) ⇒ 'b (★- [100] 100)
-derefOld :: 'a ⇒ ('a ⇒ 'b) ⇒ 'b (¬★- [100,100] 100)

translations
{ | b | } => *CONST Collect* (-*quote* (-*heap* b))

definition *sep-app* :: (*heap-state* ⇒ *bool*) ⇒ *heap-state* ⇒ *bool* **where**
 sep-app *P s* ≡ *P s*

definition *hrs-id* :: *heap-raw-state* ⇒ *heap-raw-state* **where**

hrs-id \equiv *id*

declare *hrs-id-def* [*simp add*]

parse-translation \langle

let

fun ac x = Syntax.const -antiquoteCur \$ Syntax.const x

fun aco x y = Syntax.const y \$ (Syntax.const globals \$ x)

fun hd a = a NameGeneration.global-heap-var

fun d a = Syntax.const hrs-htd \$ hd a

fun repl (Abs (s,T,t)) = Abs (s,T,repl t)

| *repl (Const (-h-t-valid,-)\$x) = Syntax.const h-t-valid \$ d ac \$ Syntax.const*

c-guard \$ x

| *repl (Const (-derefCur,-)\$x) = Syntax.const the \$*

(Syntax.const lift-t \$ hd ac \$ x)

| *repl (Const (-derefOld,-)\$x\$y) = Syntax.const the \$*

(Syntax.const lift-t \$ hd (aco x) \$ y)

| *repl (Const (-heap-state,-)) = Syntax.const hrs-id \$ hd ac*

| *repl (Const (-heap-stateOld,-)\$x) = Syntax.const hrs-id \$ hd (aco x)*

| *repl (x\$y) = repl x \$ repl y*

| *repl x = x*

fun heap-assert-tr [b] = repl b

| *heap-assert-tr ts = raise TERM (heap-assert-tr, ts);*

in [(-heap,K heap-assert-tr)] end

\rangle

parse-translation \langle

let

fun ac x = Syntax.const -antiquoteCur \$ Syntax.const x

fun aco x y = Syntax.const y \$ (Syntax.const globals \$ x)

fun hd a = Syntax.const lift-state \$ (a NameGeneration.global-heap-var)

fun st2 (Abs (s,T,t)) n = Abs (s,T,st2 t (n+1))

| *st2 (Bound k) n = Bound (if k < n then k else k + 1)*

| *st2 (x\$y) n = st2 x n \$ st2 y n*

| *st2 x - = x*

fun st1 (Abs (s,T,t)) n = Abs (s,T,st1 t (n+1))

| *st1 (Bound k) n = Bound (if k < n then k else k + 1)*

| *st1 (Const (sep-empty,-)) n = Syntax.const sep-empty \$ Bound n*

| *st1 (Const (sep-map,-)\$x\$y) n = Syntax.const sep-map \$*

(st2 x n) \$ (st2 y n) \$ Bound n

| *st1 (Const (sep-map',-)\$x\$y) n = Syntax.const sep-map' \$*

(st2 x n) \$ (st2 y n) \$ Bound n

| *st1 (Const (sep-conj,-)\$x\$y) n = Syntax.const sep-conj \$*

(nst2 x n) \$ (nst2 y n) \$ Bound n

| *st1 (Const (sep-impl,-)\$x\$y) n = Syntax.const sep-impl \$*

(nst2 x n) \$ (nst2 y n) \$ Bound n

| *st1 (x\$y) n = st1 x n \$ st1 y n*


```

| st1 x - = x
and new-heap t = Abs (s,dummyT,st1 t 0)
and nst2 x n = new-heap (st2 x n);
fun sep-tr [t] = Syntax.const sep-app $ (*new-heap *) t $ hd ac
| sep-tr ts = raise TERM (sep-tr, ts);
in [(-sep-assert,K sep-tr)] end
>

```

lemma *c-null-guard*:

```

c-null-guard (p::'a::mem-type ptr)  $\implies$  p  $\neq$  NULL
by (fastforce simp: c-null-guard-def intro: intvl-self)

```

lemma (in *mem-type*) *c-guard-no-wrap*:

```

fixes p :: 'a ptr
assumes cgrd: c-guard p
shows ptr-val p  $\leq$  ptr-val p + of-nat (size-of TYPE('a) - 1)
using cgrd unfolding c-guard-def c-null-guard-def
apply -
apply (erule conjE)
apply (erule contrapos-np)
apply (simp add: intvl-def)
apply (drule word-wrap-of-natD)
apply (erule exE)
apply (rule exI)
apply (simp add: nat-le-Suc-less size-of-def wf-size-desc-gt(1))
done

```

lemma *word-le-unat-bound*:

```

fixes a::'a ::len word
assumes a  $\leq$  a + b
shows unat a + unat b < 2 ^ LENGTH('a)
using assms no-olen-add-nat by blast

```

lemma (in *mem-type*) *c-guard-no-wrap'*:

```

fixes p :: 'a ptr
assumes cgrd: c-guard p
shows unat (ptr-val p) + size-of TYPE('a)  $\leq$  addr-card
proof -

```

```

have szb: size-of TYPE('a) < addr-card
by (simp add: local.max-size)

```

```

have not-null: 0 < size-of TYPE('a)
by (simp add: size-of-def wf-size-desc-gt(1))

```

```

have sz-le: size-of TYPE('a) - Suc 0 < 2 ^ LENGTH(addr-bitsize)

```

```

using len-of-addr-card less-imp-diff-less szb by simp

with szb
have eq: unat (word-of-nat (size-of TYPE('a) - 1)::addr-bitsize word) = size-of
(TYPE('a)) - 1
  apply (subst unat-of-nat-eq)
  apply (simp-all)
done
from word-le-unat-bound [OF c-guard-no-wrap [OF cgrd], simplified eq]
show ?thesis
  by (simp add: addr-card)
qed

definition
c-fnptr-guard-def: c-fnptr-guard (fnptr::unit ptr) ≡ ptr-val fnptr ≠ 0

lemma c-fnptr-guard-NULL [simp]: c-fnptr-guard NULL = False
  by (simp add: c-fnptr-guard-def)

lemma c-guardD:
c-guard (p::'a::mem-type ptr) ⇒ ptr-aligned p ∧ p ≠ NULL
  by (clarsimp simp: c-guard-def c-null-guard)

lemma c-guard-ptr-aligned:
c-guard p ⇒ ptr-aligned p
  by (simp add: c-guard-def)

lemma c-guard-NULL:
c-guard (p::'a::mem-type ptr) ⇒ p ≠ NULL
  by (simp add: c-guardD)

lemma c-guard-NULL-simp [simp]:
   $\neg$  c-guard (NULL::'a::mem-type ptr)
  by (force dest: c-guard-NULL)

lemma c-guard-mono:
guard-mono (c-guard::'a::mem-type ptr-guard) (c-guard::'b::mem-type ptr-guard)
  apply(clarsimp simp: guard-mono-def c-guard-def)
  subgoal premises prems for n f p
  proof -
    have guard-mono (ptr-aligned::'a::mem-type ptr-guard) (ptr-aligned::'b::mem-type
ptr-guard)
      using prems by - (rule ptr-aligned-mono)
    then show ?thesis
      using prems
      apply -
      apply(clarsimp simp: guard-mono-def ptr-aligned-def c-null-guard-def typ-uinfo-t-def)
      apply(frule field-lookup-export-uinfo-Some-rev)
      apply clarsimp

```

```

apply(drule field-tag-sub [where p=p])
apply(clarsimp simp: field-lvalue-def field-offset-def field-offset-untyped-def
typ-uinto-t-def)
apply(metis (mono-tags, opaque-lifting) export-size-of subsetD typ-uinto-t-def)
done
qed
done

```

lemma *c-guard-NULL-fl*:

```

[[ c-guard (p::'a::mem-type ptr); field-lookup (typ-info-t TYPE('a)) f 0 = Some
(t,n);
  export-uinto t = typ-uinto-t TYPE('b::mem-type) ]]
 $\implies 0 < \&(p \rightarrow f)$ 
using c-guard-mono
apply(clarsimp simp: guard-mono-def)
apply((erule allE)+, erule impE)
apply(fastforce dest: field-lookup-export-uinto-Some simp: typ-uinto-t-def)
apply(drule field-lookup-export-uinto-Some)
apply(simp add: field-lvalue-def field-offset-def field-offset-untyped-def typ-uinto-t-def)
apply(clarsimp simp: c-guard-def)
apply(drule c-null-guard)+
apply(clarsimp simp: word-neq-0-conv)
done

```

lemma *c-guard-ptr-aligned-fl*:

```

[[ c-guard (p::'a::mem-type ptr); field-lookup (typ-info-t TYPE('a)) f 0 = Some
(t,n);
  export-uinto t = typ-uinto-t TYPE('b::mem-type) ]]  $\implies$ 
ptr-aligned ((Ptr &(p  $\rightarrow$  f))::'b ptr)
using c-guard-mono
apply(clarsimp simp: guard-mono-def)
apply((erule allE)+, erule impE)
apply(fastforce dest: field-lookup-export-uinto-Some simp: typ-uinto-t-def)
apply(drule field-lookup-export-uinto-Some)
apply(simp add: c-guard-def field-lvalue-def field-offset-def field-offset-untyped-def
typ-uinto-t-def)
done

```

syntax

```

-sep-map :: 'a::c-type ptr  $\Rightarrow$  'a  $\Rightarrow$  heap-assert (-  $\mapsto$  - [56,51] 56)
-sep-map-any :: 'a::c-type ptr  $\Rightarrow$  heap-assert (-  $\mapsto$  - [56] 56)
-sep-map' :: 'a::c-type ptr  $\Rightarrow$  'a  $\Rightarrow$  heap-assert (-  $\hookrightarrow$  - [56,51] 56)
-sep-map'-any :: 'a::c-type ptr  $\Rightarrow$  heap-assert (-  $\hookrightarrow$  - [56] 56)
-tagd :: 'a::c-type ptr  $\Rightarrow$  heap-assert ( $\vdash_s$  - [99] 100)

```

translations

$p \mapsto v == p \mapsto^i (\text{CONST } c\text{-guard}) v$
 $p \mapsto - == p \mapsto^i (\text{CONST } c\text{-guard}) -$
 $p \hookrightarrow v == p \hookrightarrow^i (\text{CONST } c\text{-guard}) v$
 $p \hookrightarrow - == p \hookrightarrow^i (\text{CONST } c\text{-guard}) -$
 $\vdash_s p == \text{CONST } c\text{-guard} \vdash_s^i p$

term $x \mapsto y$
term $(x \mapsto y \wedge^* y \mapsto z) s$
term $(x \mapsto y \wedge^* y \mapsto z) \wedge^* x \mapsto y$
term $\vdash_s p$

lemma *sep-map-NULL* [*simp*]:
 $\text{NULL} \mapsto (v::'a::\text{mem-type}) = \text{sep-false}$
by (*force simp: sep-map-def sep-map-inv-def c-guard-def*
dest: lift-typ-heap-g sep-conjD c-null-guard)

lemma *sep-map'-NULL-simp* [*simp*]:
 $\text{NULL} \hookrightarrow (v::'a::\text{mem-type}) = \text{sep-false}$
apply (*simp add: sep-map'-def sep-map'-inv-def sep-conj-ac*)
apply (*subst sep-conj-com, subst sep-map-inv-def [symmetric]*)
apply *simp*
done

lemma *sep-map'-ptr-aligned*:
 $(p \hookrightarrow v) s \implies \text{ptr-aligned } p$
by (*rule c-guard-ptr-aligned*) (*erule sep-map'-g*)

lemma *sep-map'-NULL*:
 $(p \hookrightarrow (v::'a::\text{mem-type})) s \implies p \neq \text{NULL}$
by (*rule c-guard-NULL*) (*erule sep-map'-g*)

lemma *tagd-sep-false* [*simp*]:
 $\vdash_s (\text{NULL}::'a::\text{mem-type } \text{ptr}) = \text{sep-false}$
by (*auto simp: tagd-inv-def tagd-def dest!: sep-conjD s-valid-g*)

syntax (output)

$\text{-Deref} :: 'b \Rightarrow 'b (*- [1000] 1000)$
 $\text{-AssignH} :: 'b \Rightarrow 'b \Rightarrow ('a, 'p, 'f) \text{ com } ((2*- :==/ -) [30, 30] 23)$

print-translation <

let
fun *deref* (*Const* (*-antiquoteCur*, -) \$ *Const* (*h*, -)) *p* =
 if *h* = *NameGeneration.global-heap* *then* *Syntax.const -Deref* \$ *p* *else*
 raise Match
 | *deref h p* = *raise Match*
fun *lift-tr* [*h*, *p*] = *deref h p*
 | *lift-tr ts* = *raise Match*
fun *deref-update* (*Const* (*heap-update*, -) \$ *!r* \$ (*Const* (*-antiquoteCur*, -) \$

```

Const (h,-)) =
  if h=NameGeneration.global-heap then Syntax.const -AssignH $ l $ r
  else raise Match
| deref-update x = raise Match
(* First we need to determine if this is a heap update or normal assign *)
fun deref-assign (Const (-antiquoteCur,-)$Const (h,-)) r =
  if h=NameGeneration.global-heap then deref-update r else
  raise Match
| deref-assign l r = raise Match
fun assign-tr [l,r] = deref-assign l r
| assign-tr ts = raise Match
in [(CTypesDefs.lift,K lift-tr),(-Assign,K assign-tr)] end
>

```

```

print-translation <
let
  fun sep-app-tr [l,r] = Syntax.const -sep-assert $ l
  | sep-app-tr ts = raise Match
in [(sep-app,K sep-app-tr)] end
>

```

syntax *-h-t-valid* :: 'a::c-type ptr ⇒ bool (|=_t - [99] 100)

abbreviation *lift-t-c* :: heap-mem × heap-tyt-desc ⇒ 'a::c-type tyt-heap **where**
lift-t-c s == *lift-t c-guard s*

syntax *-h-t-valid* :: heap-tyt-desc ⇒ 'a::c-type ptr ⇒ bool (- |=_t - [99,99] 100)

translations

d |=_t *p* == *d*,CONST *c-guard* |=_t *p*

lemma *h-t-valid-c-guard*:

d |=_t *p* ⇒ *c-guard p*

by (*clarsimp simp: h-t-valid-def*)

lemma *h-t-valid-NULL* [*simp*]:

¬ *d* |=_t (*NULL*::'a::mem-type ptr)

by (*clarsimp simp: h-t-valid-def dest!: c-guard-NULL*)

lemma *h-t-valid-ptr-aligned*:

d |=_t *p* ⇒ *ptr-aligned p*

by (*auto simp: h-t-valid-def dest: c-guard-ptr-aligned*)

lemma *lift-t-NULL*:

lift-t-c s (*NULL*::'a::mem-type ptr) = *None*

by (*cases s*) (*auto simp: lift-t-if*)

lemma *lift-t-ptr-aligned* [*simp*]:

lift-t-c s p = Some v \implies *ptr-aligned p*
by (*auto dest: lift-t-g c-guard-ptr-aligned*)

lemmas *c-typr-rewrs = typ-rewrs h-t-valid-ptr-aligned lift-t-NULL*

datatype *'gx c-exntype = Break | Return | Continue | Goto string | Nonlocal 'gx*

definition *is-local x = (x = Break \vee x = Return \vee x = Continue \vee ($\exists l. x = Goto l$))*

lemma *is-local-simps [simp]:*

is-local Break
is-local Return
is-local Continue
is-local (Goto l)
 \neg is-local (Nonlocal x)
by (*auto simp add: is-local-def*)

primrec *the-Nonlocal where*
the-Nonlocal (Nonlocal x) = x

datatype *strictc-errortype =*

Div-0
| C-Guard
| MemorySafety
| ShiftError
| SideEffects
| ArrayBounds
| SignedArithmetic
| DontReach
| GhostStateError
| UnspecifiedSyntax
| OwnershipError
| UndefinedFunction
| AdditionalError string
| ImpossibleSpec
| AssumeError
| StackOverflow

definition *unspecified-syntax-error :: string \Rightarrow bool where*
unspecified-syntax-error s = False

record (*'g, 'l, 'e*) *state = ('g, 'l) StateSpace.state-locals +*
global-exn-var'-' :: 'e c-exntype

lemmas *hrs-simps = hrs-mem-update-def hrs-mem-def hrs-htd-update-def hrs-htd-def*

lemma *sep-map'-lift-lift*:
fixes $pa :: 'a :: \text{mem-type ptr}$ **and** $xf :: 'a \Rightarrow 'b :: \text{mem-type}$
assumes $fl: \text{field-lookup } (typ\text{-info-t } TYPE('a)) f 0 \equiv$
 $\text{Some } (adjust\text{-ti } (typ\text{-info-t } TYPE('b)) xf (xfu \circ (\lambda x -. x)), m')$
and $xf\text{-xfu}: fg\text{-cons } xf (xfu \circ (\lambda x -. x))$
shows $(pa \hookrightarrow v)(lift\text{-state } h) \Longrightarrow CTypesDefs.lift (fst h) (Ptr \&(pa \rightarrow f)) = xf v$
using $fl\ xf\text{-xfu}$
apply $(clarsimp\ simp: o\text{-def})$
apply $(rule\ sep\text{-map}'\text{-lift } [OF\ sep\text{-map}'\text{-field-map-inv, where } g1=c\text{-guard}];\ simp?)$
apply $(subst (asm)\ surjective\text{-pairing, assumption})$
apply $(simp\ add: typ\text{-uinfo-t-def export-tag-adjust-ti})$
apply $(rule\ guard\text{-mono-True})$
apply $(simp\ add: o\text{-def})$
done

lemma *lift-t-update-fld-other*:
fixes $val :: 'b :: \text{mem-type}$ **and** $ptr :: 'a :: \text{mem-type ptr}$
assumes $ft: \text{field-ti } TYPE('a) f = \text{Some } (adjust\text{-ti } (typ\text{-info-t } TYPE('b)) xf$
 $(xfu \circ (\lambda x -. x)))$
and $xf\text{-xfu}: fg\text{-cons } xf (xfu \circ (\lambda x -. x))$
and $disj: typ\text{-uinfo-t } TYPE('c :: \text{mem-type}) \perp_t typ\text{-uinfo-t } TYPE('b)$
and $cl: lift\text{-t } c\text{-guard } hp\ ptr = \text{Some } z$
shows $(lift\text{-t } c\text{-guard } (hrs\text{-mem-update } (heap\text{-update } (Ptr \&(ptr \rightarrow f))\ val)\ hp)) =$
 $(lift\text{-t } c\text{-guard } hp :: 'c :: \text{mem-type } typ\text{-heap})$
(is ?LHS = ?RHS)

proof –

let $?ati = adjust\text{-ti } (typ\text{-info-t } TYPE('b)) xf (xfu \circ (\lambda x -. x))$
have $eui: typ\text{-uinfo-t } TYPE('b) = export\text{-uinfo } (?ati)$ **using** $xf\text{-xfu}$
by $(simp\ add: typ\text{-uinfo-t-def export-tag-adjust-ti})$

have $cl': lift\text{-t } c\text{-guard } (fst\ hp, snd\ hp)\ ptr = \text{Some } z$ **using** cl **by** $simp$

show $?thesis$ **using** fl

apply $(clarsimp\ simp\ add: hrs\text{-mem-update-def split-def field-ti-def split: op\text{-tion.splits})$

by $(metis\ cl'\ disj\ eui\ fl\ h\text{-t-valid-sub lift-t-h-t-valid lift-t-heap-update-same prod.collapse)$

qed

lemma *idupdate-compupdate-fold-congE*:

assumes $idu: \bigwedge r. upd (\lambda x. accr\ r) r = r$

assumes $cpu: \bigwedge f f' r. upd f (upd f' r) = upd (f \circ f') r$

and $r: r = r'$ **and** $v: accr\ r' = v'$

and $f: \bigwedge v. v' = v \Longrightarrow f v = f' v$

shows $upd f r = upd f' r'$

proof –

have $upd (f \circ (\lambda x. accr\ r')) r' = upd (f' \circ (\lambda x. accr\ r')) r'$

by $(simp\ add: o\text{-def } f\ v)$

then show $?thesis$

by (simp add: cpu[symmetric] idu r)
qed

lemma *field-lookup-field-ti*:

field-lookup (typ-info-t TYPE('a :: c-type)) f 0 \equiv Some (a, b) \implies field-ti TYPE('a)
f = Some a

unfolding *field-ti-def* by *simp*

definition *lvar-nondet-init* ::

(('a \Rightarrow 'a) \Rightarrow (('g, 'l, 'e, 'z) state-scheme \Rightarrow ('g, 'l, 'e, 'z) state-scheme))
 \Rightarrow (('g, 'l, 'e, 'z) state-scheme, 'f, 'x) com **where**
lvar-nondet-init upd \equiv Spec {(s, t). $\exists v. t = (\text{upd } (\lambda-. v)) s$ }

axiomatization

asm-semantics :: string \Rightarrow addr list \Rightarrow (heap-mem \times 'a) \Rightarrow (addr \times (heap-mem
 \times 'a)) set **and**

asm-fetch :: 's \Rightarrow (heap-mem \times 'a) **and**

asm-store :: ('s \Rightarrow 'b) \Rightarrow (heap-mem \times 'a) \Rightarrow 's \Rightarrow 's

where

asm-semantics-enabled: $\forall iv. \text{asm-semantics } nm \text{ addr } iv \neq \{\}$ **and**

asm-store-eq: $\forall x s. \text{ghost-data } (\text{asm-store } \text{ghost-data } x s) = \text{ghost-data } s$

definition

asm-spec :: 'a itself \Rightarrow ('g \Rightarrow 'b) \Rightarrow bool \Rightarrow string
 \Rightarrow (addr \Rightarrow ('g, 'l, 'e, 'z) state-scheme \Rightarrow ('g, 'l, 'e, 'z) state-scheme)
 \Rightarrow (('g, 'l, 'e, 'z) state-scheme \Rightarrow addr list)
 \Rightarrow (('g, 'l, 'e, 'z) state-scheme \times ('g, 'l, 'e, 'z) state-scheme) set

where

asm-spec ti gdata vol spec lhs vs

= {(s, s'). $\exists (v', (gl' :: (\text{heap-mem } \times \text{'a})))$
 $\in \text{asm-semantics spec } (vs s) (\text{asm-fetch } (\text{globals } s)).$
 $s' = \text{lhs } v' (\text{globals-update } (\text{asm-store } \text{gdata } gl') s)$ }

lemma *asm-spec-enabled*:

$\exists s'. (s, s') \in \text{asm-spec ti gdata vol spec lhs vs}$

using *asm-semantics-enabled*[rule-format, **where** nm = spec

and addr=vs s **and** iv=asm-fetch (globals s)]

by (auto simp add: *asm-spec-def*)

lemma *asm-specE*:

$\llbracket (s, s') \in \text{asm-spec } (ti :: \text{'a itself}) \text{ gdata vol spec lhs vs};$
 $\bigwedge v' gl'. \llbracket (v', (gl' :: (\text{heap-mem } \times \text{'a}))) \in \text{asm-semantics spec } (vs s) (\text{asm-fetch}$
 $(\text{globals } s));$

$s' = \text{lhs } v' (\text{globals-update } (\text{asm-store } \text{gdata } gl') s);$
 $\text{gdata } (\text{asm-store } \text{gdata } gl' (\text{globals } s)) = \text{gdata } (\text{globals } s) \rrbracket$
 $\implies P \rrbracket$

$\implies P$

by (*clarsimp simp: asm-spec-def asm-store-eq*)

lemmas *state-eqE* = *arg-cong*[**where** $f = \lambda s. (globals\ s, state.more\ s)$, *elim-format*]

lemmas *asm-store-eq-helper*
 = *arg-cong2*[**where** $f = (=)$ **and** $a = asm-store\ f\ v\ s$]
arg-cong2[**where** $f = (=)$ **and** $c = asm-store\ f\ v\ s$] **for** $f\ v\ s$

definition *asm-semantic-ok-to-ignore* :: 'a itself \Rightarrow bool \Rightarrow string \Rightarrow bool **where**
asm-semantic-ok-to-ignore *ti* *volatile specifier*
 = $(\forall xs\ gl. snd\ 'asm-semantic\ specifier\ xs\ (gl :: (heap-mem \times 'a)) = \{gl\})$

lemma *heap-list-nth*:
 $m < n \Longrightarrow heap-list\ hp\ n\ p\ !\ m = hp\ (p + of-nat\ m)$
proof (*induct* m *arbitrary*: $n\ p$)
case $(0\ n'\ p')$
thus ?*case* **by** (*cases* n' , *simp-all*)
next
case $(Suc\ m'\ n'\ p')$
show ?*case*
proof (*cases* n')
case 0 **thus** ?*thesis* **using** $\langle Suc\ m' < n' \rangle$ **by** *simp*
next
case $(Suc\ n'')$
hence $m' < n''$ **using** $\langle Suc\ m' < n' \rangle$ **by** *simp*
thus ?*thesis* **using** *Suc*
by (*simp* *add*: *Suc.hyps* *ac-simps*)
qed
qed

lemma *heap-update-list-id'*:
 $heap-list\ hp\ n\ ptr = xs \Longrightarrow heap-update-list\ ptr\ xs\ hp = hp$
proof (*induct* xs *arbitrary*: $ptr\ n$)
case *Nil*
then **show** ?*case* **by** *simp*
next
case $(Cons\ x1\ xs)$
then **show** ?*case*
by (*cases* n) (*auto* *simp* *add*: *fun-upd-idem*)
qed

lemma *heap-update-list-concat-fold*:
assumes $ptr' = ptr + of-nat\ (length\ ys)$
shows $heap-update-list\ ptr'\ xs\ (heap-update-list\ ptr\ ys\ s)$
 = $heap-update-list\ ptr\ (ys\ @\ xs)\ s$
unfolding *assms*
apply (*induct* ys *arbitrary*: $ptr\ s$)
apply *simp*
apply *simp*
apply (*elim* *meta-alle*)
apply (*erule* *trans*[*rotated*])

apply (*simp add: field-simps*)
done

lemma *heap-update-list-concat-fold-hrs-mem*:
 $ptr' = ptr + of_nat\ (length\ ys) \implies$
 $hrs_mem_update\ (heap_update_list\ ptr'\ xs)$
 $\quad (hrs_mem_update\ (heap_update_list\ ptr\ ys)\ s)$
 $= hrs_mem_update\ (heap_update_list\ ptr\ (ys\ @\ xs))\ s$
by (*simp add: hrs-mem-update-def split-def*
heap-update-list-concat-fold)

lemmas *heap-update-list-concat-unfold*
 $= heap_update_list_concat_fold[OF\ refl,\ symmetric]$

lemma *intvl-nowrap*:
fixes $x :: 'a::len\ word$
shows $\llbracket y \neq 0; unat\ y + z \leq 2 \wedge len_of\ TYPE('a) \rrbracket \implies x \notin \{x + y ..+ z\}$
apply *clarsimp*
apply (*drule intvlD*)
apply *clarsimp*
apply (*simp add: unat-arith-simps*)
apply (*simp split: if-split-asm*)
by (*metis add-le-imp-le-left le-unat-voi less-imp-le-nat not-less*)

lemma *intvl-off-disj*:
fixes $x :: addr$
assumes $ylt: y \leq off$
and $zoff: z + off < 2 \wedge word_bits$
shows $\{x ..+ y\} \cap \{x + of_nat\ off ..+ z\} = \{\}$
using $ylt\ zoff$
supply *unsigned-of-nat [simp del]*
apply (*cases off = 0*)
apply *simp*
apply (*rule contrapos-pp [OF TrueI]*)
apply (*drule intvl-inter*)
apply (*erule disjE*)
subgoal **premises** *prems*
proof –
have $x \notin \{x + word_of_nat\ off ..+ z\}$
apply (*rule intvl-nowrap [where x = x and y = of-nat off :: addr and z =*
 $z]$)
using *prems*
apply –
apply (*rule of-nat-neq-0*)
apply *simp*
apply (*unfold word-bits-len-of*)

```

    apply simp
    apply (simp add: unat-of-nat word-bits-conv)
    done
  then show ?thesis using prems by simp
qed
subgoal
  apply (drule intvlD)
  apply clarsimp
  apply (drule (1) order-less-le-trans)
  apply (drule unat-cong)
  apply (simp add: unat-of-nat word-bits-conv)
  done
done

```

lemma *intvl-empty2*:
 $(\{p \text{ ..+ } n\} = \{\}) = (n = 0)$
by (*auto simp add: intvl-def*)

lemma *heap-update-list-commute*:
 $\{p \text{ ..+ } \text{length } xs\} \cap \{q \text{ ..+ } \text{length } ys\} = \{\}$
 $\implies \text{heap-update-list } p \text{ } xs \text{ } (\text{heap-update-list } q \text{ } ys \text{ } hp)$
 $= \text{heap-update-list } q \text{ } ys \text{ } (\text{heap-update-list } p \text{ } xs \text{ } hp)$
apply (*cases length xs < addr-card*)
apply (*cases length ys < addr-card*)
apply (*rule ext, simp add: heap-update-list-value*)
apply *blast*
apply (*simp-all add: addr-card intvl-overflow intvl-empty2*)
done

lemma *heap-update-commute*:
 $\llbracket \{ptr\text{-val } p \text{ ..+ } \text{size-of } TYPE('a)\} \cap \{ptr\text{-val } q \text{ ..+ } \text{size-of } TYPE('b)\} = \{\};$
 $\text{wf-fd } (typ\text{-info-t } TYPE('a)); \text{wf-fd } (typ\text{-info-t } TYPE('b)) \rrbracket$
 $\implies \text{heap-update } p \text{ } v \text{ } (\text{heap-update } q \text{ } (u :: 'b :: c\text{-type}) \text{ } h)$
 $= \text{heap-update } q \text{ } u \text{ } (\text{heap-update } p \text{ } (v :: 'a :: c\text{-type}) \text{ } h)$
apply (*simp add: heap-update-def*)
apply (*simp add: heap-update-list-commute heap-list-update-disjoint-same*
to-bytes-def length-fa-ti size-of-def Int-commute)
done

lemma *heap-update-padding-commute*:
 $\llbracket \{ptr\text{-val } p \text{ ..+ } \text{size-of } TYPE('a)\} \cap \{ptr\text{-val } q \text{ ..+ } \text{size-of } TYPE('b)\} = \{\};$
 $\text{length } bs = \text{size-of } TYPE('a); \text{length } bs' = \text{size-of } TYPE('b);$
 $\text{wf-fd } (typ\text{-info-t } TYPE('a)); \text{wf-fd } (typ\text{-info-t } TYPE('b)) \rrbracket$
 $\implies \text{heap-update-padding } p \text{ } v \text{ } bs \text{ } (\text{heap-update-padding } q \text{ } (u :: 'b :: c\text{-type}) \text{ } bs'$
 $h)$
 $= \text{heap-update-padding } q \text{ } u \text{ } bs' \text{ } (\text{heap-update-padding } p \text{ } (v :: 'a :: c\text{-type})$
 $bs \text{ } h)$
apply (*simp add: heap-update-padding-def*)

apply (*simp add: heap-update-list-commute heap-list-update-disjoint-same to-bytes-def length-fa-ti size-of-def Int-commute*)

done

lemma *heap-update-padding-heap-update-commute:*

$\llbracket \{ptr\text{-val } p \text{ ..+ size-of } TYPE('a)\} \cap \{ptr\text{-val } q \text{ ..+ size-of } TYPE('b)\} = \{\};$
 $length\ bs = size\text{-of } TYPE('a);$

$wf\text{-fd } (typ\text{-info-t } TYPE('a)); wf\text{-fd } (typ\text{-info-t } TYPE('b))\rrbracket$

$\implies heap\text{-update-padding } p\ v\ bs\ (heap\text{-update } q\ (u :: 'b :: c\text{-type})\ h)$

$= heap\text{-update } q\ u\ (heap\text{-update-padding } p\ (v :: 'a :: c\text{-type})\ bs\ h)$

apply (*simp add: heap-update-heap-update-padding-conv*)

apply (*subst heap-update-padding-commute*)

apply (*auto simp add: heap-list-update-disjoint-same heap-update-padding-def length-fa-ti size-of-def to-bytes-def*)

done

lemma *heap-update-heap-update-padding-commute:*

$\llbracket \{ptr\text{-val } p \text{ ..+ size-of } TYPE('a)\} \cap \{ptr\text{-val } q \text{ ..+ size-of } TYPE('b)\} = \{\};$
 $length\ bs' = size\text{-of } TYPE('b);$

$wf\text{-fd } (typ\text{-info-t } TYPE('a)); wf\text{-fd } (typ\text{-info-t } TYPE('b))\rrbracket$

$\implies heap\text{-update } p\ v\ (heap\text{-update-padding } q\ (u :: 'b :: c\text{-type})\ bs'\ h)$

$= heap\text{-update-padding } q\ u\ bs'\ (heap\text{-update } p\ (v :: 'a :: c\text{-type})\ h)$

apply (*simp add: heap-update-heap-update-padding-conv*)

apply (*subst heap-update-padding-commute*)

apply (*auto simp add: heap-list-update-disjoint-same heap-update-padding-def length-fa-ti size-of-def to-bytes-def*)

inf-commute)

done

lemma *addr-card-wb:*

$addr\text{-card} = 2 \wedge word\text{-bits}$

by (*simp add: addr-card-def card-word word-bits-conv*)

lemma *fold-cong':*

$a = b \implies xs = ys \implies (\bigwedge x. x \in set\ xs \implies f\ x = g\ x)$

$\implies fold\ f\ xs\ a = fold\ g\ ys\ b$

unfolding *simp-implies-def*

by (*metis fold-cong*)

lemma *arg-cong-meta:* $x = y \implies (f\ x = f\ y) \equiv True$

by *simp*

simproc-setup *arg-cong* ($\langle x = y \rangle$) = $\langle fn\ phi \implies fn\ ctxt \implies fn\ ct \implies$

let

$val\ \{f, x, y, \dots\} = @\{c\text{term-match } (fo)\ \langle ?f\ ?x = ?f\ ?y \rangle\} ct$

$val\ eq = infer\text{-instantiate } \langle x = x\ and\ y = y\ in\ cprop\ \langle x = y \rangle\ ctxt$

in

```

    try (Goal.prove-internal ctxt [] eq) (fn - => asm-full-simp-tac ctxt 1)
  |> Option.map (fn thm => (@{thm arg-cong-meta} OF [thm]))
end
handle Pattern.MATCH => NONE | Match => NONE
>

```

declare [[*simproc del: arg-cong*]]

lemma *fun-cong-meta*: $f = g \implies (f\ x = g\ x) \equiv \text{True}$
by *simp*

simproc-setup *fun-cong* ($\langle x = y \rangle$) = $\langle \text{fn phi} \Rightarrow \text{fn ctxt} \Rightarrow \text{fn ct} \Rightarrow$
let
 val {f, g, x, ...} = @{cterm-match (fo) $\langle ?f\ ?x = ?g\ ?x \rangle$ ct
 val ctxt = ctxt delsimps @{thms fun-eq-iff} (* prevent looping *)
 val eq = **infer-instantiate** $\langle f = f$ and $g = g$ in *cprop* $\langle f = g \rangle$ ctxt
in
 try (Goal.prove-internal ctxt [] eq) (fn - => asm-full-simp-tac ctxt 1)
 |> Option.map (fn thm => (@{thm fun-cong-meta} OF [thm]))
end
handle Pattern.MATCH => NONE | Match => NONE
>

declare [[*simproc del: fun-cong*]]

abbreviation

ptr-span :: $'a::\text{c-type ptr} \Rightarrow \text{addr set}$ **where**
ptr-span p $\equiv \{ \text{ptr-val } p \text{ ..+ size-of TYPE('a)} \}$

lemma *nat-index-bound*:

$j * a + k < a * b$ **if** $jk: j < b$ $k < a$ **for** $jk :: \text{nat}$

proof (*rule less-le-trans*)

show $j * a + k < (b - 1) * a + a$

using *jk* **by** (*intro add-le-less-mono mult-le-mono1*) *simp*

show $(b - 1) * a + a \leq a * b$

using *jk* **by** (*simp add: diff-mult-distrib*)

qed

lemma *disj-ptr-span-ptr-neq*:

disjnt (*ptr-span* (p::'a::mem-type ptr)) (*ptr-span* (q::'a::mem-type ptr)) \implies
 $p \neq q$

unfolding *disjnt-def*

by (*metis empty-iff inf.idem mem-type-self*)

lemma *field-lvalue-same-conv'*: $\&(p::'a::\text{c-type ptr} \rightarrow f) = \&(q::'a::\text{c-type ptr} \rightarrow f)$

$\longleftrightarrow p = q$

by (*simp add: field-lvalue-def*)

11.27 (Partial) Pointer Lenses

11.27.1 *pointer-lense*

named-theorems

read-write-same **and**

read-write-other **and**

write-cong **and**

map-other-commute

locale *pointer-lense* =

fixes $r :: 's \Rightarrow 'a :: \text{mem-type } ptr \Rightarrow 'b$

fixes $w :: 'a \ ptr \Rightarrow ('b \Rightarrow 'b) \Rightarrow 's \Rightarrow 's$

assumes *read-write-same*[*read-write-same*]: $r (w \ p \ f \ s) \ p = f (r \ s \ p)$

assumes *write-same*: $f (r \ s \ p) = (r \ s \ p) \Longrightarrow w \ p \ f \ s = s$

assumes *write-comp*[*update-compose, simp*]: $w \ p \ f (w \ p \ g \ s) = w \ p (f \ o \ g) \ s$

assumes *write-other-commute*[*map-other-commute*]: $\text{disjnt } (ptr\text{-span } p) (ptr\text{-span } q) \Longrightarrow w \ q \ g (w \ p \ f \ s) = w \ p \ f (w \ q \ g \ s)$

begin

lemma *read-write-other*[*read-write-other*]:

$\text{disjnt } (ptr\text{-span } p) (ptr\text{-span } p') \Longrightarrow r (w \ p \ f \ s) \ p' = r \ s \ p'$

apply (*subst write-same*[*symmetric, of* $\lambda x. r \ s \ p' \ s \ p'$])

apply *simp*

apply (*subst write-other-commute*[*symmetric*])

apply *assumption*

apply (*rule read-write-same*)

done

lemma *write-cong*[*write-cong*]: $f (r \ s \ p) = f' (r \ s \ p) \Longrightarrow w \ p \ f \ s = w \ p \ f' \ s$

by (*metis K-record-comp read-write-same write-comp write-same*)

lemma *write-read*: $w \ p (\lambda \cdot. r \ s \ p) \ s = s$

by (*simp add: write-same*)

lemma *write-id*: $w \ p (\lambda x. x) \ s = s$

by (*simp add: write-same*)

lemma *read-write-pointwise-id*: $r (w \ p (\lambda x. x) \ s) = r \ s$

using *write-same* **by** *simp*

end

11.27.2 *partial-pointer-lense*

locale *partial-pointer-lense* = *is-scene* m **for** $m :: 'a :: \text{c-type } scene +$

fixes $r :: 'h \Rightarrow 'a \ ptr \Rightarrow 'a \ \text{upd}$

fixes $w :: 'a \ ptr \Rightarrow 'a \Rightarrow 'h \ \text{upd}$

assumes *r-w*[*simp*]: $r (w \ p \ x \ h) \ p \ y = m \ x \ y$

assumes *w-w*[*simp*]: $w \ p \ x (w \ p \ y \ h) = w \ p \ x \ h$

assumes $w\text{-}r[\textit{simp}]$: $w\ p\ (r\ h\ p\ x)\ h = h$
assumes $w\text{-}m[\textit{simp}]$: $w\ p\ (m\ x\ y)\ h = w\ p\ x\ h$
assumes $w\text{-}w\text{-}disj$: $disjnt\ (ptr\text{-}span\ p)\ (ptr\text{-}span\ q) \implies w\ p\ x\ (w\ q\ y\ h) = w\ q\ y$
 $(w\ p\ x\ h)$
begin

lemma $m\text{-}r[\textit{simp}]$: $m\ (r\ h\ p\ x)\ y = r\ h\ p\ y$
by $(metis\ r\text{-}w\ w\text{-}r)$

lemma $r\text{-}w\text{-}disj[\textit{simp}]$: $disjnt\ (ptr\text{-}span\ p)\ (ptr\text{-}span\ q) \implies r\ (w\ q\ x\ h)\ p\ y = r\ h$
 $p\ y$
by $(metis\ r\text{-}w\ w\text{-}r\ w\text{-}w\text{-}disj)$

lemma $r\text{-}m$: $r\ h\ p\ (m\ x\ y) = r\ h\ p\ y$
by $(metis\ m\text{-}r\ right)$

end

lemma $partial\text{-}pointer\text{-}lenseI$:

fixes $get\ upd\ r\ w$
assumes $lense\ get\ upd$
assumes $pointer\text{-}lense\ r\ w$
shows $partial\text{-}pointer\text{-}lense$
 $(\lambda a\ b.\ upd\ (\lambda\cdot.\ get\ a)\ b)$
 $(\lambda h\ p.\ upd\ (\lambda\cdot.\ r\ h\ p))$
 $(\lambda p\ x.\ w\ p\ (\lambda\cdot.\ get\ x))$

proof –

interpret $lense\ get\ upd$ **by** $fact$
interpret $pointer\text{-}lense\ r\ w$ **by** $fact$
show $?thesis$
apply $unfold\text{-}locales$
subgoal **by** $simp$
subgoal **by** $(simp\ add:\ comp\text{-}def)$
subgoal **by** $simp$
subgoal **by** $(simp\ add:\ read\text{-}write\text{-}same)$
subgoal **by** $(simp\ add:\ comp\text{-}def)$
subgoal **by** $(simp\ add:\ write\text{-}same)$
subgoal **by** $simp$
subgoal **by** $(simp\ add:\ write\text{-}other\text{-}commute\ disjnt\text{-}def)$
done

qed

lemma $pointer\text{-}lense\text{-}of\text{-}lense\text{-}fld$:

assumes $lense\ r\ w$
shows $pointer\text{-}lense\ (\lambda h\ p.\ r\ h\ (PTR('a)\ \&(p\rightarrow f)))\ (\lambda p\ v.\ w\ (upd\text{-}fun\ (PTR('a)\ \&(p\rightarrow f))\ v))$

proof –

interpret $lense\ r\ w$ **by** $fact$
show $?thesis$

```

apply unfold-locales
apply simp
apply (simp add: upd-same upd-fun.upd-same)
apply simp
apply simp
apply (subst upd-fun-commute)
apply (simp-all add: field-lvalue-same-conv' disj-ptr-span-ptr-neq eq-commute)
done
qed

```

lemma *partial-pointer-lenseI-fld:*

```

fixes get :: 'a::mem-type  $\Rightarrow$  'b and upd r w
assumes 1: lense get upd
assumes 2: lense r w
shows partial-pointer-lense
  ( $\lambda a$  b. upd ( $\lambda \cdot$ . get a) b)
  ( $\lambda h$  p. upd ( $\lambda \cdot$ . r h (PTR('b)  $\&$ (p $\rightarrow$ f))))
  ( $\lambda p$  x. w (upd-fun (PTR('b)  $\&$ (p $\rightarrow$ f)) ( $\lambda \cdot$ . get x)))
by (rule partial-pointer-lenseI[OF 1 pointer-lense-of-lense-fld, OF 2])

```

lemma *partial-pointer-lense-compose:*

```

assumes partial-pointer-lense m1 r1 w1
assumes partial-pointer-lense m2 r2 w2
assumes m[simp]:  $\bigwedge a$   $b$   $c. m1$   $a$  ( $m2$   $b$   $c$ ) =  $m2$   $b$  ( $m1$   $a$   $c$ )
assumes w[simp]:  $\bigwedge p$   $a$   $q$   $b$   $h. p = q \vee disjnt$  (ptr-span p) (ptr-span q)  $\implies$ 
  w1 p a (w2 q b h) = w2 q b (w1 p a h)
shows partial-pointer-lense ( $\lambda a$  b. m1 a (m2 a b))
  ( $\lambda h$  p x. r1 h p (r2 h p x))
  ( $\lambda p$  x h. w1 p x (w2 p x h))

```

proof –

```

interpret A: partial-pointer-lense m1 r1 w1 by fact
interpret B: partial-pointer-lense m2 r2 w2 by fact

```

```

have [simp]: m1 (m2 a b) c = m1 b c for a b c
by (metis A.idem A.left m)

```

```

have r1-w2[simp]: p = q  $\vee$  disjnt (ptr-span p) (ptr-span q)  $\implies$ 
  r1 (w2 p x h) q y = r1 h q y for p q h x y
by (metis (no-types, lifting) A.r-w A.w-r disjnt-comm w)

```

```

have r1-r2: r1 h p (r2 h p x) = r2 h p (r1 h p x) for h p x
by (metis A.m-r B.m-r m)

```

show *?thesis*

```

apply unfold-locales
apply (simp-all add: A.w-w-disj B.w-w-disj disjnt-sym)
apply (simp add: r1-r2)
apply (simp flip: m)
done

```


qed

lemma *partial-pointer-lense-id*:

partial-pointer-lense ($\lambda a. id$) ($\lambda h p. id$) ($\lambda p x. id$)

by (*simp add: partial-pointer-lense-def partial-pointer-lense-axioms-def*)

lemma *partial-pointer-lense-fold*:

fixes $rs :: ('h \Rightarrow 'a :: c\text{-type } ptr \Rightarrow 'a \Rightarrow 'a) \text{ list}$

assumes $length\ ms = length\ rs$ $length\ ms = length\ ws$

assumes *list-all* ($\lambda(m, r, w). \text{partial-pointer-lense } m\ r\ w$) (*zip ms (zip rs ws)*)

assumes $\forall a\ b\ c. \text{distinct-prop } (\lambda m1\ m2. m1\ a\ (m2\ b\ c) = m2\ b\ (m1\ a\ c))\ ms$

assumes $\forall p\ a\ q\ b\ h.$

$\text{distinct-prop } (\lambda w1\ w2. p = q \vee \text{disjnt } (\text{ptr-span } p)\ (\text{ptr-span } q) \longrightarrow$

$w1\ p\ a\ (w2\ q\ b\ h) = w2\ q\ b\ (w1\ p\ a\ h))\ ws$

shows *partial-pointer-lense* ($\lambda a\ b. \text{fold } (\lambda m. m\ a)\ ms\ b$)

($\lambda h\ p\ x. \text{fold } (\lambda r. r\ h\ p)\ rs\ x$)

($\lambda p\ x\ h. \text{fold } (\lambda w. w\ p\ x)\ ws\ h$)

using *assms*

proof (*induction ms arbitrary: ws rs*)

case (*Cons m ms ws' rs'*)

from *Cons.premis(1,2)* **obtain** $r\ rs\ w\ ws$ **where** [*simp*]: $rs' = r \# rs$ $ws' = w$
 $\# ws$

by (*cases ws'; cases rs'; auto*)

have *ppl-ms*:

partial-pointer-lense

($\lambda a. \text{fold } (\lambda m. m\ a)\ ms$)

($\lambda h\ p. \text{fold } (\lambda r. r\ h\ p)\ rs$)

($\lambda p\ x. \text{fold } (\lambda w. w\ p\ x)\ ws$)

using *Cons.premis* **by** (*intro Cons.IH; simp*)

from *Cons.premis(3)* **have** *ppl-m*: *partial-pointer-lense m r w* **by** *simp*

show *?case*

apply (*simp, rule partial-pointer-lense-compose[OF ppl-ms ppl-m]*)

proof –

fix $a\ b\ c$

from *Cons.premis(4)* **have** *list-all* ($\lambda m'. \forall a\ b\ c. m\ a\ (m'\ b\ c) = m'\ b\ (m\ a\ c)$)
ms

by (*simp add: list-all-iff*)

then show $\text{fold } (\lambda m. m\ a)\ ms\ (m\ b\ c) = m\ b\ (\text{fold } (\lambda m. m\ a)\ ms\ c)$

proof (*induction ms arbitrary: c*)

case (*Cons m' ms*)

then show *?case* **by** *simp metis*

qed *simp*

next

fix $p\ q :: 'a\ ptr$ **and** $a\ b :: 'a$ **and** $h :: 'h$

assume $p = q \vee \text{disjnt } (\text{ptr-span } p)\ (\text{ptr-span } q)$

then have $q = p \vee \text{disjnt } (\text{ptr-span } q)\ (\text{ptr-span } p)$

by (*simp add: eq-commute disjnt-comm*)

with *Cons.premis(5)* **have** *list-all* ($\lambda w'. \forall b\ a\ h. w\ q\ a\ (w'\ p\ b\ h) = w'\ p\ b\ (w$
 $q\ a\ h))\ ws$

```

    by (auto simp add: list-all-iff)
  then show fold (λw. w p a) ws (w q b h) = w q b (fold (λw. w p a) ws h)
  proof (induction ws arbitrary: a b h)
    case (Cons w' ws)
    then show ?case
      by simp metis
  qed simp
qed
qed (simp add: partial-pointer-lense-id)

```

lemma *pointer-lense-of-partial-pointer-lense*:

```

  assumes partial-pointer-lense m r w
  assumes [simp]: ∧a b. m a b = a
  shows pointer-lense (λh p. r h p x) (λp f h. w p (f (r h p x)) h)
  proof -
    interpret partial-pointer-lense m r w by fact
    show ?thesis
      by unfold-locales (simp-all add: w-w-disj disjnt-def Int-commute)
  qed

```

lemma *pointer-lense-of-partials*:

```

  fixes rs :: ('h ⇒ 'a::mem-type ptr ⇒ 'a ⇒ 'a) list
  assumes *:
    length ms = length rs length ms = length ws
    list-all (λ(m, r, w). partial-pointer-lense m r w) (zip ms (zip rs ws))
  and **:
    ∧a b c. distinct-prop (λm1 m2. m1 a (m2 b c) = m2 b (m1 a c)) ms
    ∧p a q b h. distinct-prop (λw1 w2. p = q ∨ disjnt (ptr-span p) (ptr-span q) →
      w1 p a (w2 q b h) = w2 q b (w1 p a h)) ws
  assumes ms: ∧a b. fold (λm. m a) ms b = a
  assumes R: ∧h p. R h p = fold (λr. r h p) rs x
  assumes W: ∧p f h. W p f h = fold (λw. w p (f (R h p))) ws h
  shows pointer-lense R W
  unfolding R[abs-def] W[abs-def]
  apply (rule pointer-lense-of-partial-pointer-lense[OF partial-pointer-lense-fold,
    OF *])
  subgoal using **(1) by simp
  subgoal using **(2) by simp
  subgoal by (rule ms)
  done

```

end

theory *Padding-Equivalence*

```

  imports
    TypHeap
    SepCode
    CProof

```

More-Lib

begin

lemma *field-lookup-padding-field-name*:

fixes

t :: ('a, 'b) *typ-info* **and**

st :: ('a, 'b) *typ-info-struct* **and**

ts :: ('a, 'b) *typ-info-tuple list* **and**

x :: ('a, 'b) *typ-info-tuple*

shows

*field-lookup t [f] n = Some (s, m) \implies padding-field-name f \implies wf-padding t \implies
is-padding-tag s*

*field-lookup-struct st [f] n = Some (s, m) \implies padding-field-name f \implies wf-padding-struct
st \implies*

is-padding-tag s

*field-lookup-list ts [f] n = Some (s, m) \implies padding-field-name f \implies wf-padding-list
ts \implies*

is-padding-tag s

*field-lookup-tuple x [f] n = Some (s, m) \implies padding-field-name f \implies wf-padding-tuple
x \implies*

is-padding-tag s

by (*induct t and st and ts and x arbitrary: n m and n m and n m and n m*)
(*auto split: if-split-asm option.splits*)

lemma *field-lookup-access-ti-take-drop'*:

fixes *t*::('a, 'b) *typ-info*

and *st*::('a, 'b) *typ-info-struct*

and *ts*::('a, 'b) *typ-info-tuple list*

and *x*::('a, 'b) *typ-info-tuple*

shows

*field-lookup t f m = Some (s, m + n) \implies wf-fd t \implies wf-desc t \implies wf-size-desc
t \implies length bs = size-td t \implies*

*access-ti s v (take (size-td s) (drop n bs)) =
(take (size-td s) (drop n (access-ti t v bs)))*

*field-lookup-struct st f m = Some (s, m + n) \implies wf-fd-struct st \implies wf-desc-struct
st \implies wf-size-desc-struct st \implies length bs = size-td-struct st \implies*

*access-ti s v (take (size-td s) (drop n bs)) =
(take (size-td s) (drop n (access-ti-struct st v bs)))*

*field-lookup-list ts f m = Some (s, m + n) \implies wf-fd-list ts \implies wf-desc-list ts
 \implies wf-size-desc-list ts \implies length bs = size-td-list ts \implies*

*access-ti s v (take (size-td s) (drop n bs)) =
(take (size-td s) (drop n (access-ti-list ts v bs)))*

*field-lookup-tuple x f m = Some (s, m + n) \implies wf-fd-tuple x \implies wf-desc-tuple
x \implies wf-size-desc-tuple x \implies length bs = size-td-tuple x \implies*

*access-ti s v (take (size-td s) (drop n bs)) =
(take (size-td s) (drop n (access-ti-tuple x v bs)))*

```

proof (induct t and st and ts and x arbitrary: f m n s bs v and f m n s bs v and
f m n s bs v and f m n s bs v)
  case (TypDesc algn st nm)
  then show ?case
    by (auto split: if-split-asm)
      (metis TypDesc.prem1(2) TypDesc.prem1(5) access-ti.simps le-refl length-fa-ti
take-all)
next
  term TypDesc
  case (TypScalar algn st d)
  then show ?case by auto
next
  case (TypAggregate ts)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
  obtain d nm y where x: x = DTuple d nm y by (cases x) auto
  from Cons-typ-desc.prem1 obtain
    lbs: length bs = size-td-tuple x + size-td-list fs and
    wf-fd-x: wf-fd-tuple (DTuple d nm y) and
    wf-desc-x: wf-desc-tuple (DTuple d nm y) and
    wf-fd-fs: wf-fd-list fs and
    wf-size-desc-x: wf-size-desc-tuple (DTuple d nm y) and
    wf-size-desc-fs: wf-size-desc-list fs and
    nm-notin: nm ∉ dt-snd ' set fs and
    wf-desc-fs: wf-desc-list fs and
    commutes: fu-commutes (update-ti-t d) (update-ti-list-t fs)
  by (auto simp add: x)

from lbs
have lbs-drop: length (drop (size-td-tuple x) bs) = size-td-list fs
  by simp

from lbs
have lbs-take: length (take (size-td-tuple x) bs) = size-td-tuple (DTuple d nm y)
  by (simp add: x)

from lbs wf-fd-x
have eq1: length (access-ti d v (take (size-td d) bs)) = size-td d
  by (metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps x)

from Cons-typ-desc.prem1 obtain f1 fxs
  where f: f = f1 # fxs
  by (cases f) auto

show ?case

```

```

proof (cases f1 = nm)
  case True
  show ?thesis
  proof (cases field-lookup d fxs m)
    case None
    from Cons-typ-desc.premis field-lookup-list-None [OF nm-notin]
    have False
    by (simp add: True None f x)
    thus ?thesis by simp
  next
  case (Some r)
  from Cons-typ-desc.premis have r: r = (s, m + n)
    by (simp add: x True Some f)
  hence fl: field-lookup-tuple (DTuple d nm y) f m = Some (s, m + n)
    by (simp add: f True Some r)

  from td-set-wf-size-desc(4)[OF wf-size-desc-x td-set-tuple-field-lookup-tupleD,
OF fl]
  have wf-size-desc s .
  from wf-size-desc-gt(1)[OF this]
  have 0 < size-td s .

  with td-set-tuple-offset-size-m [OF td-set-tuple-field-lookup-tupleD, OF fl]
  have n-le: n < size-td d
    by auto

  have bound: size-td s + (m + n - m) ≤ size-td-tuple (DTuple d nm y) by
fact
  from bound
  have take-eq: (take (size-td s) (drop n (take (size-td d) bs))) =
    (take (size-td s) (drop n bs))
    by (simp add: take-drop)

  from Cons-typ-desc.hyps(1)[simplified x, OF fl wf-fd-x wf-desc-x wf-size-desc-x
lbs-take, of v] lbs bound
  show ?thesis by (simp add: x True Some r take-eq eq1)
  qed
next
  case False
  with Cons-typ-desc.premis
  have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, m + n)
    by (clarsimp simp add: x f False)
  hence n-bound: size-td d ≤ n
    by (meson field-lookup-offset-le(3) nat-add-left-cancel-le)

  from fl
  have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, (m + size-td
d) + (n - size-td d))

```

```

    by (metis Nat.diff-cancel field-lookup-list-offset2 field-lookup-list-offsetD)
  from n-bound
  have take-eq: (take (size-td s) (drop (n - size-td d + size-td d) bs)) =
    (take (size-td s) (drop n bs))
  by simp

  from Cons-typ-desc.hyps(2) [OF fl wf-fd-fs wf-desc-fs wf-size-desc-fs lbs-drop,
of v]
  False n-bound
  show ?thesis
  by (simp add: x f eq1)
qed
next
case (DTuple-typ-desc d nm y)
then show ?case apply (cases f) by (auto split: if-split-asm)
qed

lemma field-lookup-access-ti-take-drop:
  field-lookup t f 0 = Some (s, n)  $\implies$  wf-fd t  $\implies$  wf-desc t  $\implies$  wf-size-desc t  $\implies$ 
  length bs = size-td t  $\implies$ 
  access-ti s v (take (size-td s) (drop n bs)) =
  (take (size-td s) (drop n (access-ti t v bs)))
  using field-lookup-access-ti-take-drop' [where m=0] by auto

lemma field-lookup-nth-focus':
  fixes t::('a, 'b) typ-info
  and st::('a, 'b) typ-info-struct
  and ts::('a, 'b) typ-info-tuple list
  and x::('a, 'b) typ-info-tuple
  shows
   $\llbracket$ field-lookup t f m = Some (s, m + n);  $n \leq i$ ;  $i < n + \text{size-td } s$ ; length bs = size-td
  t;
  wf-fd t; wf-desc t; wf-size-desc t $\rrbracket \implies$ 
  access-ti t v bs ! i = access-ti s v (take (size-td s) (drop n bs)) ! (i - n)

 $\llbracket$ field-lookup-struct st f m = Some (s, m + n);  $n \leq i$ ;  $i < n + \text{size-td } s$ ; length bs
= size-td-struct st;
wf-fd-struct st; wf-desc-struct st; wf-size-desc-struct st $\rrbracket \implies$ 
  access-ti-struct st v bs ! i = access-ti s v (take (size-td s) (drop n bs)) ! (i - n)

 $\llbracket$ field-lookup-list ts f m = Some (s, m + n);  $n \leq i$ ;  $i < n + \text{size-td } s$ ; length bs =
size-td-list ts;
wf-fd-list ts; wf-desc-list ts; wf-size-desc-list ts $\rrbracket \implies$ 
  access-ti-list ts v bs ! i = access-ti s v (take (size-td s) (drop n bs)) ! (i - n)

 $\llbracket$ field-lookup-tuple x f m = Some (s, m + n);  $n \leq i$ ;  $i < n + \text{size-td } s$ ; length bs =
size-td-tuple x;
wf-fd-tuple x; wf-desc-tuple x; wf-size-desc-tuple x $\rrbracket \implies$ 

```

$access\text{-}ti\text{-}tuple\ x\ v\ bs\ !\ i = access\text{-}ti\ s\ v\ (take\ (size\text{-}td\ s)\ (drop\ n\ bs))\ !\ (i - n)$
proof (*induct* *t* **and** *st* **and** *ts* **and** *x* *arbitrary*: *f m n i s bs v* **and** *f m n i s bs v*
and *f m n i s bs v* **and** *f m n i s bs v*)
case (*TypDesc* *algn st nm*)
then show *?case* **by** (*auto split: if-split-asm*)
next
case (*TypScalar* *sz algn d*)
then show *?case* **by** *auto*
next
case (*TypAggregate* *ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc* *x fs*)
obtain *d nm y* **where** *x*: $x = DTuple\ d\ nm\ y$ **by** (*cases x*) *auto*
from *Cons-typ-desc.prem*s **obtain**
lbs: $length\ bs = size\text{-}td\text{-}tuple\ x + size\text{-}td\text{-}list\ fs$ **and**
wf-fd-x: *wf-fd-tuple* (*DTuple d nm y*) **and**
wf-desc-x: *wf-desc-tuple* (*DTuple d nm y*) **and**
wf-fd-fs: *wf-fd-list fs* **and**
wf-size-desc-x: *wf-size-desc-tuple* (*DTuple d nm y*) **and**
wf-size-desc-fs: *wf-size-desc-list fs* **and**
nm-notin: $nm \notin dt\text{-}snd\ 'set\ fs$ **and**
wf-desc-fs: *wf-desc-list fs* **and**
i-lower: $n \leq i$ **and**
i-upper: $i < n + size\text{-}td\ s$
by (*auto simp add: x*)

from *lbs*
have *lbs-drop*: $length\ (drop\ (size\text{-}td\text{-}tuple\ x)\ bs) = size\text{-}td\text{-}list\ fs$
by *simp*

from *lbs*
have *lbs-take*: $length\ (take\ (size\text{-}td\text{-}tuple\ x)\ bs) = size\text{-}td\text{-}tuple\ (DTuple\ d\ nm\ y)$
by (*simp add: x*)

from *lbs wf-fd-x*
have *eq1*: $length\ (access\text{-}ti\ d\ v\ (take\ (size\text{-}td\ d)\ bs)) = size\text{-}td\ d$
by (*metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps x*)

from *Cons-typ-desc.prem*s **obtain** *f1 fxs*
where *f*: $f = f1\ \#\ fxs$
by (*cases f*) *auto*

thm *Cons-typ-desc.hyps* [*simplified x*]

show *?case*

```

proof (cases f1 = nm)
  case True
  show ?thesis
  proof (cases field-lookup d fxs m)
    case None
    from Cons-typ-desc.prem1 field-lookup-list-None [OF nm-notin]
    have False
    by (simp add: True None f x)
    thus ?thesis by simp
  next
  case (Some r)
  from Cons-typ-desc.prem1 have r: r = (s, m + n)
    by (simp add: x True Some f)
  hence fl: field-lookup-tuple (DTuple d nm y) f m = Some (s, m + n)
    by (simp add: f True Some r)
  from td-set-wf-size-desc(4)[OF wf-size-desc-x td-set-tuple-field-lookup-tupleD,
OF fl]
  have wf-size-desc s .
  from wf-size-desc-gt(1)[OF this]
  have 0 < size-td s .

  with td-set-tuple-offset-size-m [OF td-set-tuple-field-lookup-tupleD, OF fl]
  have n-le: n < size-td d
    by auto

  have i-in-d: i < size-td d
    using ⟨size-td s + (m + n - m) ≤ size-td-tuple (DTuple d nm y)⟩ i-upper
by auto
  have bound: size-td s + (m + n - m) ≤ size-td-tuple (DTuple d nm y) by
fact
  from bound
  have take-eq: (take (size-td s) (drop n (take (size-td d) bs))) =
    (take (size-td s) (drop n bs))
    by (simp add: take-drop)
  from Cons-typ-desc.hyps(1)[simplified x, OF fl i-lower i-upper lbs-take wf-fd-x
wf-desc-x wf-size-desc-x, of v] lbs bound
  show ?thesis
    by (simp add: x True Some r take-eq eq1 nth-append i-in-d)
  qed
next
case False
with Cons-typ-desc.prem1
have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, m + n)
  by (clarsimp simp add: x f False)
hence n-bound: size-td d ≤ n
  by (meson field-lookup-offset-le(3) nat-add-left-cancel-le)

from fl

```



```

have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, (m + size-td d) + (n - size-td d))
  by (metis Nat.diff-cancel field-lookup-list-offset2 field-lookup-list-offsetD)
from n-bound
have take-eq: (take (size-td s) (drop (n - size-td d + size-td d) bs)) =
  (take (size-td s) (drop n bs))
  by simp

have i-lower': n - size-td d ≤ i - size-td d
  using diff-le-mono i-lower by blast

have i-upper': i - size-td d < n - size-td d + size-td s
  using i-lower i-upper by linarith

have i-notin-d:  $\neg i < \text{size-td } d$ 
  by (meson i-lower leD less-le-trans n-bound)

from Cons-typ-desc.hyps(2) [OF fl i-lower' i-upper' lbs-drop wf-fd-fs wf-desc-fs wf-size-desc-fs, where v=v]
  False n-bound
show ?thesis
  by (simp add: x f eq1 nth-append i-notin-d)
qed
next
  case (DTuple-typ-desc d nm y)
then show ?case apply (cases f) by (auto split: if-split-asm)
qed

```

```

lemma field-lookup-nth-focus:
   $\llbracket \text{field-lookup } t f 0 = \text{Some } (s, n); n \leq i; i < n + \text{size-td } s; \text{length } bs = \text{size-td } t; \text{wf-fd } t; \text{wf-desc } t; \text{wf-size-desc } t \rrbracket \implies$ 
   $\text{access-ti } t v bs ! i = \text{access-ti } s v (\text{take } (\text{size-td } s) (\text{drop } n bs)) ! (i - n)$ 
  using field-lookup-nth-focus' [where m=0] by auto

```

```

lemma field-lookup-nth-update-focus':
  fixes t::('a, 'b) typ-info
  and st::('a, 'b) typ-info-struct
  and ts::('a, 'b) typ-info-tuple list
  and x::('a, 'b) typ-info-tuple
shows
   $\llbracket \text{field-lookup } t f m = \text{Some } (s, m + n); n \leq i; i < n + \text{size-td } s; \text{length } bs = \text{size-td } t; \text{wf-fd } t; \text{wf-desc } t; \text{wf-size-desc } t \rrbracket \implies$ 
   $\text{access-ti } t v (bs[i := b]) =$ 
   $\text{super-update-bs } (\text{access-ti } s v ((\text{take } (\text{size-td } s) (\text{drop } n bs))[i - n := b]))$ 
   $(\text{access-ti } t v bs) n$ 

```

```

 $\llbracket \text{field-lookup-struct } st f m = \text{Some } (s, m + n); n \leq i; i < n + \text{size-td } s; \text{length } bs$ 

```

```

= size-td-struct st;
  wf-fd-struct st; wf-desc-struct st; wf-size-desc-struct st]] ==>
  access-ti-struct st v (bs[i := b]) =
    super-update-bs (access-ti s v ((take (size-td s) (drop n bs))[i - n := b]))
      (access-ti-struct st v bs) n

[[field-lookup-list ts f m = Some (s, m + n); n ≤ i; i < n + size-td s; length bs =
size-td-list ts;
  wf-fd-list ts; wf-desc-list ts; wf-size-desc-list ts]] ==>
  access-ti-list ts v (bs[i := b]) =
    super-update-bs (access-ti s v ((take (size-td s) (drop n bs))[i - n := b]))
      (access-ti-list ts v bs) n

[[field-lookup-tuple x f m = Some (s, m + n); n ≤ i; i < n + size-td s; length bs =
size-td-tuple x;
  wf-fd-tuple x; wf-desc-tuple x; wf-size-desc-tuple x]] ==>
  access-ti-tuple x v (bs[i := b]) =
    super-update-bs (access-ti s v ((take (size-td s) (drop n bs))[i - n := b]))
      (access-ti-tuple x v bs) n
proof (induct t and st and ts and x arbitrary: f m n i s bs v and f m n i s bs v
and f m n i s bs v and f m n i s bs v)
case (TypDesc algn st nm)
  then show ?case
    by (auto split: if-split-asm simp add: super-update-bs-def)
      (metis TypDesc.prem1(4) TypDesc.prem1(5) access-ti.simps le-refl length-fa-ti
length-list-update)
  next
    case (TypScalar sz algn d)
    then show ?case by auto
  next
    case (TypAggregate ts)
    then show ?case by auto
  next
    case Nil-typ-desc
    then show ?case by auto
  next
    case (Cons-typ-desc x fs)
    obtain d nm y where x: x = DTuple d nm y by (cases x) auto
    from Cons-typ-desc.prem1 obtain
      lbs: length bs = size-td-tuple x + size-td-list fs and
      wf-fd-x: wf-fd-tuple (DTuple d nm y) and
      wf-desc-x: wf-desc-tuple (DTuple d nm y) and
      wf-fd-fs: wf-fd-list fs and
      wf-size-desc-x: wf-size-desc-tuple (DTuple d nm y) and
      wf-size-desc-fs: wf-size-desc-list fs and
      nm-notin: nm ∉ dt-snd ' set fs and
      wf-desc-fs: wf-desc-list fs and
      i-lower: n ≤ i and
      i-upper: i < n + size-td s

```

```

    by (auto simp add: x)

from lbs
have lbs-drop: length (drop (size-td-tuple x) bs) = size-td-list fs
  by simp

from lbs
have lbs-take: length (take (size-td-tuple x) bs) = size-td-tuple (DTuple d nm y)
  by (simp add: x)

from lbs wf-fd-x
have eq1: length (access-ti d v (take (size-td d) bs)) = size-td d
  by (metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps x)

from Cons-typ-desc.prems obtain f1 fxs
  where f: f = f1 # fxs
  by (cases f) auto

thm Cons-typ-desc.hyps [simplified x]

show ?case
proof (cases f1 = nm)
  case True
  show ?thesis
  proof (cases field-lookup d fxs m)
    case None
    from Cons-typ-desc.prems field-lookup-list-None [OF nm-notin]
    have False
      by (simp add: True None f x)
    thus ?thesis by simp
  next
  case (Some r)
  from Cons-typ-desc.prems have r: r = (s, m + n)
    by (simp add: x True Some f)
  hence fl: field-lookup-tuple (DTuple d nm y) f m = Some (s, m + n)
    by (simp add: f True Some r)
  from td-set-wf-size-desc(4)[OF wf-size-desc-x td-set-tuple-field-lookup-tupleD,
OF fl]
  have wf-size-desc s .
  from wf-size-desc-gt(1)[OF this]
  have 0 < size-td s .

with td-set-tuple-offset-size-m [OF td-set-tuple-field-lookup-tupleD, OF fl]
have n-le: n < size-td d
  by auto

have i-in-d: i < size-td d
  using (size-td s + (m + n - m) ≤ size-td-tuple (DTuple d nm y)) i-upper

```

```

by auto
  have bound:  $\text{size-td } s + (m + n - m) \leq \text{size-td-tuple } (DTuple\ d\ nm\ y)$  by
fact
  from bound
  have take-eq:  $\text{take } (\text{size-td } s) (\text{drop } n (\text{take } (\text{size-td } d) bs)) =$ 
 $\text{take } (\text{size-td } s) (\text{drop } n bs)$ 
  by (simp add: take-drop)
  from bound
  have take-upd-eq:  $((\text{take } (\text{size-td } d) bs)[i := b]) = (\text{take } (\text{size-td } d) (bs[i :=$ 
b]))
  by (simp add: take-update-swap)

  have take-eq1:  $\text{take } (\text{size-td } s) (\text{drop } n (\text{take } (\text{size-td } d) bs)) = \text{take } (\text{size-td}$ 
s) (drop n bs)
  using take-eq by blast

  have l-take-s:  $\text{length } (\text{take } (\text{size-td } s) (\text{drop } n bs)) = \text{size-td } s$ 
  using bound lbs-take by auto

  have sz-s:  $\text{size-td } s \leq \text{length } bs - n$ 
  using l-take-s by auto

  from fl
  have fd-cons-s: fd-cons s
  using wf-fd-consD wf-fd-field-lookup(4) wf-fd-x by blast
  have l-acc-s:  $\text{length } (\text{access-ti } s\ v ((\text{take } (\text{size-td } s) (\text{drop } n bs))[i - n := b]))$ 
   $= \text{size-td } s$ 
  by (simp add: fd-cons-length [OF fd-cons-s] sz-s eq1)

  have take-eq2:  $\text{take } (\text{size-td } s) (\text{drop } n (\text{take } (\text{size-td } d) bs))[i - n := b] =$ 
 $\text{take } (\text{size-td } s) (\text{drop } n bs)[i - n := b]$ 
  using take-eq1 by presburger

  note hyp = Cons-typ-desc.hyps(1)[simplified x, OF fl i-lower i-upper lbs-take
wf-fd-x wf-desc-x wf-size-desc-x, of v,
  simplified, simplified x, simplified, simplified take-upd-eq]
  from lbs bound
  show ?thesis
  apply (simp add: x True Some r take-eq take-upd-eq eq1 list-update-append
i-in-d)
  apply (simp add: hyp)
  apply (subst super-update-bs-append1)
  apply (simp add: l-acc-s eq1)
  apply (simp add: take-eq2)
  done
qed
next
case False
with Cons-typ-desc.prems

```

```

have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, m + n)
  by (clarsimp simp add: x f False)
hence n-bound: size-td d ≤ n
  by (meson field-lookup-offset-le(3) nat-add-left-cancel-le)

from fl
have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, (m + size-td
d) + (n - size-td d))
  by (metis Nat.diff-cancel field-lookup-list-offset2 field-lookup-list-offsetD)
from n-bound
have take-eq: (take (size-td s) (drop (n - size-td d + size-td d) bs)) =
  (take (size-td s) (drop n bs))
  by simp

have i-lower': n - size-td d ≤ i - size-td d
  using diff-le-mono i-lower by blast

have i-upper': i - size-td d < n - size-td d + size-td s
  using i-lower i-upper by linarith

have i-notin-d: ¬ i < size-td d
  by (meson i-lower leD less-le-trans n-bound)

from i-notin-d have take-d-eq: take (size-td d) (bs[i := b]) = take (size-td d)
bs
  by simp
have drop-shift: (drop (size-td d) bs)[i - size-td d := b] = drop (size-td d) (bs[i
:= b])
  by (metis drop-update-swap i-notin-d le-def)

note hyp = Cons-typ-desc.hyps(2) [OF fl i-lower' i-upper' lbs-drop wf-fd-fs
wf-desc-fs wf-size-desc-fs, where v=v,
simplified x, simplified, simplified drop-shift]
from False n-bound
show ?thesis
  apply (simp add: x f eq1 nth-append i-notin-d)
  apply (subst super-update-bs-append2)
  apply (simp add: eq1)
  apply (simp add: take-d-eq add: eq1)
  apply (simp add: hyp)
done
qed
next
case (DTuple-typ-desc d nm y)
then show ?case apply (cases f) by (auto split: if-split-asm)
qed

lemma field-lookup-nth-update-focus:
shows

```

```

[[field-lookup t f 0 = Some (s, n); n ≤ i; i < n + size-td s; length bs = size-td t;
 wf-fd t; wf-desc t; wf-size-desc t]] ⇒
  access-ti t v (bs[i := b]) =
    super-update-bs (access-ti s v ((take (size-td s) (drop n bs))[i - n := b]))
      (access-ti t v bs) n
using field-lookup-nth-update-focus' [where m=0] by auto

```

context *mem-type*

begin

lemma *mem-type-field-lookup-access-ti-take-drop*:

assumes *fl*: field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)

assumes *lbs*: length bs = size-of TYPE('a)

shows access-ti s v (take (size-td s) (drop n bs)) =

take (size-td s) (drop n (access-ti (typ-info-t TYPE('a)) v bs))

proof –

have *wf-fd*: wf-fd (typ-info-t TYPE('a))

by (simp add: wf-fdp-fdD wf-lf-fdp)

have *wf-desc*: wf-desc (typ-info-t TYPE('a)) **by** simp

have *wf-size*: wf-size-desc (typ-info-t TYPE('a)) **by** simp

from field-lookup-access-ti-take-drop [OF *fl wf-fd wf-desc wf-size lbs* [simplified size-of-def]]

show ?thesis .

qed

lemma *mem-type-update-ti-super-update-bs*:

assumes *fl*: field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)

assumes *lbs*: length bs = size-of TYPE('a)

assumes *lv*: length v = size-td s

shows update-ti s v (update-ti (typ-info-t TYPE('a)) bs w) =

update-ti (typ-info-t TYPE('a)) (super-update-bs v bs n) w

proof –

have *wf-fd*: wf-fd (typ-info-t TYPE('a))

by (simp add: wf-fdp-fdD wf-lf-fdp)

have *lsuper*: length (super-update-bs v bs n) = size-td (typ-info-t TYPE('a))

by (metis add.commute field-lookup-offset-size' *fl lbs length-super-update-bs local.size-of-def lv*)

from fi-fu-consistentD [OF *fl wf-fd lbs* [simplified size-of-def] *lv, of w*] *lbs*

show ?thesis

by (simp add: update-ti-t-def *lsuper lv size-of-def*)

qed

lemma *mem-type-update-ti-from-bytes-super-update-bs*:

assumes *fl*: field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)

assumes *lbs*: length bs = size-of TYPE('a)

assumes *lv*: length v = size-td s

shows update-ti s v (from-bytes bs) = update-ti (typ-info-t TYPE('a)) (super-update-bs

v bs n) undefined

proof –

from *mem-type-update-ti-super-update-bs* [OF *fl lbs lv*, of undefined]

show *?thesis*

by (*simp add: from-bytes-def update-ti-t-def lbs size-of-def*)

qed

lemma *mem-type-field-lookup-nth-focus*:

assumes *fl*: *field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)*

assumes *i-lower*: $n \leq i$

assumes *i-upper*: $i < n + \text{size-td } s$

assumes *lbs* : $\text{length } lbs = \text{size-of } TYPE('a)$

shows *access-ti (typ-info-t TYPE('a)) v bs ! i =*

access-ti s v (take (size-td s) (drop n bs)) ! (i - n)

proof –

have *wf-fd*: *wf-fd (typ-info-t TYPE('a))*

by (*simp add: wf-fdp-fdD wf-lf-fdp*)

have *wf-desc*: *wf-desc (typ-info-t TYPE('a))* **by** *simp*

have *wf-size*: *wf-size-desc (typ-info-t TYPE('a))* **by** *simp*

from *field-lookup-nth-focus* [OF *fl i-lower i-upper lbs [simplified size-of-def]* *wf-fd wf-desc wf-size*]

show *?thesis* .

qed

lemma *mem-type-field-lookup-nth-update-focus*:

assumes *fl*: *field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)*

assumes *i-lower*: $n \leq i$

assumes *i-upper*: $i < n + \text{size-td } s$

assumes *lbs* : $\text{length } lbs = \text{size-of } TYPE('a)$

shows

access-ti (typ-info-t TYPE('a)) v (bs[i := b]) =

super-update-bs (access-ti s v ((take (size-td s) (drop n bs))[i - n := b]))

(access-ti (typ-info-t TYPE('a)) v bs) n

proof –

have *wf-fd*: *wf-fd (typ-info-t TYPE('a))*

by (*simp add: wf-fdp-fdD wf-lf-fdp*)

have *wf-desc*: *wf-desc (typ-info-t TYPE('a))* **by** *simp*

have *wf-size*: *wf-size-desc (typ-info-t TYPE('a))* **by** *simp*

from *field-lookup-nth-update-focus* [OF *fl i-lower i-upper lbs [simplified size-of-def]* *wf-fd wf-desc wf-size*]

show *?thesis* .

qed

end

ML $\langle @\{term \ xs[i := v]\} \rangle$

lemma *nth-take-drop-index-shift*:
 $n \leq i \implies i < m + n \implies m + n \leq \text{length } xs \implies \text{take } m (\text{drop } n \text{ } xs) ! (i - n) = xs ! i$
proof (*induct xs arbitrary: n i m*)
 case *Nil*
 then show *?case by simp*
next
 case (*Cons x1 xs*)
 then show *?case by auto*
qed

lemma *super-update-bs-update-index-shift*: $n \leq i \implies i - n < \text{length } bs \implies n + \text{length } bs \leq \text{length } xbs \implies$
 $(\text{super-update-bs } bs \text{ } xbs \text{ } n)[i := b] = \text{super-update-bs } (bs[i - n := b]) \text{ } xbs \text{ } n$
 apply (*simp add: super-update-bs-def*)
 apply (*simp add: list-update-append*)
 done

lemma *super-update-bs-nth-shift*:
 $n \leq i \implies i - n < \text{length } bs \implies n + \text{length } bs \leq \text{length } xbs \implies \text{super-update-bs } bs \text{ } xbs \text{ } n ! i = bs ! (i - n)$
 apply (*simp add: super-update-bs-def*)
 apply (*simp add: nth-append*)
 done

lemma (*in mem-type*) *field-lookup-is-padding-byte-outer-to-inner*:
 assumes *fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)*
 assumes *lower-bound-x: n ≤ x*
 assumes *upper-bound-x: x < n + size-td s*
 assumes *is-padding: padding-base.is-padding-byte (access-ti (typ-info-t (TYPE('a)))) (update-ti (typ-info-t (TYPE('a)))) (size-td (typ-info-t (TYPE('a)))) x*
 shows *padding-base.is-padding-byte (access-ti s) (update-ti s) (size-td s) (x - n)*
proof (*rule padding-base.is-padding-byteI*)
 from *lower-bound-x upper-bound-x* **show** $x - n < \text{size-td } s$ **by** *simp*
next
 fix *v::'a and bs::byte list*
 assume $x - n < \text{size-td } s$
 assume *lbs: length bs = size-td s*
 from *fl* **have** $\text{sz: size-td } s + n \leq \text{size-of } TYPE('a)$
 by (*simp add: field-lookup-offset-size' size-of-def*)

 obtain *px sfx xbs* **where**
 xbs: xbs = px @ bs @ sfx **and**
 lpx: length px = n **and**
 lsfx: length sfx = size-of TYPE('a) - n - size-td s
 by (*metis Ex-list-of-length*)
 with *sz* **have** *lbs: length lbs = size-of TYPE('a)*
 by (*metis add-leD2 add-le-imp-le-diff lbs le-add-diff-inverse length-append*)
 from *xbs lpx lsfx xbs lbs* **have** *bs: (take (size-td s) (drop n xbs)) = bs*

by *simp*

have *bound*: $x - n < \text{size-td } s$ **by** *fact*

have *lacc*: $\text{length } (\text{access-ti } (\text{typ-info-t } \text{TYPE}('a)) \ v \ xbs) = \text{size-of } \text{TYPE}('a)$
by (*simp add*: *length-fa-ti lxs size-of-def*)

from *mem-type-field-lookup-access-ti-take-drop* [*OF fl lxs, simplified bs, of v*]

have *acc-conv*: $\text{access-ti } s \ v \ bs =$
 $\text{take } (\text{size-td } s) (\text{drop } n (\text{access-ti } (\text{typ-info-t } \text{TYPE}('a)) \ v \ xbs)) .$

from *padding-base.is-padding-byte-acc-eq* [*OF is-padding, of xbs v*] *lxs*

have *access-ti* (*typ-info-t* *TYPE*('a)) *v xbs ! x = xbs ! x*
by (*simp add*: *size-of-def*)

moreover **have** *xbs ! x = bs ! (x - n)*
using *xbs lpx lsfx lxs bound*

apply (*simp add*: *xbs*)

by (*metis bs drop-append-miracle le-def lower-bound-x nth-append nth-take xbs*)

moreover **have** $\text{take } (\text{size-td } s) (\text{drop } n (\text{access-ti } (\text{typ-info-t } \text{TYPE}('a)) \ v \ xbs))$
 $! (x - n) =$
 $\text{access-ti } (\text{typ-info-t } \text{TYPE}('a)) \ v \ xbs ! x$
using *lacc lxs bound lower-bound-x upper-bound-x*
by (*metis less-diff-conv2 nth-take-drop-index-shift sz*)

ultimately show $\text{access-ti } s \ v \ bs ! (x - n) = bs ! (x - n)$
by (*simp add*: *acc-conv*)

next

fix *v::'a* **and** *bs::byte list* **and** *b::byte*

assume $x - n < \text{size-td } s$

assume *lbs*: $\text{length } bs = \text{size-td } s$

from *fl* **have** *sz*: $\text{size-td } s + n \leq \text{size-of } \text{TYPE}('a)$
by (*simp add*: *local.field-lookup-offset-size local.size-of-def*)

obtain *pfx sfx xbs* **where**

xbs: $xbs = pfx @ bs @ sfx$ **and**

lpx: $\text{length } pfx = n$ **and**

lsfx: $\text{length } sfx = \text{size-of } \text{TYPE}('a) - n - \text{size-td } s$

by (*metis Ex-list-of-length*)

with *sz* **have** *lxs*: $\text{length } xbs = \text{size-of } \text{TYPE}('a)$
by (*metis add-leD2 add-le-imp-le-diff lbs le-add-diff-inverse length-append*)

from *xbs lpx lsfx xbs lbs* **have** *bs*: $(\text{take } (\text{size-td } s) (\text{drop } n \ xbs)) = bs$
by *simp*

have *v-upd-conv*: $(\text{update-ti } (\text{typ-info-t } \text{TYPE}('a)) (\text{to-bytes } v \ xbs) \ v) = v$
by (*simp add*: *fu-fa length-fa-ti local.size-of-def local.to-bytes-def lxs update-ti-update-ti-t*)

have *lxto*: $\text{length } (\text{to-bytes } v \ xbs) = \text{size-of } \text{TYPE}('a)$
by (*simp add*: *length-fa-ti lxs size-of-def to-bytes-def*)

```

from lxto lbs
have lsuper: length (super-update-bs bs (to-bytes v xbs) n) = size-of TYPE('a)
  using sz by auto

have bound:  $x - n < \text{size-td } s$  by fact

have xbs-super:  $xbs = \text{super-update-bs } bs \ xbs \ n$ 
  by (simp add: lpx super-update-bs-def xbs)
from mem-type-update-ti-super-update-bs [OF fl lxto lbs, of v, simplified v-upd-conv]
have upd-eq1:
  update-ti s bs v =
    update-ti (typ-info-t TYPE('a)) (super-update-bs bs (to-bytes v xbs) n) v .

have lbs': length (bs[x - n := b]) = size-td s
  using lbs by auto

from mem-type-update-ti-super-update-bs [OF fl lxto lbs', of v, simplified v-upd-conv]
have upd-eq2:
  update-ti s (bs[x - n := b]) v =
    update-ti (typ-info-t TYPE('a)) (super-update-bs (bs[x - n := b]) (to-bytes v
xbs) n) v .

from lxto lxbs lbs lower-bound-x upper-bound-x bound sz
have super-eq:  $(\text{super-update-bs } bs \ (to\text{-bytes } v \ xbs) \ n)[x := b] = \text{super-update-bs}$ 
 $(bs[x - n := b]) \ (to\text{-bytes } v \ xbs) \ n$ 
  by (simp add: super-update-bs-update-index-shift)

from padding-base.is-padding-byte-upd-eq [OF is-padding, of (super-update-bs bs
 $(to\text{-bytes } v \ xbs) \ n) \ v \ b] \ lsuper
have upd-pad: update-ti (typ-info-t TYPE('a)) (super-update-bs bs (to-bytes v
 $xbs) \ n) \ v =$ 
  update-ti (typ-info-t TYPE('a)) ((super-update-bs bs (to-bytes v xbs) n)[x
 $:= b]) \ v$ 
  by (simp add: size-of-def)
show update-ti s bs v = update-ti s (bs[x - n := b]) v
  apply (simp add: upd-eq1)
  apply (simp add: upd-pad)
  apply (simp add: upd-eq2)
  apply (simp add: super-eq)
  done
qed

lemma (in mem-type) field-lookup-is-padding-byte-inner-to-outer:
  assumes fl: field-lookup (typ-info-t (TYPE('a))) f 0 = Some (s, n)
  assumes lower-bound-x:  $n \leq x$ 
  assumes upper-bound-x:  $x < n + \text{size-td } s$ 
  assumes is-padding: padding-base.is-padding-byte (access-ti s) (update-ti s) (size-td
 $s) \ (x - n)$$ 
```

```

shows padding-base.is-padding-byte (access-ti (typ-info-t (TYPE('a)))) (update-ti
(typ-info-t (TYPE('a)))) (size-td (typ-info-t (TYPE('a)))) x
proof (rule padding-base.is-padding-byteI)
  from lower-bound-x upper-bound-x fl
  show x < size-td (typ-info-t TYPE('a))
    using field-lookup-offset-size' by fastforce
next
  fix v::'a and bs::byte list
  assume sz: x < size-td (typ-info-t TYPE('a))
  assume lbs: length bs = size-td (typ-info-t TYPE('a))
  from mem-type-field-lookup-nth-focus [OF fl lower-bound-x upper-bound-x, sim-
simplified size-of-def, OF lbs]
  have eq1:
    access-ti (typ-info-t TYPE('a)) v bs ! x =
      access-ti s v (take (size-td s) (drop n bs)) ! (x - n) .
  from lbs fl
  have l-take-drop: length (take (size-td s) (drop n bs)) = size-td s
    using field-lookup-offset-size' by fastforce

from padding-base.is-padding-byte-acc-eq [OF is-padding l-take-drop, of v]
have eq2:
  access-ti s v (take (size-td s) (drop n bs)) ! (x - n) =
    take (size-td s) (drop n bs) ! (x - n) .
have eq3:
  take (size-td s) (drop n bs) ! (x - n) = bs ! x
  by (metis add.commute field-lookup-offset-size' fl lbs lower-bound-x nth-take-drop-index-shift
upper-bound-x)

show access-ti (typ-info-t TYPE('a)) v bs ! x = bs ! x
  apply (simp add: eq1)
  apply (simp add: eq2)
  apply (simp add: eq3)
done
next
  fix v::'a and bs::byte list and b::byte
  assume sz: x < size-td (typ-info-t TYPE('a))
  assume lbs: length bs = size-td (typ-info-t TYPE('a))

  have v-upd-conv: (update-ti (typ-info-t TYPE('a)) (to-bytes v bs) v) = v
    by (simp add: fu-fa lbs length-fa-ti to-bytes-def update-ti-update-ti-t)

  have lto: length (to-bytes v bs) = size-td (typ-info-t TYPE('a))
    by (simp add: lbs length-fa-ti to-bytes-def)

  note upd-focus = mem-type-update-ti-super-update-bs [OF fl, simplified size-of-def,
OF lto, where w=v, simplified v-upd-conv]

from lbs lower-bound-x upper-bound-x sz
have lbs1: length (take (size-td s) (drop n bs)) = size-td s

```

by (*metis add-le-imp-le-diff field-lookup-offset-size' fl length-drop length-take min.absorb2*)

from *lbs lower-bound-x upper-bound-x*
have *bs-upd-eq*: (*take (size-td s) (drop n bs)[x - n := b]* = *take (size-td s) (drop n (bs[x := b]))*)
by (*metis drop-update-swap take-update-swap*)

from *lbs lbs1 lower-bound-x upper-bound-x*
have *bs2*: *length (take (size-td s) (drop n (bs[x := b])))* = *size-td s*
by (*metis bs-upd-eq length-list-update*)

from *lbs lbs1*
have *bs-super-conv*: (*super-update-bs (take (size-td s) (drop n bs)) bs n*) = *bs*
by (*metis append-take-drop-id super-update-bs-def take-drop-append*)

from *lbs lbs1 lower-bound-x upper-bound-x*
have *bs-upd-super-conv*: (*super-update-bs (take (size-td s) (drop n (bs[x := b]))) bs n*) = *bs[x := b]*
by (*metis add.commute bs-super-conv bs-upd-eq field-lookup-offset-size' fl nat-diff-less super-update-bs-update-index-shift*)

from *mem-type-update-ti-super-update-bs [OF fl, simplified size-of-def, OF lbs lbs1, simplified bs-super-conv, where w=v]*
have *eq1*:
update-ti (typ-info-t TYPE('a)) bs v =
update-ti s (take (size-td s) (drop n bs)) (update-ti (typ-info-t TYPE('a)) bs v) **by** *simp*

from *mem-type-update-ti-super-update-bs [OF fl, simplified size-of-def, OF lbs lbs2, simplified bs-upd-super-conv, where w=v]*
have *eq2*:
update-ti s (take (size-td s) (drop n (bs[x := b])))
(update-ti (typ-info-t TYPE('a)) bs v) =
update-ti (typ-info-t TYPE('a)) (bs[x := b]) v .

thm *upd-focus [OF lbs1]*

note *pad-eq= padding-base.is-padding-byte-upd-eq [OF is-padding, OF lbs1, where b=b]*

show *update-ti (typ-info-t TYPE('a)) bs v* =
update-ti (typ-info-t TYPE('a)) (bs[x := b]) v
apply (*subst eq1*)
apply (*simp add: pad-eq*)
apply (*simp add: bs-upd-eq*)
apply (*simp add: eq2*)
done

qed

lemma (in *mem-type*) *field-lookup-is-padding-byte*:
assumes *fl*: *field-lookup* (*typ-info-t* *TYPE('a)*) *f* 0 = *Some* (*s*, *n*)
assumes *lower-bound-x*: $n \leq x$
assumes *upper-bound-x*: $x < n + \text{size-td } s$
shows
padding-base.is-padding-byte (*access-ti* *s*) (*update-ti* *s*) (*size-td* *s*) ($x - n$) \longleftrightarrow
padding-base.is-padding-byte
(*access-ti* (*typ-info-t* (*TYPE('a)*))) (*update-ti* (*typ-info-t* (*TYPE('a)*))) (*size-td*
(*typ-info-t* (*TYPE('a)*))) *x*
using *field-lookup-is-padding-byte-outer-to-inner* [*OF fl lower-bound-x upper-bound-x*]
field-lookup-is-padding-byte-inner-to-outer [*OF fl lower-bound-x upper-bound-x*]
by *blast*

lemma (in *mem-type*) *field-lookup-is-value-byte-outer-to-inner*:
assumes *fl*: *field-lookup* (*typ-info-t* *TYPE('a)*) *f* 0 = *Some* (*s*, *n*)
assumes *lower-bound-x*: $n \leq x$
assumes *upper-bound-x*: $x < n + \text{size-td } s$
assumes *is-value*: *padding-base.is-value-byte* (*access-ti* (*typ-info-t* (*TYPE('a)*)))
(*update-ti* (*typ-info-t* (*TYPE('a)*))) (*size-td* (*typ-info-t* (*TYPE('a)*))) *x*
shows *padding-base.is-value-byte* (*access-ti* *s*) (*update-ti* *s*) (*size-td* *s*) ($x - n$)
proof (*rule padding-base.is-value-byteI*)
from *lower-bound-x upper-bound-x* **show** $x - n < \text{size-td } s$ **by** *simp*
next
fix *v::'a* **and** *bs::byte list* **and** *bs'::byte list*
assume $x - n < \text{size-td } s$
assume *lbs*: *length* *bs* = *size-td* *s*
assume *lbs'*: *length* *bs'* = *size-td* *s*
from *fl* **have** *sz*: *size-td* *s* + *n* \leq *size-of* *TYPE('a)*
by (*metis* (*no-types*, *lifting*) *diff-add-zero diff-is-0-eq dual-order.trans field-lookup-offset-size'*
size-of-def)

obtain *pfx sfx xbs* **where**
xbs: *xbs* = *pfx* @ *bs* @ *sfx* **and**
lpfx: *length* *pfx* = *n* **and**
lsfx: *length* *sfx* = *size-of* *TYPE('a)* - *n* - *size-td* *s*
by (*metis* *Ex-list-of-length*)

with *sz* **have** *lbs*: *length* *xbs* = *size-of* *TYPE('a)*
by (*metis* *add-leD2 add-le-imp-le-diff lbs le-add-diff-inverse length-append*)
from *xbs lpfx lsfx xbs lbs* **have** *bs*: (*take* (*size-td* *s*) (*drop* *n* *xbs*)) = *bs*
by *simp*

obtain *pfx' sfx' xbs'* **where**
xbs': *xbs'* = *pfx'* @ *bs'* @ *sfx'* **and**
lpfx': *length* *pfx'* = *n* **and**
lsfx': *length* *sfx'* = *size-of* *TYPE('a)* - *n* - *size-td* *s*
by (*metis* *Ex-list-of-length*)

```

with sz have lbs': length lbs' = size-of TYPE('a)
  by (metis add-leD2 add-le-imp-le-diff lbs' le-add-diff-inverse length-append)

from lbs' lpx' lsx' lbs' lbs' have bs': (take (size-td s) (drop n lbs')) = bs'
  by simp

have bound: x - n < size-td s by fact

have lacc: length (access-ti (typ-info-t TYPE('a)) v lbs) = size-of TYPE('a)
  by (simp add: fd-cons-length lbs size-of-def wf-fd-consD)

have v-upd-conv: (update-ti (typ-info-t TYPE('a)) (to-bytes v lbs) v) = v
  by (simp add: fu-fa lacc lbs size-of-def to-bytes-def update-ti-update-ti-t)

have lxto: length (to-bytes v lbs) = size-of TYPE('a)
  by (simp add: lacc to-bytes-def)

from lxto lbs
have lsuper: length (super-update-bs bs (to-bytes v lbs) n) = size-of TYPE('a)
  using sz by auto

have bound: x - n < size-td s by fact

have lbs-super: lbs = super-update-bs bs lbs n
  by (simp add: lpx super-update-bs-def lbs)

from mem-type-update-ti-super-update-bs [OF fl lxto lbs, of v, simplified v-upd-conv]
have upd-eq1:
  update-ti s bs v =
    update-ti (typ-info-t TYPE('a)) (super-update-bs bs (to-bytes v lbs) n) v .

have idx-shift1:
  take (size-td s) (drop n
    (access-ti (typ-info-t TYPE('a))
      (update-ti (typ-info-t TYPE('a)) (super-update-bs bs (to-bytes v lbs) n)
v) lbs'))
    ! (x - n)
    = access-ti (typ-info-t TYPE('a))
      (update-ti (typ-info-t TYPE('a)) (super-update-bs bs (to-bytes v lbs) n)
v) lbs'
    ! x
  by (metis bound fd-cons-length less-diff-conv2 local.wf-fd lower-bound-x lbs'
    nth-take-drop-index-shift size-of-def sz wf-fd-consD)

from lxto lbs lower-bound-x upper-bound-x sz
have super-nth: super-update-bs bs (to-bytes v lbs) n ! x = bs ! (x - n)
  by (simp add: super-update-bs-nth-shift)
note val-eq = padding-base.is-value-byte-acc-upd-cancel [OF is-value lsuper [simplified
size-of-def] lbs' [simplified size-of-def]]

```

```

note acc-eq = mem-type-field-lookup-access-ti-take-drop [OF fl lxs', simplified
bs']
show access-ti s (update-ti s bs v) bs' ! (x - n) = bs ! (x - n)
  apply (simp add: upd-eq1)
  apply (simp add: acc-eq)
  apply (simp add: idx-shift1)
  apply (simp add: val-eq)
  apply (simp add: super-nth)
  done
next
fix v::'a and bs::byte list and b::byte
assume x - n < size-td s
assume lbs: length bs = size-td s

from fl have sz: size-td s + n ≤ size-of TYPE('a)
by (metis (no-types, lifting) diff-add-zero diff-is-0-eq dual-order.trans field-lookup-offset-size'
size-of-def)

obtain pfx sfx xbs where
  xbs: xbs = pfx @ bs @ sfx and
  lpfx: length pfx = n and
  lsfx: length sfx = size-of TYPE('a) - n - size-td s
by (metis Ex-list-of-length)

with sz have lxs: length xbs = size-of TYPE('a)
by (metis add-leD2 add-le-imp-le-diff lbs le-add-diff-inverse length-append)
from xbs lpfx lsfx xbs lbs have bs: (take (size-td s) (drop n xbs)) = bs
by simp

obtain xbs' where
  xbs': xbs' = pfx @ bs [x - n := b] @ sfx by blast
from lxs lbs xbs have lxs': length xbs' = size-of TYPE('a)
using xbs' by auto
from xbs' lpfx lxs' lbs have bs': (take (size-td s) (drop n xbs')) = bs[x - n := b]
by simp

from lbs lower-bound-x upper-bound-x sz lpfx lsfx
have xbs'-conv: xbs' = xbs[x := b]
by (simp add: xbs xbs' list-update-append)

note eq1 = mem-type-field-lookup-access-ti-take-drop [OF fl lxs, simplified bs, of
v]
note eq2 = mem-type-field-lookup-access-ti-take-drop [OF fl lxs', simplified bs',
of v]
note val-eq = padding-base.is-value-byte-acc-eq [OF is-value lxs [simplified size-of-def],
b=b]
where b=b]
show access-ti s v bs = access-ti s v (bs[x - n := b])
  apply (simp add: eq1)

```

```

  apply (simp add: val-eq)
  apply (simp add: eq2)
  apply (simp add: xbs'-conv)
done
qed

```

```

lemma (in mem-type) field-lookup-is-value-byte-inner-to-outer:
  assumes fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)
  assumes lower-bound-x:  $n \leq x$ 
  assumes upper-bound-x:  $x < n + \text{size-td } s$ 
  assumes is-value: padding-base.is-value-byte (access-ti s) (update-ti s) (size-td s)
  (x - n)
  shows padding-base.is-value-byte (access-ti (typ-info-t (TYPE('a)))) (update-ti
  (typ-info-t (TYPE('a)))) (size-td (typ-info-t (TYPE('a)))) x
proof (rule padding-base.is-value-byteI)
  from lower-bound-x upper-bound-x fl
  show  $x < \text{size-td } (\text{typ-info-t } \text{TYPE}('a))$ 
  using field-lookup-offset-size' by fastforce
next
  fix v::'a and bs::byte list and bs'::byte list
  assume sz:  $x < \text{size-td } (\text{typ-info-t } \text{TYPE}('a))$ 
  assume lbs: length bs = size-td (typ-info-t TYPE('a))
  assume lbs': length bs' = size-td (typ-info-t TYPE('a))

  from fl have sz: size-td s + n  $\leq$  size-of TYPE('a)
  by (metis (no-types, lifting) diff-add-zero diff-is-0-eq dual-order.trans field-lookup-offset-size'
  size-of-def)

  from sz lbs
  have super-bs: (super-update-bs (take (size-td s) (drop n bs)) bs n) = bs
  by (metis (no-types, lifting) append.assoc append-len2 append-take-drop-id
  diff-diff-cancel
  length-drop nat-move-sub-le size-of-def super-update-bs-def take-add)
  from lower-bound-x upper-bound-x lbs sz
  have l-take-drop: length (take (size-td s) (drop n bs)) = size-td s
  by (simp add: size-of-def)

  from lower-bound-x upper-bound-x lbs' sz
  have l-take-drop': length (take (size-td s) (drop n bs')) = size-td s
  by (simp add: size-of-def)

  from lbs lower-bound-x upper-bound-x sz
  have take-drop-eq: take (size-td s) (drop n bs) ! (x - n) = bs ! x
  by (simp add: size-of-def)

note upd-focus = mem-type-update-ti-super-update-bs [OF fl, simplified size-of-def,
OF lbs l-take-drop, simplified super-bs, symmetric ]
note acc-focus = mem-type-field-lookup-nth-focus [OF fl lower-bound-x upper-bound-x,

```



```

simplified size-of-def, OF lbs]
  note cancel = padding-base.is-value-byte-acc-upd-cancel [OF is-value l-take-drop
l-take-drop]
  show access-ti (typ-info-t TYPE('a)) (update-ti (typ-info-t TYPE('a)) bs v) bs'
! x =
  bs ! x
  apply (subst upd-focus)
  apply (simp add: acc-focus)
  apply (simp add: cancel)
  apply (simp add: take-drop-eq)
  done
next
fix v::'a and bs::byte list and b::byte
assume sz: x < size-td (typ-info-t TYPE('a))
assume lbs: length bs = size-td (typ-info-t TYPE('a))

from fl have sz: size-td s + n ≤ size-of TYPE('a)
by (metis (no-types, lifting) diff-add-zero diff-is-0-eq dual-order.trans field-lookup-offset-size'
size-of-def)

note eq1 = mem-type-field-lookup-access-ti-take-drop [OF fl, simplified size-of-def,
OF lbs ]

from sz lower-bound-x upper-bound-x lbs
have l-take-s: length (take (size-td s) (drop n bs)) = size-td s
by (simp add: size-of-def)

from lbs sz lower-bound-x upper-bound-x
have super-eq:
  super-update-bs (access-ti s v (take (size-td s) (drop n bs)))
  (access-ti (typ-info-t TYPE('a)) v bs) n = (access-ti (typ-info-t TYPE('a))
v bs)
by (metis append-take-drop-id eq1 l-take-s length-drop length-fa-ti
length-take local.wf-fd super-update-bs-def take-drop-append)

note focus1 = mem-type-field-lookup-nth-update-focus [OF fl lower-bound-x up-
per-bound-x, simplified size-of-def, OF lbs]
note pad-eq = padding-base.is-value-byte-acc-eq [OF is-value l-take-s, symmetric]
show access-ti (typ-info-t TYPE('a)) v bs =
  access-ti (typ-info-t TYPE('a)) v (bs[x := b])

  apply (simp add: focus1)
  apply (subst pad-eq)
  apply (simp add: super-eq)
  done
qed

```

```

lemma (in mem-type) field-lookup-is-value-byte:
  assumes fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)
  assumes lower-bound-x: n ≤ x
  assumes upper-bound-x: x < n + size-td s
  shows padding-base.is-value-byte (access-ti s) (update-ti s) (size-td s) (x - n)
  ←→
  padding-base.is-value-byte (access-ti (typ-info-t (TYPE('a))))
    (update-ti (typ-info-t (TYPE('a)))) (size-td (typ-info-t (TYPE('a)))) x
  using field-lookup-is-value-byte-inner-to-outer [OF fl lower-bound-x upper-bound-x]
  field-lookup-is-value-byte-outer-to-inner [OF fl lower-bound-x upper-bound-x]
  by blast

thm padding-base.eq-padding-def
thm padding-base.eq-upto-padding-def

```

```

lemma td-set-component-descs-independent:
  fixes t::'a xtyp-info
  and st::'a xtyp-info-struct
  and ts::'a xtyp-info-tuple list
  and x::'a xtyp-info-tuple
shows
  (s, n) ∈ td-set t m ⇒ component-descs-independent t ⇒ component-descs-independent
  s
  (s, n) ∈ td-set-struct st m ⇒ component-descs-independent-struct st ⇒ com-
  ponent-descs-independent s
  (s, n) ∈ td-set-list ts m ⇒ component-descs-independent-list ts ⇒ compo-
  nent-descs-independent s
  (s, n) ∈ td-set-tuple x m ⇒ component-descs-independent-tuple x ⇒ compo-
  nent-descs-independent s
proof (induct t and st and ts and x arbitrary: m and m and m and m)
case (TypDesc algn st nm)
then show ?case by auto
next
  case (TypScalar sz algn d)
  then show ?case by auto
next
  case (TypAggregate ts)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
  then show ?case by auto
next
  case (DTuple-typ-desc d nm y)

```

then show *?case by auto*
qed

lemma *td-set-wf-component-descs*:

fixes *t::'a xtyp-info*
and *st::'a xtyp-info-struct*
and *ts::'a xtyp-info-tuple list*
and *x::'a xtyp-info-tuple*

shows

$(s, n) \in \text{td-set } t \ m \implies \text{wf-component-descs } t \implies \text{wf-component-descs } s$
 $(s, n) \in \text{td-set-struct } st \ m \implies \text{wf-component-descs-struct } st \implies \text{wf-component-descs } s$
 $(s, n) \in \text{td-set-list } ts \ m \implies \text{wf-component-descs-list } ts \implies \text{wf-component-descs } s$
 $(s, n) \in \text{td-set-tuple } x \ m \implies \text{wf-component-descs-tuple } x \implies \text{wf-component-descs } s$

proof (*induct t and st and ts and x arbitrary: m and m and m and m*)

case (*TypDesc* *algn st nm*)

then show *?case by auto*

next

case (*TypScalar* *sz algn d*)

then show *?case by auto*

next

case (*TypAggregate* *ts*)

then show *?case by auto*

next

case *Nil-typ-desc*

then show *?case by auto*

next

case (*Cons-typ-desc* *x fs*)

then show *?case by auto*

next

case (*DTuple-typ-desc* *d nm y*)

then show *?case by auto*

qed

lemma *td-set-field-descs*:

fixes *t::'a xtyp-info*
and *st::'a xtyp-info-struct*
and *ts::'a xtyp-info-tuple list*
and *x::'a xtyp-info-tuple*

shows

$(s, n) \in \text{td-set } t \ m \implies \text{wf-component-descs } t \implies \text{component-desc } s \in \text{insert } (\text{component-desc } t) \ (\text{set } (\text{field-descs } t))$
 $(s, n) \in \text{td-set-struct } st \ m \implies \text{wf-component-descs-struct } st \implies \text{component-desc } s \in (\text{set } (\text{field-descs-struct } st))$
 $(s, n) \in \text{td-set-list } ts \ m \implies \text{wf-component-descs-list } ts \implies \text{component-desc } s \in (\text{set } (\text{field-descs-list } ts))$
 $(s, n) \in \text{td-set-tuple } x \ m \implies \text{wf-component-descs-tuple } x \implies \text{component-desc } s$

```

∈ (set (field-descs-tuple x))
proof (induct t and st and ts and x arbitrary: m and m and m and m)
case (TypDesc algn st nm)
then show ?case by auto
next
  case (TypScalar sz algn d)
  then show ?case by auto
next
  case (TypAggregate ts)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
  then show ?case by auto
next
  case (DTuple-typ-desc d nm y)
  then show ?case by auto
qed

```

lemma *td-set-wf-field-descs:*

```

fixes t::'a xtyp-info
and st::'a xtyp-info-struct
and ts::'a xtyp-info-tuple list
and x::'a xtyp-info-tuple

```

shows

```

(s, n) ∈ td-set t m ⇒ wf-field-descs (set (field-descs t)) ⇒ wf-field-descs (set
(field-descs s))

```

```

(s, n) ∈ td-set-struct st m ⇒ wf-field-descs (set (field-descs-struct st)) ⇒
wf-field-descs (set (field-descs s))

```

```

(s, n) ∈ td-set-list ts m ⇒ wf-field-descs (set (field-descs-list ts)) ⇒ wf-field-descs
(set (field-descs s))

```

```

(s, n) ∈ td-set-tuple x m ⇒ wf-field-descs (set (field-descs-tuple x)) ⇒ wf-field-descs
(set (field-descs s))

```

proof (induct t and st and ts and x arbitrary: m and m and m and m)

```

case (TypDesc algn st nm)

```

```

then show ?case by auto

```

```

next

```

```

  case (TypScalar sz algn d)

```

```

  then show ?case by auto

```

```

next

```

```

  case (TypAggregate ts)

```

```

  then show ?case by auto

```

```

next

```

```

  case Nil-typ-desc

```

```

  then show ?case by auto

```

```

next

```

```

  case (Cons-typ-desc x fs)

```

```

then show ?case by auto
next
  case (DTuple-typ-desc d nm y)
  then show ?case by auto
qed

```

```

context xmem-type

```

```

begin

```

```

lemma xmem-type-td-set-field-descs:

```

```

  (s,n) ∈ td-set (typ-info-t TYPE('a)) m ⇒
  component-desc s ∈ insert (component-desc (typ-info-t TYPE('a))) (set (field-descs
  (typ-info-t TYPE('a))))
  using td-set-field-descs local.wf-component-descs by blast

```

```

lemma field-lookup-component-desc:

```

```

  field-lookup (typ-info-t TYPE('a)) f m = Some (s, n) ⇒
  component-desc s ∈ insert (component-desc (typ-info-t TYPE('a))) (set (field-descs
  (typ-info-t TYPE('a))))
  using xmem-type-td-set-field-descs td-set-field-lookupD
  by blast

```

```

lemma field-lookup-wf-field-desc:

```

```

  field-lookup (typ-info-t TYPE('a)) f m = Some (s, n) ⇒
  wf-field-desc (component-desc s)
  using field-lookup-component-desc
  by (meson local.wf-field-descs' wf-field-descs-def)

```

```

lemma field-lookup-component-descs-independent:

```

```

  assumes fl: field-lookup (typ-info-t TYPE('a)) f m = Some (s, n)
  shows component-descs-independent s
  using field-lookup-component-desc [OF fl]
  by (meson fl local.component-descs-independent td-set-component-descs-independent(1)
  td-set-field-lookupD)

```

```

lemma field-lookup-wf-component-descs:

```

```

  assumes fl: field-lookup (typ-info-t TYPE('a)) f m = Some (s, n)
  shows wf-component-descs s
  using field-lookup-component-desc [OF fl] td-set-wf-component-descs fl
  local.wf-component-descs td-set-field-lookupD by blast

```

```

lemma field-lookup-wf-field-descs:

```

```

  assumes fl: field-lookup (typ-info-t TYPE('a)) f m = Some (s, n)
  shows wf-field-descs (set (field-descs s))
  using td-set-wf-field-descs fl local.wf-field-descs td-set-field-lookupD by blast

```

```

lemma field-lookup-field-access-access-ti:

```

```

  assumes fl: field-lookup (typ-info-t TYPE('a)) f m = Some (s, n)
  shows field-access (component-desc s) = access-ti s

```

using *access-ti-component-desc-compatible(1)* [*OF field-lookup-wf-component-descs*
[*OF fl*]
field-lookup-component-descs-independent [*OF fl*] *field-lookup-wf-field-descs*
[*OF fl*]
by (*simp add: fun-eq-iff*)

lemma *field-lookup-field-update-update-ti*:
assumes *fl: field-lookup (typ-info-t TYPE('a)) f m = Some (s, n)*
shows *field-update (component-desc s) = update-ti s*
using *update-ti-component-desc-compatible(1)* [*OF field-lookup-wf-component-descs*
[*OF fl*]
field-lookup-component-descs-independent [*OF fl*] *field-lookup-wf-field-descs*
[*OF fl*]
by (*simp add: fun-eq-iff*)

lemma *field-lookup-field-sz-size-td*:
assumes *fl: field-lookup (typ-info-t TYPE('a)) f m = Some (s, n)*
shows *field-sz (component-desc s) = size-td s*
using *size-td-field-sz(1)* [*OF field-lookup-wf-component-descs* [*OF fl*]]
by *simp*

lemma *field-lookup-component-desc-complement-padding*:
field-lookup (typ-info-t TYPE('a)) f m = Some (s, n) \implies
complement-padding (field-access (component-desc s)) (field-update (component-desc
s)) (field-sz (component-desc s))
using *field-lookup-component-desc*
by (*meson field-lookup-wf-field-desc padding-lense.axioms(1) wf-field-desc.axioms*)

lemma *field-lookup-component-desc-complement-padding'*:
assumes *fl: field-lookup (typ-info-t TYPE('a)) f m = Some (s, n)*
shows *complement-padding (access-ti s) (update-ti s) (size-td s)*
using *field-lookup-component-desc-complement-padding*
field-lookup-field-access-access-ti [*OF fl*]
field-lookup-field-update-update-ti [*OF fl*]
field-lookup-field-sz-size-td [*OF fl*]
by (*metis fl*)

lemma *field-lookup-padding-lense*:
field-lookup (typ-info-t TYPE('a)) f m = Some (s, n) \implies
padding-lense (access-ti s) (update-ti s) (size-td s)
by (*metis field-lookup-field-access-access-ti field-lookup-field-update-update-ti*
field-lookup-wf-component-descs field-lookup-wf-field-desc size-td-field-sz(1)
wf-field-desc-def)

sublocale *lense: padding-lense*
access-ti (typ-info-t TYPE('a))
update-ti (typ-info-t TYPE('a))
size-of TYPE('a)

using *local.field-access-access-ti local.field-sz-size-of*
local.field-update-update-ti local.xmem-type-wf-field-desc.padding-lense-axioms
 by *simp*

end

lemma *eq-padding-access-update-field-cancel:*

assumes *fl: field-lookup (typ-info-t (TYPE('a::xmem-type))) f 0 = Some (s, n)*
assumes *lower-bound-x: n ≤ x*
assumes *upper-bound-x: x < n + size-of TYPE('b)*
assumes *match: export-uinfo s = typ-uinfo-t TYPE('b::xmem-type)*
assumes *lbs: length bs = size-of TYPE('b)*
assumes *lbs': length bs' = size-of TYPE('a)*
assumes *eq-padding: padding-base.eq-padding (access-ti s) (size-td s) bs (take (size-of TYPE('b)) (drop n bs'))*
shows *access-ti (typ-info-t TYPE('a::xmem-type)) (update-ti s bs v) bs' ! x = bs ! (x - n)*

proof –

obtain *i where i: i = x - n by blast*
from *match fl have sz-b: size-of TYPE('b) = size-td s*
 using *export-size-of by blast*

interpret *cps: complement-padding access-ti s update-ti s size-td s*
 by (*rule field-lookup-component-desc-complement-padding' [OF fl]*)

from *lower-bound-x upper-bound-x sz-b*
have *x-n-bound: x - n < size-td s*
 by *simp*
from *eq-padding*
have *l-take-drop-bs': length (take (size-td s) (drop n bs')) = size-td s*
 by (*metis lbs padding-base.eq-padding-length-eq sz-b*)

from *lower-bound-x upper-bound-x fl lbs'*
have *acc-bs': take (size-td s) (drop n bs') ! (x - n) = bs' ! x*
 by (*metis add.commute field-lookup-offset-size nth-take-drop-index-shift size-of-def sz-b*)

from *mem-type-field-lookup-nth-focus [OF fl lower-bound-x [simplified size-of-def] upper-bound-x [simplified sz-b] lbs']*

have *access-ti (typ-info-t TYPE('a)) (update-ti s bs v) bs' ! x =*
access-ti s (update-ti s bs v) (take (size-td s) (drop n bs')) ! (x - n) .

also have *... = bs ! (x - n) (is ?lhs = ?rhs)*

proof (*cases padding-base.is-padding-byte (access-ti s) (update-ti s) (size-td s) (x - n)*)

case *True*

from *padding-base.is-padding-byte-acc-eq [OF True l-take-drop-bs'] acc-bs'*
have *?lhs = bs' ! x by simp*

```

also have  $bs' ! x = bs ! (x - n)$ 
  using cps.eq-padding-padding-byte-id [OF eq-padding True] acc-bs' sz-b
  by simp
  finally show ?thesis .
next
  case False
  then have is-value: cps.is-value-byte (x - n) using cps.complement x-n-bound
by simp
  from cps.is-value-byte-acc-upd-cancel [OF is-value lbs [simplified sz-b] l-take-drop-bs']
  show ?thesis .
  qed
  finally show ?thesis .
qed

```

```

context xmem-type
begin

```

```

sublocale xmem-type-padding: complement-padding
  access-ti (typ-info-t TYPE('a))
  update-ti (typ-info-t TYPE('a))
  size-of TYPE('a)
  using xmem-type-wf-field-desc.complement-padding-axioms
  by (simp add: field-access-access-ti field-update-update-ti field-sz-size-td size-of-def)
end

```

```

lemma drop-heap-list-le2:
  heap-list h n (x + of-nat k)
    = drop k (heap-list h (n + k) x)
  by (simp add: drop-heap-list-le)

```

```

lemma heap-list-take-drop:
  assumes N-bound: unat a + N ≤ 2 ^ len-of TYPE(addr-bitsize)
  shows  $n + m ≤ N \implies \text{take } m (\text{drop } n (\text{heap-list hp } N a)) =$ 
     $\text{heap-list hp } m (a + \text{word-of-nat } n)$ 
  apply (induct m arbitrary: n)
  apply simp
  apply simp
  using N-bound
  by auto
  (metis (no-types, opaque-lifting) add.left-commute add.right-neutral add-Suc-right
drop-heap-list-le2 heap-list-rec take-drop take-heap-list-le)

```

```

lemma heap-list-take-drop':
  assumes N-bound: unat a + N ≤ addr-card
  assumes bound: n + m ≤ N
  shows  $\text{take } m (\text{drop } n (\text{heap-list hp } N a)) =$ 
     $\text{heap-list hp } m (a + \text{word-of-nat } n)$ 
proof –
  have  $\text{addr-card} = 2 \wedge \text{len-of TYPE(addr-bitsize)}$ 

```


by (*simp add: addr-card*)
from *heap-list-take-drop* [*OF N-bound [simplified this] bound*] **show** *?thesis* **by**
blast
qed

experiment

fixes *proj*:: 'a::xmem-type \Rightarrow 'b::xmem-type
fixes *t*::'a xtyp-info

assumes *fl*: *field-lookup* (*typ-info-t* TYPE('a)) *f* 0 = *Some* (*t*, *n*)
and *eu*: *export-uinfo* *t* = *typ-uinfo-t* TYPE('b)
assumes *access-comp*: *access-ti* *t* *v* = *access-ti* (*typ-info-t* TYPE('b)) (*proj* *v*)
assumes *surj*: *surj* *proj*

begin

lemma *sz*: *size-td* *t* = *size-of* TYPE('b)
using *eu*
by (*metis export-uinfo-size size-of-def typ-uinfo-size*)

lemma *padding-base.eq-padding* (*access-ti* *t*) (*size-td* *t*) =
padding-base.eq-padding (*access-ti* (*typ-info-t* TYPE('b))) (*size-of* TYPE('b))
unfolding *padding-base.eq-padding-def*
apply (*rule ext*)
apply (*rule ext*)
apply (*simp add: sz*)
apply (*simp add: access-comp*)
using *surj*
by (*metis surj-def*)
end

definition *is-padding-byte*::*typ-uinfo* \Rightarrow *nat* \Rightarrow *bool* **where**
is-padding-byte *t* *i* \equiv *i* < *size-td* *t* \wedge
 $(\forall$ *bs* *b*. *length* *bs* = *size-td* *t* \longrightarrow *norm-tu* *t* (*bs*[*i* := *b*]) = *norm-tu* *t* *bs*)

definition *is-value-byte*::*typ-uinfo* \Rightarrow *nat* \Rightarrow *bool* **where**
is-value-byte *t* *i* \equiv *i* < *size-td* *t* \wedge
 $(\exists$ *bs* *b*. *length* *bs* = *size-td* *t* \wedge *norm-tu* *t* (*bs*[*i* := *b*]) \neq *norm-tu* *t* *bs*)

lemma *is-padding-byteI*:
assumes *i* < *size-td* *t*
assumes \bigwedge *i* *bs* *b*. *length* *bs* = *size-td* *t* \Longrightarrow *norm-tu* *t* (*bs*[*i* := *b*]) = *norm-tu* *t* *bs*
shows *is-padding-byte* *t* *i*
using *assms* **by** (*simp add: is-padding-byte-def*)

lemma *complement-tu-padding*:
 i < *size-td* *t* \Longrightarrow *is-padding-byte* *t* *i* \neq *is-value-byte* *t* *i*
by (*auto simp add: is-padding-byte-def is-value-byte-def*)

definition *eq-padding*::*typ-uinfo* \Rightarrow *byte list* \Rightarrow *byte list* \Rightarrow *bool* **where**
eq-padding *t* *bs* *bs'* \equiv *length* *bs* = *size-td* *t* \wedge *length* *bs'* = *size-td* *t* \wedge
 $(\forall i. \text{is-padding-byte } t \ i \longrightarrow \text{bs} \ ! \ i = \text{bs}' \ ! \ i)$

definition *eq-upto-padding*::*typ-uinfo* \Rightarrow *byte list* \Rightarrow *byte list* \Rightarrow *bool* **where**
eq-upto-padding *t* *bs* *bs'* \equiv *length* *bs* = *size-td* *t* \wedge *length* *bs'* = *size-td* *t* \wedge
 $(\forall i. \text{is-value-byte } t \ i \longrightarrow \text{bs} \ ! \ i = \text{bs}' \ ! \ i)$

lemma *eq-padding-refl*[*simp*]: *length* *bs* = *size-td* *t* \Longrightarrow *eq-padding* *t* *bs* *bs*
by (*auto simp add: eq-padding-def*)

lemma *eq-upto-padding-refl*[*simp*]: *length* *bs* = *size-td* *t* \Longrightarrow *eq-upto-padding* *t* *bs* *bs*
by (*auto simp add: eq-upto-padding-def*)

lemma *eq-padding-sym*: *eq-padding* *t* *bs* *bs'* \longleftrightarrow *eq-padding* *t* *bs'* *bs*
by (*auto simp add: eq-padding-def*)

lemma *eq-padding-symp*: *symp* (*eq-padding* *t*)
by (*simp add: symp-def eq-padding-sym*)

lemma *eq-upto-padding-sym*: *eq-upto-padding* *t* *bs* *bs'* \longleftrightarrow *eq-upto-padding* *t* *bs'* *bs*
by (*auto simp add: eq-upto-padding-def*)

lemma *eq-upto-padding-symp*: *symp* (*eq-upto-padding* *t*)
by (*simp add: symp-def eq-upto-padding-sym*)

lemma *eq-padding-trans*: *eq-padding* *t* *bs1* *bs2* \Longrightarrow *eq-padding* *t* *bs2* *bs3* \Longrightarrow *eq-padding* *t* *bs1* *bs3*
by (*auto simp add: eq-padding-def*)

lemma *eq-padding-transp*: *transp* (*eq-padding* *t*)
unfolding *transp-def*
by (*auto intro: eq-padding-trans*)

lemma *eq-upto-padding-trans*: *eq-upto-padding* *t* *bs1* *bs2* \Longrightarrow *eq-upto-padding* *t* *bs2* *bs3* \Longrightarrow *eq-upto-padding* *t* *bs1* *bs3*
by (*auto simp add: eq-upto-padding-def*)

lemma *eq-upto-padding-transp*: *transp* (*eq-upto-padding* *t*)
unfolding *transp-def*
by (*auto intro: eq-upto-padding-trans*)

lemma *eq-padding-eq-upto-padding-eq*: *eq-padding* *t* *bs* *bs'* \Longrightarrow *eq-upto-padding* *t* *bs* *bs'* \Longrightarrow *bs* = *bs'*

proof –

assume *a1*: *eq-padding* *t* *bs* *bs'*

assume *a2*: *eq-upto-padding* *t* *bs* *bs'*

```

have f3: length bs' = size-td t
  using a1 by (simp add: eq-padding-def)
have length bs = size-td t
  using a1 eq-padding-def by blast
then show ?thesis
  using f3 a2 a1 by (metis (full-types) complement-tu-padding eq-padding-def
eq-upto-padding-def nth-equalityI)
qed

```

thm padding-base.is-padding-byte-def

lemma is-padding-byte-access-ti':

```

fixes t::'a xtyp-info
and st::'a xtyp-info-struct
and ts::'a xtyp-info-tuple list
and x::'a xtyp-info-tuple

```

shows

```

[[ wf-desc t; wf-size-desc t; wf-field-descs (set (field-descs t)); wf-component-descs
t; wf-fd t;
  i < size-td t; length bs = size-td t;
  ∀ bs b. length bs = size-td t ⟶ norm-tu (map-td field-norm (λ-. ()) t) (bs[i :=
b]) = norm-tu (export-uinfo t) bs]]
⟹ access-ti t v bs ! i = bs ! i

```

```

[[ wf-desc-struct st; wf-size-desc-struct st; wf-field-descs (set (field-descs-struct st));
wf-component-descs-struct st; wf-fd-struct st; i < size-td-struct st; length bs =
size-td-struct st;
  ∀ bs b. length bs = size-td-struct st ⟶ norm-tu-struct (map-td-struct field-norm
(λ-. ()) st) (bs[i := b]) = norm-tu-struct ((map-td-struct field-norm (λ-. ()) st)
bs)]
⟹ access-ti-struct st v bs ! i = bs ! i

```

```

[[ wf-desc-list ts; wf-size-desc-list ts; wf-field-descs (set (field-descs-list ts));
wf-component-descs-list ts; wf-fd-list ts; i < size-td-list ts; length bs = size-td-list
ts;
  ∀ bs b. length bs = size-td-list ts ⟶ norm-tu-list (map-td-list field-norm (λ-. ())
ts) (bs[i := b]) = norm-tu-list ((map-td-list field-norm (λ-. ()) ts) bs)]
⟹ access-ti-list ts v bs ! i = bs ! i

```

```

[[ wf-desc-tuple x; wf-size-desc-tuple x; wf-field-descs (set (field-descs-tuple x));
wf-component-descs-tuple x; wf-fd-tuple x; i < size-td-tuple x; length bs = size-td-tuple
x;
  ∀ bs b. length bs = size-td-tuple x ⟶ norm-tu-tuple (map-td-tuple field-norm
(λ-. ()) x) (bs[i := b]) = norm-tu-tuple ((map-td-tuple field-norm (λ-. ()) x) bs)]
⟹ access-ti-tuple x v bs ! i = bs ! i

```

proof (induct t and st and ts and x arbitrary: bs i and bs i and bs i and bs i)

```

case (TypDesc algn st nm)
then show ?case by auto
next
  case (TypScalar sz algn d)
  from TypScalar obtain
    wf-d: wf-field-desc d and
    field-sz: field-sz d = sz and
    i-bound: i < sz and
    lbs: length bs = sz and
    padding:  $\bigwedge bs\ b. \text{length } bs = sz \implies$ 
      field-norm sz algn d (bs[i := b]) = field-norm sz algn d bs
    by simp

interpret wf: wf-field-desc d by (rule wf-d)
show ?case
proof (cases wf.is-padding-byte i)
  case True
    then show ?thesis
    using field-sz lbs wf.is-padding-byte-acc-id by auto
  next
    case False
    note not-padding = this
    with wf.complement field-sz i-bound have is-value: wf.is-value-byte i by blast
    from is-value padding have False
    by (auto simp add: field-norm-def)
      (metis (mono-tags, opaque-lifting) field-sz i-bound list-update-id not-padding
padding-base.is-value-byte-upd-neg
wf.access-result-size wf.is-padding-byte-def wf.update-access wf.update-access-unequal)
    then show ?thesis by blast
  qed
next
case (TypAggregate ts)
then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
  obtain d nm y where x: x = DTuple d nm y by (cases x) auto
  from Cons-typ-desc.prems obtain
    wf-desc-d: wf-desc d and
    wf-fd-x: wf-fd-tuple (DTuple d nm y) and
    wf-fd-d: wf-fd d and
    wf-fd-fs: wf-fd-list fs and
    nm-notin: nm  $\notin$  dt-snd ' set fs and
    wf-desc-fs: wf-desc-list fs and
    wf-sz-d: wf-size-desc d and
    wf-sz-fs: wf-size-desc-list fs and
    wf-field-d: wf-field-desc (component-desc d) and

```

```

wf-fields-d: wf-field-descs (set (field-descs d)) and
wf-fields-fs: wf-field-descs (set (field-descs-list fs)) and
y: y = component-desc d and
wf-comp-d: wf-component-descs d and
wf-comp-fs: wf-component-descs-list fs and
i-bound: i < size-td d + size-td-list fs and
lbs: length bs = size-td d + size-td-list fs and
wf-desc-x: wf-desc-tuple x and
wf-sz-x: wf-size-desc-tuple x and
wf-decs-x: wf-field-descs (set (field-descs-tuple x)) and
wf-comp-x: wf-component-descs-tuple x and

padding:  $\bigwedge bs\ b . \text{length } bs = \text{size-td } d + \text{size-td-list } fs \implies$ 
  norm-tu (map-td field-norm ( $\lambda\cdot$ . ()) d)
    (take (size-td d) (bs[i := b])) @
  norm-tu-list (map-td-list field-norm ( $\lambda\cdot$ . ()) fs)
    (drop (size-td d) (bs[i := b])) =
  norm-tu (map-td field-norm ( $\lambda\cdot$ . ()) d) (take (size-td d) bs) @
  norm-tu-list (map-td-list field-norm ( $\lambda\cdot$ . ()) fs)
    (drop (size-td d) bs)
by (auto simp add: x)

have padding-d:  $\forall bs . \text{length } bs = \text{size-td } d \longrightarrow$ 
  ( $\forall b . \text{norm-tu (map-td field-norm } (\lambda\cdot . ()) d) (bs[i := b]) =$ 
    norm-tu (map-td field-norm ( $\lambda\cdot$ . ()) d) bs)
proof (safe)
  fix bs::byte list and b::byte
  assume lbs: length bs = size-td d
  from lbs obtain xbs sfx where xbs: xbs = bs @ sfx and lbs: length xbs =
size-td d + size-td-list fs
  by (metis Ex-list-of-length length-append)

  from lbs
  have eq1: (take (size-td d) ((bs @ sfx)[i := b])) = bs[i := b] by (simp add:
list-update-append)

  from lbs
  have lbs': length (bs[i:=b]) = size-td d by simp

  from wf-fd-norm-tu(1)[rule-format, OF wf-fd-d lbs ] lbs
  have lnorm1: length (norm-tu (map-td field-norm ( $\lambda\cdot$ . ()) d) bs) = size-td d
  by (simp add: export-uinfo-def length-fa-ti wf-fd-d)

  from wf-fd-norm-tu(1)[rule-format, OF wf-fd-d lbs' ] lbs'
  have lnorm2: length (norm-tu (map-td field-norm ( $\lambda\cdot$ . ()) d) (bs[i:=b])) =
size-td d
  by (simp add: export-uinfo-def length-fa-ti wf-fd-d)

  from padding [OF lbs, of b] xbs lbs

```

```

show norm-tu (map-td field-norm (λ-. ()) d) (bs[i := b]) =
  norm-tu (map-td field-norm (λ-. ()) d) bs
apply simp
apply (subst (asm) list-append-eq-split-conv)
apply (simp add: eq1 lnorm1 lnorm2)
apply (simp add: eq1)
done
qed

from lbs
have lbs-drop: length (drop (size-td d) bs) = size-td-list fs
  by (simp add: x)

from lbs
have lbs-take: length (take (size-td d) bs) = size-td d
  by (simp add: x)

have lacc: length (access-ti d v (take (size-td d) bs)) = size-td d
  by (metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps wf-fd-x x)

show ?case

proof (cases i < size-td d)
  case True
  note hyp = Cons-typ-desc.hyps(1) [OF wf-desc-x wf-sz-x wf-decs-x wf-comp-x,
  simplified x, OF wf-fd-x, simplified, OF True lbs-take padding-d]
  from hyp lbs
  show ?thesis
  by (simp add: x True nth-append lacc )
  next
  case False
  from False i-bound have i-bound': i - size-td d < size-td-list fs by simp
  have padding-fs: ∀ bs b.
    length bs = size-td-list fs →
    norm-tu-list (map-td-list field-norm (λ-. ()) fs) (bs[i - size-td d := b]) =
    norm-tu-list (map-td-list field-norm (λ-. ()) fs) bs
  proof (safe)
  fix bs::byte list and b::byte
  assume lbs: length bs = size-td-list fs
  from lbs obtain xbs pfx where xbs: xbs = pfx @ bs and lxbs: length xbs =
  size-td d + size-td-list fs
  and lpfx: length pfx = size-td d
  by (metis Ex-list-of-length length-append)

  from lpfx
  have drop-eq1: drop (size-td d) xbs = bs
  by (simp add: xbs)
  from lpfx False have drop-eq2: (drop (size-td d) (xbs[i := b])) = bs [i -

```

```

size-td d := b]
  by (auto simp add: xbs list-update-append)
from lpfx False
have take-eq1: take (size-td d) (xbs[i := b]) = pfx
  by (simp add: xbs list-update-append)
from lpfx False
have take-eq2: take (size-td d) xbs = pfx
  by (simp add: xbs)

  have leq1: length (norm-tu (map-td field-norm (λ-. ())) d) (take (size-td d)
(xbs[i := b]))) = size-td d
  apply (simp add: take-eq1)
  using wf-fd-norm-tu(1)[rule-format, OF wf-fd-d lpfx ] lpfx
  by (simp add: export-uinfo-def length-fa-ti wf-fd-d)
  have leq2: length (norm-tu (map-td field-norm (λ-. ())) d) (take (size-td d)
xbs)) = size-td d
  apply (simp add: take-eq2)
  using wf-fd-norm-tu(1)[rule-format, OF wf-fd-d lpfx ] lpfx
  by (simp add: export-uinfo-def length-fa-ti wf-fd-d)

from padding [OF lxs, of b, simplified drop-eq1 drop-eq2]
show norm-tu-list (map-td-list field-norm (λ-. ())) fs) (bs[i - size-td d := b])
=
norm-tu-list (map-td-list field-norm (λ-. ())) fs) bs
  apply -
  apply (subst (asm) list-append-eq-split-conv)
  apply (simp add: leq1 leq2)
  apply simp
  done
qed
from False lbs have drop-eq: drop (size-td d) bs ! (i - size-td d) = bs ! i by
simp
note hyp = Cons-typ-desc.hyps(2) [OF wf-desc-fs wf-sz-fs wf-fields-fs wf-comp-fs
wf-fd-fs i-bound' lbs-drop padding-fs]
from hyp
show ?thesis by (simp add: x False nth-append lacc drop-eq)
qed
next
case (DTuple-typ-desc d nm y)
then show ?case by (auto simp add: export-uinfo-def)
qed

lemma is-padding-byte-access-ti:
  assumes wf: wf-desc (t::'a xtyp-info)
  and wf-sz: wf-size-desc t
  and wf-descs: wf-field-descs (set (field-descs t))
  and wf-comp: wf-component-descs t
  and wf-fd: wf-fd t
  and i-bound: i < size-td t

```

```

    and lbs: length bs = size-td t
    and is-padding: is-padding-byte (export-uinfo t) i
  shows access-ti t v bs ! i = bs ! i
    using is-padding-byte-access-ti'(1) [OF wf wf-sz wf-descs wf-comp wf-fd i-bound
lbs] is-padding
    by (simp add: is-padding-byte-def export-uinfo-def)

```

context *xmem-type*

begin

lemma *xmem-type-is-padding-byte-access-ti*:

assumes padding: is-padding-byte (typ-uinfo-t TYPE('a)) i

and i-bound: i < size-of TYPE('a)

and lbs: length bs = size-of TYPE('a)

shows access-ti (typ-uinfo-t TYPE('a)) v bs ! i = bs ! i

proof –

have wf: wf-desc (typ-uinfo-t TYPE('a)) by simp

have wf-sz: wf-size-desc (typ-uinfo-t TYPE('a)) by simp

have wf-descs: wf-field-descs (set (field-descs (typ-uinfo-t TYPE('a))))

using local.wf-field-descs by blast

have wf-comp: wf-component-descs (typ-uinfo-t TYPE('a))

by (simp add: local.wf-component-descs)

have wf-fd: wf-fd (typ-uinfo-t TYPE('a))

using local.wf-desc local.wf-lf wf-fdp-fd(1) wf-lf-fdp by blast

from is-padding-byte-access-ti [OF wf wf-sz wf-descs wf-comp wf-fd

i-bound [simplified size-of-def] lbs [simplified size-of-def]

padding [simplified typ-uinfo-t-def]]

show ?thesis .

qed

end

lemma *is-padding-byte-update-ti-id'*:

fixes t::'a xtyp-info

and st::'a xtyp-info-struct

and ts::'a xtyp-info-tuple list

and x::'a xtyp-info-tuple

shows

[[wf-desc t; wf-size-desc t; wf-field-descs (set (field-descs t)); wf-component-descs

t; wf-fd t;

i < size-td t; length bs = size-td t;

∀ bs b. length bs = size-td t → norm-tu (map-td field-norm (λ-. ()) t) (bs[i :=

b]) = norm-tu (export-uinfo t) bs]]

⇒ update-ti t (bs[i := b]) v = update-ti t bs v

[[wf-desc-struct st; wf-size-desc-struct st; wf-field-descs (set (field-descs-struct st));

wf-component-descs-struct st; wf-fd-struct st; i < size-td-struct st; length bs =


```

size-td-struct st;
  ∀ bs b. length bs = size-td-struct st → norm-tu-struct (map-td-struct field-norm
(λ-. ()) st) (bs[i := b]) = norm-tu-struct ((map-td-struct field-norm (λ-. ()) st))
bs]]
  ⇒ update-ti-struct st (bs[i := b]) v = update-ti-struct st bs v

[[ wf-desc-list ts; wf-size-desc-list ts; wf-field-descs (set (field-descs-list ts));
  wf-component-descs-list ts; wf-fd-list ts; i < size-td-list ts; length bs = size-td-list
ts;
  ∀ bs b. length bs = size-td-list ts → norm-tu-list (map-td-list field-norm (λ-. ()))
ts) (bs[i := b]) = norm-tu-list ((map-td-list field-norm (λ-. ())) ts) bs]]
  ⇒ update-ti-list ts (bs[i := b]) v = update-ti-list ts bs v

[[ wf-desc-tuple x; wf-size-desc-tuple x; wf-field-descs (set (field-descs-tuple x));
  wf-component-descs-tuple x; wf-fd-tuple x; i < size-td-tuple x; length bs = size-td-tuple
x;
  ∀ bs b. length bs = size-td-tuple x → norm-tu-tuple (map-td-tuple field-norm
(λ-. ()) x) (bs[i := b]) = norm-tu-tuple ((map-td-tuple field-norm (λ-. ()) x) bs]]
  ⇒ update-ti-tuple x (bs[i := b]) v = update-ti-tuple x bs v
proof (induct t and st and ts and x arbitrary: bs i v and bs i v and bs i v and
bs i v)
case (TypDesc algn st nm)
then show ?case by auto
next
  case (TypScalar sz algn d)
  from TypScalar obtain
    wf-d: wf-field-desc d and
    field-sz: field-sz d = sz and
    i-bound: i < sz and
    lbs: length bs = sz and
    padding: ∧ bs b. length bs = sz ⇒
      field-norm sz algn d (bs[i := b]) = field-norm sz algn d bs
  by simp

interpret wf: wf-field-desc d by (rule wf-d)
show ?case
proof (cases wf.is-padding-byte i)
  case True
  then show ?thesis
    using field-sz lbs wf.is-padding-byte-upd-eq by auto
next
  case False
  note not-padding = this
  with wf.complement field-sz i-bound have is-value: wf.is-value-byte i by blast
  from is-value padding have False
  by (auto simp add: field-norm-def)
  (metis (mono-tags, opaque-lifting) field-sz i-bound list-update-id not-padding
padding-base.is-value-byte-upd-neq
wf.access-result-size wf.is-padding-byte-def wf.update-access wf.update-access-unequal)

```

```

    then show ?thesis by blast
qed
next
case (TypAggregate ts)
then show ?case by auto
next
case Nil-typ-desc
then show ?case by auto
next
case (Cons-typ-desc x fs)
obtain d nm y where x: x = DTuple d nm y by (cases x) auto
from Cons-typ-desc.prem1 obtain
  wf-desc-d: wf-desc d and
  wf-fd-x: wf-fd-tuple (DTuple d nm y) and
  wf-fd-d: wf-fd d and
  wf-fd-fs: wf-fd-list fs and
  nm-notin: nm ∉ dt-snd ' set fs and
  wf-desc-fs: wf-desc-list fs and
  wf-sz-d: wf-size-desc d and
  wf-sz-fs: wf-size-desc-list fs and
  wf-field-d: wf-field-desc (component-desc d) and
  wf-fields-d: wf-field-descs (set (field-descs d)) and
  wf-fields-fs: wf-field-descs (set (field-descs-list fs)) and
  y: y = component-desc d and
  wf-comp-d: wf-component-descs d and
  wf-comp-fs: wf-component-descs-list fs and
  i-bound: i < size-td d + size-td-list fs and
  lbs: length bs = size-td d + size-td-list fs and
  wf-desc-x: wf-desc-tuple x and
  wf-sz-x: wf-size-desc-tuple x and
  wf-decs-x: wf-field-descs (set (field-descs-tuple x)) and
  wf-comp-x: wf-component-descs-tuple x and

padding:  $\bigwedge bs b . \text{length } bs = \text{size-td } d + \text{size-td-list } fs \implies$ 
  norm-tu (map-td field-norm (λ-. ()) d)
    (take (size-td d) (bs[i := b])) @
  norm-tu-list (map-td-list field-norm (λ-. ()) fs)
    (drop (size-td d) (bs[i := b])) =
  norm-tu (map-td field-norm (λ-. ()) d) (take (size-td d) bs) @
  norm-tu-list (map-td-list field-norm (λ-. ()) fs)
    (drop (size-td d) bs)
by (auto simp add: x)

have padding-d:  $\forall bs . \text{length } bs = \text{size-td } d \implies$ 
  ( $\forall b . \text{norm-tu (map-td field-norm (λ-. ()) d) (bs[i := b]) =$ 
    norm-tu (map-td field-norm (λ-. ()) d) bs)
proof (safe)
  fix bs::byte list and b::byte
  assume lbs: length bs = size-td d

```

```

from lbs obtain xbs sfx where xbs:  $xbs = bs @ sfx$  and lbs:  $length\ xbs =$ 
 $size-td\ d + size-td-list\ fs$ 
  by (metis Ex-list-of-length length-append)

from lbs
  have eq1:  $(take\ (size-td\ d)\ ((bs @ sfx)[i := b])) = bs[i := b]$  by (simp add:
list-update-append)

from lbs
  have lbs':  $length\ (bs[i:=b]) = size-td\ d$  by simp

from wf-fd-norm-tu(1)[rule-format, OF wf-fd-d lbs ] lbs
  have lnorm1:  $length\ (norm-tu\ (map-td\ field-norm\ (\lambda-. ())\ d)\ bs) = size-td\ d$ 
  by (simp add: export-winfo-def length-fa-ti wf-fd-d)

from wf-fd-norm-tu(1)[rule-format, OF wf-fd-d lbs' ] lbs'
  have lnorm2:  $length\ (norm-tu\ (map-td\ field-norm\ (\lambda-. ())\ d)\ (bs[i:=b])) =$ 
 $size-td\ d$ 
  by (simp add: export-winfo-def length-fa-ti wf-fd-d)

from padding [OF lbs, of b] xbs lbs
  show  $norm-tu\ (map-td\ field-norm\ (\lambda-. ())\ d)\ (bs[i := b]) =$ 
 $norm-tu\ (map-td\ field-norm\ (\lambda-. ())\ d)\ bs$ 
  apply simp
  apply (subst (asm) list-append-eq-split-conv)
  apply (simp add: eq1 lnorm1 lnorm2)
  apply (simp add: eq1)
  done
qed

from lbs
  have lbs-drop:  $length\ (drop\ (size-td\ d)\ bs) = size-td-list\ fs$ 
  by (simp add: x)

from lbs
  have lbs-take:  $length\ (take\ (size-td\ d)\ bs) = size-td\ d$ 
  by (simp add: x)

have lacc:  $length\ (access-ti\ d\ v\ (take\ (size-td\ d)\ bs)) = size-td\ d$ 
  by (metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps wf-fd-x x)

show ?case

proof (cases i < size-td d)
  case True
  from True lbs
  have take-eq:  $(take\ (size-td\ d)\ bs)[i := b] = take\ (size-td\ d)\ (bs[i := b])$ 
  by (simp add: take-update-swap)

```

```

note hyp = Cons-typ-desc.hyps(1) [OF wf-desc-x wf-sz-x wf-decs-x wf-comp-x,
simplified x, OF wf-fd-x, simplified, OF True lbs-take padding-d]
from hyp lbs
show ?thesis
  by (simp add: x True nth-append lacc take-eq)
next
case False
from False i-bound have i-bound': i - size-td d < size-td-list fs by simp
have padding-fs:  $\forall$  bs b.
  length bs = size-td-list fs  $\longrightarrow$ 
  norm-tu-list (map-td-list field-norm ( $\lambda$ -. ()) fs) (bs[i - size-td d := b]) =
  norm-tu-list (map-td-list field-norm ( $\lambda$ -. ()) fs) bs
proof (safe)
  fix bs::byte list and b::byte
  assume lbs: length bs = size-td-list fs
  from lbs obtain xbs pfx where xbs: xbs = pfx @ bs and lbs: length xbs =
size-td d + size-td-list fs
  and lpx: length pfx = size-td d
  by (metis Ex-list-of-length length-append)

from lpx
have drop-eq1: drop (size-td d) xbs = bs
  by (simp add: xbs)
  from lpx False have drop-eq2: (drop (size-td d) (xbs[i := b])) = bs [i -
size-td d := b]
  by (auto simp add: xbs list-update-append)
from lpx False
have take-eq1: take (size-td d) (xbs[i := b]) = pfx
  by (simp add: xbs list-update-append)
from lpx False
have take-eq2: take (size-td d) xbs = pfx
  by (simp add: xbs)

  have leq1: length (norm-tu (map-td field-norm ( $\lambda$ -. ()) d) (take (size-td d)
(xbs[i := b]))) = size-td d
  apply (simp add: take-eq1)
  using wf-fd-norm-tu(1)[rule-format, OF wf-fd-d lpx ] lpx
  by (simp add: export-uinfo-def length-fa-ti wf-fd-d)
  have leq2: length (norm-tu (map-td field-norm ( $\lambda$ -. ()) d) (take (size-td d)
xbs)) = size-td d
  apply (simp add: take-eq2)
  using wf-fd-norm-tu(1)[rule-format, OF wf-fd-d lpx ] lpx
  by (simp add: export-uinfo-def length-fa-ti wf-fd-d)

from padding [OF lbs, of b, simplified drop-eq1 drop-eq2]
show norm-tu-list (map-td-list field-norm ( $\lambda$ -. ()) fs) (bs[i - size-td d := b])
=
norm-tu-list (map-td-list field-norm ( $\lambda$ -. ()) fs) bs
apply -

```

```

    apply (subst (asm) list-append-eq-split-conv)
    apply (simp add: leq1 leq2)
    apply simp
  done
qed
from False lbs have drop-eq: ((drop (size-td d) bs)[i - size-td d := b]) =
  (drop (size-td d) (bs[i := b]))
  by (simp add: drop-update-swap)
from False lbs have take-eq: take (size-td d) (bs[i := b]) = take (size-td d) bs
by simp
note hyp = Cons-typ-desc.hyps(2) [OF wf-desc-fs wf-sz-fs wf-fields-fs wf-comp-fs
wf-fd-fs i-bound' lbs-drop padding-fs]
from hyp
show ?thesis by (simp add: x False nth-append lacc drop-eq take-eq)
qed
next
case (DTuple-typ-desc d nm y)
then show ?case by (auto simp add: export-uinfo-def)
qed

```

```

lemma is-padding-byte-update-ti-id:
  assumes wf: wf-desc (t::'a xtyp-info)
  and wf-sz: wf-size-desc t
  and wf-descs: wf-field-descs (set (field-descs t))
  and wf-comp: wf-component-descs t
  and wf-fd: wf-fd t
  and i-bound: i < size-td t
  and lbs: length bs = size-td t
  and is-padding: is-padding-byte (export-uinfo t) i
shows update-ti t (bs[i := b]) v = update-ti t bs v
  using is-padding-byte-update-ti-id'(1) [OF wf wf-sz wf-descs wf-comp wf-fd i-bound
lbs] is-padding
  by (simp add: is-padding-byte-def export-uinfo-def)

```

```

context xmem-type
begin
lemma xmem-type-is-padding-byte-update-ti-id:
  assumes padding: is-padding-byte (typ-uinfo-t TYPE('a)) i
  and i-bound: i < size-of TYPE('a)
  and lbs: length bs = size-of TYPE('a)
shows update-ti (typ-info-t TYPE('a)) (bs[i:=b]) v = update-ti (typ-info-t TYPE('a))
bs v
proof -
  have wf: wf-desc (typ-info-t TYPE('a)) by simp
  have wf-sz: wf-size-desc (typ-info-t TYPE('a)) by simp
  have wf-descs: wf-field-descs (set (field-descs (typ-info-t TYPE('a))))
    using local.wf-field-descs by blast
  have wf-comp: wf-component-descs (typ-info-t TYPE('a))
    by (simp add: local.wf-component-descs)

```

```

have wf-fd: wf-fd (typ-info-t TYPE('a))
  using local.wf-desc local.wf-lf wf-fdp-fd(1) wf-lf-fdp by blast
from is-padding-byte-update-ti-id [OF wf wf-sz wf-descs wf-comp wf-fd
  i-bound [simplified size-of-def] lbs [simplified size-of-def]
  padding [simplified typ-uinfo-t-def]]
show ?thesis .
qed
end

```

```

lemma heap-update-fold:
sz = size-of TYPE('a)  $\implies$ 
  heap-update-list a (to-bytes (v::'a:: c-type) (heap-list h sz a)) h = heap-update (Ptr a) v h
by (simp add:heap-update-def)

```

```

lemma heap-update-padding-fold:
sz = size-of TYPE('a)  $\implies$ 
  heap-update-list a (to-bytes (v::'a:: c-type) bs) h = heap-update-padding (Ptr a) v
  bs h
by (simp add:heap-update-padding-def)

```

```

context padding-base
begin
thm is-padding-byte-def
thm is-value-byte-def
end

```

```

lemma is-value-byte-access-ti-id':
  fixes t::'a xtyp-info
  and st::'a xtyp-info-struct
  and ts::'a xtyp-info-tuple list
  and x::'a xtyp-info-tuple
shows
[[ wf-desc t; wf-size-desc t; wf-field-descs (set (field-descs t)); wf-component-descs
t; wf-fd t;
  i < size-td t; length bs = size-td t;
   $\exists$  bs b. length bs = size-td t  $\wedge$  norm-tu (map-td field-norm ( $\lambda$ -. ()) t) (bs[i := b])
 $\neq$  norm-tu (export-uinfo t) bs]]
 $\implies$  access-ti t v (bs[i := b]) = access-ti t v bs

```

```

[[ wf-desc-struct st; wf-size-desc-struct st; wf-field-descs (set (field-descs-struct st));
  wf-component-descs-struct st; wf-fd-struct st; i < size-td-struct st; length bs =
size-td-struct st;
   $\exists$  bs b. length bs = size-td-struct st  $\wedge$  norm-tu-struct (map-td-struct field-norm
( $\lambda$ -. ()) st) (bs[i := b])  $\neq$  norm-tu-struct ((map-td-struct field-norm ( $\lambda$ -. ()) st))
bs]]

```

$\implies \text{access-ti-struct } st \ v \ (bs[i := b]) = \text{access-ti-struct } st \ v \ bs$

$\llbracket \text{wf-desc-list } ts; \text{wf-size-desc-list } ts; \text{wf-field-descs } (\text{set } (\text{field-descs-list } ts));$
 $\text{wf-component-descs-list } ts; \text{wf-fd-list } ts; \ i < \text{size-td-list } ts; \text{length } bs = \text{size-td-list}$
 $ts;$
 $\exists bs \ b. \text{length } bs = \text{size-td-list } ts \wedge \text{norm-tu-list } (\text{map-td-list } \text{field-norm } (\lambda-. \ ()))$
 $ts) \ (bs[i := b]) \neq \text{norm-tu-list } ((\text{map-td-list } \text{field-norm } (\lambda-. \ ())) \ ts) \ bs \rrbracket$
 $\implies \text{access-ti-list } ts \ v \ (bs[i := b]) = \text{access-ti-list } ts \ v \ bs$

$\llbracket \text{wf-desc-tuple } x; \text{wf-size-desc-tuple } x; \text{wf-field-descs } (\text{set } (\text{field-descs-tuple } x));$
 $\text{wf-component-descs-tuple } x; \text{wf-fd-tuple } x; \ i < \text{size-td-tuple } x; \text{length } bs = \text{size-td-tuple}$
 $x;$
 $\exists bs \ b. \text{length } bs = \text{size-td-tuple } x \wedge \text{norm-tu-tuple } (\text{map-td-tuple } \text{field-norm } (\lambda-. \ ()))$
 $x) \ (bs[i := b]) \neq \text{norm-tu-tuple } ((\text{map-td-tuple } \text{field-norm } (\lambda-. \ ())) \ x) \ bs \rrbracket$
 $\implies \text{access-ti-tuple } x \ v \ (bs[i := b]) = \text{access-ti-tuple } x \ v \ bs$

proof (*induct t and st and ts and x arbitrary: bs i and bs i and bs i and bs i*)

case (*TypDesc algn st nm*)
then show *?case by auto*
next

case (*TypScalar sz algn d*)
from *TypScalar obtain*
wf-d: wf-field-desc d and
field-sz: field-sz d =sz and
i-bound: i < sz and
lbs: length bs = sz and
is-val: $\exists bs \ b. \text{length } bs = sz \wedge$
field-norm sz algn d (bs[i := b]) \neq field-norm sz algn d bs
by *simp*

interpret *wf: wf-field-desc d by (rule wf-d)*
show *?case*
proof (*cases wf.is-padding-byte i*)
case *True*
then show *?thesis*
using *field-sz lbs wf.is-padding-byte-upd-eq*
by (*metis field-norm-def length-list-update is-val*)

next
case *False*
note *not-padding = this*
with *wf.complement field-sz i-bound have is-value: wf.is-value-byte i by blast*
from *is-value is-val*
show *?thesis*
apply *simp*
by (*metis field-sz lbs padding-base.is-value-byte-acc-eq*)

qed
next
case (*TypAggregate ts*)

```

then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
  obtain d nm y where x: x = DTuple d nm y by (cases x) auto
  from Cons-typ-desc.prem obtain
    wf-desc-d: wf-desc d and
    wf-fd-x: wf-fd-tuple (DTuple d nm y) and
    wf-fd-d: wf-fd d and
    wf-fd-fs: wf-fd-list fs and
    nm-notin: nm ∉ dt-snd ' set fs and
    wf-desc-fs: wf-desc-list fs and
    wf-sz-d: wf-size-desc d and
    wf-sz-fs: wf-size-desc-list fs and
    wf-field-d: wf-field-desc (component-desc d) and
    wf-fields-d: wf-field-descs (set (field-descs d)) and
    wf-fields-fs: wf-field-descs (set (field-descs-list fs)) and
    y: y = component-desc d and
    wf-comp-d: wf-component-descs d and
    wf-comp-fs: wf-component-descs-list fs and
    i-bound: i < size-td d + size-td-list fs and
    lbs: length bs = size-td d + size-td-list fs and
    wf-desc-x: wf-desc-tuple x and
    wf-sz-x: wf-size-desc-tuple x and
    wf-decs-x: wf-field-descs (set (field-descs-tuple x)) and
    wf-comp-x: wf-component-descs-tuple x and

    is-value: ∃ bs b. length bs = size-td d + size-td-list fs ∧
      norm-tu (map-td field-norm (λ-. ()) d)
        (take (size-td d) (bs[i := b])) @
      norm-tu-list (map-td-list field-norm (λ-. ()) fs)
        (drop (size-td d) (bs[i := b])) ≠
      norm-tu (map-td field-norm (λ-. ()) d) (take (size-td d) bs) @
      norm-tu-list (map-td-list field-norm (λ-. ()) fs)
        (drop (size-td d) bs)
  by (auto simp add: x)

from is-value obtain xbs xb where
  lbs: length xbs = size-td d + size-td-list fs and
  norm-neq: norm-tu (map-td field-norm (λ-. ()) d)
    (take (size-td d) (xbs[i := xb])) @
  norm-tu-list (map-td-list field-norm (λ-. ()) fs)
    (drop (size-td d) (xbs[i := xb])) ≠
  norm-tu (map-td field-norm (λ-. ()) d) (take (size-td d) xbs) @
  norm-tu-list (map-td-list field-norm (λ-. ()) fs)
    (drop (size-td d) xbs)
  by blast

```



```

from lbs have l-take1: length (take (size-td d) (xbs[i := xb])) = size-td d
  by simp
from wf-fd-norm-tu(1)[rule-format, OF wf-fd-d l-take1]
have l-norm-upd-d: length (norm-tu (map-td field-norm ( $\lambda$ -. ()) d) (take (size-td
d) (xbs[i := xb])))) = size-td d
  by (metis access-ti0-def export-uinfo-def fd-cons-length-p wf-fd-consD wf-fd-d)

from lbs have l-take2: length (take (size-td d) (xbs)) = size-td d
  by simp
from wf-fd-norm-tu(1)[rule-format, OF wf-fd-d l-take2]
have l-norm-d: length (norm-tu (map-td field-norm ( $\lambda$ -. ()) d) (take (size-td d)
xbs)) = size-td d
  by (metis access-ti0-def export-uinfo-def fd-cons-length-p wf-fd-consD wf-fd-d)

from lbs
have lbs-drop: length (drop (size-td d) bs) = size-td-list fs
  by (simp add: x)

from lbs
have lbs-take: length (take (size-td d) bs) = size-td d
  by (simp add: x)

have lacc: length (access-ti d v (take (size-td d) bs)) = size-td d
  by (metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps wf-fd-x x)

show ?case

proof (cases i < size-td d)
  case True
    from True lbs
    have take-eq: (take (size-td d) bs)[i := b] = take (size-td d) (bs[i := b])
      by (simp add: take-update-swap)
    from True
    have drop-eq: drop (size-td d) (xbs[i := xb]) = drop (size-td d) xbs
      by simp
    from lbs True
    have take-xbs-eq: (take (size-td d) (xbs[i := xb])) = (take (size-td d) xbs)[i :=
xb]
      by (simp add: take-update-swap)
    from norm-neq
    have norm-tu (map-td field-norm ( $\lambda$ -. ()) d) (take (size-td d) (xbs[i := xb]))  $\neq$ 
      norm-tu (map-td field-norm ( $\lambda$ -. ()) d) (take (size-td d) xbs)
      by (simp add: drop-eq)
    with take-xbs-eq
    have value-d:  $\exists$  bs. length bs = size-td d  $\wedge$ 
      ( $\exists$  b. norm-tu (map-td field-norm ( $\lambda$ -. ()) d) (bs[i := b])  $\neq$ 
norm-tu (map-td field-norm ( $\lambda$ -. ()) d) bs)
      apply simp

```

```

    using l-take2 by blast
    note hyp = Cons-typ-desc.hyps(1) [OF wf-desc-x wf-sz-x wf-decs-x wf-comp-x,
simplified x, OF wf-fd-x, simplified, OF True lbs-take value-d]
    from hyp lbs
    show ?thesis
    by (simp add: x True nth-append lacc take-eq)
next
case False
from False i-bound have i-bound': i - size-td d < size-td-list fs by simp

from False lxs
have take-xs: take (size-td d) (xbs[i := xb]) = take (size-td d) xbs by simp

from norm-neq
have norm-tu-list (map-td-list field-norm (λ-. ()) fs) (drop (size-td d) (xbs[i :=
xb])) ≠
    norm-tu-list (map-td-list field-norm (λ-. ()) fs) (drop (size-td d) xbs)
by (simp add: take-xs)
then
have value-fs: ∃ bs b.
    length bs = size-td-list fs ∧
    norm-tu-list (map-td-list field-norm (λ-. ()) fs) (bs[i - size-td d := b]) ≠
    norm-tu-list (map-td-list field-norm (λ-. ()) fs) bs
by (metis False drop-update-swap lbs lbs-drop length-drop lxs not-le)

from False lbs have drop-eq: ((drop (size-td d) bs)[i - size-td d := b]) =
    (drop (size-td d) (bs[i := b]))
by (simp add: drop-update-swap)
from False lbs have take-eq: take (size-td d) (bs[i := b]) = take (size-td d) bs
by simp
note hyp = Cons-typ-desc.hyps(2) [OF wf-desc-fs wf-sz-fs wf-fields-fs wf-comp-fs
wf-fd-fs i-bound' lbs-drop value-fs]
from hyp
show ?thesis by (simp add: x False nth-append lacc drop-eq take-eq)
qed
next
case (DTuple-typ-desc d nm y)
then show ?case by (auto simp add: export-uinfo-def)
qed

```

```

lemma is-value-byte-access-ti-id:
  assumes wf: wf-desc t
  assumes wf-sz: wf-size-desc t
  assumes wf-decs: wf-field-decs (set (field-decs t))
  assumes wf-comps: wf-component-decs t
  assumes wf-fd: wf-fd t
  assumes i-bound: i < size-td t
  assumes lbs: length bs = size-td t
  assumes is-value: is-value-byte (export-uinfo t) i

```

shows $\text{access-ti } t \ v \ (\text{bs}[i := b]) = \text{access-ti } t \ v \ \text{bs}$
using $\text{is-value-byte-access-ti-id}'(1)[OF \ \text{wf} \ \text{wf-sz} \ \text{wf-descs} \ \text{wf-comps} \ \text{wf-fd} \ \text{i-bound}$
 $\text{lbs, where } v=v \ \text{and } b=b \] \ \text{is-value}$
by ($\text{simp add: is-value-byte-def export-uinfo-def}$)

lemma $\text{is-value-byte-access-ti-update-ti-cancel}'$:

fixes $t::'a \ \text{xtyp-info}$
and $st::'a \ \text{xtyp-info-struct}$
and $ts::'a \ \text{xtyp-info-tuple list}$
and $x::'a \ \text{xtyp-info-tuple}$

shows

$\llbracket \text{wf-desc } t; \text{wf-size-desc } t; \text{wf-field-descs } (\text{set } (\text{field-descs } t)); \text{wf-component-descs}$
 $t; \text{wf-fd } t;$

$i < \text{size-td } t; \text{length } \text{bs} = \text{size-td } t; \text{length } \text{bs}' = \text{size-td } t;$

$\exists \text{bs } b. \text{length } \text{bs} = \text{size-td } t \wedge \text{norm-tu } (\text{map-td field-norm } (\lambda-. ()) \ t) \ (\text{bs}[i := b])$
 $\neq \text{norm-tu } (\text{export-uinfo } t) \ \text{bs} \rrbracket$

$\implies \text{access-ti } t \ (\text{update-ti } t \ \text{bs } v) \ \text{bs}' ! i = \text{bs} ! i$

$\llbracket \text{wf-desc-struct } st; \text{wf-size-desc-struct } st; \text{wf-field-descs } (\text{set } (\text{field-descs-struct } st));$
 $\text{wf-component-descs-struct } st; \text{wf-fd-struct } st; i < \text{size-td-struct } st; \text{length } \text{bs} =$
 $\text{size-td-struct } st; \text{length } \text{bs}' = \text{size-td-struct } st;$

$\exists \text{bs } b. \text{length } \text{bs} = \text{size-td-struct } st \wedge \text{norm-tu-struct } (\text{map-td-struct field-norm}$
 $(\lambda-. ()) \ st) \ (\text{bs}[i := b]) \neq \text{norm-tu-struct } ((\text{map-td-struct field-norm } (\lambda-. ()) \ st))$
 $\text{bs} \rrbracket$

$\implies \text{access-ti-struct } st \ (\text{update-ti-struct } st \ \text{bs } v) \ \text{bs}' ! i = \text{bs} ! i$

$\llbracket \text{wf-desc-list } ts; \text{wf-size-desc-list } ts; \text{wf-field-descs } (\text{set } (\text{field-descs-list } ts));$
 $\text{wf-component-descs-list } ts; \text{wf-fd-list } ts; i < \text{size-td-list } ts; \text{length } \text{bs} = \text{size-td-list}$
 $ts; \text{length } \text{bs}' = \text{size-td-list } ts;$

$\exists \text{bs } b. \text{length } \text{bs} = \text{size-td-list } ts \wedge \text{norm-tu-list } (\text{map-td-list field-norm } (\lambda-. ())$
 $ts) \ (\text{bs}[i := b]) \neq \text{norm-tu-list } ((\text{map-td-list field-norm } (\lambda-. ()) \ ts)) \ \text{bs} \rrbracket$

$\implies \text{access-ti-list } ts \ (\text{update-ti-list } ts \ \text{bs } v) \ \text{bs}' ! i = \text{bs} ! i$

$\llbracket \text{wf-desc-tuple } x; \text{wf-size-desc-tuple } x; \text{wf-field-descs } (\text{set } (\text{field-descs-tuple } x));$
 $\text{wf-component-descs-tuple } x; \text{wf-fd-tuple } x; i < \text{size-td-tuple } x; \text{length } \text{bs} = \text{size-td-tuple}$
 $x; \text{length } \text{bs}' = \text{size-td-tuple } x;$

$\exists \text{bs } b. \text{length } \text{bs} = \text{size-td-tuple } x \wedge \text{norm-tu-tuple } (\text{map-td-tuple field-norm } (\lambda-.$
 $(()) \ x) \ (\text{bs}[i := b]) \neq \text{norm-tu-tuple } ((\text{map-td-tuple field-norm } (\lambda-. ()) \ x)) \ \text{bs} \rrbracket$

$\implies \text{access-ti-tuple } x \ (\text{update-ti-tuple } x \ \text{bs } v) \ \text{bs}' ! i = \text{bs} ! i$

proof ($\text{induct } t \ \text{and } st \ \text{and } ts \ \text{and } x \ \text{arbitrary: } \text{bs } \text{bs}' \ i \ v \ \text{and } \text{bs } \text{bs}' \ i \ v \ \text{and } \text{bs}$
 $\text{bs}' \ i \ v \ \text{and } \text{bs } \text{bs}' \ i \ v$)

case ($\text{TypDesc } \text{algn } st \ nm$)

then show $?case \ \text{by } \text{auto}$

next

case ($\text{TypScalar } sz \ \text{algn } d$)

from TypScalar **obtain**

$\text{wf-d: wf-field-desc } d \ \text{and}$

$\text{field-sz: field-sz } d = sz \ \text{and}$

```

i-bound:  $i < sz$  and
lbs:  $length\ bs = sz$  and
lbs':  $length\ bs' = sz$  and
is-val:  $\exists bs\ b. length\ bs = sz \wedge$ 
       $field\text{-}norm\ sz\ algn\ d\ (bs[i := b]) \neq field\text{-}norm\ sz\ algn\ d\ bs$ 
by simp

interpret wf: wf-field-desc d by (rule wf-d)
show ?case
proof (cases wf.is-padding-byte i)
  case True
  then show ?thesis
    using field-sz lbs wf.is-padding-byte-upd-eq
    by (metis field-norm-def length-list-update is-val)

next
  case False
  note not-padding = this
  with wf.complement field-sz i-bound have is-value: wf.is-value-byte i by blast
  from is-value is-val lbs'
  show ?thesis
    by (simp add: field-sz lbs wf.is-value-byte-acc-upd-cancel)
qed
next
  case (TypAggregate ts)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
  obtain d nm y where x: x = DTuple d nm y by (cases x) auto

from Cons-typ-desc.prems obtain
  wf-desc-d: wf-desc d and
  wf-fd-x: wf-fd-tuple (DTuple d nm y) and
  wf-fd-d: wf-fd d and
  wf-fd-fs: wf-fd-list fs and
  nm-notin:  $nm \notin dt\text{-}snd\ 'set\ fs$  and
  wf-desc-fs: wf-desc-list fs and
  wf-sz-d: wf-size-desc d and
  wf-sz-fs: wf-size-desc-list fs and
  wf-field-d: wf-field-desc (component-desc d) and
  wf-fields-d: wf-field-descs (set (field-descs d)) and
  wf-fields-fs: wf-field-descs (set (field-descs-list fs)) and
  y:  $y = component\text{-}desc\ d$  and
  wf-comp-d: wf-component-descs d and
  wf-comp-fs: wf-component-descs-list fs and
  i-bound:  $i < size\text{-}td\ d + size\text{-}td\text{-}list\ fs$  and

```

lbs: $\text{length } bs = \text{size-td } d + \text{size-td-list } fs$ **and**
lbs': $\text{length } bs' = \text{size-td } d + \text{size-td-list } fs$ **and**
wf-desc-x: *wf-desc-tuple* *x* **and**
wf-sz-x: *wf-size-desc-tuple* *x* **and**
wf-decs-x: *wf-field-descs* (*set* (*field-descs-tuple* *x*)) **and**
wf-comp-x: *wf-component-descs-tuple* *x* **and**

is-value: $\exists bs \ b. \text{length } bs = \text{size-td } d + \text{size-td-list } fs \wedge$
 $\text{norm-tu } (\text{map-td } \text{field-norm } (\lambda-. ()) \ d)$
 $(\text{take } (\text{size-td } d) \ (bs[i := b])) \ @$
 $\text{norm-tu-list } (\text{map-td-list } \text{field-norm } (\lambda-. ()) \ fs)$
 $(\text{drop } (\text{size-td } d) \ (bs[i := b])) \neq$
 $\text{norm-tu } (\text{map-td } \text{field-norm } (\lambda-. ()) \ d) \ (\text{take } (\text{size-td } d) \ bs) \ @$
 $\text{norm-tu-list } (\text{map-td-list } \text{field-norm } (\lambda-. ()) \ fs)$
 $(\text{drop } (\text{size-td } d) \ bs)$ **and**
commutes: *fu-commutes* (*update-ti-t* *d*) (*update-ti-list-t* *fs*)
by (*auto simp add: x*)

from *is-value* **obtain** *xbs* *xb* **where**

lxs: $\text{length } xbs = \text{size-td } d + \text{size-td-list } fs$ **and**
norm-neg: $\text{norm-tu } (\text{map-td } \text{field-norm } (\lambda-. ()) \ d)$
 $(\text{take } (\text{size-td } d) \ (xbs[i := xb])) \ @$
 $\text{norm-tu-list } (\text{map-td-list } \text{field-norm } (\lambda-. ()) \ fs)$
 $(\text{drop } (\text{size-td } d) \ (xbs[i := xb])) \neq$
 $\text{norm-tu } (\text{map-td } \text{field-norm } (\lambda-. ()) \ d) \ (\text{take } (\text{size-td } d) \ xbs) \ @$
 $\text{norm-tu-list } (\text{map-td-list } \text{field-norm } (\lambda-. ()) \ fs)$
 $(\text{drop } (\text{size-td } d) \ xbs)$

by *blast*

from *lxs* **have** *l-take1*: $\text{length } (\text{take } (\text{size-td } d) \ (xbs[i := xb])) = \text{size-td } d$

by *simp*

from *wf-fd-norm-tu(1)*[*rule-format*, *OF wf-fd-d l-take1*]

have *l-norm-upd-d*: $\text{length } (\text{norm-tu } (\text{map-td } \text{field-norm } (\lambda-. ()) \ d) \ (\text{take } (\text{size-td } d) \ (xbs[i := xb]))) = \text{size-td } d$

by (*metis access-ti₀-def export-uinfo-def fd-cons-length-p wf-fd-consD wf-fd-d*)

from *lxs* **have** *l-take2*: $\text{length } (\text{take } (\text{size-td } d) \ (xbs)) = \text{size-td } d$

by *simp*

from *wf-fd-norm-tu(1)*[*rule-format*, *OF wf-fd-d l-take2*]

have *l-norm-d*: $\text{length } (\text{norm-tu } (\text{map-td } \text{field-norm } (\lambda-. ()) \ d) \ (\text{take } (\text{size-td } d) \ xbs)) = \text{size-td } d$

by (*metis access-ti₀-def export-uinfo-def fd-cons-length-p wf-fd-consD wf-fd-d*)

from *lbs*

have *lbs-drop*: $\text{length } (\text{drop } (\text{size-td } d) \ bs) = \text{size-td-list } fs$

by (*simp add: x*)

from *lbs'*

```

have lbs'-drop: length (drop (size-td d) bs') = size-td-list fs
  by (simp add: x)

from lbs
have lbs-take: length (take (size-td d) bs) = size-td d
  by (simp add: x)

from lbs'
have lbs'-take: length (take (size-td d) bs') = size-td d
  by (simp add: x)

have lacc: length (access-ti d v (take (size-td d) bs)) = size-td d
  by (metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps wf-fd-x x)

from lbs lbs'-take
have l-acc-upd:
  length (access-ti d (update-ti d (take (size-td d) bs)
    (update-ti-list fs (drop (size-td d) bs) v)) (take (size-td d) bs')) =
size-td d
  by (meson length-fa-ti wf-fd-d)
show ?case

proof (cases i < size-td d)
  case True
  from True lbs
  have take-eq: (take (size-td d) bs)[i := b] = take (size-td d) (bs[i := b])
    by (simp add: take-update-swap)
  from True
  have drop-eq: drop (size-td d) (xbs[i := xb]) = drop (size-td d) xbs
    by simp
  from lxb True
  have take-xbs-eq: (take (size-td d) (xbs[i := xb])) = (take (size-td d) xbs)[i :=
xb]
    by (simp add: take-update-swap)
  from norm-neg
  have norm-tu (map-td field-norm (λ-. ()) d (take (size-td d) (xbs[i := xb])))  $\neq$ 
    norm-tu (map-td field-norm (λ-. ()) d (take (size-td d) xbs))
    by (simp add: drop-eq)
  with take-xbs-eq
  have value-d:  $\exists$  bs. length bs = size-td d  $\wedge$ 
     $(\exists b. \text{norm-tu (map-td field-norm (}\lambda\text{-. ()) d (bs[i := b])} \neq$ 
    norm-tu (map-td field-norm (}\lambda\text{-. ()) d bs)
    apply simp
    using l-take2 by blast

```

note *hyp* = *Cons-tyt-desc.hyps(1)* [*OF wf-desc-x wf-sz-x wf-decs-x wf-comp-x, simplified x, OF wf-fd-x, simplified,*

```

      OF True lbs-take lbs'-take value-d, where v = (update-ti-list fs (drop (size-td
d) bs) v)]
    from lbs lbs'
    show ?thesis
    by (simp add: x True nth-append lacc take-eq l-acc-upd hyp)
  next
    case False
    from False i-bound have i-bound': i - size-td d < size-td-list fs by simp

    from False lxs
    have take-xs: take (size-td d) (xbs[i := xb]) = take (size-td d) xbs by simp

    from norm-neq
    have norm-tu-list (map-td-list field-norm (λ-. ()) fs) (drop (size-td d) (xbs[i :=
xb])) ≠
      norm-tu-list (map-td-list field-norm (λ-. ()) fs) (drop (size-td d) xbs)
    by (simp add: take-xs)
    then
    have value-fs: ∃ bs b.
      length bs = size-td-list fs ∧
      norm-tu-list (map-td-list field-norm (λ-. ()) fs) (bs[i - size-td d := b]) ≠
      norm-tu-list (map-td-list field-norm (λ-. ()) fs) bs
    by (metis False drop-update-swap lbs lbs-drop length-drop lxs not-le)

    from commutes have commute:
      (update-ti d (take (size-td d) bs)
      (update-ti-list fs (drop (size-td d) bs) v)) =
      (update-ti-list fs (drop (size-td d) bs) (update-ti d (take (size-td d) bs) v))
    apply (simp add: update-ti-t-def update-ti-list-t-def fu-commutes-def)
    apply (erule allE [where x=v])
    apply (erule allE [where x=(take (size-td d) bs)])
    apply (erule allE [where x=(drop (size-td d) bs)])
    using lbs-take lbs-drop
    apply simp
    done

    from False lbs have drop-eq: drop (size-td d) bs ! (i - size-td d) = bs ! i
    by (simp add: drop-update-swap)

    note hyp = Cons-typ-desc.hyps(2) [OF wf-desc-fs wf-sz-fs wf-fields-fs wf-comp-fs
wf-fd-fs i-bound' lbs-drop lbs'-drop value-fs]
    from hyp
    show ?thesis
    apply (simp add: x False nth-append lacc drop-eq l-acc-upd)
    apply (simp add: commute)
    done
  qed
next

```

case (*DTuple-typ-desc* *d nm y*)
then show *?case* **by** (*auto simp add: export-uinfo-def*)
qed

lemma *is-value-byte-access-ti-update-ti-cancel*:

assumes *wf: wf-desc t*
assumes *wf-sz: wf-size-desc t*
assumes *wf-descs: wf-field-descs (set (field-descs t))*
assumes *wf-comps: wf-component-descs t*
assumes *wf-fd: wf-fd t*
assumes *i-bound: i < size-td t*
assumes *lbs: length bs = size-td t*
assumes *lbs': length bs' = size-td t*
assumes *is-value: is-value-byte (export-uinfo t) i*
shows *access-ti t (update-ti t bs v) bs' ! i = bs ! i*
using *is-value-byte-access-ti-update-ti-cancel'(1) [OF wf wf-sz wf-descs wf-comps*
wf-fd i-bound lbs lbs'] is-value
by (*simp add: export-uinfo-def is-value-byte-def*)

lemma *field-lookup-is-padding-byte-focus'*:

fixes *t::('a, 'b) typ-info*
and *st::('a, 'b) typ-info-struct*
and *ts::('a, 'b) typ-info-tuple list*
and *x::('a, 'b) typ-info-tuple*

shows

$\llbracket \text{field-lookup } t f m = \text{Some } (s, m + n); n \leq i; i < n + \text{size-td } s; \text{length } bs = \text{size-td } t;$
 $\text{wf-desc } t; \text{wf-size-desc } t \rrbracket \implies$
 $\text{norm-tu } (\text{map-td field-norm } (\lambda-. ()) t) (bs[i := b]) = \text{norm-tu } (\text{map-td field-norm } (\lambda-. ()) t) bs$
 \longleftrightarrow
 $\text{norm-tu } (\text{export-uinfo } s) ((\text{take } (\text{size-td } s) (\text{drop } n bs))[i - n := b]) =$
 $\text{norm-tu } (\text{export-uinfo } s) (\text{take } (\text{size-td } s) (\text{drop } n bs))$

$\llbracket \text{field-lookup-struct } st f m = \text{Some } (s, m + n); n \leq i; i < n + \text{size-td } s; \text{length } bs = \text{size-td-struct } st;$
 $\text{wf-desc-struct } st; \text{wf-size-desc-struct } st \rrbracket \implies$

$\text{norm-tu-struct } (\text{map-td-struct field-norm } (\lambda-. ()) st) (bs[i := b]) = \text{norm-tu-struct } (\text{map-td-struct field-norm } (\lambda-. ()) st) bs$
 \longleftrightarrow
 $\text{norm-tu } (\text{export-uinfo } s) ((\text{take } (\text{size-td } s) (\text{drop } n bs))[i - n := b]) =$
 $\text{norm-tu } (\text{export-uinfo } s) (\text{take } (\text{size-td } s) (\text{drop } n bs))$

$\llbracket \text{field-lookup-list } ts f m = \text{Some } (s, m + n); n \leq i; i < n + \text{size-td } s; \text{length } bs = \text{size-td-list } ts;$
 $\text{wf-desc-list } ts; \text{wf-size-desc-list } ts \rrbracket \implies$

$\text{norm-tu-list } (\text{map-td-list field-norm } (\lambda-. ()) ts) (bs[i := b]) = \text{norm-tu-list } (\text{map-td-list field-norm } (\lambda-. ()) ts) bs$

\longleftrightarrow
 $norm\text{-}tu\ (export\text{-}uinfo\ s)\ ((take\ (size\text{-}td\ s)\ (drop\ n\ bs))[i - n := b]) =$
 $norm\text{-}tu\ (export\text{-}uinfo\ s)\ (take\ (size\text{-}td\ s)\ (drop\ n\ bs))$

$\llbracket field\text{-}lookup\text{-}tuple\ x\ f\ m = Some\ (s,\ m + n); n \leq i; i < n + size\text{-}td\ s; length\ bs =$
 $size\text{-}td\text{-}tuple\ x;$
 $wf\text{-}desc\text{-}tuple\ x; wf\text{-}size\text{-}desc\text{-}tuple\ x \rrbracket \implies$
 $norm\text{-}tu\text{-}tuple\ (map\text{-}td\text{-}tuple\ field\text{-}norm\ (\lambda\cdot.\ ())\ x)\ (bs[i := b]) = norm\text{-}tu\text{-}tuple$
 $(map\text{-}td\text{-}tuple\ field\text{-}norm\ (\lambda\cdot.\ ())\ x)\ bs$

\longleftrightarrow
 $norm\text{-}tu\ (export\text{-}uinfo\ s)\ ((take\ (size\text{-}td\ s)\ (drop\ n\ bs))[i - n := b]) =$
 $norm\text{-}tu\ (export\text{-}uinfo\ s)\ (take\ (size\text{-}td\ s)\ (drop\ n\ bs))$

proof (*induct t and st and ts and x arbitrary: f m n i s bs and f m n i s bs and*
f m n i s bs and f m n i s bs)
case (*TypDesc* *algn st nm*)
then show *?case by (auto split: if-split-asm)*
next
case (*TypScalar* *sz algn d*)
then show *?case by auto*
next
case (*TypAggregate* *ts*)
then show *?case by auto*
next
case *Nil-typ-desc*
then show *?case by auto*
next
case (*Cons-typ-desc* *x fs*)
obtain *d nm y where x: x = DTuple d nm y by (cases x) auto*
from *Cons-typ-desc.prem*s **obtain**
lbs: length bs = size-td-tuple x + size-td-list fs and
wf-desc-x: wf-desc-tuple (DTuple d nm y) and
wf-size-desc-x: wf-size-desc-tuple (DTuple d nm y) and
wf-size-desc-fs: wf-size-desc-list fs and
nm-notin: nm ∉ dt-snd ' set fs and
wf-desc-fs: wf-desc-list fs and
i-lower: n ≤ i and
i-upper: i < n + size-td s
by (*auto simp add: x*)

from *lbs*
have *lbs-drop: length (drop (size-td-tuple x) bs) = size-td-list fs*
by *simp*

from *lbs*
have *lbs-take: length (take (size-td-tuple x) bs) = size-td-tuple (DTuple d nm y)*
by (*simp add: x*)

from *lbs-take*
have *take-eq1: ((take (size-td d) bs)[i := b]) = (take (size-td d) (bs[i := b]))*

```

    by (simp add: take-update-swap)

from Cons-typ-desc.prems obtain f1 fxs
  where f: f = f1 # fxs
  by (cases f) auto

thm Cons-typ-desc.hyps [simplified x]

show ?case
proof (cases f1 = nm)
  case True
  show ?thesis
  proof (cases field-lookup d fxs m)
    case None
    from Cons-typ-desc.prems field-lookup-list-None [OF nm-notin]
    have False
      by (simp add: True None f x)
    thus ?thesis by simp
  next
  case (Some r)
  from Cons-typ-desc.prems have r: r = (s, m + n)
    by (simp add: x True Some f)
  hence fl: field-lookup-tuple (DTuple d nm y) f m = Some (s, m + n)
    by (simp add: f True Some r)
  from td-set-wf-size-desc(4)[OF wf-size-desc-x td-set-tuple-field-lookup-tupleD,
OF fl]
  have wf-size-desc s .
  from wf-size-desc-gt(1)[OF this]
  have 0 < size-td s .

  with td-set-tuple-offset-size-m [OF td-set-tuple-field-lookup-tupleD, OF fl]
  have n-le: n < size-td d
    by auto

  have i-in-d: i < size-td d
    using ⟨size-td s + (m + n - m) ≤ size-td-tuple (DTuple d nm y)⟩ i-upper
  by auto
  have bound: size-td s + (m + n - m) ≤ size-td-tuple (DTuple d nm y) by
  fact
  from bound

  have take-eq: (take (size-td s) (drop n (take (size-td d) bs))) =
    (take (size-td s) (drop n bs))
    by (simp add: take-drop)
  from Cons-typ-desc.hyps(1)[simplified x, OF fl i-lower i-upper lbs-take
wf-desc-x wf-size-desc-x] lbs bound
  show ?thesis
    by (simp add: x True Some r take-eq nth-append i-in-d take-eq1)

```

```

    qed
  next
  case False
  with Cons-typ-desc.prems
  have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, m + n)
    by (clarsimp simp add: x f False)
  hence n-bound: size-td d ≤ n
    by (meson field-lookup-offset-le(3) nat-add-left-cancel-le)

  from fl
  have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, (m + size-td d) + (n - size-td d))
    by (metis Nat.diff-cancel field-lookup-list-offset2 field-lookup-list-offsetD)
  from n-bound
  have take-eq: (take (size-td s) (drop (n - size-td d + size-td d) bs)) = (take (size-td s) (drop n bs))
    by simp

  have i-lower': n - size-td d ≤ i - size-td d
    using diff-le-mono i-lower by blast

  have i-upper': i - size-td d < n - size-td d + size-td s
    using i-lower i-upper by linarith

  have i-notin-d:  $\neg i < \text{size-td } d$ 
    by (meson i-lower leD less-le-trans n-bound)

  from i-notin-d lbs-take have take-eq: take (size-td d) (bs[i := b]) = take (size-td d) bs
    by simp

  from i-notin-d lbs i-upper not-le have drop-eq:
     $((\text{drop } (\text{size-td } d) \text{ bs})[i - \text{size-td } d := b]) = (\text{drop } (\text{size-td } d) (\text{bs}[i := b]))$ 
    by (simp add: drop-update-swap)

  from Cons-typ-desc.hyps(2) [OF fl i-lower' i-upper' lbs-drop wf-desc-fs wf-size-desc-fs]
    False n-bound
  show ?thesis
    by (simp add: x f nth-append i-notin-d take-eq drop-eq)
  qed
next
case (DTuple-typ-desc d nm y)
then show ?case apply (cases f) by (auto split: if-split-asm)
qed

lemma field-lookup-is-padding-byte-focus:
  assumes fl: field-lookup t f 0 = Some (s, n)
  assumes i-lower:  $n \leq i$ 
  assumes i-upper:  $i < n + \text{size-td } s$ 

```

```

assumes wf: wf-desc t
assumes wf-sz: wf-size-desc t
shows is-padding-byte (export-uinfo t) i = is-padding-byte (export-uinfo s) (i - n)
proof
  assume is-padding: is-padding-byte (export-uinfo t) i
  show is-padding-byte (export-uinfo s) (i - n)
    apply (simp add: is-padding-byte-def, safe)
  proof -
    from i-upper i-lower show i - n < size-td s by simp
  next
    fix bs::byte list and b::byte
    assume lbs: length bs = size-td s
    from fl have le: n + size-td s ≤ size-td t
      by (metis add.commute field-lookup-offset-size')
    with lbs i-lower i-upper obtain pfx sfx xbs where
      xbs: xbs = pfx @ bs @ sfx and
      lbs: length xbs = size-td t and
      lpx: length pfx = n

    by (metis Ex-list-of-length ab-semigroup-add-class.add-ac(1) le-Suc-ex length-append)

    have take-eq: take (size-td s) (drop n xbs) = bs
      using xbs lbs lpx le
      by simp
    from field-lookup-is-padding-byte-focus'(1) [where m = 0, simplified, OF fl
i-lower i-upper lbs wf wf-sz, where b=b]
      is-padding lbs
    show norm-tu (export-uinfo s) (bs[i - n := b]) =
      norm-tu (export-uinfo s) bs
      by (simp add: take-eq export-uinfo-def is-padding-byte-def)
    qed
  next
    assume is-padding: is-padding-byte (export-uinfo s) (i - n)
    from fl i-upper i-lower
    have i-t: i < size-td t
      using field-lookup-offset-size' by fastforce
    show is-padding-byte (export-uinfo t) i
      apply (simp add: is-padding-byte-def, safe)
    proof -
      from i-t
      show sz: i < size-td t .
    next
      fix bs::byte list and b::byte
      assume lbs: length bs = size-td t
      from fl i-upper i-lower i-t
      have less: i - n < size-td t
        by simp

      from less lbs i-lower i-upper i-t

```

have $sz-s$: $size-td\ s \leq length\ (drop\ n\ bs)$
by (*metis add-le-imp-le-diff field-lookup-offset-size' fl length-drop*)

from *field-lookup-is-padding-byte-focus'(1)* [**where** $m = 0$, *simplified, OF fl*
i-lower i-upper lbs wf wf-sz, where $b=b$]
is-padding sz-s
show $norm-tu\ (export-uinfo\ t)\ (bs[i := b]) =$
 $norm-tu\ (export-uinfo\ t)\ bs$
by (*auto simp add: is-padding-byte-def export-uinfo-def*)
qed
qed

lemma *field-lookup-is-value-byte-focus*:
assumes fl : *field-lookup t f 0 = Some (s, n)*
assumes $i-lower$: $n \leq i$
assumes $i-upper$: $i < n + size-td\ s$
assumes wf : *wf-desc t*
assumes $wf-sz$: *wf-size-desc t*
shows *is-value-byte (export-uinfo t) i = is-value-byte (export-uinfo s) (i - n)*
proof –
from $fl\ i-upper\ i-lower$
have $i-t$: $i < size-td\ t$
using *field-lookup-offset-size'* **by** *fastforce*
from $i-lower\ i-upper$
have $i - n < size-td\ s$ **by** *simp*
with *field-lookup-is-padding-byte-focus [OF fl i-lower i-upper wf wf-sz] comple-*
ment-tu-padding i-t
show *?thesis*
by (*simp add: export-uinfo-size*)
qed

lemma *field-lookup-eq-padding-focus*:
assumes fl : *field-lookup t f 0 = Some (s, n)*
assumes lbs : $length\ bs = size-td\ t$
assumes lbs' : $length\ bs' = size-td\ t$
assumes wf : *wf-desc t*
assumes $wf-sz$: *wf-size-desc t*
assumes *eq-padding: eq-padding (export-uinfo t) bs bs'*
shows *eq-padding (export-uinfo s) (take (size-td s) (drop n bs)) (take (size-td s)*
(drop n bs'))
proof –
from $fl\ lbs$ **have** $bound-s$: $size-td\ s \leq length\ bs - n$
by (*metis add.commute add-diff-cancel-left' diff-le-mono field-lookup-offset-size'*)
from $fl\ lbs'$ **have** $bound-s'$: $size-td\ s \leq length\ bs' - n$
by (*metis add.commute add-diff-cancel-left' diff-le-mono field-lookup-offset-size'*)
from $bound-s\ lbs$
have $l-take$: $length\ (take\ (size-td\ s)\ (drop\ n\ bs)) = size-td\ s$
by *simp*

```

from bound-s' lbs'
have l-take': length (take (size-td s) (drop n bs')) = size-td s
  by simp

show ?thesis
  apply (simp add: eq-padding-def, safe)
proof -
  show size-td s ≤ length bs - n by (rule bound-s)
next
  show size-td s ≤ length bs' - n by (rule bound-s')
next
  fix i assume is-padding: is-padding-byte (export-uinfo s) i
  from is-padding have i-s: i < size-td s
    by (simp add: is-padding-byte-def)
  have n-i: n ≤ i + n by simp
  from i-s have i-n-bound: i + n < n + size-td s by simp
  show take (size-td s) (drop n bs) ! i =
    take (size-td s) (drop n bs') ! i

  using l-take l-take' is-padding eq-padding lbs lbs'
    field-lookup-is-padding-byte-focus [OF fl n-i i-n-bound wf wf-sz, simplified]
  by (auto simp add: eq-padding-def add.commute is-padding-byte-def)
qed
qed

```

```

lemma field-lookup-eq-padding-focus-eq:
assumes fl: field-lookup t f 0 = Some (s, n)
assumes lbs: length bs = size-td t
assumes lbs': length bs' = size-td t
assumes pfx-eq:  $\bigwedge i. i < n \implies bs ! i = bs' ! i$ 
assumes sfx-eq:  $\bigwedge i. n + size-td s \leq i \implies i < size-td t \implies bs ! i = bs' ! i$ 
assumes wf: wf-desc t
assumes wf-sz: wf-size-desc t
shows eq-padding (export-uinfo t) bs bs' =
  eq-padding (export-uinfo s) (take (size-td s) (drop n bs)) (take (size-td s) (drop
n bs'))
proof -
  from fl lbs have bound-s: size-td s ≤ length bs - n
  by (metis add.commute add-diff-cancel-left' diff-le-mono field-lookup-offset-size')
  from fl lbs' have bound-s': size-td s ≤ length bs' - n
  by (metis add.commute add-diff-cancel-left' diff-le-mono field-lookup-offset-size')
  from bound-s lbs
  have l-take: length (take (size-td s) (drop n bs)) = size-td s
  by simp
  from bound-s' lbs'
  have l-take': length (take (size-td s) (drop n bs')) = size-td s
  by simp

```

```

show ?thesis
proof
  assume eq-padding: eq-padding (export-uinfo t) bs bs'
  show eq-padding (export-uinfo s) (take (size-td s) (drop n bs)) (take (size-td s)
(drop n bs^))
    by (rule field-lookup-eq-padding-focus [OF fl lbs lbs' wf wf-sz eq-padding])
  next
    assume eq-padding: eq-padding (export-uinfo s) (take (size-td s) (drop n bs))
(take (size-td s) (drop n bs^))
    show eq-padding (export-uinfo t) bs bs'
    apply (simp add: eq-padding-def, safe)
  proof -
    show length bs = size-td t by (rule lbs)
  next
    show length bs' = size-td t by (rule lbs')
  next
    fix i
    assume is-padding: is-padding-byte (export-uinfo t) i
    from is-padding have i-t: i < size-td t
      by (simp add: is-padding-byte-def)
    show bs ! i = bs' ! i
    proof (cases i < n)
      case True
        with pfx-eq show ?thesis by simp
      next
        case False
          hence i-lower: n ≤ i by simp
          show ?thesis
          proof (cases i < n + size-td s)
            case True
              from field-lookup-is-padding-byte-focus [OF fl i-lower True wf wf-sz,
simplified]
                eq-padding is-padding True i-lower l-take l-take'
              show ?thesis
              by (auto simp add: eq-padding-def add.commute is-padding-byte-def)
            next
              case False
                with fl i-t have i-t: i < size-td t by simp
                with sfx-eq i-lower False show ?thesis by simp
          qed
        qed
      qed
    qed
  qed

```

```

lemma field-lookup-eq-padding-super-update-bs:
  assumes fl: field-lookup t f 0 = Some (s, n)
  assumes lbs: length xbs = size-td t

```

```

assumes lbs: length bs = size-td s
assumes lbs': length bs' = size-td s
assumes wf: wf-desc t
assumes wf-sz: wf-size-desc t
shows eq-padding (export-uinfo t) (super-update-bs bs xbs n) (super-update-bs bs'
xbs n) =
  eq-padding (export-uinfo s) bs bs'
proof -
  from lbs lbs' fl
  have pfx-eq:  $\bigwedge i. i < n \implies (\text{super-update-bs } bs \ xbs \ n) ! i = (\text{super-update-bs } bs' \ xbs \ n) ! i$ 
  by (metis (no-types, opaque-lifting) add-leD2 assms(2) field-lookup-offset-size'
length-take less-le-trans min-def nth-append super-update-bs-def)
  from lbs lbs' fl
  have sfx-eq:  $\bigwedge i. n + \text{size-td } s \leq i \implies i < \text{size-td } t \implies (\text{super-update-bs } bs \ xbs \ n) ! i = (\text{super-update-bs } bs' \ xbs \ n) ! i$ 
  proof -
  fix i :: nat
  assume a1:  $n + \text{size-td } s \leq i$ 
  have n ≤ size-td t
  by (meson add-leD2 field-lookup-offset-size' fl)
  then show super-update-bs bs xbs n ! i = super-update-bs bs' xbs n ! i
  using a1 lxs lbs lbs' by (smt (verit) Many-More.nat-min-simps(1) ap-
pend.assoc
  length-append length-take not-less nth-append super-update-bs-def)
  qed
from lxs lbs
have lsuper: length (super-update-bs bs xbs n) = size-td t
  by (metis add.commute field-lookup-offset-size' fl length-super-update-bs)
from lxs lbs'
have lsuper': length (super-update-bs bs' xbs n) = size-td t
  by (metis add.commute field-lookup-offset-size' fl length-super-update-bs)
from lxs lbs lsuper
have bs: (take (size-td s) (drop n (super-update-bs bs xbs n))) = bs
  by (metis (no-types, lifting) append-take-drop-id append-eq-conv-conj
append-take-drop-id length-take super-update-bs-def)
from lxs lbs' lsuper'
have bs': (take (size-td s) (drop n (super-update-bs bs' xbs n))) = bs'
  by (metis (no-types, lifting) append-take-drop-id append-eq-conv-conj
append-take-drop-id length-take super-update-bs-def)
from field-lookup-eq-padding-focus-eq [OF fl lsuper lsuper' pfx-eq sfx-eq wf wf-sz]
show ?thesis
  by (simp add: bs bs')
qed

```

```

lemma field-lookup-eq-upto-padding-focus:
assumes fl: field-lookup t f 0 = Some (s, n)
assumes lbs: length bs = size-td t

```



```

assumes lbs': length bs' = size-td t
assumes wf: wf-desc t
assumes wf-sz: wf-size-desc t
assumes eq-upto-padding: eq-upto-padding (export-uinfo t) bs bs'
shows eq-upto-padding (export-uinfo s) (take (size-td s) (drop n bs)) (take (size-td
s) (drop n bs'))
proof -
  from fl lbs have bound-s: size-td s ≤ length bs - n
  by (metis add.commute add-diff-cancel-left' diff-le-mono field-lookup-offset-size')
  from fl lbs' have bound-s': size-td s ≤ length bs' - n
  by (metis add.commute add-diff-cancel-left' diff-le-mono field-lookup-offset-size')
  from bound-s lbs
  have l-take: length (take (size-td s) (drop n bs)) = size-td s
  by simp
  from bound-s' lbs'
  have l-take': length (take (size-td s) (drop n bs')) = size-td s
  by simp

show ?thesis
  apply (simp add: eq-upto-padding-def, safe)
proof -
  show size-td s ≤ length bs - n by (rule bound-s)
next
  show size-td s ≤ length bs' - n by (rule bound-s')
next
  fix i assume is-value: is-value-byte (export-uinfo s) i
  from is-value have i-s: i < size-td s
  by (simp add: is-value-byte-def)
  have n-i: n ≤ i + n by simp
  from i-s have i-n-bound: i + n < n + size-td s by simp
  show take (size-td s) (drop n bs) ! i =
    take (size-td s) (drop n bs') ! i

  using l-take l-take' is-value eq-upto-padding lbs lbs'
    field-lookup-is-value-byte-focus [OF fl n-i i-n-bound wf wf-sz, simplified]
  by (auto simp add: eq-upto-padding-def add.commute is-value-byte-def)
qed
qed

```

```

lemma field-lookup-eq-upto-padding-focus-eq:
assumes fl: field-lookup t f 0 = Some (s, n)
assumes lbs: length bs = size-td t
assumes lbs': length bs' = size-td t
assumes pfx-eq:  $\bigwedge i. i < n \implies bs ! i = bs' ! i$ 
assumes sfx-eq:  $\bigwedge i. n + size-td s \leq i \implies i < size-td t \implies bs ! i = bs' ! i$ 
assumes wf: wf-desc t
assumes wf-sz: wf-size-desc t
shows eq-upto-padding (export-uinfo t) bs bs' =

```

```

    eq-upto-padding (export-uinfo s) (take (size-td s) (drop n bs)) (take (size-td s)
(drop n bs'))
proof -
  from fl lbs have bound-s: size-td s ≤ length bs - n
  by (metis add.commute add-diff-cancel-left' diff-le-mono field-lookup-offset-size')
  from fl lbs' have bound-s': size-td s ≤ length bs' - n
  by (metis add.commute add-diff-cancel-left' diff-le-mono field-lookup-offset-size')
  from bound-s lbs
  have l-take: length (take (size-td s) (drop n bs)) = size-td s
  by simp
  from bound-s' lbs'
  have l-take': length (take (size-td s) (drop n bs')) = size-td s
  by simp

show ?thesis
proof
  assume eq-upto-padding: eq-upto-padding (export-uinfo t) bs bs'
  show eq-upto-padding (export-uinfo s) (take (size-td s) (drop n bs)) (take (size-td
s) (drop n bs'))
  by (rule field-lookup-eq-upto-padding-focus [OF fl lbs lbs' wf wf-sz eq-upto-padding])
  next
  assume eq-upto-padding: eq-upto-padding (export-uinfo s) (take (size-td s) (drop
n bs)) (take (size-td s) (drop n bs'))
  show eq-upto-padding (export-uinfo t) bs bs'
  apply (simp add: eq-upto-padding-def, safe)
  proof -
    show length bs = size-td t by (rule lbs)
  next
    show length bs' = size-td t by (rule lbs')
  next
  fix i
  assume is-value: is-value-byte (export-uinfo t) i
  from is-value have i-t: i < size-td t
  by (simp add: is-value-byte-def)
  show bs ! i = bs' ! i
  proof (cases i < n)
    case True
      with pfx-eq show ?thesis by simp
    next
      case False
        hence i-lower: n ≤ i by simp
        show ?thesis
        proof (cases i < n + size-td s)
          case True
            from field-lookup-is-value-byte-focus [OF fl i-lower True wf wf-sz, simplified]
            eq-upto-padding is-value True i-lower l-take l-take'
            show ?thesis
            by (auto simp add: eq-upto-padding-def add.commute )
          next

```

```

      case False
      with fl i-t have i-t: i < size-td t by simp
      with sfx-eq i-lower False show ?thesis by simp
    qed
  qed
  qed
  qed
  qed

```

lemma *field-lookup-eq-upto-padding-super-update-bs:*

```

  assumes fl: field-lookup t f 0 = Some (s, n)
  assumes lxs: length xbs = size-td t
  assumes lbs: length bs = size-td s
  assumes lbs': length bs' = size-td s
  assumes wf: wf-desc t
  assumes wf-sz: wf-size-desc t
  shows eq-upto-padding (export-uinfo t) (super-update-bs bs xbs n) (super-update-bs
    bs' xbs n) =
    eq-upto-padding (export-uinfo s) bs bs'
proof -
  from lbs lbs' fl
  have pfx-eq:  $\bigwedge i. i < n \implies (super-update-bs bs xbs n) ! i = (super-update-bs bs'
    xbs n) ! i$ 
    by (metis (no-types, opaque-lifting) add-leD2 assms(2) field-lookup-offset-size'
      length-take less-le-trans min-def nth-append super-update-bs-def)
  from lbs lbs' fl
  have sfx-eq:  $\bigwedge i. n + size-td s \leq i \implies i < size-td t \implies (super-update-bs bs xbs
    n) ! i = (super-update-bs bs' xbs n) ! i$ 
proof -
  fix i :: nat
  assume a1:  $n + size-td s \leq i$ 
  have n  $\leq$  size-td t
    by (meson add-leD2 field-lookup-offset-size' fl)
  then show super-update-bs bs xbs n ! i = super-update-bs bs' xbs n ! i
    using a1 lxs lbs lbs' by (smt (verit) Many-More.nat-min-simps(1) ap-
      pend.assoc
        length-append length-take not-less nth-append super-update-bs-def)
  qed
  from lxs lbs
  have lsuper: length (super-update-bs bs xbs n) = size-td t
    by (metis add.commute field-lookup-offset-size' fl length-super-update-bs)
  from lxs lbs'
  have lsuper': length (super-update-bs bs' xbs n) = size-td t
    by (metis add.commute field-lookup-offset-size' fl length-super-update-bs)
  from lxs lbs lsuper
  have bs: (take (size-td s) (drop n (super-update-bs bs xbs n))) = bs
    by (metis (no-types, lifting) append-take-drop-id append-eq-conv-conj
      append-take-drop-id length-take super-update-bs-def)
  from lxs lbs' lsuper'

```

have bs' : (take (size-td s) (drop n (super-update-bs bs' xbs n))) = bs'
by (metis (no-types, lifting) append-take-drop-id append-eq-conv-conj
append-take-drop-id length-take super-update-bs-def)
from field-lookup-eq-upto-padding-focus-eq [OF fl lsuper lsuper' pfx-eq sfx-eq wf
wf-sz]
show ?thesis
by (simp add: bs bs')
qed

lemma field-lookup-wf-size-desc-pres:

fixes $t::('a, 'b)$ typ-desc
and $st::('a, 'b)$ typ-struct
and $ts::('a, 'b)$ typ-tuple list
and $x::('a, 'b)$ typ-tuple

shows

$wf\text{-size-desc } t \implies \text{field-lookup } t f n = \text{Some } (s, m) \implies wf\text{-size-desc } s$
 $wf\text{-size-desc-struct } st \implies \text{field-lookup-struct } st f n = \text{Some } (s, m) \implies wf\text{-size-desc } s$
 $wf\text{-size-desc-list } ts \implies \text{field-lookup-list } ts f n = \text{Some } (s, m) \implies wf\text{-size-desc } s$
 $wf\text{-size-desc-tuple } x \implies \text{field-lookup-tuple } x f n = \text{Some } (s, m) \implies wf\text{-size-desc } s$
by (induct t **and** st **and** ts **and** x arbitrary: n s m f **and** n s m f **and** n s m f
and n s m f)
(auto split: if-split-asm option.splits)

lemma field-lookup-update-ti-super-update-bs-conv:

fixes $t::('a, 'b)$ typ-info
and $st::('a, 'b)$ typ-info-struct
and $ts::('a, 'b)$ typ-info-tuple list
and $x::('a, 'b)$ typ-info-tuple

shows

$\llbracket \text{field-lookup } t f m = \text{Some } (s, m + n); \text{ length } bs = \text{size-td } s; \text{ length } xbs = \text{size-td } t;$
 $wf\text{-fd } t; wf\text{-desc } t; wf\text{-size-desc } t \rrbracket \implies$
 $\text{update-ti } s bs v =$
 $\text{update-ti } t (\text{super-update-bs } bs (\text{access-ti } t v xbs) n) v$

$\llbracket \text{field-lookup-struct } st f m = \text{Some } (s, m + n); \text{ length } bs = \text{size-td } s; \text{ length } xbs = \text{size-td-struct } st;$
 $wf\text{-fd-struct } st; wf\text{-desc-struct } st; wf\text{-size-desc-struct } st \rrbracket \implies$
 $\text{update-ti } s bs v =$
 $\text{update-ti-struct } st (\text{super-update-bs } bs (\text{access-ti-struct } st v xbs) n) v$

$\llbracket \text{field-lookup-list } ts f m = \text{Some } (s, m + n); \text{ length } bs = \text{size-td } s; \text{ length } xbs = \text{size-td-list } ts;$
 $wf\text{-fd-list } ts; wf\text{-desc-list } ts; wf\text{-size-desc-list } ts \rrbracket \implies$
 $\text{update-ti } s bs v =$
 $\text{update-ti-list } ts (\text{super-update-bs } bs (\text{access-ti-list } ts v xbs) n) v$

```

[[field-lookup-tuple x f m = Some (s, m + n); length bs = size-td s; length xbs =
size-td-tuple x;
 wf-fd-tuple x; wf-desc-tuple x; wf-size-desc-tuple x]] ==>
update-ti s bs v =
  update-ti-tuple x (super-update-bs bs (access-ti-tuple x v xbs) n) v
proof (induct t and st and ts and x arbitrary: f m n xbs v and f m n xbs v and
f m n xbs v and f m n xbs v)
case (TypDesc algn st nm)
  then show ?case
    by (auto split: if-split-asm)
      (metis TypDesc.prem3(3) TypDesc.prem4(4) access-ti.simps length-fa-ti su-
per-update-bs-same-length)
  next
    case (TypScalar algn st d)
      then show ?case by auto
  next
    case (TypAggregate ts)
      then show ?case by auto
  next
    case Nil-typ-desc
then show ?case by auto
next
  case (Cons-typ-desc x fs)
obtain d nm y where x: x = DTuple d nm y by (cases x) auto
from Cons-typ-desc.prem3 obtain
  lxs: length xbs = size-td-tuple x + size-td-list fs and
  lbs: length bs = size-td s and
  wf-fd-x: wf-fd-tuple (DTuple d nm y) and
  wf-desc-x: wf-desc-tuple (DTuple d nm y) and
  wf-fd-fs: wf-fd-list fs and
  wf-size-desc-x: wf-size-desc-tuple (DTuple d nm y) and
  wf-size-desc-fs: wf-size-desc-list fs and
  nm-notin: nm ∉ dt-snd ' set fs and
  wf-desc-fs: wf-desc-list fs and
  commutes: fu-commutes (update-ti-t d) (update-ti-list-t fs)
by (auto simp add: x)

from wf-fd-cons-listD [OF wf-fd-fs]
have fd-cons-fs: fd-cons-list fs .

from wf-fd-x have fd-cons-d: fd-cons d by (simp add: x wf-fd-consD)

from lxs
have lxs-drop: length (drop (size-td-tuple x) xbs) = size-td-list fs
  by simp

from lxs
have lxs-take: length (take (size-td-tuple x) xbs) = size-td-tuple (DTuple d nm
y)

```

```

    by (simp add: x)

from lbs
have ldrop-xbs: length (drop (size-td d) xbs) = size-td-list fs
  by (simp add: x)

from lbs wf-fd-x
have eq1: length (access-ti d v (take (size-td d) xbs)) = size-td d
  by (metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps x)

from Cons-typ-desc.prem obtain f1 fxs
  where f: f = f1 # fxs
  by (cases f) auto

show ?case
proof (cases f1 = nm)
  case True
  show ?thesis
  proof (cases field-lookup d fxs m)
    case None
    from Cons-typ-desc.prem field-lookup-list-None [OF nm-notin]
    have False
      by (simp add: True None f x)
    thus ?thesis by simp
  next
  case (Some r)
  from Cons-typ-desc.prem have r: r = (s, m + n)
    by (simp add: x True Some f)
  hence fl: field-lookup-tuple (DTuple d nm y) f m = Some (s, m + n)
    by (simp add: f True Some r)

  from td-set-wf-size-desc(4)[OF wf-size-desc-x td-set-tuple-field-lookup-tupleD,
OF fl]
  have wf-size-desc s .
  from wf-size-desc-gt(1)[OF this]
  have 0 < size-td s .

  with td-set-tuple-offset-size-m [OF td-set-tuple-field-lookup-tupleD, OF fl]
  have n-le: n < size-td d
    by auto

  have bound: size-td s + (m + n - m) ≤ size-td-tuple (DTuple d nm y) by
fact
  have bound1: n + length bs ≤ size-td d
    by (metis add.commute bound diff-add-inverse lbs size-td-tuple.simps)

```

```

from fd-cons-fs ldrop-xbs
have upd-id: (update-ti-list fs (access-ti-list fs v (drop (size-td d) xbs)) v) = v
apply (clarsimp simp add: fd-cons-list-def fd-cons-desc-def update-ti-list-t-def)
  apply (simp add: fd-cons-update-access-def fd-cons-length-def)
done

  note hyp = Cons-typ-desc.hyps(1)[simplified x, OF fl lbs lxs-take wf-fd-x
wf-desc-x wf-size-desc-x, where v = v]
  show ?thesis apply (simp add: x True r hyp)
  by (simp add: super-update-bs-append-drop-first eq1 bound1 super-update-bs-append-take-first
upd-id)
  qed
next
case False
with Cons-typ-desc.prems
have fl1: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, m + n)
  by (clarsimp simp add: x f False)
hence n-bound: size-td d ≤ n
  by (meson field-lookup-offset-le(3) nat-add-left-cancel-le)

from fl1
have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, (m + size-td
d) + (n - size-td d))
  by (metis Nat.diff-cancel field-lookup-list-offset2 field-lookup-list-offsetD)
from n-bound
have take-eq: (take (size-td s) (drop (n - size-td d + size-td d) xbs)) =
  (take (size-td s) (drop n xbs))
  by simp

from fd-cons-d lxs-take eq1
have upd-d-id: update-ti d (access-ti d v (take (size-td d) xbs)) v = v
apply (clarsimp simp add: fd-cons-def fd-cons-desc-def update-ti-t-def)
apply (simp add: fd-cons-update-access-def x)
apply (erule allE [where x = v])
apply (erule allE [where x = (take (size-td d) xbs)])
apply (simp add: eq1)
done

from ldrop-xbs fd-cons-fs
have lacc-fs: length (access-ti-list fs v (drop (size-td d) xbs)) = size-td-list fs
  by (simp add: fd-cons-list-def fd-cons-desc-def fd-cons-length-def)

from td-set-field-lookup-rev''(3) [rule-format] fl
have (s, (m + size-td d) + (n - size-td d)) ∈ td-set-list fs (m + size-td d)
  by blast
from td-set-offset-size''(3)[rule-format, OF this, simplified]
have bound: size-td s + (n - size-td d) ≤ size-td-list fs.

from lacc-fs n-bound bound lbs

```

have *lsuper*: *length (super-update-bs bs (access-ti-list fs v (drop (size-td d) xbs)) (n - size-td d)) = size-td-list fs*
by (*simp add: super-update-bs-length*)

from commutes have
commute: update-ti d (access-ti d v (take (size-td d) xbs))
(update-ti-list fs (super-update-bs bs (access-ti-list fs v (drop (size-td d) xbs)) (n - size-td d)) v) =
update-ti-list fs (super-update-bs bs (access-ti-list fs v (drop (size-td d) xbs)) (n - size-td d))
(update-ti d (access-ti d v (take (size-td d) xbs)) v)
apply (*simp add: fu-commutes-def*)
apply (*erule allE [where x = v]*)
apply (*erule allE [where x = access-ti d v (take (size-td d) xbs)]*)
apply (*erule allE [where x = super-update-bs bs (access-ti-list fs v (drop (size-td d) xbs)) (n - size-td d)]*)
apply (*simp add: update-ti-list-t-def update-ti-t-def eq1 lsuper*)
done

note *hyp = Cons-typ-desc.hyps(2)[simplified x, OF fl lbs lxs-drop wf-fd-fs wf-desc-fs wf-size-desc-fs, simplified x, simplified]*

show *?thesis*

apply (*simp add: x f eq1*)

apply (*simp add: super-update-bs-append-drop-second eq1 n-bound super-update-bs-append-take-second*)

apply (*simp add: commute*)

apply (*simp add: hyp*)

apply (*simp add: upd-d-id*)

done

qed

next

case (*DTuple-typ-desc d nm y*)

then show *?case by (auto split: if-split-asm)*

qed

lemma *access-ti-super-update-bs-of-wf:*

fixes *t::('a, 'b) typ-info*

and *st::('a, 'b) typ-info-struct*

and *ts::('a, 'b) typ-info-tuple list*

and *x::('a, 'b) typ-info-tuple*

shows

$\llbracket \text{field-lookup } t \text{ f } m = \text{Some } (s, m + n); \text{length } bs' = \text{size-td } s; \text{length } bs = \text{size-td } t; \text{wf-fd } t; \text{wf-desc } t; \text{wf-size-desc } t \rrbracket \implies$

$\text{access-ti } t \text{ (update-ti } s \text{ (access-ti}_0 \text{ s } w) \text{ v) (super-update-bs } bs' \text{ bs } n) =$
 $\text{super-update-bs (access-ti } s \text{ w } bs') \text{ (access-ti } t \text{ v } bs) \text{ n}$

$\llbracket \text{field-lookup-struct } st \text{ f } m = \text{Some } (s, m + n); \text{length } bs' = \text{size-td } s; \text{length } bs = \text{size-td-struct } st; \rrbracket$

$\llbracket \text{wf-fd-struct } st; \text{wf-desc-struct } st; \text{wf-size-desc-struct } st \rrbracket \implies$
 $\text{access-ti-struct } st \text{ (update-ti } s \text{ (access-ti}_0 \text{ } s \text{ } w) \text{ } v) \text{ (super-update-bs } bs' \text{ } bs \text{ } n) =$
 $\text{super-update-bs (access-ti } s \text{ } w \text{ } bs') \text{ (access-ti-struct } st \text{ } v \text{ } bs) \text{ } n$

$\llbracket \text{field-lookup-list } ts \text{ } f \text{ } m = \text{Some } (s, m + n); \text{length } bs' = \text{size-td } s; \text{length } bs =$
 $\text{size-td-list } ts;$
 $\text{wf-fd-list } ts; \text{wf-desc-list } ts; \text{wf-size-desc-list } ts \rrbracket \implies$
 $\text{access-ti-list } ts \text{ (update-ti } s \text{ (access-ti}_0 \text{ } s \text{ } w) \text{ } v) \text{ (super-update-bs } bs' \text{ } bs \text{ } n) =$
 $\text{super-update-bs (access-ti } s \text{ } w \text{ } bs') \text{ (access-ti-list } ts \text{ } v \text{ } bs) \text{ } n$

$\llbracket \text{field-lookup-tuple } x \text{ } f \text{ } m = \text{Some } (s, m + n); \text{length } bs' = \text{size-td } s; \text{length } bs =$
 $\text{size-td-tuple } x;$
 $\text{wf-fd-tuple } x; \text{wf-desc-tuple } x; \text{wf-size-desc-tuple } x \rrbracket \implies$
 $\text{access-ti-tuple } x \text{ (update-ti } s \text{ (access-ti}_0 \text{ } s \text{ } w) \text{ } v) \text{ (super-update-bs } bs' \text{ } bs \text{ } n) =$
 $\text{super-update-bs (access-ti } s \text{ } w \text{ } bs') \text{ (access-ti-tuple } x \text{ } v \text{ } bs) \text{ } n$

proof (induct t and st and ts and x arbitrary: $f \text{ } m \text{ } n \text{ } bs$ and $f \text{ } m \text{ } n \text{ } bs$ and $f \text{ } m$
 $n \text{ } bs$ and $f \text{ } m \text{ } n \text{ } bs$)
case ($\text{TypDesc } \text{algn } st \text{ } nm$)
note $fd = \text{wf-fd-field-lookupD}[OF \text{TypDesc}(2, 5), \text{THEN } \text{wf-fd-consD}]$
have $\text{length } bs = \text{size-td } s \implies$
 $\text{access-ti } s \text{ (update-ti } s \text{ (access-ti}_0 \text{ } s \text{ } w) \text{ } v) \text{ } bs = \text{access-ti } s \text{ } w \text{ } bs$ **for** bs
using fd
unfolding $fd\text{-cons-def } fd\text{-cons-desc-def } fd\text{-cons-access-update-def } fd\text{-cons-update-access-def}$
by $\text{simp (metis access-ti}_0 \text{ } fd \text{ } fd\text{-cons-length-p length-replicate update-ti-update-ti-t)}$
then show $?case$
using $\text{TypDesc}(2-)\text{ length-fa-ti}[OF \langle \text{wf-fd } (\text{TypDesc } \text{algn } st \text{ } nm) \rangle]$
by (auto split: if-splits
 $\text{simp: super-update-bs-same-length TypDesc}(1))$

next
case ($\text{TypScalar } sz \text{ } \text{algn } d$)
then show $?case$ **by** auto

next
case ($\text{TypAggregate } ts$)
then show $?case$ **by** auto

next
case Nil-typ-desc
then show $?case$ **by** auto

next
case ($\text{Cons-typ-desc } x \text{ } fs$)
obtain $d \text{ } nm \text{ } y$ **where** $x = \text{DTuple } d \text{ } nm \text{ } y$ **by** (cases x) auto
from $\text{Cons-typ-desc.premis}$ **have**
 $lbs: \text{length } bs = \text{size-td-tuple } x + \text{size-td-list } fs$ **and**
 $\text{wf-fd-x: wf-fd-tuple } (\text{DTuple } d \text{ } nm \text{ } y)$ **and**
 $\text{wf-desc-x: wf-desc-tuple } (\text{DTuple } d \text{ } nm \text{ } y)$ **and**
 $\text{wf-fd-fs: wf-fd-list } fs$ **and**
 $\text{wf-size-desc-x: wf-size-desc-tuple } (\text{DTuple } d \text{ } nm \text{ } y)$ **and**
 $\text{wf-size-desc-fs: wf-size-desc-list } fs$ **and**
 $nm\text{-notin: } nm \notin \text{dt-snd ' set } fs$ **and**
 $\text{wf-desc-fs: wf-desc-list } fs$ **and**

```

    bs': length bs' = size-td s
  by (auto simp add: x)

{ fix bs bs' :: byte list and v
  assume bs: length bs = size-td d and bs': length bs' = size-td-list fs
  with ⟨wf-fd-list (x # fs)⟩ have x-fs:
    access-ti-list fs (update-ti-tuple-t x bs v) bs' = access-ti-list fs v bs'
  by (simp add: fa-fu-ind-def x)
  then have access-ti-list fs (update-ti d bs v) bs' = access-ti-list fs v bs'
  by (simp add: update-ti-tuple-t-def bs x) }
note fs-x = this

from lbs
have lbs-drop: length (drop (size-td-tuple x) bs) = size-td-list fs
  by simp

from lbs
have lbs-take: length (take (size-td-tuple x) bs) = size-td-tuple (DTuple d nm y)
  by (simp add: x)

from lbs wf-fd-x
have eq1: length (access-ti d v (take (size-td d) bs)) = size-td d
  by (metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps x)

from Cons-typ-desc.prems obtain f1 fxs
  where f: f = f1 # fxs
  by (cases f) auto

have wf-s: wf-fd s
  using Cons-typ-desc.prems(1) Cons-typ-desc.prems(4) wf-fd-field-lookup(3) by
blast

let ?v = update-ti s (access-ti0 s w) v

have len-acc-w[simp]: length (access-ti0 s w) = size-td s
  using length-fa-ti[OF wf-s] by (simp add: access-ti0-def)

show ?case
proof (cases f1 = nm)
case True
  show ?thesis
  proof (cases field-lookup d fxs m)
  case None
    from Cons-typ-desc.prems field-lookup-list-None [OF nm-notin]
    have False
      by (simp add: True None f x)
    thus ?thesis by simp
  case next

```

```

case (Some r)
from Cons-typ-desc.prems have r: r = (s, m + n)
  by (simp add: x True Some f)
hence fl: field-lookup-tuple (DTuple d nm y) f m = Some (s, m + n)
  by (simp add: f True Some r)
from td-set-wf-size-desc(4)[OF wf-size-desc-x td-set-tuple-field-lookup-tupleD,
OF fl]
  have wf-size-desc s .
  from wf-size-desc-gt(1)[OF this]
  have 0 < size-td s .

with td-set-tuple-offset-size-m [OF td-set-tuple-field-lookup-tupleD, OF fl]
have n-le: n < size-td d
  by auto

have bound: size-td s + (m + n - m) ≤ size-td-tuple (DTuple d nm y) by
fact
from bound
have take-eq: (take (size-td s) (drop n (take (size-td d) bs))) =
  (take (size-td s) (drop n bs))
  by (simp add: take-drop)

have take-eq1: take (size-td s) (drop n (take (size-td d) bs)) = take (size-td
s) (drop n bs)
  using take-eq by blast

have l-take-s: length (take (size-td s) (drop n bs)) = size-td s
  using bound lbs-take by auto

have sz-s: size-td s ≤ length bs - n
  using l-take-s by auto

from fl
have fd-cons-s: fd-cons s
  using wf-fd-consD wf-fd-field-lookup(4) wf-fd-x by blast

have sz: size-td s + n ≤ size-td d length bs = size-td d + size-td-list fs
  using lbs x bound by auto
with bs' have super-update-bs-simps[simp]:
  take (size-td d) (super-update-bs bs' bs n) =
  super-update-bs bs' (take (size-td d) bs) n
  drop (size-td d) (super-update-bs bs' bs n) = drop (size-td d) bs
  by (auto simp: super-update-bs-def take-drop)

have hyp:
  access-ti d (update-ti s (access-ti0 s w) v)
  (super-update-bs bs' (take (size-td d) bs) n) =
  super-update-bs (access-ti s w bs') (access-ti d v (take (size-td d) bs)) n
  using Cons-typ-desc.hyps(1) fl lbs-take wf-fd-x wf-desc-x wf-size-desc-x bs'

```

```

    by (auto simp: x)

  have length-take-bs: length (take (size-td d) bs) = size-td d
    using sz by auto

  have fl-d-s: field-lookup d fxs m = Some (s, m + n)
    using fl by (simp add: True f)

  have wf-d: wf-fd d wf-desc d wf-size-desc d
    using wf-fd-x wf-desc-x wf-size-desc-x by auto
  note length-fa-ti[OF wf-d(1), simp]

  show ?thesis using lbs bound fd-cons-length[OF ⟨fd-cons s⟩ bs']
    apply (simp add: x True Some r eq1 hyp bs' super-update-bs-append1)
  apply (subst field-lookup-update-ti-super-update-bs-conv(1)[OF fl-d-s len-acc-w
length-take-bs wf-d])
    apply (subst fs-x)
    apply simp-all
  done
qed
next
case False
with Cons-typ-desc.prem
have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, m + n)
  by (clarsimp simp add: x f False)
hence n-bound: size-td d ≤ n
  by (meson field-lookup-offset-le(3) nat-add-left-cancel-le)

have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) =
  Some (s, (m + size-td d) + (n - size-td d))
  by (metis Nat.diff-cancel field-lookup-list-offset2 field-lookup-list-offsetD fl)

have hyp:
  access-ti-list fs (update-ti s (access-ti0 s w) v)
    (super-update-bs bs' (drop (size-td d) bs) (n - size-td d)) =
  super-update-bs (access-ti s w bs')
    (access-ti-list fs v (drop (size-td d) bs)) (n - size-td d)
  using Cons-typ-desc.hyps(2) fl bs' lbs-drop wf-fd-fs wf-desc-fs wf-size-desc-fs
  by (simp add: x)

have super-update-bs-simps[simp]:
  take (size-td d) (super-update-bs bs' bs n) = take (size-td d) bs
  drop (size-td d) (super-update-bs bs' bs n) =
  super-update-bs bs' (drop (size-td d) bs) (n - size-td d)
  using n-bound lbs x by (simp-all add: super-update-bs-def drop-take)

have fl-x: field-lookup-list (x # fs) [nm] 0 = Some (d, 0)
  by (simp add: x)
have disj: disj-fn [nm] f

```

```

using False f by (auto simp: disj-fn-def)

from fa-fu-lookup-disj(3)[rule-format, OF fl-x Cons-typ-desc(3) disj ⟨wf-fd-list
(x # fs)]
have acc-upd: length bs = size-td s  $\implies$  length bs' = size-td d  $\implies$ 
access-ti d (update-ti s bs v) bs' = access-ti d v bs' for bs bs' v
by (simp add: fa-fu-ind-def update-ti-t-def)

from False n-bound lbs
show ?thesis
by (simp add: x f hyp eq1 nth-append super-update-bs-append2 acc-upd)
qed
next
case (DTuple-typ-desc d nm y)
then show ?case apply (cases f) by (auto split: if-split-asm)
qed

lemma heap-update-list-same:
assumes p-no-overflow: unat p + length bs  $\leq$  addr-card
assumes same-px:  $\bigwedge i. i < \text{length } bs \implies bs ! i = hp (p + \text{word-of-nat } i)$ 
shows heap-update-list p bs hp = hp
using p-no-overflow same-px
proof (induct bs arbitrary: p hp)
case Nil
then show ?case by simp
next
case (Cons b bs)
from Cons.prem obtain
no-overflow: Suc (unat p + length bs)  $\leq$  addr-card and
same: ( $\bigwedge i. i < \text{Suc } (\text{length } bs) \implies (b \# bs) ! i = hp (p + \text{word-of-nat } i)$ )
by clarsimp

from no-overflow have no-overflow': unat (p + 1) + length bs  $\leq$  addr-card
by (metis (no-types, opaque-lifting) Suc-leD
add.commute add-Suc-right add-leD1 comm-monoid-add-class.add-0 unatSuc
unat-0)

{
fix i
assume bound: i < length bs
from same [where i = Suc i, simplified, OF bound]
have bs ! i = hp (p + (1 + word-of-nat i)) .
then
have bs ! i = hp (p + 1 + word-of-nat i)
by (metis group-cancel.add1)
} note same' = this

from same [where i=0] have hp-eq: hp(p := b) = hp by simp

```

note $hyp = Cons.hyps$ [**where** $p=p+1$, *OF no-overflow' same'*]
then show *?case* **by** (*simp add: hp-eq*)
qed

lemma *heap-update-list-same-prefix*:

assumes *p-no-overflow*: $unat\ p + length\ pfx + length\ bs \leq addr-card$
assumes *same-pfx*: $\bigwedge i. i < length\ pfx \implies pfx\ !\ i = hp\ (p + word-of-nat\ i)$
shows $heap-update-list\ p\ (pfx\ @\ bs)\ hp = heap-update-list\ (p + word-of-nat\ (length\ pfx))\ bs\ hp$
using *p-no-overflow same-pfx*
proof (*induct pfx arbitrary: p bs hp*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x pfx*)
from *Cons.prem*s **obtain**
no-overflow: $Suc\ (unat\ p + length\ pfx + length\ bs) \leq addr-card$ **and**
same: $(\bigwedge i. i < Suc\ (length\ pfx) \implies (x\ \# \ pfx)\ !\ i = hp\ (p + word-of-nat\ i))$
by *clarsimp*

from *no-overflow* **have** *no-overflow'*: $unat\ (p + 1) + length\ pfx + length\ bs \leq addr-card$
by (*metis (no-types, opaque-lifting) Suc-leD ab-semigroup-add-class.add-ac(1) add commute add-Suc-right add-leD1 comm-monoid-add-class.add-0 unatSuc unat-0*)

{
fix *i*
assume *bound*: $i < length\ pfx$
from *same* [**where** $i = Suc\ i$, *simplified, OF bound*]
have $pfx\ !\ i = hp\ (p + (1 + word-of-nat\ i))$.
then
have $pfx\ !\ i = hp\ (p + 1 + word-of-nat\ i)$
by (*metis group-cancel.add1*)
} note *same' = this*

have *add-eq*: $p + 1 + word-of-nat\ (length\ pfx) = p + (1 + word-of-nat\ (length\ pfx))$
using *add.assoc* **by** *blast*
from *same* [**where** $i=0$] **have** *hp-eq*: $hp(p := x) = hp$ **by** *simp*
note $hyp = Cons.hyps$ [**where** $p=p+1$, *OF no-overflow' same'*]
then show *?case* **by** (*simp add: hp-eq add-eq*)
qed

lemma *heap-update-list-same-suffix*:

assumes *p-no-overflow*: $unat\ p + length\ sfx + length\ bs \leq addr-card$
assumes *same-sfx*: $\bigwedge i. i < length\ sfx \implies sfx\ !\ i = hp\ (p + word-of-nat\ (length\ bs + i))$

```

shows heap-update-list p (bs @ sfx) hp = heap-update-list p bs hp
using p-no-overflow same-sfx
proof (induct bs arbitrary: p sfx hp)
  case Nil
  then show ?case by (simp add: heap-update-list-same)
next
  case (Cons b bs)
  from Cons.prem obtain no-overflow: Suc (unat p + length sfx + length bs) ≤
  addr-card and
    same :  $\bigwedge i. i < \text{length } sfx \implies sfx ! i = hp (p + \text{word-of-nat } (Suc (\text{length } bs) + i))$ 
  by clarsimp

  from no-overflow have no-overflow': unat (p + 1) + length sfx + length bs ≤
  addr-card
  by (metis (no-types, opaque-lifting) Suc-leD ab-semigroup-add-class.add-ac(1)
    add commute add-Suc-right add-leD1 comm-monoid-add-class.add-0 unatSuc
    unat-0)

  {
    fix i
    assume i-bound: i < length sfx
    from same [OF i-bound] have prem: sfx ! i = hp (p + word-of-nat (Suc (length
  bs) + i)) .
    from i-bound no-overflow
    have Suc (i + length bs) < addr-card
      by simp
    then
    have neq: p ≠ p + 1 + word-of-nat (length bs + i)
    by (metis (no-types, opaque-lifting) ab-semigroup-add-class.add-ac(1) add commute
  add-cancel-right-right
    len-of-addr-card not-gr-zero of-nat-Suc unat-of-nat-eq unsigned-0 zero-less-Suc)

    from neq [symmetric]
    have hp-eq: (hp(p := b)) (p + 1 + word-of-nat (length bs + i)) = hp (p + 1
  + word-of-nat (length bs + i))
      by (rule fun-upd-other)
    have add-eq: (p + 1 + word-of-nat (length bs + i)) = (p + word-of-nat (Suc
  (length bs) + i))
      by simp

    have sfx ! i = (hp(p := b)) (p + 1 + word-of-nat (length bs + i))
      apply (subst hp-eq)
      apply (subst prem )
      apply (simp only: add-eq)
      done
  } note same' = this
note hyp = Cons.hyps [OF no-overflow', where hp=(hp(p := b)), OF same']
show ?case apply (simp add: hyp) by (simp add: fun-upd-def)

```

qed

lemma *heap-update-list-same-prefix-suffix*:

assumes *p-no-overflow*: $\text{unat } p + \text{length } pfx + \text{length } bs + \text{length } sfx \leq \text{addr-card}$

assumes *same-pfx*: $\bigwedge i. i < \text{length } pfx \implies pfx ! i = hp (p + \text{word-of-nat } i)$

assumes *same-sfx*: $\bigwedge i. i < \text{length } sfx \implies sfx ! i = hp (p + \text{word-of-nat } (\text{length } pfx + \text{length } bs + i))$

shows $\text{heap-update-list } p (pfx @ bs @ sfx) hp = \text{heap-update-list } (p + \text{word-of-nat } (\text{length } pfx)) bs hp$

proof –

from *p-no-overflow*

have *no-overflow-pfx*: $\text{unat } p + \text{length } pfx + \text{length } (bs @ sfx) \leq \text{addr-card}$ **by** *simp*

from *heap-update-list-same-prefix* [**where** *pfx=pfx* **and** *bs=bs@sfx*, *OF no-overflow-pfx same-pfx*]

have $\text{heap-update-list } p (pfx @ bs @ sfx) hp = \text{heap-update-list } (p + \text{word-of-nat } (\text{length } pfx)) (bs @ sfx) hp$.

also

from *p-no-overflow* **have** $\text{unat } p + \text{length } pfx \leq \text{addr-card}$

by *simp*

then

have $\text{unat } (p + \text{word-of-nat } (\text{length } pfx)) \leq \text{unat } p + \text{length } pfx$

by (*simp add: unat-le-helper*)

with *p-no-overflow*

have *no-overflow-sfx*: $\text{unat } (p + \text{word-of-nat } (\text{length } pfx)) + \text{length } sfx + \text{length } bs \leq \text{addr-card}$

by *simp*

from *same-sfx p-no-overflow* **have** *same-sfx'*: $\bigwedge i. i < \text{length } sfx \implies$

$sfx ! i = hp (p + \text{word-of-nat } (\text{length } pfx) + \text{word-of-nat } (\text{length } bs + i))$

apply *simp*

by (*simp add: ab-semigroup-add-class.add-ac(1)*)

from *heap-update-list-same-suffix* [**where** *p=p + word-of-nat (length pfx)* **and** *bs=bs* **and** *sfx=sfx* **and** *hp=hp*,
OF no-overflow-sfx same-sfx']

have $\text{heap-update-list } (p + \text{word-of-nat } (\text{length } pfx)) (bs @ sfx) hp = \text{heap-update-list } (p + \text{word-of-nat } (\text{length } pfx)) bs hp$.

finally show *?thesis* .

qed

lemma *heap-update-list-super-update-bs-heap-list*:

assumes *p-no-overflow*: $\text{unat } p + m \leq \text{addr-card}$

assumes *n-m*: $n + \text{length } bs \leq m$

shows $\text{heap-update-list } (p + \text{word-of-nat } n) bs hp = \text{heap-update-list } p (\text{super-update-bs}$


```

bs (heap-list hp m p) n) hp
proof –
  have heap-list-hp:  $\bigwedge i. i < m \implies (\text{heap-list hp m p}) ! i = \text{hp } (p + \text{word-of-nat } i)$ 
    by (rule heap-list-nth)

obtain pfx sfx where
  super: super-update-bs bs (heap-list hp m p) n = pfx @ bs @ sfx and
  pfx: pfx = take n (heap-list hp m p) and
  sfx: sfx = drop (n + length bs) (heap-list hp m p) and
  lpx: length pfx = n and
  lsfx: length sfx = m - n - length bs
  using n-m
  by (simp add: super-update-bs-def )

have bounds: unat p + length pfx + length bs + length sfx ≤ addr-card
  using lpx p-no-overflow n-m lsfx
  by simp

have same-pfx:  $\bigwedge i. i < \text{length pfx} \implies \text{pfx} ! i = \text{hp } (p + \text{word-of-nat } i)$ 
  using heap-list-hp
  by (simp add: pfx)

have same-sfx:  $\bigwedge i. i < \text{length sfx} \implies \text{sfx} ! i = \text{hp } (p + \text{word-of-nat } (\text{length pfx} + \text{length bs} + i))$ 
  using heap-list-hp
  by (simp add: sfx lpx)
from heap-update-list-same-prefix-suffix [OF bounds same-pfx same-sfx]
show ?thesis
  by (simp add: super lpx)
qed

lemma append-take-dropI:  $xs = \text{take } (\text{length } xs) \ zs \implies ys = \text{drop } (\text{length } xs) \ zs \implies xs @ ys = zs$ 
  by (metis append-take-drop-id)

lemma heap-list-heap-update-list-id:
  assumes bound:  $n \leq \text{addr-card}$ 
  assumes lbs:  $\text{length } bs = n$ 
  shows  $(\text{heap-list } (\text{heap-update-list } a \ bs \ h) \ n \ a) = bs$ 
  using bound lbs
proof (induct n arbitrary: a bs h)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then show ?case
  apply (cases bs)
  apply clarsimp

```

apply *clarsimp*
by (*metis* (*no-types*, *lifting*) *Suc-le-D Suc-le-eq add-cancel-right-right add-diff-cancel-left'*
fun-upd-same heap-update-list-value' linorder-not-less one-neq-zero plus-1-eq-Suc
unat-minus-abs unsigned-1)
qed

lemma *heap-update-list-nth-conv:*

length bs ≤ addr-card ⇒
heap-update-list a bs h a' =
(if (unat a' + (if a' < a then addr-card else 0) - unat a) < length bs then
bs ! (unat a' + (if a' < a then addr-card else 0) - unat a)
else
h a')

proof (*induct bs arbitrary: a h*)

case *Nil*

then show *?case by simp*

next

case (*Cons b bs*)

from *Cons.prem*s **have** *bound: Suc (length bs) ≤ addr-card by simp*

hence *bound'*: *length bs ≤ addr-card by simp*

note *hyp = Cons.hyps [OF this, of a+1 (h(a := b))]*

show *?case*

proof (*cases a' < a*)

case *True*

show *?thesis apply (simp add: hyp True)*

using *True*

by (*smt* (*verit*) *Nat.add-diff-assoc One-nat-def Suc-diff-Suc add-diff-cancel-left'*
cancel-comm-monoid-add-class.diff-cancel diff-add diff-diff-left
diff-right-commute fun-upd-apply le-add-diff len-of-addr-card not-less-eq
not-less-eq-eq nth-Cons-pos of-nat-add order-less-le trans-less-add2
un-ui-le unat-0 unat-sub-if' unsigned-greater-eq unsigned-less word-less-def
word-overflow-unat word-unat.Rep-inverse zero-less-diff)

next

case *False*

show *?thesis apply (simp add: hyp False)*

using *False bound'*

by (*smt* (*verit*) *Nat.add-diff-assoc One-nat-def Suc-eq-plus1 Suc-le-eq*
add-diff-cancel-left bound cancel-comm-monoid-add-class.diff-cancel
diff-zero fun-upd-def le-eq-less-or-eq len-of-addr-card linorder-not-le nth-Cons-0
nth-Cons-Suc order-less-le plus-1-eq-Suc trans-le-add2 unatSuc2 unat-1
unat-add-lem unat-mono unsigned-0 unsigned-less)

qed

qed

lemma *heap-update-list-overwrite-all-nth-conv:*

assumes *lbs: length bs = addr-card*

shows *heap-update-list a bs h a' =*

$bs ! (unat\ a' + (if\ a' < a\ then\ addr-card\ else\ 0) - unat\ a)$

proof –
from lbs **have** $leq: length\ bs \leq addr-card$ **by** $simp$
show $?thesis$
using lbs
apply ($simp\ add: heap-update-list-nth-conv\ [OF\ leq]$)
by ($metis\ len-of-addr-card\ linorder-not-less\ unat-sub-if'\ unsigned-less\ word-less-nat-alt$)
qed

lemma $heap-update-list-overwrite'$:
assumes $bound: length\ bs \leq addr-card$
assumes $len-eq: length\ bs = length\ bs'$
shows ($heap-update-list\ a\ bs\ (heap-update-list\ a\ bs'\ h)$) = ($heap-update-list\ a\ bs\ h$)
proof ($rule\ ext$)
using $bound\ len-eq$
apply ($simp\ add: heap-update-list-nth-conv$)
done

lemma $heap-list-tail-addr-card$:
assumes $len: length\ ys = addr-card$
shows $heap-update-list\ p\ (xs\ @\ ys)\ h = heap-update-list\ (p + word-of-nat\ (length\ xs))\ ys\ h$
proof –
from len **have** $len': length\ ys \leq addr-card$ **by** $simp$
show $?thesis$
apply ($simp\ add: heap-update-list-concat-unfold$)
apply ($rule\ ext$)
apply ($simp\ add: heap-update-list-nth-conv\ [OF\ len']$)
using len
by ($metis\ len-of-addr-card\ linorder-not-less\ unat-sub-if'\ unsigned-less\ word-less-nat-alt$)
qed

lemma $heap-update-list-overwrite$:
assumes $leq: length\ bs = length\ bs'$
shows $heap-update-list\ p\ bs\ (heap-update-list\ p\ bs'\ h) = heap-update-list\ p\ bs\ h$
proof ($cases\ length\ bs \leq addr-card$)
case $True$
from $heap-update-list-overwrite'\ [OF\ True\ leq]$ **show** $?thesis$ **by** $simp$
next
case $False$
from $False$ **obtain** $xs\ ys$ **where** $bs: bs=xs@ys$ **and** $lys: length\ ys = addr-card$
by ($metis\ append-take-drop-id\ diff-diff-cancel\ length-drop\ linorder-not-less\ order-less-le$)
from $False\ leq$ **obtain** $xs'\ ys'$ **where** $bs': bs'=xs'@ys'$ **and** $lys': length\ ys' = addr-card$
by ($metis\ append-take-drop-id\ diff-diff-cancel\ length-drop\ linorder-not-less\ or-$

```

der-less-le)
from lys have leq-ys: length ys ≤ addr-card by simp
from lys' have leq-ys': length ys' ≤ addr-card by simp
note eqs = heap-list-tail-addr-card [OF lys, where xs = xs]
  heap-list-tail-addr-card [OF lys', where xs = xs^]
  heap-update-list-overwrite-all-nth-conv [OF lys]
  heap-update-list-overwrite-all-nth-conv [OF lys^]

show ?thesis
  apply (rule ext)
  apply (simp add: bs bs' eqs)
done
qed

context xmem-type
begin

lemma xmem-type-is-value-byte-access-ti-update-ti-cancel:
  assumes i-bound: i < size-of TYPE('a)
  assumes lbs: length bs = size-of TYPE('a)
  assumes lbs': length bs' = size-of TYPE('a)
  assumes is-value: is-value-byte (typ-uinfo-t TYPE('a)) i
  shows access-ti (typ-info-t TYPE('a)) (update-ti (typ-info-t TYPE('a)) bs v) bs'
  ! i = bs ! i
  using is-value-byte-access-ti-update-ti-cancel [OF - - - - i-bound [simplified
size-of-def]
  lbs [simplified size-of-def] lbs' [simplified size-of-def] is-value [simplified typ-uinfo-t-def]]
  by (simp add: local.wf-component-descs local.wf-field-descs wf-fdp-fd(1) wf-lf-fdp)

lemma xmem-type-is-value-byte-access-ti-id:
  assumes i-bound: i < size-of TYPE('a)
  assumes lbs: length bs = size-of TYPE('a)
  assumes is-value: is-value-byte (typ-uinfo-t TYPE('a)) i
  shows access-ti (typ-info-t TYPE('a)) v (bs[i:=b]) = access-ti (typ-info-t TYPE('a))
v bs
  using is-value-byte-access-ti-id [OF - - - - i-bound [simplified size-of-def]
  lbs [simplified size-of-def] is-value [simplified typ-uinfo-t-def]]
  by (simp add: local.wf-component-descs local.wf-field-descs wf-fdp-fd(1) wf-lf-fdp)

lemma is-padding-byte-to-lense: is-padding-byte (typ-uinfo-t TYPE('a)) i
  ⇒ lense.is-padding-byte i
  by (metis is-padding-byte-def local.lense.complement local.lense.is-padding-byte-def
  local.lense.is-value-byte-update-depends local.xmem-type-is-padding-byte-update-ti-id
size-of-def typ-uinfo-size)

lemma is-value-byte-to-lense: is-value-byte (typ-uinfo-t TYPE('a)) i
  ⇒ lense.is-value-byte i

```

```

using xmem-type-is-padding-byte-access-ti
xmem-type-is-padding-byte-update-ti-id xmem-type-is-value-byte-access-ti-update-ti-cancel
using is-value-byte-def local.lense.is-value-byteI local.size-of-def
xmem-type-is-value-byte-access-ti-id by auto

```

```

lemma is-padding-byte-from-lense:
assumes padding: lense.is-padding-byte i
shows is-padding-byte (typ-uinfo-t TYPE('a)) i
proof –
  from padding have i-sz: i < size-of (TYPE('a))
    using local.lense.is-padding-byte-def by blast
  show ?thesis
  proof (cases is-padding-byte (typ-uinfo-t TYPE('a)) i)
    case True
    then show ?thesis by simp
  next
    case False
    with complement-tu-padding [OF i-sz [simplified size-of-def, simplified typ-uinfo-size
  [symmetric]] show ?thesis
    using i-sz is-value-byte-to-lense local.lense.complement padding by blast
  qed
qed

```

```

lemma is-padding-byte-lense-conv: is-padding-byte (typ-uinfo-t TYPE('a)) i = lense.is-padding-byte
i
using is-padding-byte-from-lense is-padding-byte-to-lense by blast

```

```

lemma is-value-byte-from-lense:
assumes is-value: lense.is-value-byte i
shows is-value-byte (typ-uinfo-t TYPE('a)) i
proof –
  from is-value have i-sz: i < size-of (TYPE('a))
    using local.lense.is-value-byte-def by blast
  show ?thesis
  proof (cases is-value-byte (typ-uinfo-t TYPE('a)) i)
    case True
    then show ?thesis by simp
  next
    case False
    with complement-tu-padding [OF i-sz [simplified size-of-def, simplified typ-uinfo-size
  [symmetric]] show ?thesis
    using assms i-sz is-padding-byte-to-lense local.lense.complement by blast
  qed
qed

```

```

lemma is-value-byte-lense-conv: is-value-byte (typ-uinfo-t TYPE('a)) i = lense.is-value-byte
i
using is-value-byte-from-lense is-value-byte-to-lense by blast

```

lemma *eq-padding-lense-conv*: *eq-padding* (*typ-uinfo-t* *TYPE('a)*) *bs* *bs'* = *lense.eq-padding* *bs* *bs'*

apply (*simp* *add*: *lense.eq-padding-is-padding-byte-conv* *eq-padding-def*)
by (*simp* *add*: *is-padding-byte-lense-conv* *size-of-def*)

lemma *eq-upto-padding-lense-conv*: *eq-upto-padding* (*typ-uinfo-t* *TYPE('a)*) *bs* *bs'*
= *lense.eq-upto-padding* *bs* *bs'*

apply (*simp* *add*: *lense.eq-upto-padding-is-value-byte-conv* *eq-upto-padding-def*)
by (*simp* *add*: *is-value-byte-lense-conv* *size-of-def*)

lemma *lense-eq-upto-padding-from-bytes-eg*:

assumes *eq-upto-padding*: *lense.eq-upto-padding* *bs* *bs'*

shows ((*from-bytes* *bs*::'*a*) = *from-bytes* *bs'*)

using *eq-upto-padding*

by (*metis* *field-desc.select-convs*(2) *field-desc-def* *from-bytes-def* *local.field-sz-size-of*
local.field-update-update-ti *local.xmem-type-wf-field-desc.eq-upto-padding-upd*
padding-base.eq-upto-padding-length-eg *update-ti-t-def*)

lemma *eq-upto-padding-from-bytes-eg*:

assumes *eq-upto-padding*: *eq-upto-padding* (*typ-uinfo-t* *TYPE('a)*) *bs* *bs'*

shows ((*from-bytes* *bs*::'*a*) = *from-bytes* *bs'*)

using *lense-eq-upto-padding-from-bytes-eg* *eq-upto-padding-lense-conv* *eq-upto-padding*
by (*simp* *add*: *typ-uinfo-t-def*)

lemma *lense-eg-padding-to-bytes-eg*:

assumes *eq-padding*: *lense.eq-padding* *bs* *bs'*

shows (*to-bytes* (*v*::'*a*) *bs*) = *to-bytes* *v* *bs'*

proof –

from *lense.eq-padding-acc* [*OF* *eq-padding*]

show *?thesis*

by (*simp* *add*: *to-bytes-def*)

qed

lemma *eq-padding-to-bytes-eg*:

assumes *eq-padding*: *eq-padding* (*typ-uinfo-t* *TYPE('a)*) *bs* *bs'*

shows (*to-bytes* (*v*::'*a*) *bs*) = *to-bytes* *v* *bs'*

using *lense-eg-padding-to-bytes-eg* *eq-padding-lense-conv* *eq-padding*

by (*simp* *add*: *typ-uinfo-t-def*)

lemma *eq-padding-to-bytes*:

length *bs* = *size-of* *TYPE('a)* \implies *eq-padding* (*typ-uinfo-t* *TYPE('a)*) (*to-bytes*
(*v*::'*a*) *bs*) *bs*

by (*simp* *add*: *eq-padding-lense-conv* *local.lense.field-access-eg-padding1* *to-bytes-def*)

lemma *lense-eg-padding-to-bytes*:

length *bs* = *size-of* *TYPE('a)* \implies *lense.eq-padding* (*to-bytes* (*v*::'*a*) *bs*) *bs*

using *eq-padding-to-bytes* *eq-padding-lense-conv*

by *simp*

lemma *heap-list-heap-update-id*:
fixes *p::'a ptr*
shows (*heap-list* (heap-update-list (ptr-val *p*) (to-bytes (u::'a) (heap-list *h* (size-of TYPE('a)) (ptr-val *p*))) *h*) (size-of TYPE('a)) (ptr-val *p*)) = (to-bytes *u* (heap-list *h* (size-of TYPE('a)) (ptr-val *p*))))

proof –
have *bound*: *size-of TYPE('a) ≤ addr-card*
using *max-size nless-le by blast*
have *lbs*: *length (to-bytes u (heap-list h (size-of TYPE('a)) (ptr-val p))) = size-of TYPE('a)*
by (*simp add: local.lense.access-result-size to-bytes-def*)
note *eq1 = heap-list-heap-update-list-id [OF bound lbs]*
show *?thesis*
by (*simp add: eq1*)

qed

lemma *heap-update-collapse*:
fixes *p::'a ptr*
shows *heap-update p v (heap-update p u h) = heap-update p v h*

proof –

have *lv*: *length (to-bytes v (to-bytes u (heap-list h (size-of TYPE('a)) (ptr-val p)))) = size-of TYPE('a)*
by (*simp add: local.lense.access-result-size to-bytes-def*)

have *lu*: *length (to-bytes u (heap-list h (size-of TYPE('a)) (ptr-val p))) = size-of TYPE('a)*
by (*simp add: local.lense.access-result-size to-bytes-def*)

have *leq*: *length (to-bytes v (to-bytes u (heap-list h (size-of TYPE('a)) (ptr-val p)))) = length (to-bytes u (heap-list h (size-of TYPE('a)) (ptr-val p)))*
by (*simp add: lv lu*)

have *eq*:
(to-bytes v (to-bytes u (heap-list h (size-of TYPE('a)) (ptr-val p)))) = (to-bytes v (heap-list h (size-of TYPE('a)) (ptr-val p)))
by (*metis eq-padding-lense-conv eq-padding-to-bytes-eq heap-list-length local.lense.field-access-eq-padding1 to-bytes-def*)

show *?thesis*
apply (*simp add: heap-update-def*)
apply (*simp add: heap-list-heap-update-id [OF]*)
apply (*subst heap-update-list-overwrite [OF leq]*)
apply (*simp add: eq*)
done

qed

lemma *heap-update-padding-collapse*:
fixes $p::'a \text{ ptr}$
assumes $lbs: \text{length } bs = \text{size-of } \text{TYPE}('a)$
assumes $lbs': \text{length } bs' = \text{size-of } \text{TYPE}('a)$
shows $\text{heap-update-padding } p \ v \ bs \ (\text{heap-update-padding } p \ u \ bs' \ h) = \text{heap-update-padding } p \ v \ bs \ h$
apply (*simp add: heap-update-padding-def*)
using $lbs \ lbs' \ \text{heap-update-list-overwrite}$
by *auto*

lemma *heap-update-padding-heap-update-collapse*:
fixes $p::'a \text{ ptr}$
assumes $lbs: \text{length } bs = \text{size-of } \text{TYPE}('a)$
shows $\text{heap-update-padding } p \ v \ bs \ (\text{heap-update } p \ u \ h) = \text{heap-update-padding } p \ v \ bs \ h$
by (*simp add: heap-update-heap-update-padding-conv heap-update-padding-collapse [OF lbs]*)

lemma *to-bytes-from-bytes-id*:
 $\text{length } bs = \text{size-of } \text{TYPE}('a) \implies \text{to-bytes } ((\text{from-bytes } bs)::'a) \ bs = bs$
by (*metis (no-types, lifting) field-desc.select-convs(2) field-desc-def from-bytes-def lense-eq-padding-to-bytes local.field-sz-size-of local.field-sz-size-td local.lense.eq-upto-paddingI local.lense.padding-eq-complement local.lense.update-access local.upd padding-base.eq-padding-length1 to-bytes-def update-ti-update-ti-t*)

lemma *to-bytes-heap-list-id*:
 $\text{to-bytes } ((\text{from-bytes } (\text{heap-list } h \ (\text{size-of } \text{TYPE}('a)) \ a))::'a)$
 $(\text{heap-list } h \ (\text{size-of } \text{TYPE}('a)) \ a) =$
 $\text{heap-list } h \ (\text{size-of } \text{TYPE}('a)) \ a$
by (*simp add: to-bytes-from-bytes-id*)

lemma *heap-update-id*:
fixes $p::'a \text{ ptr}$
shows $\text{heap-update } p \ (h\text{-val } h \ p) \ h = h$
by (*simp add: heap-update-def h-val-def to-bytes-heap-list-id heap-update-list-id'*)

context
fixes $s \ f \ n$
assumes $fl: \text{field-lookup } (\text{typ-info-t } \text{TYPE}('a)) \ f \ 0 = \text{Some } (s, n)$
begin
interpretation $flense: \text{padding-lense access-ti } s \ \text{update-ti } s \ \text{size-td } s$
using $\text{field-lookup-padding-lense [OF fl]}$.

private lemma *n-t*: $n < \text{size-of } \text{TYPE}('a)$
using fl

by (*metis add-diff-cancel-right' add-leD2 cancel-comm-monoid-add-class.diff-cancel field-lookup-offset-size'*
field-lookup-wf-size-desc-gt
local.field-sz-size-of local.field-sz-size-td local.wf-size-desc nat-less-le not-add-less2)

private lemma *wf-desc-s: wf-desc s*
using *fl field-lookup-wf-desc-pres(1) local.wf-desc* **by** *blast*

private lemma *wf-size-desc-s: wf-size-desc s*
using *fl field-lookup-wf-size-desc-pres(1) local.wf-size-desc* **by** *blast*

private lemma *wf-field-descs-s: wf-field-descs (set (field-descs s))*
using *fl local.field-lookup-wf-field-descs* **by** *blast*

private lemma *wf-component-descs-s: wf-component-descs s*
using *fl local.field-lookup-wf-component-descs* **by** *blast*

private lemma *wf-fd-s: wf-fd s*
using *fl local.wf-desc local.wf-lf wf-fd-field-lookupD wf-fdp-fd(1) wf-lf-fdp*
by *blast*

lemma *xmem-type-field-lookup-is-padding-byte-focus:*
assumes *i-lower: n ≤ i*
assumes *i-upper : i < n + size-td s*
shows *is-padding-byte (typ-uinfo-t TYPE('a)) i = is-padding-byte (export-uinfo s) (i - n)*
using *field-lookup-is-padding-byte-focus [OF fl i-lower i-upper]*
by (*simp add: typ-uinfo-t-def*)

lemma *xmem-type-field-lookup-is-padding-byte-focus-rev1:*
assumes *is-padding: is-padding-byte (export-uinfo s) i*
shows *is-padding-byte (typ-uinfo-t TYPE('a)) (i + n)*
proof –
from *is-padding* **have** *i-s: i < size-td s*
using *is-padding-byte-def* **by** *simp*
have *lower: n ≤ i + n* **by** *simp*
from *i-s* **have** *upper: i + n < n + size-td s* **by** *simp*

from *xmem-type-field-lookup-is-padding-byte-focus [OF lower upper]* *is-padding*
show *?thesis* **by** *simp*
qed

lemma *xmem-type-field-lookup-is-padding-byte-focus-rev2:*
assumes *is-padding: is-padding-byte (typ-uinfo-t TYPE('a)) (i + n)*
assumes *i-bound: i < size-td s*
shows *is-padding-byte (export-uinfo s) i*
proof –
have *lower: n ≤ i + n* **by** *simp*
from *i-bound* **have** *upper: i + n < n + size-td s* **by** *simp*

```

from xmem-type-field-lookup-is-padding-byte-focus [OF lower upper] is-padding
show ?thesis
  by simp
qed

```

lemma *xmem-type-field-lookup-lense-is-padding-byte-focus-rev1*:

```

assumes is-padding: flense.is-padding-byte i
shows lense.is-padding-byte (i + n)

```

proof –

```

from is-padding have i-s: i < size-td s
  using flense.is-padding-byte-def by simp
have lower: n ≤ i + n by simp
from i-s have upper: i + n < n + size-td s by simp

```

```

from field-lookup-is-padding-byte-inner-to-outer [OF fl lower upper] is-padding
show ?thesis by (simp add: size-of-def)

```

qed

lemma *xmem-type-field-lookup-lense-is-padding-byte-focus-rev2*:

```

assumes is-padding: lense.is-padding-byte (i + n)
assumes i-bound: i < size-td s
shows flense.is-padding-byte i

```

proof –

```

have lower: n ≤ i + n by simp
from i-bound have upper: i + n < n + size-td s by simp
from field-lookup-is-padding-byte-outer-to-inner [OF fl lower upper] is-padding
show ?thesis
  by (simp add: size-of-def)

```

qed

lemma *xmem-type-field-lookup-is-value-byte-focus*:

```

assumes i-lower: n ≤ i
assumes i-upper : i < n + size-td s
shows is-value-byte (typ-uinfo-t TYPE('a)) i = is-value-byte (export-uinfo s) (i
– n)
using field-lookup-is-value-byte-focus [OF fl i-lower i-upper]
by (simp add: typ-uinfo-t-def)

```

lemma *xmem-type-field-lookup-is-value-byte-focus-rev1*:

```

assumes is-value: is-value-byte (export-uinfo s) i
shows is-value-byte (typ-uinfo-t TYPE('a)) (i + n)

```

proof –

```

from is-value have i-s: i < size-td s
  using is-value-byte-def by simp
have lower: n ≤ i + n by simp
from i-s have upper: i + n < n + size-td s by simp

```

```

from xmem-type-field-lookup-is-value-byte-focus [OF lower upper] is-value

```

show *?thesis* **by** *simp*
qed

lemma *xmem-type-field-lookup-is-value-byte-focus-rev2*:
assumes *is-value*: *is-value-byte* (*typ-uinfo-t* *TYPE('a)*) (*i + n*)
assumes *i-bound*: *i < size-td s*
shows *is-value-byte* (*export-uinfo s*) *i*
proof –
 have *lower*: $n \leq i + n$ **by** *simp*
 from *i-bound* **have** *upper*: $i + n < n + \text{size-td } s$ **by** *simp*
 from *xmem-type-field-lookup-is-value-byte-focus* [*OF lower upper*] *is-value*
 show *?thesis*
 by *simp*
qed

lemma *xmem-type-field-lookup-lense-is-value-byte-focus-rev1*:
assumes *is-value*: *flense.is-value-byte i*
shows *lense.is-value-byte* (*i + n*)
proof –
 from *is-value* **have** *i-s*: *i < size-td s*
 using *flense.is-value-byte-def* **by** *simp*
 have *lower*: $n \leq i + n$ **by** *simp*
 from *i-s* **have** *upper*: $i + n < n + \text{size-td } s$ **by** *simp*

 from *field-lookup-is-value-byte-inner-to-outer* [*OF fl lower upper*] *is-value*
 show *?thesis* **by** (*simp add: size-of-def*)
qed

lemma *xmem-type-field-lookup-lense-is-value-byte-focus-rev2*:
assumes *is-value*: *lense.is-value-byte* (*i + n*)
assumes *i-bound*: *i < size-td s*
shows *flense.is-value-byte i*
proof –
 have *lower*: $n \leq i + n$ **by** *simp*
 from *i-bound* **have** *upper*: $i + n < n + \text{size-td } s$ **by** *simp*
 from *field-lookup-is-value-byte-outer-to-inner* [*OF fl lower upper*] *is-value*
 show *?thesis*
 by (*simp add: size-of-def*)
qed

lemma *field-lookup-is-padding-byte-access-ti*:
assumes *i-bound*: *i < size-td s*
assumes *lbs*: *length bs = size-td s*
assumes *is-padding*: *is-padding-byte* (*export-uinfo s*) *i*
shows *access-ti s v bs ! i = bs ! i*
using *is-padding-byte-access-ti* [*OF wf-desc-s wf-size-desc-s wf-field-descs-s*
 wf-component-descs-s wf-fd-s i-bound lbs is-padding] .

```

lemma field-lookup-is-padding-byte-update-ti-id:
  assumes i-bound:  $i < \text{size-td } s$ 
  assumes lbs:  $\text{length } bs = \text{size-td } s$ 
  assumes is-padding: is-padding-byte (export-uinfo s) i
  shows update-ti s (bs[i := b]) v = update-ti s bs v
  using is-padding-byte-update-ti-id [OF wf-desc-s wf-size-desc-s wf-field-descs-s
    wf-component-descs-s wf-fd-s i-bound lbs is-padding] .

lemma field-lookup-is-value-byte-access-ti-update-ti-cancel:
  assumes i-bound:  $i < \text{size-td } s$ 
  assumes lbs:  $\text{length } bs = \text{size-td } s$ 
  assumes lbs':  $\text{length } bs' = \text{size-td } s$ 
  assumes is-value: is-value-byte (export-uinfo s) i
  shows access-ti s (update-ti s bs v) bs' ! i = bs ! i
using is-value-byte-access-ti-update-ti-cancel [OF wf-desc-s wf-size-desc-s wf-field-descs-s
  wf-component-descs-s wf-fd-s i-bound lbs lbs' is-value] .

lemma field-lookup-is-value-byte-access-ti-id:
  assumes i-bound:  $i < \text{size-td } s$ 
  assumes lbs:  $\text{length } bs = \text{size-td } s$ 
  assumes is-value: is-value-byte (export-uinfo s) i
  shows access-ti s v (bs[i := b]) = access-ti s v bs
using is-value-byte-access-ti-id [OF wf-desc-s wf-size-desc-s wf-field-descs-s
  wf-component-descs-s wf-fd-s i-bound lbs is-value] .

lemma field-lookup-is-padding-byte-to-lense:
  assumes is-padding: is-padding-byte (export-uinfo s) i
  shows flense.is-padding-byte i
proof (rule flense.is-padding-byteI)
  from is-padding show  $i < \text{size-td } s$ 
    using is-padding-byte-def by simp
next
  fix v::'a and bs::byte list
  assume i-bound:  $i < \text{size-td } s$ 
  assume lbs:  $\text{length } bs = \text{size-td } s$ 
  show access-ti s v bs ! i = bs ! i
    by (rule field-lookup-is-padding-byte-access-ti [OF i-bound lbs is-padding])
next
  fix v::'a and bs::byte list and b::byte
  assume i-bound:  $i < \text{size-td } s$ 
  assume lbs:  $\text{length } bs = \text{size-td } s$ 
  show update-ti s bs v = update-ti s (bs[i := b]) v
    using field-lookup-is-padding-byte-update-ti-id [OF i-bound lbs is-padding] by
simp
qed

lemma field-lookup-is-padding-byte-from-lense:
  assumes is-padding: flense.is-padding-byte i

```

shows *is-padding-byte* (*export-uinfo s*) *i*
proof –
from *is-padding* **have** *i-bound: i < size-td s* **by** (*simp add: flense.is-padding-byte-def*)
from *xmem-type-field-lookup-lense-is-padding-byte-focus-rev1* [*OF is-padding*]
have *lense.is-padding-byte (i + n)*.
from *is-padding-byte-from-lense* [*OF this*]
have *is-padding-byte (typ-uinfo-t TYPE('a)) (i + n)*.
with *xmem-type-field-lookup-is-padding-byte-focus* [**where** *i=i + n*] *i-bound*
show *?thesis* **by** *simp*
qed

lemma *field-lookup-is-padding-byte-lense-conv*:
is-padding-byte (export-uinfo s) i \longleftrightarrow *flense.is-padding-byte i*
using *field-lookup-is-padding-byte-from-lense field-lookup-is-padding-byte-to-lense*
by *blast*

lemma *field-lookup-is-value-byte-from-lense*:
assumes *is-value: flense.is-value-byte i*
shows *is-value-byte (export-uinfo s) i*
proof –
from *is-value* **have** *i-bound: i < size-td s* **by** (*simp add: flense.is-value-byte-def*)
from *xmem-type-field-lookup-lense-is-value-byte-focus-rev1* [*OF is-value*]
have *lense.is-value-byte (i + n)*.
from *is-value-byte-from-lense* [*OF this*]
have *is-value-byte (typ-uinfo-t TYPE('a)) (i + n)*.
with *xmem-type-field-lookup-is-value-byte-focus* [**where** *i=i + n*] *i-bound*
show *?thesis* **by** *simp*
qed

lemma *field-lookup-is-value-byte-to-lense*:
assumes *is-value: is-value-byte (export-uinfo s) i*
shows *flense.is-value-byte i*
proof (*rule flense.is-value-byteI*)
from *is-value* **show** *i < size-td s*
using *is-value-byte-def* **by** *simp*
next
fix *v::'a* **and** *bs::byte list* **and** *bs'::byte list*
assume *i-bound: i < size-td s*
assume *lbs: length bs = size-td s*
assume *lbs': length bs' = size-td s*
show *access-ti s (update-ti s bs v) bs' ! i = bs ! i*
by (*rule field-lookup-is-value-byte-access-ti-update-ti-cancel* [*OF i-bound lbs lbs'*
is-value])
next
fix *v::'a* **and** *bs::byte list* **and** *b::byte*
assume *i-bound: i < size-td s*
assume *lbs: length bs = size-td s*
show *access-ti s v bs = access-ti s v (bs[i := b])*
using *field-lookup-is-value-byte-access-ti-id* [*OF i-bound lbs is-value*] **by** *simp*

qed

lemma *field-lookup-is-value-byte-lense-conv*:

is-value-byte (*export-uinfo* *s*) *i* \longleftrightarrow *flense.is-value-byte* *i*

using *field-lookup-is-value-byte-from-lense* *field-lookup-is-value-byte-to-lense*

by *blast*

lemma *field-lookup-eq-padding-lense-conv*: *eq-padding* (*export-uinfo* *s*) *bs* *bs'* \longleftrightarrow *flense.eq-padding* *bs* *bs'*

using *field-lookup-is-padding-byte-lense-conv*

by (*simp add: eq-padding-def flense.eq-padding-is-padding-byte-conv*)

lemma *field-lookup-eq-upto-padding-lense-conv*: *eq-upto-padding* (*export-uinfo* *s*) *bs* *bs'* \longleftrightarrow *flense.eq-upto-padding* *bs* *bs'*

using *field-lookup-is-value-byte-lense-conv*

by (*simp add: eq-upto-padding-def flense.eq-upto-padding-is-value-byte-conv*)

lemma *xmem-type-field-lookup-eq-padding-focus*:

assumes *lbs*: *length* *bs* = *size-of* *TYPE*('a)

assumes *lbs'*: *length* *bs'* = *size-of* *TYPE*('a)

assumes *eq-padding*: *eq-padding* (*typ-uinfo-t* *TYPE*('a)) *bs* *bs'*

shows *eq-padding* (*export-uinfo* *s*) (*take* (*size-td* *s*) (*drop* *n* *bs*)) (*take* (*size-td* *s*) (*drop* *n* *bs'*))

using *field-lookup-eq-padding-focus* [*OF fl lbs [simplified size-of-def] lbs'[simplified size-of-def]* - -

eq-padding [simplified typ-uinfo-t-def]]

by (*simp add: typ-uinfo-t-def size-of-def*)

lemma *xmem-type-field-lookup-eq-padding-focus-eq*:

assumes *lbs*: *length* *bs* = *size-of* *TYPE*('a)

assumes *lbs'*: *length* *bs'* = *size-of* *TYPE*('a)

assumes *pfx-eq*: $\bigwedge i. i < n \implies bs ! i = bs' ! i$

assumes *sfx-eq*: $\bigwedge i. n + \text{size-td } s \leq i \implies i < \text{size-of } \text{TYPE}('a) \implies bs ! i = bs' ! i$

shows *eq-padding* (*typ-uinfo-t* *TYPE*('a)) *bs* *bs'* =

eq-padding (*export-uinfo* *s*) (*take* (*size-td* *s*) (*drop* *n* *bs*)) (*take* (*size-td* *s*) (*drop* *n* *bs'*))

using *field-lookup-eq-padding-focus-eq* [*OF fl lbs [simplified size-of-def] lbs'[simplified size-of-def] pfx-eq sfx-eq*]

by (*simp add: typ-uinfo-t-def size-of-def*)

lemma *xmem-type-field-lookup-lense-eq-padding-focus*:

assumes *lbs*: *length* *bs* = *size-of* *TYPE*('a)

assumes *lbs'*: *length* *bs'* = *size-of* *TYPE*('a)

assumes *eq-padding*: *lense.eq-padding* *bs* *bs'*

shows *flense.eq-padding* (*take* (*size-td* *s*) (*drop* *n* *bs*)) (*take* (*size-td* *s*) (*drop* *n* *bs'*))

using *xmem-type-field-lookup-eq-padding-focus* [*OF lbs lbs'*] *eq-padding*

eq-padding-lense-conv *field-lookup-eq-padding-lense-conv*

by *simp*

lemma *xmem-type-field-lookup-lense-eq-padding-focus-eq*:
assumes *lbs*: $\text{length } bs = \text{size-of } TYPE('a)$
assumes *lbs'*: $\text{length } bs' = \text{size-of } TYPE('a)$
assumes *px-eq*: $\bigwedge i. i < n \implies bs ! i = bs' ! i$
assumes *sfx-eq*: $\bigwedge i. n + \text{size-td } s \leq i \implies i < \text{size-of } TYPE('a) \implies bs ! i = bs' ! i$
shows $\text{lense.eq-padding } bs \ bs' =$
 $\text{flense.eq-padding } (\text{take } (\text{size-td } s) (\text{drop } n \ bs)) (\text{take } (\text{size-td } s) (\text{drop } n \ bs'))$
using *xmem-type-field-lookup-eq-padding-focus-eq* [*OF lbs lbs' px-eq sfx-eq*]
eq-padding-lense-conv field-lookup-eq-padding-lense-conv
by *simp*

lemma *xmem-type-field-lookup-eq-upto-padding-focus*:
assumes *lbs*: $\text{length } bs = \text{size-of } TYPE('a)$
assumes *lbs'*: $\text{length } bs' = \text{size-of } TYPE('a)$
assumes *eq-upto-padding*: $\text{eq-upto-padding } (\text{typ-uinfo-t } TYPE('a)) \ bs \ bs'$
shows $\text{eq-upto-padding } (\text{export-uinfo } s) (\text{take } (\text{size-td } s) (\text{drop } n \ bs)) (\text{take } (\text{size-td } s) (\text{drop } n \ bs'))$
using *field-lookup-eq-upto-padding-focus* [*OF fl lbs [simplified size-of-def] lbs' [simplified size-of-def]*] - -
 $\text{eq-upto-padding } [\text{simplified typ-uinfo-t-def}]]$
by (*simp add: typ-uinfo-t-def size-of-def*)

lemma *xmem-type-field-lookup-eq-upto-padding-focus-eq*:
assumes *lbs*: $\text{length } bs = \text{size-of } TYPE('a)$
assumes *lbs'*: $\text{length } bs' = \text{size-of } TYPE('a)$
assumes *px-eq*: $\bigwedge i. i < n \implies bs ! i = bs' ! i$
assumes *sfx-eq*: $\bigwedge i. n + \text{size-td } s \leq i \implies i < \text{size-of } TYPE('a) \implies bs ! i = bs' ! i$
shows $\text{eq-upto-padding } (\text{typ-uinfo-t } TYPE('a)) \ bs \ bs' =$
 $\text{eq-upto-padding } (\text{export-uinfo } s) (\text{take } (\text{size-td } s) (\text{drop } n \ bs)) (\text{take } (\text{size-td } s) (\text{drop } n \ bs'))$
using *field-lookup-eq-upto-padding-focus-eq* [*OF fl lbs [simplified size-of-def] lbs' [simplified size-of-def] px-eq sfx-eq*]
by (*simp add: typ-uinfo-t-def size-of-def*)

lemma *xmem-type-field-lookup-lense-eq-upto-padding-focus*:
assumes *lbs*: $\text{length } bs = \text{size-of } TYPE('a)$
assumes *lbs'*: $\text{length } bs' = \text{size-of } TYPE('a)$
assumes *eq-upto-padding*: $\text{lense.eq-upto-padding } bs \ bs'$
shows $\text{flense.eq-upto-padding } (\text{take } (\text{size-td } s) (\text{drop } n \ bs)) (\text{take } (\text{size-td } s) (\text{drop } n \ bs'))$
using *xmem-type-field-lookup-eq-upto-padding-focus* [*OF lbs lbs'*] *eq-upto-padding eq-upto-padding-lense-conv field-lookup-eq-upto-padding-lense-conv*
by *simp*

lemma *xmem-type-field-lookup-lense-eq-upto-padding-focus-eq*:
assumes *lbs*: $\text{length } bs = \text{size-of } TYPE('a)$

assumes lbs' : $length\ bs' = size-of\ TYPE('a)$
assumes $px-eq$: $\bigwedge i. i < n \implies bs\ !\ i = bs'\ !\ i$
assumes $sfx-eq$: $\bigwedge i. n + size-td\ s \leq i \implies i < size-of\ TYPE('a) \implies bs\ !\ i = bs'\ !\ i$
shows $lense.eq-upto-padding\ bs\ bs' =$
 $flense.eq-upto-padding\ (take\ (size-td\ s)\ (drop\ n\ bs))\ (take\ (size-td\ s)\ (drop\ n\ bs^\wedge))$
using $xmem-type-field-lookup-eq-upto-padding-focus-eq$ [$OF\ lbs\ lbs'\ px-eq\ sfx-eq$]
 $eq-upto-padding-lense-conv\ field-lookup-eq-upto-padding-lense-conv$
by $simp$

lemma $xmem-type-field-lookup-eq-padding-super-update-bs$:
assumes lbs : $length\ lbs = size-of\ TYPE('a)$
assumes bs : $length\ bs = size-td\ s$
assumes bs' : $length\ bs' = size-td\ s$
shows $eq-padding\ (typ-uinfo-t\ TYPE('a))\ (super-update-bs\ lbs\ lbs\ n)\ (super-update-bs\ bs'\ lbs\ n) \longleftrightarrow$
 $eq-padding\ (export-uinfo\ s)\ lbs\ bs'$
using $field-lookup-eq-padding-super-update-bs$ [$OF\ fl\ lbs\ [simplified\ size-of-def]\ lbs\ lbs^\wedge$]
by ($simp\ add: typ-uinfo-t-def$)

lemma $xmem-type-field-lookup-lense-eq-padding-super-update-bs$:
assumes lbs : $length\ lbs = size-of\ TYPE('a)$
assumes bs : $length\ bs = size-td\ s$
assumes bs' : $length\ bs' = size-td\ s$
shows $lense.eq-padding\ (super-update-bs\ lbs\ lbs\ n)\ (super-update-bs\ bs'\ lbs\ n) \longleftrightarrow$
 $flense.eq-padding\ bs\ bs'$
using $xmem-type-field-lookup-eq-padding-super-update-bs$ [$OF\ lbs\ lbs\ lbs^\wedge$]
 $eq-padding-lense-conv\ field-lookup-eq-padding-lense-conv$
by $simp$

lemma $xmem-type-field-lookup-eq-upto-padding-super-update-bs$:
assumes lbs : $length\ lbs = size-of\ TYPE('a)$
assumes bs : $length\ bs = size-td\ s$
assumes bs' : $length\ bs' = size-td\ s$
shows $eq-upto-padding\ (typ-uinfo-t\ TYPE('a))\ (super-update-bs\ lbs\ lbs\ n)\ (super-update-bs\ bs'\ lbs\ n) \longleftrightarrow$
 $eq-upto-padding\ (export-uinfo\ s)\ lbs\ bs'$
using $field-lookup-eq-upto-padding-super-update-bs$ [$OF\ fl\ lbs\ [simplified\ size-of-def]\ lbs\ lbs^\wedge$]
by ($simp\ add: typ-uinfo-t-def$)

lemma $xmem-type-field-lookup-lense-eq-upto-padding-super-update-bs$:
assumes lbs : $length\ lbs = size-of\ TYPE('a)$
assumes bs : $length\ bs = size-td\ s$
assumes bs' : $length\ bs' = size-td\ s$
shows $lense.eq-upto-padding\ (super-update-bs\ lbs\ lbs\ n)\ (super-update-bs\ bs'\ lbs\ n)$

n) \longleftrightarrow
flense.eq-upto-padding *bs bs'*
using *xmem-type-field-lookup-eq-upto-padding-super-update-bs* [*OF lbs bs lbs'*]
eq-upto-padding-lense-conv *field-lookup-eq-upto-padding-lense-conv*
by *simp*

lemma *access-ti-update-ti-lense-eq-upto-padding*:
assumes *lbs: length bs = size-td s*
assumes *lbs': length bs' = size-td s*
shows *flense.eq-upto-padding (access-ti s (update-ti s bs v) bs') bs*
by (*simp add: flense.value-byte-to-eq-upto-padding-eq lbs lbs' length-fa-ti*
flense.is-value-byte-acc-upd-cancel wf-fd-s)

lemma *access-ti-update-ti-eq-upto-padding*:
assumes *lbs: length bs = size-td s*
assumes *lbs': length bs' = size-td s*
shows *eq-upto-padding (export-uinfo s) (access-ti s (update-ti s bs v) bs') bs*
using *access-ti-update-ti-lense-eq-upto-padding* [*OF lbs lbs'*]
field-lookup-eq-upto-padding-lense-conv
by *simp*

lemma *access-ti-update-ti-lense-eq-padding*:
assumes *flense.eq-padding bs bs'*
shows *access-ti s (update-ti s bs v) bs' = bs*
by (*metis (no-types, lifting) access-ti-update-ti-lense-eq-upto-padding assms flense.field-access-eq-padding1*
flense.padding-eq-complement padding-base.eq-padding-length1
flense.eq-padding-length2 flense.eq-padding-trans)

lemma *access-ti-update-ti-eq-padding*:
assumes *eq-padding (export-uinfo s) bs bs'*
shows *access-ti s (update-ti s bs v) bs' = bs*
using *access-ti-update-ti-lense-eq-padding assms field-lookup-eq-padding-lense-conv*
by *simp*

lemma
assumes *match: export-uinfo s = typ-uinfo-t TYPE('b::xmem-type)*
assumes *lbs: length bs = size-td s*
shows *access-ti s v bs = to-bytes ((from-bytes (access-ti s v bs))::'b) bs*
proof –

from *match*
have *sz-acc-s: length (access-ti s v bs) = size-td (typ-info-t TYPE('b))*
by (*metis export-size-of flense.access-result-size fold-typ-uinfo-t*)
show *?thesis*
by (*smt (verit, best) field-lookup-eq-padding-lense-conv flense.field-access-eq-padding1*
lbs match sz-acc-s xmem-type-class.eq-padding-lense-conv xmem-type-class.field-sz-size-of
xmem-type-class.field-sz-size-td xmem-type-class.lense-eq-padding-to-bytes-eq
xmem-type-class.to-bytes-from-bytes-id)

qed

```

context
  assumes match: export-uinfo s = typ-uinfo-t TYPE('b::xmem-type)
begin

lemma field-lookup-lense-eq-padding-fieldtyp-conv:
  flense.eq-padding bs bs' = xmem-type-class.lense.eq-padding TYPE('b) bs bs'
  using field-lookup-eq-padding-lense-conv match xmem-type-class.eq-padding-lense-conv
by auto

lemma field-lookup-lense-eq-upto-padding-fieldtyp-conv:
  flense.eq-upto-padding bs bs' = xmem-type-class.lense.eq-upto-padding TYPE('b) bs bs'
  using field-lookup-eq-upto-padding-lense-conv match xmem-type-class.eq-upto-padding-lense-conv
by auto

lemma field-lookup-is-value-byte-fieldtyp-conv:
  flense.is-value-byte i = xmem-type-class.lense.is-value-byte TYPE('b) i
  using field-lookup-is-value-byte-lense-conv match xmem-type-class.is-value-byte-lense-conv
by auto

lemma field-lookup-is-padding-byte-fieldtyp-conv:
  flense.is-padding-byte i = xmem-type-class.lense.is-padding-byte TYPE('b) i
  using field-lookup-is-padding-byte-lense-conv match xmem-type-class.is-padding-byte-lense-conv
by auto

lemma field-lookup-access-ti-to-bytes-field-conv:
  assumes eq-upto-padding: eq-upto-padding (export-uinfo s) (access-ti s v bs) vs
  assumes eq-padding: eq-padding (export-uinfo s) bs bs'
  shows access-ti s v bs = to-bytes ((from-bytes vs)::'b) bs'
proof –
  from eq-upto-padding have flense-eq-upto: flense.eq-upto-padding (access-ti s v bs) vs
  by (simp add: field-lookup-eq-upto-padding-lense-conv)
  hence blense-eq-upto: xmem-type-class.lense.eq-upto-padding TYPE('b) (access-ti s v bs) vs
  by (simp add: field-lookup-lense-eq-upto-padding-fieldtyp-conv)

from eq-padding have flense-eq: flense.eq-padding bs bs'
  by (simp add: field-lookup-eq-padding-lense-conv)
hence blense-eq: xmem-type-class.lense.eq-padding TYPE('b) bs bs'
  by (simp add: field-lookup-lense-eq-padding-fieldtyp-conv)

show ?thesis
  using flense-eq flense-eq-upto blense-eq blense-eq-upto

apply (simp add: to-bytes-def from-bytes-def c-type-class.to-bytes-def c-type-class.from-bytes-def)
by (smt (verit, ccfv-threshold) c-type-class.to-bytes-def field-lookup-lense-eq-padding-fieldtyp-conv)

```

*flense.eq-padding-length1 flense.field-access-eq-padding1 padding-base.eq-upto-padding-length2
update-ti-s-from-bytes xmem-type-class.lense.eq-padding-acc xmem-type-class.lense-eq-upto-padding-from-b
xmem-type-class.to-bytes-from-bytes-id)*

qed

lemma *field-lookup-access-ti-eq-upto-padding:*

length bs = size-td s \implies

eq-upto-padding (export-uinfo s) (access-ti s v bs) (access-ti s v bs')

by (*metis access-ti-update-ti-eq-upto-padding flense.access-result-size flense.update-access*)

lemma *field-lookup-access-ti-eq-padding-value:*

*length bs = size-td s \implies eq-padding (export-uinfo s) (access-ti s v bs) (access-ti
s v' bs)*

by (*meson eq-padding-sym eq-padding-trans field-lookup-eq-padding-lense-conv flense.field-access-eq-padding1*)

lemma *field-lookup-access-ti-eq-padding-bytes:*

length bs = size-td s \implies eq-padding (export-uinfo s) (access-ti s v bs) bs

using *field-lookup-eq-padding-lense-conv flense.field-access-eq-padding1* **by** *blast*

lemma *field-lookup-access-ti-to-bytes-field-conv':*

assumes *eq-upto-padding: eq-upto-padding (export-uinfo s) (access-ti s v bs) vs*

assumes *lbs: length bs = size-td s*

shows *access-ti s v bs = to-bytes ((from-bytes vs)::'b) bs*

using *field-lookup-access-ti-to-bytes-field-conv [OF eq-upto-padding eq-padding-refl]*

lbs

by *simp*

lemma *field-lookup-update-ti-from-bytes-field-conv:*

fixes *v::'a and vf::'b*

assumes *lbs: length bs = size-td s*

assumes *lbs: length lbs = size-of TYPE('a)*

shows *update-ti (typ-info-t TYPE('b)) bs vf =*

(from-bytes (access-ti s (update-ti s bs v) lbs))

proof –

from *lbs lbs*

have *eq-upto-padding (export-uinfo s) (access-ti s (update-ti s bs v) lbs) bs*

by (*metis access-ti-update-ti-eq-upto-padding access-ti-update-ti-lense-eq-padding
flense.access-result-size flense.eq-padding-refl flense.update-access*)

hence *from-bytes (access-ti s (update-ti s bs v) lbs) = (from-bytes bs::'b)*

by (*simp add: match xmem-type-class.eq-upto-padding-from-bytes-eq*)

moreover have *update-ti (typ-info-t TYPE('b)) bs vf = from-bytes bs*

using *lbs*

by (*metis (mono-tags, lifting) c-type-class.typ-uinfo-size export-size-of export-uinfo-size
match*)

update-ti-s-from-bytes update-ti-update-ti-t)

ultimately show *?thesis* **by** *simp*

qed

lemma *xmem-type-field-lookup-update-ti-super-update-bs-conv*:
fixes $v::'a$
assumes lbs : $length\ bs = size-td\ s$
assumes lbs : $length\ lbs = size-of\ TYPE('a)$
shows $update-ti\ s\ bs\ v =$
 $update-ti\ (typ-info-t\ TYPE('a))\ (super-update-bs\ bs\ (access-ti\ (typ-info-t$
 $TYPE('a))\ v\ lbs)\ n)\ v$
using *field-lookup-update-ti-super-update-bs-conv(1)* [**where** $m=0$, *simplified*, *OF*
 $fl\ lbs\ lbs$ [*simplified size-of-def*], **where** $v=v$]
by *simp*

lemma *heap-update-list-field-to-root*:
fixes $p::'a\ ptr$
assumes $cgrd$: *c-guard* p
assumes lbs : $length\ bs = size-td\ s$
shows $heap-update-list\ (\&(p\to f))\ bs\ hp =$
 $heap-update-list\ (ptr-val\ p)\ (super-update-bs\ bs\ (heap-list\ hp\ (size-of\ TYPE('a))$
 $(ptr-val\ p))\ n)\ hp$
proof –
from fl
have off : $word-of-nat\ (field-offset\ TYPE('a)\ f) = word-of-nat\ n$
by (*simp*)
from *c-guard-no-wrap'* [*OF* $cgrd$]
have $no-overflow$: $unat\ (ptr-val\ p) + size-of\ TYPE('a) \leq addr-card$.
from $lbs\ fl$ **have** $n-bound$: $n + length\ bs \leq size-of\ TYPE('a)$
by (*metis add.commute field-lookup-offset-size size-of-def*)

show *?thesis*
apply (*simp add: field-lvalue-def off*)
apply (*rule heap-update-list-super-update-bs-heap-list* [*OF no-overflow n-bound*])
done
qed

lemma *heap-list-field-to-root*:
fixes $p::'a\ ptr$
shows $heap-list\ hp\ (size-td\ s)\ \&(p\to f) =$
 $take\ (size-td\ s)\ ((drop\ n)\ (heap-list\ hp\ (size-of\ TYPE('a))\ (ptr-val\ (p::'a$
 $ptr))))$
proof –
from fl **have** off : $(field-offset\ TYPE('a)\ f) = n$ **by** *simp*
from fl
have $n-bound$: $n \leq size-of\ TYPE('a)$
by (*meson n-t nat-less-le*)
from fl
have $s-bound$: $size-td\ s \leq size-of\ TYPE('a) - n$
by (*metis field-lookup-offset-size nat-move-sub-le size-of-def*)

show ?thesis
by (simp add: field-lvalue-def off drop-heap-list-le [OF n-bound] take-heap-list-le [OF s-bound])
qed

lemma heap-list-field-super-update-bs-root-conv:

fixes p::'a ptr

shows super-update-bs (heap-list hp (size-td s) (&(p→f))) (heap-list hp (size-of TYPE('a)) (ptr-val p)) n =
(heap-list hp (size-of TYPE('a)) (ptr-val p))

proof –

have l: length (heap-list hp (size-of TYPE('a)) (ptr-val p)) = size-of TYPE('a)
by simp

from l fl **have** l-take: (length (take n (heap-list hp (size-of TYPE('a)) (ptr-val p)))) = n
by (simp add: le-less n-t)

from l fl n-t

have l-s-take: length ((take (size-td s) (drop n (heap-list hp (size-of TYPE('a)) (ptr-val p)))))) = size-td s

by (metis heap-list-field-to-root heap-list-length)

have com: (n + size-td s) = (size-td s + n)

by simp

show ?thesis

apply (simp add: super-update-bs-def)

apply (subst heap-list-field-to-root)

apply (rule append-take-dropI)

apply (simp only: l-take)

apply (rule append-take-dropI)

apply (simp only: l-s-take l-take)

apply (simp only: l-s-take l-take)

apply (simp)

apply (subst com)

apply (rule refl)

done

qed

lemma heap-update-field-root-conv:

fixes p::'a ptr

assumes cgrd: c-guard p

shows heap-update (PTR('b) &(p→f)) v hp =
heap-update p (update-ti s (to-bytes v (heap-list hp (size-of TYPE('b)) (&(p→f)))) (h-val hp p)) hp

```

unfolding heap-update-def c-type-class.heap-update-def ptr-val-def
proof –
  show heap-update-list &(p→f) (to-bytes v (heap-list hp (size-of TYPE('b)) &(p→f)))
hp =
  heap-update-list (ptr-val p)
  (to-bytes
    (update-ti s (to-bytes v (heap-list hp (size-of TYPE('b)) &(p→f))) (h-val hp
p)))
    (heap-list hp (size-of TYPE('a)) (ptr-val p)))
  hp (is ?lhs = ?rhs)
proof –
  define bs where bs = to-bytes v (heap-list hp (size-of TYPE('b)) (&(p→f)))
  have lbs: length bs = size-td s
  unfolding bs-def
  by (simp add: export-size-of match)

  from heap-update-list-field-to-root [OF cgrd lbs]
  have eq1:
    heap-update-list (&(p→f)) bs hp =
      heap-update-list (ptr-val p) (super-update-bs bs (heap-list hp (size-of
TYPE('a)) (ptr-val p)) n) hp
  by simp

  have lhl: length (heap-list hp (size-of TYPE('a)) (ptr-val p)) = size-of TYPE('a)
  by simp
  from xmem-type-field-lookup-update-ti-super-update-bs-conv [OF lbs this, where
v=h-val hp p]
  have eq2:
    update-ti s bs (h-val hp p) =
      update-ti (typ-info-t TYPE('a))
      (super-update-bs bs
        (access-ti (typ-info-t TYPE('a)) (h-val hp p) (heap-list hp (size-of
TYPE('a)) (ptr-val p)))
        n)
      (h-val hp p) .

  have eq3: access-ti (typ-info-t TYPE('a)) (h-val hp p) (heap-list hp (size-of
TYPE('a)) (ptr-val p)) =
    heap-list hp (size-of TYPE('a)) (ptr-val p)
  apply (simp add: h-val-def from-bytes-def update-ti-t-def size-of-def)
  using lhl
  by (metis local.field-sz-size-of local.field-sz-size-td local.lense.access-result-size
local.lense.eq-padding-neq-is-value-byte local.lense.field-access-eq-padding1
local.lense.is-value-byte-acc-upd-cancel nth-equalityI)

  have lsuper: length (super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val
p)) n) = size-of TYPE('a)
  using flense.eq-padding-refl lbs lhl padding-base.eq-padding-length1

```

```

xmem-type-field-lookup-lense-eq-padding-super-update-bs by blast

thm field-lookup-eq-padding-super-update-bs
  have eq-padding-bs: eq-padding (export-uinfo s) bs (heap-list hp (size-td s)
(&(p→f)))
  apply (simp add: bs-def to-bytes-def c-type-class.to-bytes-def)
  by (metis (mono-tags, lifting) c-type-class.to-bytes-def export-size-of heap-list-length
match
      xmem-type-class.eq-padding-to-bytes)

  have l-take-drop: length (take (size-td s) (drop n (heap-list hp (size-of TYPE('a)
(ptr-val p)))) = size-td s
  by (metis heap-list-field-to-root heap-list-length)
  thm heap-list-field-super-update-bs-root-conv

from xmem-type-field-lookup-eq-padding-super-update-bs [OF lhl lbs,
  where bs' = take (size-td s) (drop n (heap-list hp (size-of TYPE('a)
(ptr-val p))), OF l-take-drop,
  simplified heap-list-field-to-root [symmetric], simplified heap-list-field-super-update-bs-root-conv]
  eq-padding-bs
  have eq-padding (typ-uinfo-t TYPE('a)) (super-update-bs bs (heap-list hp (size-of
TYPE('a)) (ptr-val p)) n
  (heap-list hp (size-of TYPE('a)) (ptr-val p))
  by (simp)

  hence eq-padding: lense.eq-padding (super-update-bs bs (heap-list hp (size-of
TYPE('a)) (ptr-val p)) n
  (heap-list hp (size-of TYPE('a)) (ptr-val p))
  by (simp add: eq-padding-lense-conv)

from eq-padding
have eq4:
  (to-bytes
  (update-ti (typ-info-t TYPE('a))
  (super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n) (h-val
hp p))
  (heap-list hp (size-of TYPE('a)) (ptr-val p))) =
  (super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n)
  apply (simp add: to-bytes-def)
by (smt (verit) lhl local.lense.access-result-size local.lense.eq-padding-neq-is-value-byte
  local.lense.field-access-eq-padding1 local.lense.is-value-byte-acc-upd-cancel
lsuper nth-equalityI)

show ?thesis
  unfolding bs-def [symmetric]
  apply (simp add: eq1)
  apply (simp add: eq2)
  apply (simp add: eq3)

```

```

    apply (simp add: eq4)
  done
qed
qed

lemma heap-update-padding-field-root-conv:
  fixes p: 'a ptr
  assumes cgrd: c-guard p
  assumes lbs: length bs = size-of TYPE ('b)
  shows heap-update-padding (PTR('b) &(p→f)) v bs hp =
    heap-update-padding p (update-ti s (to-bytes v bs) (h-val hp p))
      (super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n) hp
  unfolding heap-update-padding-def c-type-class.heap-update-padding-def ptr-val-def
  proof -
    show heap-update-list &(p→f) (to-bytes v bs) hp =
      heap-update-list (ptr-val p)
        (to-bytes
          (update-ti s (to-bytes v bs) (h-val hp p))
          (super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n))
      hp (is ?lhs = ?rhs)
    proof -
      define bs' where bs' = to-bytes v bs
      have lbs': length bs' = size-td s
        unfolding bs'-def
        by (simp add: export-size-of lbs match)

      from heap-update-list-field-to-root [OF cgrd lbs']
      have eq1:
        heap-update-list (&(p→f)) bs' hp =
          heap-update-list (ptr-val p) (super-update-bs bs' (heap-list hp (size-of
            TYPE('a)) (ptr-val p)) n) hp
        by simp

      have lhl: length (heap-list hp (size-of TYPE('a)) (ptr-val p)) = size-of TYPE('a)
        by simp
      from xmem-type-field-lookup-update-ti-super-update-bs-conv [OF lbs' this, where
        v=h-val hp p]
      have eq2:
        update-ti s bs' (h-val hp p) =
          update-ti (typ-info-t TYPE('a))
            (super-update-bs bs'
              (access-ti (typ-info-t TYPE('a)) (h-val hp p) (heap-list hp (size-of
                TYPE('a)) (ptr-val p)))
                n)
            (h-val hp p) .

      have eq3: access-ti (typ-info-t TYPE('a)) (h-val hp p) (heap-list hp (size-of

```



```

TYPE('a) (ptr-val p)) =
  heap-list hp (size-of TYPE('a)) (ptr-val p)
apply (simp add: h-val-def from-bytes-def update-ti-t-def size-of-def)
using h1
by (metis local.field-sz-size-of local.field-sz-size-td local.lense.access-result-size
  local.lense.eq-padding-neq-is-value-byte local.lense.field-access-eq-padding1
  local.lense.is-value-byte-acc-upd-cancel nth-equalityI)

have lsuper: length (super-update-bs bs' (heap-list hp (size-of TYPE('a)) (ptr-val
p)) n) = size-of TYPE('a)
using flense.eq-padding-refl lbs' h1 padding-base.eq-padding-length1
  xmem-type-field-lookup-lense-eq-padding-super-update-bs by blast

thm field-lookup-eq-padding-super-update-bs

have l-take-drop: length (take (size-td s) (drop n (heap-list hp (size-of TYPE('a))
(ptr-val p)))) = size-td s
by (metis heap-list-field-to-root heap-list-length)
have eq4:
  (to-bytes
    (update-ti (typ-info-t TYPE('a))
      (super-update-bs bs' (heap-list hp (size-of TYPE('a)) (ptr-val p)) n) (h-val
hp p))
    (super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n) =
    (super-update-bs bs' (heap-list hp (size-of TYPE('a)) (ptr-val p)) n)
apply (simp add: to-bytes-def)
by (smt (verit, ccfv-threshold) bs'-def export-size-of field-lookup-lense-eq-padding-fieldtyp-conv
  lbs h1 local.lense.access-result-size local.lense.complement local.lense.eq-padding-acc
  local.lense.is-padding-byte-def local.lense.is-value-byte-def lsuper match mem-type-class.mem-type-simps(2)
  nth-equalityI wf-type-class.len xmem-type-class.lense-eq-padding-to-bytes xmem-type-field-lookup-lense-eq-paddin

show ?thesis
  unfolding bs'-def [symmetric]
  apply (simp add: eq1)
  apply (simp add: eq2)
  apply (simp add: eq3)
  apply (simp add: eq4)
  done
qed
qed

lemma heap-update-field-root-conv':
  fixes p::'a ptr
  assumes cgrd: c-guard p
  shows heap-update (PTR('b) &(p→f)) v hp =
    heap-update p (update-ti s (to-bytes-p v) (h-val hp p)) hp
proof –
  have eq-upto-padding (export-uinfo s) (to-bytes-p v) (to-bytes v (heap-list hp
(size-of TYPE('b)) (&(p→f))))

```

```

apply (simp add: to-bytes-def c-type-class.to-bytes-def to-bytes-p-def c-type-class.to-bytes-p-def)
by (simp add: field-lookup-eq-upto-padding-lense-conv field-lookup-lense-eq-upto-padding-fieldtyp-conv
  xmem-type-class.lense.field-access-eq-upto-padding)
hence update-ti s (to-bytes-p v) (h-val hp p) =
  update-ti s (to-bytes v (heap-list hp (size-of TYPE('b)) (&(p→f)))) (h-val
hp p)
by (simp add: field-lookup-eq-upto-padding-lense-conv padding-base.eq-upto-padding-def)
with heap-update-field-root-conv [OF cgrd] show ?thesis by simp
qed

```

lemma heap-update-padding-field-root-conv':

```

fixes p::'a ptr
assumes cgrd: c-guard p
assumes lbs: length bs = size-of TYPE ('b)
shows heap-update-padding (PTR('b) &(p→f)) v bs hp =
  heap-update-padding p (update-ti s (to-bytes-p v) (h-val hp p))
  (super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n) hp
proof –
have eq-upto-padding (export-uinfo s) (to-bytes-p v) (to-bytes v bs)
apply (simp add: to-bytes-def c-type-class.to-bytes-def to-bytes-p-def c-type-class.to-bytes-p-def)
by (simp add: field-lookup-eq-upto-padding-lense-conv field-lookup-lense-eq-upto-padding-fieldtyp-conv
  xmem-type-class.lense.field-access-eq-upto-padding)
hence update-ti s (to-bytes-p v) (h-val hp p) =
  update-ti s (to-bytes v bs) (h-val hp p)
by (simp add: field-lookup-eq-upto-padding-lense-conv padding-base.eq-upto-padding-def)
with heap-update-padding-field-root-conv [OF cgrd lbs] show ?thesis by simp
qed

```

end
end

lemma heap-update-field-root-conv'':

```

fixes p::'a ptr
assumes fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (adjust-ti (typ-info-t
TYPE('b::xmem-type)) fld (fld-update ∘ (λx -. x)), n)
assumes fg-cons: fg-cons fld (fld-update ∘ (λx -. x))
assumes cgrd: c-guard p
shows heap-update (PTR('b) &(p→f)) v hp =
  heap-update p (fld-update (λ-. v) (h-val hp p)) hp
proof –
have match: export-uinfo (adjust-ti (typ-info-t TYPE('b)) fld (fld-update ∘ (λx
-. x))) = typ-uinfo-t TYPE('b)
apply (subst c-type-class.typ-uinfo-t-def)
apply (rule export-tag-adjust-ti(1) )
apply (rule fg-cons)
apply (simp)
done

```

from *field-desc-adjust-ti(1)* [*rule-format, OF fg-cons, of (typ-info-t TYPE('b))*]
have *upd-eq*:

$$\text{update-ti-t } (\text{adjust-ti } (\text{typ-info-t } \text{TYPE}('b)) \text{ fld } (\text{fld-update } \circ (\lambda x -. x))) \text{ bs } v =$$

$$\text{fld-update } (\lambda -. \text{update-ti-t } (\text{typ-info-t } \text{TYPE}('b)) \text{ bs } (\text{fld } v)) \text{ v } \text{ for } \text{bs } v$$
by (*simp add: update-desc-def*)
from *this* [*of (to-bytes-p v) h-val hp p*]
have *val-conv*:

$$(\text{update-ti } (\text{adjust-ti } (\text{typ-info-t } \text{TYPE}('b)) \text{ fld } (\text{fld-update } \circ (\lambda x -. x))) (\text{to-bytes-p}$$

$$v) (\text{h-val } hp \text{ p})) =$$

$$(\text{fld-update } (\lambda -. v) (\text{h-val } hp \text{ p}))$$
apply (*simp add: update-ti-t-def size-of-def c-type-class.size-of-def*)
by (*metis (mono-tags, opaque-lifting) inv-p length-to-bytes-p update-ti-s-from-bytes*
update-ti-t-def)

note *heap-conv = heap-update-field-root-conv'* [*OF fl match cgrd, of v hp*]
show *?thesis*
apply (*simp add: heap-conv*)
apply (*simp add: val-conv*)
done
qed

lemma *heap-update-field-root-conv-pointless'*:
fixes *p::'a ptr*
assumes *fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (adjust-ti (typ-info-t*
TYPE('b::xmem-type)) fld (fld-update \circ ($\lambda x -. x$)), n)
assumes *fg-cons: fg-cons fld (fld-update \circ ($\lambda x -. x$))*
assumes *cgrd: c-guard p*
shows *heap-update (PTR('b) $\&$ ($p \rightarrow f$)) v =*

$$(\lambda hp. \text{heap-update } p (\text{fld-update } (\lambda -. v) (\text{h-val } hp \text{ p})) \text{ hp})$$
using *heap-update-field-root-conv''* [*OF fl fg-cons cgrd*]
by (*simp add: fun-eq-iff*)

lemma *heap-update-padding-field-root-conv''*:
fixes *p::'a ptr*
assumes *fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (adjust-ti (typ-info-t*
TYPE('b::xmem-type)) fld (fld-update \circ ($\lambda x -. x$)), n)
assumes *fg-cons: fg-cons fld (fld-update \circ ($\lambda x -. x$))*
assumes *cgrd: c-guard p*
assumes *lbs: length bs = size-of TYPE ('b)*
shows *heap-update-padding (PTR('b) $\&$ ($p \rightarrow f$)) v bs hp =*

$$\text{heap-update-padding } p (\text{fld-update } (\lambda -. v) (\text{h-val } hp \text{ p}))$$

$$(\text{super-update-bs } bs (\text{heap-list } hp (\text{size-of } \text{TYPE}('a)) (\text{ptr-val } p)) \text{ n}) \text{ hp}$$

proof –
have *match: export-uinfo (adjust-ti (typ-info-t TYPE('b)) fld (fld-update \circ (λx*
- . x))) = typ-uinfo-t TYPE('b)
apply (*subst c-type-class.typ-uinfo-t-def*)
apply (*rule export-tag-adjust-ti(1)*)
apply (*rule fg-cons*)
apply (*simp*)

done
from *field-desc-adjust-ti*(1) [*rule-format*, *OF fg-cons*, of (*typ-info-t* *TYPE*('b))]
have *upd-eq*:
update-ti-t (*adjust-ti* (*typ-info-t* *TYPE*('b)) *fld* (*fld-update* \circ ($\lambda x -. x$))) *bs* *v* =
fld-update ($\lambda -. \text{update-ti-t}$ (*typ-info-t* *TYPE*('b)) *bs* (*fld* *v*)) *v* **for** *bs* *v*
by (*simp add: update-desc-def*)
from *this* [of (*to-bytes-p* *v*) *h-val* *hp* *p*]
have *val-conv*:
(*update-ti* (*adjust-ti* (*typ-info-t* *TYPE*('b)) *fld* (*fld-update* \circ ($\lambda x -. x$))) (*to-bytes-p*
v) (*h-val* *hp* *p*)) =
(*fld-update* ($\lambda -. v$) (*h-val* *hp* *p*))
apply (*simp add: update-ti-t-def size-of-def c-type-class.size-of-def*)
by (*metis* (*mono-tags*, *opaque-lifting*) *inv-p length-to-bytes-p update-ti-s-from-bytes*
update-ti-t-def)

note *heap-conv* = *heap-update-padding-field-root-conv'* [*OF fl match cgrd lbs*, of
v *hp*]
show *?thesis*
apply (*simp add: heap-conv*)
apply (*simp add: val-conv*)
done
qed

lemma *heap-update-field-root-conv''*:
fixes *p*::'a *ptr*
assumes *fl*: *field-ti* *TYPE*('a) *f* = *Some* *s*
assumes *cgrd*: *c-guard* *p*
assumes *match*: *export-uinfo* *s* = *typ-uinfo-t* *TYPE*('b::*xmem-type*)
shows *heap-update* (*PTR*('b) $\&$ (*p* \rightarrow *f*)) *v* *hp* =
heap-update *p* (*update-ti* *s* (*to-bytes-p* *v*) (*h-val* *hp* *p*)) *hp*
proof –
from *fl* **obtain** *n* **where** *fl'*: *field-lookup* (*typ-info-t* *TYPE*('a)) *f* 0 = *Some* (*s*,
n)
by (*auto simp add: field-ti-def split: option.splits*)

from *heap-update-field-root-conv'* [*OF fl' match cgrd*, of *v* *hp*]
show *?thesis*
by (*simp*)
qed

lemma *heap-update-padding-field-root-conv''*:
fixes *p*::'a *ptr*
assumes *fl*: *field-lookup* (*typ-info-t* *TYPE*('a)) *f* 0 = *Some* (*s*, *n*)
assumes *cgrd*: *c-guard* *p*
assumes *match*: *export-uinfo* *s* = *typ-uinfo-t* *TYPE*('b::*xmem-type*)
assumes *lbs*: *length* *bs* = *size-of* *TYPE* ('b)
shows *heap-update-padding* (*PTR*('b) $\&$ (*p* \rightarrow *f*)) *v* *bs* *hp* =
heap-update-padding *p* (*update-ti* *s* (*to-bytes-p* *v*) (*h-val* *hp* *p*))

```

      (super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n) hp
proof –
  from heap-update-padding-field-root-conv' [OF fl match cgrd lbs, of v hp]
  show ?thesis
    by (simp)
qed

end

```

```

lemma length-array-to-bytes:
  fixes arr::('a::array-outer-max-size['b::array-max-count])
  shows length (to-bytes arr (heap-list h (CARD('b) * size-of TYPE('a)) (ptr-val
p))) =
    size-of TYPE('a) * CARD('b)
  by (simp add: lense.access-result-size)

```

```

lemma take-drop-append-first: m + n ≤ length xs ⇒ take n (drop m (xs @ ys))
= take n (drop m xs)
  by simp

```

```

lemma size-td-list-array:
size-td-list
  (map (λn. DTuple
    (adjust-ti (typ-info-t TYPE('a::xmem-type)) (λx. x.[n])
      (λx f. Arrays.update f n x))
    (replicate n CHR "1")
    (field-access = xto-bytes ∘ (λx. x.[n]),
      field-update = (λx f. Arrays.update f n x) ∘ xfrom-bytes,
      field-sz = size-of TYPE('a)))
    [0.. $n$ ]) = n * size-of TYPE('a)
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then show ?case by (simp add: size-of-def)
qed

```

```

lemma length-access-ti-list-array:
  fixes arr::('a::array-outer-max-size['b::array-max-count])
  assumes lbs: length xbs = n * size-of TYPE('a)
  shows
length (access-ti-list
  (map (λn. DTuple
    (adjust-ti (typ-info-t TYPE('a)) (λx. x.[n])
      (λx f. Arrays.update f n x))
    (replicate n CHR "1")
    (field-access = xto-bytes ∘ (λx. x.[n]),

```

```

      field-update = (λx f. Arrays.update f n x) ∘ xfrom-bytes,
      field-sz = size-of TYPE('a))
    [0..<n]
    arr xbs) = n * size-of TYPE('a)
  using lxs
proof (induct n arbitrary: xbs)
  case 0
  then show ?case by simp
next
  case (Suc n)

  have lxs': length xbs =
    size-td-list
    (map (λn. DTuple
      (adjust-ti (typ-info-t TYPE('a)) (λx. x.[n])
        (λx f. Arrays.update f n x))
      (replicate n CHR "1")
      (λfield-access = λa. xto-bytes (a.[n]),
        field-update = λa b. Arrays.update b n (xfrom-bytes a),
        field-sz = size-td (typ-info-t TYPE('a))))
    [0..<n] @
    [DTuple (adjust-ti (typ-info-t TYPE('a)) (λx. x.[n]) (λx f. Arrays.update f n
x))
      (replicate n CHR "1")
      (λfield-access = λa. xto-bytes (a.[n]),
        field-update = λa b. Arrays.update b n (xfrom-bytes a),
        field-sz = size-td (typ-info-t TYPE('a))))])
  apply simp
  apply (subst size-td-list-array [simplified size-of-def comp-def])
  using Suc.prem
  apply (simp add: size-of-def)
  done

from Suc.prem
have ltake: length (take (n * size-td (typ-info-t TYPE('a))) xbs) =
  n * size-td (typ-info-t TYPE('a))
  by (auto simp add: size-of-def)

have llast: length
  (access-ti (typ-info-t TYPE('a)) (arr.[n])
    (take (size-td (typ-info-t TYPE('a)))
      (drop
        (size-td-list
          (map (λn. DTuple
            (adjust-ti (typ-info-t TYPE('a)) (λx. x.[n])
              (λx f. Arrays.update f n x))
            (replicate n CHR "1")
            (λfield-access = λa. xto-bytes (a.[n]),
              field-update = λa b. Arrays.update b n (xfrom-bytes a),

```

```

      field-sz = size-td (typ-info-t TYPE('a)))
    [0..<n])
  xbs))) =
size-td (typ-info-t TYPE('a))
by (simp add: lense.access-result-size size-of-def)

show ?case apply (simp add: size-of-def)
  apply (subst access-ti-append [OF lxs'])
  apply (subst size-td-list-array [simplified size-of-def comp-def])
  apply simp
  apply (subst Suc.hyps [simplified size-of-def comp-def, OF ltake])
  apply (simp add: llast)
done
qed

lemma take-drop-take: m + k ≤ n ⇒ n ≤ length xbs ⇒ take m (drop k (take
n xbs)) = take m (drop k xbs)
proof (induct xbs arbitrary: m k n)
  case Nil
  then show ?case by simp
next
  case (Cons x xbs)
  then show ?case by (simp add: take-drop)
qed

lemma access-ti-array-index:
  fixes arr::('a::array-outer-max-size['b::array-max-count])
  assumes bound: i < n
  assumes lxs: length xbs = n * size-of TYPE('a)
  assumes bs: bs = take (size-of TYPE('a)) (drop (i * (size-of TYPE('a))) xbs)
  shows
    access-ti (typ-info-t TYPE('a)) (arr.[i])
      (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) xbs)) =
      take (size-of TYPE('a))
        (drop (i * size-of TYPE('a)) (access-ti (array-tag-n n) arr xbs))
  using bound lxs bs
proof (induct n arbitrary: i xbs bs)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prem1 obtain
    i-bound: i < Suc n and
    lxs: length xbs = size-of TYPE('a) + n * size-of TYPE('a) and
    bs: bs = take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) xbs)
  by clarsimp

show ?case

```

```

proof (cases i = n)
  case True
  show ?thesis
    apply (simp add: array-tag-n-eq)
    apply (subst access-ti-append)
    apply (simp add: size-td-list-array)
    apply (simp add: size-of-def lxs)
    apply (subst drop-append-miracle)
    apply (subst length-access-ti-list-array)
    apply (subst size-td-list-array)
  using lxs
    apply (simp add: True)
    apply (simp add: True)
    apply (simp add: True size-td-list-array)
    apply (simp add: size-of-def)
    by (simp add: lense.access-result-size size-of-def)
  next
  case False
  from False i-bound have i-bound': i < n by simp
  from lxs have lxs': length (take (n * size-of TYPE('a)) lxs) = n * size-of
TYPE('a) by simp
  from bs i-bound' lxs' lxs False
  have bs':
    bs = take (size-of TYPE('a))
      (drop (i * size-of TYPE('a)) (take (n * size-of TYPE('a)) lxs))

  apply (simp)
  apply (subst take-drop-take)
  using less-le-mult-nat sz-nzero apply blast
  apply simp
  apply simp
  done
  show ?thesis
    apply (simp add: array-tag-n-eq)
    apply (subst access-ti-append)
    apply (simp add: size-td-list-array)
    apply (simp add: size-of-def lxs)
    apply (subst take-drop-append-first)
    apply (subst length-access-ti-list-array)
    apply (subst size-td-list-array)
  using lxs apply (simp add: )
  using lxs i-bound False
  apply (metis add.commute less-le-mult-nat mem-type-simps(3) not-less-less-Suc-eq)
  apply (subst size-td-list-array)
  apply (simp add: bs [symmetric])

  using Suc.hyps [where lxs=(take (n * size-of TYPE('a)) lxs), OF i-bound'
lxs' bs', simplified bs'[symmetric]]
  apply (simp)

```



```

    apply (subst array-tag-n-eq)
    apply simp
  done
qed
qed

```

```

lemma access-ti-array-index':
  fixes arr::('a::array-outer-max-size['b::array-max-count])
  assumes bound:  $i < \text{CARD}('b)$ 
  assumes lbs:  $\text{length } xbs = (\text{CARD}('b) * \text{size-of } \text{TYPE}('a))$ 
  assumes bs:  $bs = \text{take } (\text{size-of } \text{TYPE}('a)) (\text{drop } (i * (\text{size-of } \text{TYPE}('a))) xbs)$ 
  shows
    access-ti (typ-info-t TYPE('a)) (arr.[i]) bs =
      take (size-of TYPE('a))
        (drop (i * size-of TYPE('a))
          (access-ti (typ-info-t TYPE('a['b])) arr xbs))
  apply (simp add: typ-info-array array-tag-def)
  using access-ti-array-index [OF bound lbs bs, where arr=arr, simplified bs
    [symmetric]]
  apply simp
  done

```

```

lemma fold-index-shift:  $\text{fold } f [n..<n + m] = \text{fold } (\lambda i. f (n + i)) [0..<m]$ 
proof (induct m arbitrary: n)
  case 0
  then show ?case by simp
next
  case (Suc m)
  thus ?case by simp
qed

```

```

lemma fold-Suc-index-shift:  $\text{fold } f [1..<\text{Suc } n] = \text{fold } (\lambda i. f (\text{Suc } i)) [0..<n]$ 
  using fold-index-shift [where n=1 and m=n and f = f]
  by simp

```

```

lemma sum-list-index-shift:  $\text{sum-list } (\text{map } f [n..<n+m]) = \text{sum-list } (\text{map } (\lambda i. f
  (n + i)) [0..<m])$ 
  apply (induct m)
  apply auto
  done

```

```

lemma sum-list-Suc-index-shift:  $\text{sum-list } (\text{map } f [1..<\text{Suc } n]) = \text{sum-list } (\text{map}
  (\lambda i. f (\text{Suc } i)) [0..<n])$ 
  using sum-list-index-shift [where n=1 and m=n and f = f]
  by simp

```

```

lemma upt-Suc-snoc:  $[0..<\text{Suc } n] = [0..<n] @ [n]$ 

```

by (induct n) auto

```
lemma sum-list-le-prefix:
  fixes sz:: nat  $\Rightarrow$  nat
  assumes lower:  $n \leq m$ 
  assumes le:  $m \leq k$ 
  shows sum-list (map sz [n.. $m$ ])  $\leq$  sum-list (map sz [n.. $k$ ])
  apply (subst upt-add-eq-append' [OF lower le])
  apply simp
  done
```

```
lemma intvl-off-disj':
  fixes x :: addr
  assumes ylt:  $y < \text{off}$ 
  and zoff:  $\text{unat } x + \text{off} + z \leq \text{addr-card}$ 
  shows  $\{x \text{ .. } y\} \cap \{x + \text{of-nat } \text{off} \text{ .. } z\} = \{\}$ 
  proof (cases  $\text{unat } x + \text{off} + z = \text{addr-card}$ )
  case True
  then show ?thesis
    using ylt zoff
    apply (simp add: intvl-def)
    apply (safe, clarsimp)
  proof -
    fix k :: nat and ka :: nat
    assume a1:  $\text{unat } x + \text{off} + z = \text{addr-card}$ 
    assume a2:  $y \leq \text{off}$ 
    assume a3:  $k < y$ 
    assume a4:  $(\text{word-of-nat } k :: \text{addr-bitsize word}) = \text{word-of-nat } \text{off} + \text{word-of-nat } ka$ 
    assume ka < z
    then have ka + off < addr-card
      using a1 by linarith
    then show False
      using a4 a3 a2 by (metis (no-types) add.commute add-lessD1 len-of-addr-card
less-le-trans nat-less-le unat-of-nat-eq word-arith-nat-add)
  qed
next
case False
with zoff have le:  $\text{off} + z < \text{addr-card}$  by simp
show ?thesis
  apply (rule intvl-off-disj [OF ylt])
  using le by (simp add: addr-card-wb)
qed
```

```
lemma heap-update-list-padding-fold-partition:
  fixes v:: nat  $\Rightarrow$  byte list  $\Rightarrow$  byte list
  and sz:: nat  $\Rightarrow$  nat
  and off:: nat  $\Rightarrow$  nat
```

```

assumes no-overflow:  $\text{unat } a + \text{length } bs \leq \text{addr-card}$ 
assumes partition:  $bs = \text{concat } (\text{map } (\lambda i. \text{take } (sz \ i) \ (\text{drop } (\text{off } \ i) \ bs))) \ [0..\<m]$ 
assumes lbs:  $\text{length } bs = \text{sum-list } (\text{map } sz \ [0..\<m])$ 
assumes partition-pbs:  $pbs = \text{concat } (\text{map } (\lambda i. \text{take } (sz \ i) \ (\text{drop } (\text{off } \ i) \ pbs))) \ [0..\<m]$ 
assumes lpbs:  $\text{length } pbs = \text{sum-list } (\text{map } sz \ [0..\<m])$ 
assumes off-sz:  $\bigwedge i. i < m \implies \text{off } i = \text{sum-list } (\text{map } sz \ [0..\<i])$ 
assumes val:  $\bigwedge i. i < m \implies$ 
       $v \ i \ (\text{take } (sz \ i) \ (\text{drop } (\text{off } \ i) \ pbs)) =$ 
       $(\text{take } (sz \ i) \ (\text{drop } (\text{off } \ i) \ bs))$ 

shows
   $\text{heap-update-list } a \ bs \ h =$ 
     $\text{fold } (\lambda i \ h. \text{heap-update-list } (a + \text{word-of-nat } (\text{off } \ i))$ 
       $(v \ i \ (\text{take } (sz \ i) \ (\text{drop } (\text{off } \ i) \ pbs)))$ 
       $h)$ 
     $[0..\<m] \ h$ 
using no-overflow partition lbs partition-pbs lpbs off-sz val
proof (induct m arbitrary: a bs pbs h )
case 0
then show ?case by simp
next
case (Suc m)
from Suc.premis obtain
  no-overflow:  $\text{unat } a + \text{length } bs \leq \text{addr-card}$  and
  bs-partition:  $bs = \text{concat } (\text{map } (\lambda i. \text{take } (sz \ i) \ (\text{drop } (\text{off } \ i) \ bs))) \ [0..\<Suc \ m]$ 
and
  pbs-partition:  $pbs = \text{concat } (\text{map } (\lambda i. \text{take } (sz \ i) \ (\text{drop } (\text{off } \ i) \ pbs))) \ [0..\<Suc \ m]$ 
and
  lbs:  $\text{length } bs = \text{sum-list } (\text{map } sz \ [0..\<Suc \ m])$  and
  lpbs:  $\text{length } pbs = \text{sum-list } (\text{map } sz \ [0..\<Suc \ m])$  and
  off-sz:  $\bigwedge i. i < Suc \ m \implies \text{off } i = \text{sum-list } (\text{map } sz \ [0..\<i])$  and
  val:  $\bigwedge i. i < Suc \ m \implies$ 
     $v \ i \ (\text{take } (sz \ i) \ (\text{drop } (\text{off } \ i) \ pbs)) =$ 
     $\text{take } (sz \ i) \ (\text{drop } (\text{off } \ i) \ bs)$ 
by clarsimp
have unroll:  $[0..\<Suc \ m] = [0..\<m] \ @ \ [m]$ 
by (rule upt-Suc-snoc)

define bs' where  $bs' = \text{concat } (\text{map } (\lambda i. \text{take } (sz \ i) \ (\text{drop } (\text{off } \ i) \ bs))) \ [0..\<m]$ 
define bsm where  $bsm = \text{take } (sz \ m) \ (\text{drop } (\text{off } \ m) \ bs)$ 

define pbs' where  $pbs' = \text{concat } (\text{map } (\lambda i. \text{take } (sz \ i) \ (\text{drop } (\text{off } \ i) \ pbs))) \ [0..\<m]$ 
define pbsm where  $pbsm = \text{take } (sz \ m) \ (\text{drop } (\text{off } \ m) \ pbs)$ 

from bs-partition have bs:  $bs = bs' \ @ \ bsm$ 
unfolding bsm-def bs'-def
apply (subst (asm) unroll)
apply simp

```

```

done

from pbs-partition have pbs: pbs = pbs' @ pbsm
  unfolding pbsm-def pbs'-def
  apply (subst (asm) unroll)
  apply simp
done

from lbs have lbs1: length bs = sum-list (map sz [0..\bigwedge i. i < m \implies \text{off } i = \text{sum-list (map sz [0..) by
auto

{
  fix i
  assume bound: i < m
  hence bound': i < Suc m by simp
  from val [OF this]
  have v-i: v i (take (sz i) (drop (off i) pbs)) =

```

$take\ (sz\ i)\ (drop\ (off\ i)\ bs)$.

from $off\text{-}sz\ [OF\ bound']$ **have** $off\text{-}i$: $off\ i = sum\text{-}list\ (map\ sz\ [0..\lt i])$.
from $off\text{-}sz\ [of\ m]$ **have** $off\text{-}m$: $off\ m = sum\text{-}list\ (map\ sz\ [0..\lt m])$ **by** $simp$

from $bound$ **have** $Suc\ i \leq m$ **by** $simp$
from $sum\text{-}list\text{-}le\text{-}prefix$ [**where** $n=0$, OF - $this$]
have $*$: $sum\text{-}list\ (map\ sz\ [0..\lt Suc\ i]) \leq sum\text{-}list\ (map\ sz\ [0..\lt m])$ **by** $simp$
then **have** $bound\text{-}bs'$: $off\ i + sz\ i \leq length\ bs'$ **using** $off\text{-}i\ lbs'$ **by** $simp$
from $*$ **have** $bound\text{-}pbs'$: $off\ i + sz\ i \leq length\ pbs'$ **using** $off\text{-}i\ lpbs'$ **by** $simp$

have $take\text{-}eq1$: $take\ (sz\ i)\ (drop\ (off\ i)\ bs) = take\ (sz\ i)\ (drop\ (off\ i)\ bs')$
apply $(subst\ bs)$
apply $(subst\ take\text{-}drop\text{-}append\text{-}first\ [OF\ bound\text{-}bs'])$
apply $simp$
done

have $take\text{-}eq2$: $take\ (sz\ i)\ (drop\ (off\ i)\ pbs) = take\ (sz\ i)\ (drop\ (off\ i)\ pbs')$
apply $(subst\ pbs)$
apply $(subst\ take\text{-}drop\text{-}append\text{-}first\ [OF\ bound\text{-}pbs'])$
apply $simp$
done

from $v\text{-}i$
have $v\ i\ (take\ (sz\ i)\ (drop\ (off\ i)\ pbs')) =$
 $take\ (sz\ i)\ (drop\ (off\ i)\ bs')$
by $(simp\ add: take\text{-}eq1\ take\text{-}eq2)$
} note $val' = this$

from $no\text{-}overflow$
have $no\text{-}overflow'$: $unat\ a + length\ bs' \leq addr\text{-}card$
by $(simp\ add: bs)$
{
fix i
assume $bound$: $i < m$
hence $bound'$: $i < Suc\ m$ **by** $simp$

from $bound$ **have** $Suc\ i \leq m$ **by** $simp$
from $sum\text{-}list\text{-}le\text{-}prefix$ [**where** $n=0$, OF - $this$]
have $sum\text{-}le$: $sum\text{-}list\ (map\ sz\ [0..\lt Suc\ i]) \leq sum\text{-}list\ (map\ sz\ [0..\lt m])$ **by**
 $simp$

from $off\text{-}sz\ [OF\ bound']$ **have** $off\text{-}i$: $off\ i = sum\text{-}list\ (map\ sz\ [0..\lt i])$.
have $take\ (sz\ i)\ (drop\ (off\ i)\ bs) = take\ (sz\ i)\ (drop\ (off\ i)\ bs')$
apply $(subst\ bs)$
using $lbs'\ sum\text{-}le\ off\text{-}i$
apply $simp$
done

} note $take\text{-}drop\text{-}eq = this$

from $take\text{-}drop\text{-}eq$

```

have map-eq: map ( $\lambda i.$  take (sz i) (drop (off i) bs)) [0.. $m$ ] =
      map ( $\lambda i.$  take (sz i) (drop (off i) bs')) [0.. $m$ ]
by simp

{
  fix i
  assume bound:  $i < m$ 
  hence bound':  $i < \text{Suc } m$  by simp

  from bound have Suc  $i \leq m$  by simp
  from sum-list-le-prefix [where  $n=0$ , OF - this]
  have sum-le: sum-list (map sz [0.. $\text{Suc } i$ ])  $\leq$  sum-list (map sz [0.. $m$ ]) by
simp
  from off-sz [OF bound'] have off-i: off i = sum-list (map sz [0.. $i$ ]).
  have take (sz i) (drop (off i) pbs) = take (sz i) (drop (off i) pbs')
  apply (subst pbs)
  using lpbs' sum-le off-i
  apply simp
  done
} note take-drop-eq' = this

from take-drop-eq'
have map-eq': map ( $\lambda i.$  take (sz i) (drop (off i) pbs)) [0.. $m$ ] =
      map ( $\lambda i.$  take (sz i) (drop (off i) pbs')) [0.. $m$ ]
by simp

from bs-partition have bs'-partition: bs' = concat (map ( $\lambda i.$  take (sz i) (drop
(off i) bs')) [0.. $m$ ])
  apply (subst map-eq [symmetric])
  apply (simp add: bs'-def)
  done

from pbs-partition have pbs'-partition: pbs' = concat (map ( $\lambda i.$  take (sz i) (drop
(off i) pbs')) [0.. $m$ ])
  apply (subst map-eq' [symmetric])
  apply (simp add: pbs'-def)
  done

from lbs' off-sz [of m]
have off-m: off m = length bs'
by simp

have vm-eq: ( $v$  m (take (sz m) (drop (length bs') pbs')) @
take (sz m - (length pbs' - length bs')) (drop (length bs' - length
pbs') pbsm))) = bsm
by (metis bsm-def drop-append length-drop lessI off-m pbs take-append val)

```

```

have fold-pbs':
  (fold
    (λi. heap-update-list (a + word-of-nat (off i)) (v i (take (sz i) (drop (off i)
pbs))))
    [0..<m] h) =
  (fold
    (λi. heap-update-list (a + word-of-nat (off i)) (v i (take (sz i) (drop (off i)
pbs^))))
    [0..<m] h)
  apply (rule fold-cong)
  using take-drop-eq' by auto

```

```

note hyp = Suc.hyps [where a=a and bs=bs' and pbs=pbs' and h=h, OF
no-overflow' bs'-partition lbs' pbs'-partition lpbs' off-sz' val']

```

```

show ?case
  apply (subst unroll)
  apply (subst fold-append)
  apply (simp only: comp-def)
  apply (subst bs)
  apply (subst fold-pbs')
  apply (subst pbs)
  apply (subst heap-update-list-concat-unfold)
  apply (subst hyp [ symmetric])
  apply assumption
  apply (simp add: off-m vm-eq)
  done

```

qed

lemma heap-update-list-fold-partition:

```

fixes v:: nat ⇒ byte list ⇒ byte list
  and sz:: nat ⇒ nat
  and off:: nat ⇒ nat

```

assumes no-overflow: unat a + length bs ≤ addr-card

assumes partition: bs = concat (map (λi. take (sz i) (drop (off i) bs)) [0..<m])

assumes lbs: length bs = sum-list (map sz [0..<m])

assumes off-sz: $\bigwedge i. i < m \implies \text{off } i = \text{sum-list } (\text{map } sz [0..<i])$

assumes val: $\bigwedge i. i < m \implies$

$$v \ i \ (\text{take } (sz \ i) \ (\text{drop } (off \ i) \ (\text{heap-list } h \ (\text{length } bs) \ a))) =$$

$$(\text{take } (sz \ i) \ (\text{drop } (off \ i) \ bs))$$

shows

```

heap-update-list a bs h =
  fold (λi h. heap-update-list
    (a + word-of-nat (off i))
    (v i ((heap-list h (sz i) (a + word-of-nat (off i)))))
    h)
  [0..<m] h

```

using no-overflow partition lbs off-sz val

proof (induct m arbitrary: a bs h)

case 0

```

then show ?case by simp
next
case (Suc m)
from Suc.prem1 obtain
  no-overflow: unat a + length bs ≤ addr-card and
  bs-partition: bs = concat (map (λi. take (sz i) (drop (off i) bs)) [0..

```



```

have v-i: v i (take (sz i) (drop (off i) (heap-list h (length bs) a))) =
  take (sz i) (drop (off i) bs) .

from off-sz [OF bound'] have off-i: off i = sum-list (map sz [0..<i]).
from off-sz [of m] have off-m: off m = sum-list (map sz [0..<m]) by simp

from bound have Suc i ≤ m by simp
from sum-list-le-prefix [where n=0, OF - this]
have sum-list (map sz [0..<Suc i]) ≤ sum-list (map sz [0..<m]) by simp
then have bound-bs': off i + sz i ≤ length bs' using off-i lbs' by simp

have take-eq1: take (sz i) (drop (off i) bs) = take (sz i) (drop (off i) bs')
  apply (subst bs)
  apply (subst take-drop-append-first [OF bound-bs'])
  apply simp
  done
have take-eq2:
  take (sz i) (drop (off i) (heap-list h (length bs) a)) =
    take (sz i) (drop (off i) (heap-list h (length bs') a))
  apply (subst lbs1)
  apply (subst lbs')
  apply (subst heap-list-split2)
  using bound-bs' [simplified lbs']
  apply simp
  done
from v-i
have v i (take (sz i) (drop (off i) (heap-list h (length bs') a))) =
  take (sz i) (drop (off i) bs')
  by (simp add: take-eq1 take-eq2)
} note val' = this

from no-overflow
have no-overflow': unat a + length bs' ≤ addr-card
  by (simp add: bs)
{
  fix i
  assume bound: i < m
  hence bound': i < Suc m by simp

from bound have Suc i ≤ m by simp
from sum-list-le-prefix [where n=0, OF - this]
have sum-le: sum-list (map sz [0..<Suc i]) ≤ sum-list (map sz [0..<m]) by
simp
from off-sz [OF bound'] have off-i: off i = sum-list (map sz [0..<i]).
have take (sz i) (drop (off i) bs) = take (sz i) (drop (off i) bs')
  apply (subst bs)

```

```

    using bs' sum-le off-i
    apply simp
    done
} note take-drop-eq = this

from take-drop-eq
have map-eq:  $\text{map } (\lambda i. \text{take } (sz\ i) (\text{drop } (\text{off } i) bs)) [0..<m] =$ 
            $\text{map } (\lambda i. \text{take } (sz\ i) (\text{drop } (\text{off } i) bs')) [0..<m]$ 
by simp

from bs-partition have bs'-partition:  $bs' = \text{concat } (\text{map } (\lambda i. \text{take } (sz\ i) (\text{drop } (\text{off } i) bs')) [0..<m])$ 
apply (subst map-eq [symmetric])
apply (simp add: bs'-def)
done

from bs' off-sz [of m]
have off-m:  $\text{off } m = \text{length } bs'$ 
by simp

have vm-eq:  $v\ m (\text{heap-list } (\text{heap-update-list } a\ bs'\ h) (sz\ m) (a + \text{word-of-nat } (\text{length } bs')))) = bsm$ 
proof -
  from val [of m]
  have  $v\ m (\text{take } (sz\ m) (\text{drop } (\text{off } m) (\text{heap-list } h (\text{length } bs) a))) = bsm$  by
  (simp add: bsm-def)
  moreover
  have disj:  $\{a..+\text{length } bs'\} \cap \{a + \text{word-of-nat } (\text{length } bs')..+sz\ m\} = \{\}$ 
  apply (simp add: bs')
  apply (rule intvl-off-disj')
  apply simp
  using no-overflow [simplified bs1]
  apply (simp)
  done

have  $\text{heap-list } (\text{heap-update-list } a\ bs'\ h) (sz\ m) (a + \text{word-of-nat } (\text{length } bs'))$ 
=
   $\text{take } (sz\ m) (\text{drop } (\text{off } m) (\text{heap-list } h (\text{length } bs) a))$ 
  apply (subst heap-list-update-disjoint-same [OF disj])
  apply (subst bs1)
  apply (subst bs')
  apply (subst heap-list-take-drop' [OF no-overflow [simplified bs1]])
  apply (simp add: off-m [simplified bs'])
  apply (simp add: off-m [simplified bs'])
  done

ultimately show ?thesis by simp
qed

```

```

show ?case
  apply (subst unroll)
  apply (subst fold-append)
  apply (simp only: comp-def)
  apply (subst bs)
  apply (subst heap-update-list-concat-unfold)
  apply (subst Suc.hyps [where a=a and bs=bs' and h=h, OF no-overflow'
bs'-partition lbs' off-sz' val', symmetric])
  apply assumption
  apply (simp add: off-m vm-eq)
  done
qed

```

lemma *heap-list-map-partition*:

```

fixes sz:: nat  $\Rightarrow$  nat
  and off:: nat  $\Rightarrow$  nat
assumes no-overflow: unat a + n  $\leq$  addr-card
assumes lbs: n = sum-list (map sz [0.. $m$ ])
assumes off-sz:  $\bigwedge i. i < m \Rightarrow$  off i = sum-list (map sz [0.. $i$ ])
shows
  heap-list h n a =
    concat (map ( $\lambda i. \text{heap-list } h (sz\ i) (a + \text{word-of-nat } (off\ i))$ ) [0.. $m$ ])
using no-overflow lbs off-sz
proof (induct m arbitrary: n a)
  case 0
  then show ?case by simp
next
  case (Suc m)
  from Suc.prem1 obtain
    no-overflow: unat a + n  $\leq$  addr-card and
    lbs: n = sum-list (map sz [0.. $Suc\ m$ ]) and
    off-sz:  $\bigwedge i. i < Suc\ m \Rightarrow$  off i = sum-list (map sz [0.. $i$ ])
  by clarsimp
  have unroll: [0.. $Suc\ m$ ] = [0.. $m$ ] @ [m]
  by (rule upt-Suc-snoc)

  from lbs have lbs1: n = sum-list (map sz [0.. $m$ ]) + sz m
  apply (subst (asm) unroll)
  apply simp
  done

  from no-overflow lbs1
  have no-overflow': unat a + (n - sz m)  $\leq$  addr-card
  by simp
  from lbs1
  have lbs1': n - sz m = sum-list (map sz [0.. $m$ ])
  by simp
  from lbs1 lbs1'

```

```

have sz-m: (n - (n - sz m)) = sz m
  by simp

from heap-list-split [where k=n - sz m and n = n and h=h and x=a]
have split: heap-list h n a = heap-list h (n - sz m) a @ heap-list h (sz m) (a +
word-of-nat (n - sz m))
  by (simp add: sz-m)

have hyp: heap-list h (n - sz m) a = concat (map (λi. heap-list h (sz i) (a +
word-of-nat (off i))) [0..<m])
  apply (rule Suc.hyps [OF no-overflow' lbs1 ])
  using off-sz
  by auto

from off-sz [of m] lbs1
have off-m: (n - sz m) = off m
  by simp
show ?case
  apply (subst split)
  apply (subst unroll)
  apply (simp add: hyp )
  apply (simp add: off-m)
  done
qed

lemma array-partition:
assumes lbs: length bs = m * n
shows concat (map (λi. take n (drop (i * n) bs)) [0..<m]) = bs
  using lbs
proof (induct m arbitrary: bs n)
  case 0
  then show ?case by simp
next
  case (Suc m)
  have lbs: length bs = Suc m * n by fact
  hence lbs: length bs = (m * n) + n
    by simp
  define bs' where bs' = take (m * n) bs
  define bs-last where bs-last = drop (m * n) bs
  from lbs have bs: bs = bs' @ bs-last
    by (simp add: bs'-def bs-last-def)

from lbs bs
have lbs': length bs' = m * n
  by (metis add-diff-cancel-left' add-diff-cancel-right' bs-last-def eq-concat-lenD
length-drop)

from lbs bs lbs' have lbs-last: length bs-last = n
  by auto

```

```

{
  fix i
  assume bound: i < m
  have take n (drop (i * n) bs) = take n (drop (i * n) bs')
  apply (subst bs)
  using lbs' lbs-last bound
  apply simp
  by (metis Suc-leI mult-Suc mult-le-mono order-refl)
}
then
  have map-eq: (map (λi. take n (drop (i * n) bs)) [0..<m]) = (map (λi. take n
(drop (i * n) bs')) [0..<m])
  by simp

  note hyp = Suc.hyps [OF lbs']
  show ?case
  apply simp
  apply (simp add: map-eq hyp)
  apply (simp add: bs-last-def [symmetric])
  using lbs-last
  apply (simp add: bs)
  done
qed

lemma sum-list-const-fun: sum-list (map (λ-. n::nat) [0..<m]) = m * n
  by (induct m) auto

lemmas export-uinfo-adjust-ti [simp] = export-tag-adjust-ti(1)[rule-format]

lemma heap-update-list-array:
  fixes arr:: ('a::array-outer-max-size['b::array-max-count])
  fixes p:: ('a['b]) ptr
  assumes cgrd: c-guard p
  shows
    heap-update-list (ptr-val p) (to-bytes arr (heap-list h (size-of TYPE('a['b])) (ptr-val
p))) h =
      fold
        (λi h. heap-update-list (ptr-val (array-ptr-index p False i))
          (to-bytes (arr.[i])
            (heap-list h (size-of TYPE('a)) (ptr-val (array-ptr-index p False i))))
          h)
        [0..<CARD('b)] h
proof -
  define xbs where
    xbs = (to-bytes arr (heap-list h (size-of TYPE('a['b])) (ptr-val p)))
  define sz::nat ⇒ nat where
    sz = (λ-. size-of TYPE('a))

```

```

define off::nat  $\Rightarrow$  nat where
  off = ( $\lambda i. i * \text{size-of } \text{TYPE}('a)$ )
define v::nat  $\Rightarrow$  byte list  $\Rightarrow$  byte list where
  v = ( $\lambda i \text{ bs. } (\text{to-bytes } (\text{arr}.[i]) \text{ bs})$ )

have lbs: length lbs = CARD('b) * size-of TYPE('a)
  by (simp add: lbs-def)

from c-guard-no-wrap' [OF cgrd] lbs
have no-overflow: unat (ptr-val p) + length lbs  $\leq$  addr-card
  by simp

have partition: lbs = concat (map ( $\lambda i. \text{take } (\text{sz } i) (\text{drop } (\text{off } i) \text{ lbs})$ ) [0..CARD('b)])

  apply (simp add: sz-def off-def)
  apply (rule array-partition [symmetric, OF lbs])
  done

have sum-list (map sz [0..CARD('b)]) = CARD('b) * size-of TYPE('a)
  apply (simp add: sz-def)
  apply (rule sum-list-const-fun)
  done

with lbs have lbs-sum: length lbs = sum-list (map sz [0..CARD('b)]) by
simp

have off-sz: ( $\bigwedge i. i < \text{CARD}('b) \implies \text{off } i = \text{sum-list } (\text{map } \text{sz } [0..i])$ )
  unfolding off-def sz-def using sum-list-const-fun
  by auto

{
  fix i
  assume i-bound: i < CARD('b)
  have eq-padding:
    eq-padding (typ-uinfo-t TYPE('a['b]))
      lbs
      (heap-list h (size-of TYPE('a['b])) (ptr-val p))
  using eq-padding-to-bytes heap-list-length lbs-def by blast

  from field-lookup-array [OF i-bound, where i=0, simplified]
  have fl: field-lookup (typ-uinfo-t TYPE('a['b])) [replicate i CHR "1"] 0 =
    Some (adjust-ti (typ-uinfo-t TYPE('a)) ( $\lambda x. x.[i]$ ) ( $\lambda x f. \text{Arrays.update } f$ 
i x), i * size-of TYPE('a)) .

  have exp:
    (export-uinfo
      (adjust-ti (typ-uinfo-t TYPE('a)) ( $\lambda x::'a['b]. (x.[i])$ ) ( $\lambda x f. \text{Arrays.update}$ 
f i x))) =
      typ-uinfo-t TYPE('a)

```

```

apply (subst export-uinfo-adjust-ti)
  apply (rule fg-cons-array [OF i-bound] )
  apply simp
apply (simp add: typ-uinfo-t-def)
done

from xmem-type-field-lookup-eq-padding-focus [OF fl - - eq-padding ]
have
  eq-padding (typ-uinfo-t TYPE('a))
    (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) xbs))
    (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) (heap-list h (size-of
TYPE('a['b])) (ptr-val p))))
  using lxs
  by simp (simp add: exp size-of-def)

hence eq1:
  to-bytes (arr.[i]) (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) xbs))
=
  to-bytes (arr.[i]) (take (size-of TYPE('a)) (drop (i * size-of TYPE('a))
(heap-list h (size-of TYPE('a['b])) (ptr-val p))))
  by (meson eq-padding-to-bytes-eq)
have v i (take (sz i) (drop (off i) (heap-list h (length xbs) (ptr-val p)))) =
  take (sz i) (drop (off i) xbs)
  unfolding v-def lxs sz-def off-def
  apply (simp add: xbs-def)

  using access-ti-array-index' [OF i-bound lxs, where arr=arr, OF refl,
simplified xbs-def, simplified to-bytes-def [symmetric]]
  apply (simp only: xbs-def [symmetric])
  apply (subst (asm) eq1)
  apply simp
  using eq-padding eq-padding-to-bytes-eq by fastforce
} note v-padding = this

{
  fix i
  assume i-bound: i < CARD('b)
  have ptr-val p + word-of-nat i * word-of-nat (size-of TYPE('a))
=
  ptr-val (PTR-COERCE('a['b] → 'a) p +p int i)
  by (simp add: CTypesDefs.ptr-add-def)
} note deref-conv = this
note partition = heap-update-list-fold-partition [where a=ptr-val p and bs=xbs
and sz=sz and off=off and v=v
and h=h and
m=CARD('b), OF no-overflow partition lxs-sum off-sz v-padding, simplified]
show ?thesis
  apply (simp only: xbs-def [symmetric])

```

```

apply (subst partition)
apply (rule fold-cong)
  apply simp
  apply simp
apply (simp add: v-def sz-def off-def array-ptr-index-def deref-conv)
done
qed

```

```

lemma heap-update-array:
  fixes arr:: ('a::array-outer-max-size['b::array-max-count])
  fixes p:: ('a['b]) ptr
  assumes cgrd: c-guard p
  shows
heap-update p arr h =
  fold
    ( $\lambda i h. \text{heap-update } (\text{array-ptr-index } p \text{ False } i) (\text{arr}.[i]) h$ )
    [ $0..< \text{CARD}('b)$ ] h
  using heap-update-list-array [OF cgrd]
  by (simp only: heap-update-def)

```

```

lemma heap-update-array-pointless:
  fixes arr:: ('a::array-outer-max-size['b::array-max-count])
  fixes p:: ('a['b]) ptr
  assumes cgrd: c-guard p
  shows
heap-update p arr =
  fold
    ( $\lambda i h. \text{heap-update } (\text{array-ptr-index } p \text{ False } i) (\text{arr}.[i]) h$ )
    [ $0..< \text{CARD}('b)$ ]
  using heap-update-array [OF cgrd]
  by (simp add: fun-eq-iff)

```

```

lemma heap-update-padding-list-array:
  fixes arr:: ('a::array-outer-max-size['b::array-max-count])
  fixes p:: ('a['b]) ptr
  assumes cgrd: c-guard p
  assumes lbs: length bs = CARD('b) * size-of TYPE('a)
  shows
heap-update-list (ptr-val p) (to-bytes arr bs) h =
  fold
    ( $\lambda i h. \text{heap-update-list } (\text{ptr-val } (\text{array-ptr-index } p \text{ False } i))$ 
      (to-bytes (arr.[i])
        (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs)))
      h)
    [ $0..< \text{CARD}('b)$ ] h
proof –
  define xbs where

```



```

  xbs = (to-bytes arr bs)
define sz::nat ⇒ nat where
  sz = (λ-. size-of TYPE('a))
define off::nat ⇒ nat where
  off = (λi. i * size-of TYPE('a))
define v::nat ⇒ byte list ⇒ byte list where
  v = (λi bs. (to-bytes (arr.[i] bs))
have lxs: length xbs = CARD('b) * size-of TYPE('a)
  by (simp add: xbs-def lbs)

from c-guard-no-wrap' [OF cgrd] lxs
have no-overflow: unat (ptr-val p) + length xbs ≤ addr-card
  by simp

have partition: xbs = concat (map (λi. take (sz i) (drop (off i) xbs)) [0..apply (simp add: sz-def off-def)
  apply (rule array-partition [symmetric, OF lxs])
  done

have partition-bs: bs = concat (map (λi. take (sz i) (drop (off i) bs)) [0..apply (simp add: sz-def off-def)
  apply (rule array-partition [symmetric, OF lbs])
  done

have *: sum-list (map sz [0..apply (simp add: sz-def)
  apply (rule sum-list-const-fun)
  done

  with lxs have lxs-sum: length xbs = sum-list (map sz [0..by
  simp

from lbs * have lbs-sum: length bs = sum-list (map sz [0..by simp
have off-sz: (∧i. i < CARD('b) ⇒ off i = sum-list (map sz [0..unfolding off-def sz-def using sum-list-const-fun
  by auto

have val-eq: ∑i. i < CARD('b) ⇒ v i (take (sz i) (drop (off i) bs)) = take (sz
i) (drop (off i) xbs)
  unfolding off-def sz-def v-def xbs-def
  apply (simp add: to-bytes-def)
  apply (subst access-ti-array-index' [where xbs=bs, OF - lbs])
  apply auto
  done

have val-eq':

```

```

 $\bigwedge i. i < \text{CARD}('b) \implies (v\ i\ (\text{take}\ (sz\ i)\ (\text{drop}\ (\text{off}\ i)\ bs))) =$ 
  (to-bytes (arr.[i]) (take (size-of TYPE('a)) (drop (i * size-of TYPE('a))
bs)))
  unfolding off-def sz-def v-def
  by simp

have idx-eq:  $\bigwedge i. i < \text{CARD}('b) \implies$ 
  (ptr-val p + word-of-nat (off i)) = (ptr-val (array-ptr-index p False i))
  unfolding off-def array-ptr-index-def
  by (auto simp add: c-type-class.ptr-add-def)

note partition = heap-update-list-padding-fold-partition [where a=ptr-val p and
bs= xbs and pbs = bs and sz=sz and off=off and v=v and h=h and m=CARD('b),
OF no-overflow partition lxs-sum partition-bs lxs-sum off-sz val-eq]
show ?thesis
  apply (simp only: xbs-def [symmetric])
  apply (subst partition)
  apply (assumption)
  apply (rule fold-cong)
  apply (auto simp add: val-eq' idx-eq)
done
qed

```

```

lemma heap-update-padding-array:
  fixes arr:: ('a::array-outer-max-size['b::array-max-count])
  fixes p:: ('a['b]) ptr
  assumes cgrd: c-guard p
  assumes lbs: length bs = CARD('b) * size-of TYPE('a)
  shows
    heap-update-padding p arr bs h =
      fold
        ( $\lambda i\ h. \text{heap-update-padding}\ (\text{array-ptr-index}\ p\ \text{False}\ i)\ (\text{arr}.[i])\ (\text{take}\ (\text{size-of}\ \text{TYPE}('a))\ (\text{drop}\ (i * \text{size-of}\ \text{TYPE}('a))\ bs))\ h)$ 
        [0..<CARD('b)] h
      using heap-update-padding-list-array [OF cgrd lbs]
      by (simp only: heap-update-padding-def)

```

```

lemma heap-update-padding-array-pointless:
  fixes arr:: ('a::array-outer-max-size['b::array-max-count])
  fixes p:: ('a['b]) ptr
  assumes cgrd: c-guard p
  assumes lbs: length bs = CARD('b) * size-of TYPE('a)
  shows
    heap-update-padding p arr bs =
      fold
        ( $\lambda i\ h. \text{heap-update-padding}\ (\text{array-ptr-index}\ p\ \text{False}\ i)\ (\text{arr}.[i])\ (\text{take}\ (\text{size-of}\ \text{TYPE}('a))\ (\text{drop}\ (i * \text{size-of}\ \text{TYPE}('a))\ bs))\ h)$ 
        [0..<CARD('b)]

```

```

using heap-update-padding-array [OF cgrd lbs]
by (simp add: fun-eq-iff)

lemma c-guard-array-ptr-index:
  fixes p: (('a :: mem-type)['b :: finite]) ptr
  assumes cgrd: c-guard p
  assumes bound: n < CARD('b)
  shows c-guard (array-ptr-index p coerce n)
  using cgrd bound
  apply (simp add: array-ptr-index-def c-guard-def)
  apply (clarsimp, safe)
  subgoal
    apply (simp add: ptr-aligned-def)
    by (metis ptr-aligned-def ptr-aligned-plus ptr-val-ptr-coerce)
  subgoal
    supply fun-upd-apply[simp del]
    apply (simp add: c-null-guard-def)
    apply (simp add: CTypesDefs.ptr-add-def field-simps)
    by (auto simp add: intvl-def)
    (metis (no-types, opaque-lifting) Abs-fnat-hom-mult add commute group-cancel.add1
nat-index-bound word-of-nat-plus)
  done

lemma heap-update-array-update:
  assumes n: n < CARD('b :: array-max-count)
  assumes size: CARD('b) * size-of TYPE('a :: array-outer-max-size) < 2 ^
addr-bitsize
  assumes cgrd: c-guard p
  shows heap-update p (Arrays.update (arr :: 'a['b]) n v) hp
    = heap-update (array-ptr-index p False n) v (heap-update p arr hp)
proof -

  have P:  $\bigwedge x k. \llbracket x < \text{CARD}('b); k < \text{size-of TYPE}('a) \rrbracket$ 
     $\implies \text{unat}(\text{of-nat } x * \text{of-nat}(\text{size-of TYPE}('a)) + (\text{of-nat } k :: \text{addr}))$ 
    = x * size-of TYPE('a) + k
  using size
  supply unsigned-of-nat [simp del]
  apply (cases size-of TYPE('a), simp-all)
  apply (cases CARD('b), simp-all)
  apply (subst unat-add-lem[THEN iffD1])
  apply (simp add: unat-word-ariths unat-of-nat less-Suc-eq-le)
  subgoal premises prems for x k nat nata
  proof -
    have Suc x * size-of TYPE('a) < 2 ^ addr-bitsize
      using prems
      apply simp
      apply (erule order-le-less-trans[rotated], simp add: add-mono)
      done
    then show ?thesis using prems by simp

```

```

qed
apply (subst unat-mult-lem[THEN iffD1])
apply (simp add: unat-of-nat unat-add-lem[THEN iffD1])
apply (rule order-less-le-trans, erule order-le-less-trans[rotated],
      rule add-mono, simp+)
  apply (simp add: less-Suc-eq-le trans-le-add2)
  apply simp
apply (simp add: unat-of-nat unat-add-lem[THEN iffD1])
done

let ?key-upd = heap-update (array-ptr-index p False n) v
note commute = fold-commute-apply[where h=?key-upd
  and xs=[Suc n ..< CARD('b)], where g=f' and f=f' for f']

from cgrd n
have cgrd': c-guard (array-ptr-index p False n)
  by (rule c-guard-array-ptr-index)

show ?thesis using n
  apply (simp add: heap-update-array [OF cgrd] split-upt-on-n[OF n]
        )
  apply (subst commute)
  subgoal for x
    apply (rule ext, simp)
    apply (rule heap-update-commute, simp-all add: ptr-add-def)
  apply (simp add: array-ptr-index-def CTypesDefs.ptr-add-def intvl-def Suc-le-eq)
  apply (rule set-eqI, clarsimp)
  apply (drule word-unat.Rep-inject[THEN iffD2])
  apply (clarsimp simp: P nat-eq-add-iff1)
  apply (cases x, simp-all add: less-Suc-eq-le Suc-diff-le)
  done
  subgoal
    apply (subst heap-update-collapse)
    apply (simp cong: fold-cong')
  done
done
qed

lemma heap-update-array-element':
  fixes p' :: (('a :: array-outer-max-size)['b::array-max-count]) ptr
  fixes p :: ('a :: array-outer-max-size) ptr
  fixes hp w
  assumes p: p = array-ptr-index p' False n
  assumes n: n < CARD('b)
  assumes cgrd: c-guard p'
  assumes size: CARD('b) * size-of TYPE('a) < 2 ^ addr-bitsize
  shows heap-update p' (Arrays.update (h-val hp p') n w) hp
    = heap-update p w hp

```

```

apply (subst heap-update-array-update[OF n size cgrd])
apply (simp add: heap-update-id p)
done

```

```

lemmas heap-update-array-element'
  = heap-update-array-element'[simplified array-ptr-index-simps]

```

```

lemma (in array-outer-max-size) array-count-size:
  CARD('b :: array-max-count) * size-of TYPE('a) < 2 ^ addr-bitsize
using array-outer-max-size-ax[] array-max-count-ax[where 'a='b]
apply (clarsimp dest!: nat-le-Suc-less-imp)
apply (drule(1) mult-mono, simp+)
done

```

```

lemmas heap-update-array-element
  = heap-update-array-element'[OF refl - - array-count-size]

```

primrec

```

field-names-u :: typ-uinfo ⇒ typ-uinfo ⇒
  (qualified-field-name) list and
field-names-struct-u :: typ-uinfo-struct ⇒ typ-uinfo ⇒
  (qualified-field-name) list and
field-names-list-u :: typ-uinfo-tuple list ⇒ typ-uinfo ⇒
  (qualified-field-name) list and
field-names-tuple-u :: typ-uinfo-tuple ⇒ typ-uinfo ⇒
  (qualified-field-name) list

```

where

```

tufs0: field-names-u (TypDesc align st nm) t = (if t = (TypDesc align st nm) then
  [[]] else field-names-struct-u st t)

```

```

| tufs1: field-names-struct-u (TypScalar m align d) t = []
| tufs2: field-names-struct-u (TypAggregate xs) t = field-names-list-u xs t

```

```

| tufs3: field-names-list-u [] t = []
| tufs4: field-names-list-u (x#xs) t = field-names-tuple-u x t@field-names-list-u xs
t

```

```

| tufs5: field-names-tuple-u (DTuple s f d) t = map (λfs. f#fs) (field-names-u s t)

```

lemma field-names-u-field-names-export-uinfo-conv:

```

fixes t :: ('a, 'b) typ-uinfo and
  st :: ('a, 'b) typ-uinfo-struct and
  ts :: ('a, 'b) typ-uinfo-tuple list and
  x :: ('a, 'b) typ-uinfo-tuple

```

shows

```

field-names-u (export-uinfo t) s = field-names t s
field-names-struct-u (map-td-struct field-norm (λ-. ()) st) s = field-names-struct
st s
field-names-list-u (map-td-list field-norm (λ-. ()) ts) s = field-names-list ts s

```

field-names-tuple-u (map-td-tuple field-norm (λ-. ()) x) s = *field-names-tuple* x^s

by (induct t and st and ts and x) (auto simp add: export-uinfo-def)

primrec

all-field-names :: ('a, 'b) typ-desc ⇒

(qualified-field-name) list and

all-field-names-struct :: ('a, 'b) typ-struct ⇒

(qualified-field-name) list and

all-field-names-list :: (('a, 'b) typ-desc, field-name, 'b) dt-tuple list ⇒

(qualified-field-name) list and

all-field-names-tuple :: (('a, 'b) typ-desc, field-name, 'b) dt-tuple ⇒

(qualified-field-name) list

where

afs0: *all-field-names* (TypDesc align st nm) =

[] @ *all-field-names-struct* st

| *afs1*: *all-field-names-struct* (TypScalar m align d) = []

| *afs2*: *all-field-names-struct* (TypAggregate xs) = *all-field-names-list* xs

| *afs3*: *all-field-names-list* [] = []

| *afs4*: *all-field-names-list* (x#xs) = *all-field-names-tuple* x @ *all-field-names-list* xs

| *afs5*: *all-field-names-tuple* (DTuple s f d) = map (λfs. f#fs) (*all-field-names* s)

lemma field-lookup-all-field-names:

fixes t::('a, 'b) typ-desc

and st::('a, 'b) typ-struct

and ts::('a, 'b) typ-tuple list

and x::('a, 'b) typ-tuple

shows

field-lookup t f m = Some (s, n) ⇒ f ∈ set (*all-field-names* t) and

field-lookup-struct st f m = Some (s, n) ⇒ f ∈ set (*all-field-names-struct* st)

and

field-lookup-list ts f m = Some (s, n) ⇒ f ∈ set (*all-field-names-list* ts) and

field-lookup-tuple x f m = Some (s, n) ⇒ f ∈ set (*all-field-names-tuple* x)

proof (induct t and st and ts and x arbitrary: f m n s and f m n s and f m n s and f m n s)

case (TypDesc nat typ-struct list)

then show ?case by (auto split: if-split-asm)

next

case (TypScalar nat1 nat2 a)

then show ?case by auto

next

case (TypAggregate list)

then show ?case by auto

next

```

case Nil-typ-desc
then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case by (auto split: option.splits)
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by (auto split: if-split-asm)
    (metis imageI list.exhaust-sel)
qed

```

lemma *field-names-subset-all-field-names:*

```

fixes t :: typ-uinfo and
  st :: typ-uinfo-struct and
  ts :: typ-uinfo-tuple list and
  x :: typ-uinfo-tuple
shows
  set (field-names-u t s) ⊆ set (all-field-names t)
  set (field-names-struct-u st s) ⊆ set (all-field-names-struct st)
  set (field-names-list-u ts s) ⊆ set (all-field-names-list ts)
  set (field-names-tuple-u x s) ⊆ set (all-field-names-tuple x)
by (induct t and st and ts and x arbitrary: s and s and s and s) auto

```

lemma *empty-all-field-names:*

```

fixes t :: typ-uinfo and
  st :: typ-uinfo-struct and
  ts :: typ-uinfo-tuple list and
  x :: typ-uinfo-tuple
shows
   $\square \in \text{set (all-field-names t)}$ 
   $\square \notin \text{set (all-field-names-struct st)}$ 
   $\square \notin \text{set (all-field-names-list ts)}$ 
   $\square \notin \text{set (all-field-names-tuple x)}$ 
by (induct t and st and ts and x) auto

```

lemma *empty-field-names-u:*

```

fixes t :: typ-uinfo and
  st :: typ-uinfo-struct and
  ts :: typ-uinfo-tuple list and
  x :: typ-uinfo-tuple
shows
  True
   $\square \notin \text{set (field-names-struct-u st s)}$ 
   $\square \notin \text{set (field-names-list-u ts s)}$ 
   $\square \notin \text{set (field-names-tuple-u x s)}$ 
by (induct t and st and ts and x) auto

```

lemma non-empty-field-names-u:
fixes $t :: \text{typ-uinfo}$ **and**
 $st :: \text{typ-uinfo-struct}$ **and**
 $ts :: \text{typ-uinfo-tuple list}$ **and**
 $x :: \text{typ-uinfo-tuple}$
shows
 $\text{field-names-u } t \ s \neq [] \implies \exists n. (s, n) \in \text{td-set } t \ m$
 $(\text{field-names-struct-u } st \ s) \neq [] \implies \exists n. (s, n) \in \text{td-set-struct } st \ m$
 $(\text{field-names-list-u } ts \ s) \neq [] \implies \exists n. (s, n) \in \text{td-set-list } ts \ m$
 $(\text{field-names-tuple-u } x \ s) \neq [] \implies \exists n. (s, n) \in \text{td-set-tuple } x \ m$
by (*induct t and st and ts and x arbitrary: s m and s m and s m and s m*)
fastforce+

lemma td-set-size:
 $(s, n) \in \text{td-set } t \ m \implies \text{size } s \leq \text{size } t$
 $(s, n) \in \text{td-set-struct } st \ m \implies \text{size } s \leq \text{size } st$
 $(s, n) \in \text{td-set-list } ts \ m \implies \exists t \in \text{set } ts. \text{size } s \leq \text{size } (\text{dt-fst } t)$
 $(s, n) \in \text{td-set-tuple } x \ m \implies \text{size } s \leq \text{size } (\text{dt-fst } x)$
proof (*induct t and st and ts and x arbitrary: s n m and s n m and s n m and s n m*)
case (*TypDesc nat typ-struct list*)
then show *?case by fastforce*
next
case (*TypScalar nat1 nat2 a*)
then show *?case by auto*
next
case (*TypAggregate list*)
then show *?case by (auto simp add: less-Suc-eq less-imp-le td-set-list-size-lte)*
next
case *Nil-typ-desc*
then show *?case by simp*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case by auto*
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case by auto*
qed

lemma td-set-field-names-nonempty:
 $(s, n) \in \text{td-set } t \ m \implies \text{field-names } t \ (\text{export-uinfo } s) \neq []$
 $(s, n) \in \text{td-set-struct } st \ m \implies \text{field-names-struct } st \ (\text{export-uinfo } s) \neq []$
 $(s, n) \in \text{td-set-list } ts \ m \implies \text{field-names-list } ts \ (\text{export-uinfo } s) \neq []$
 $(s, n) \in \text{td-set-tuple } x \ m \implies \text{field-names-tuple } x \ (\text{export-uinfo } s) \neq []$
by (*induct t and st and ts and x arbitrary: n m and n m and n m and n m*)
auto

lemma sub-typ-field-names-nonempty:


```

assumes  $s-t: s \leq t$ 
shows  $\text{field-names } t (\text{export-uinfo } s) \neq []$ 
proof -
  from  $s-t$  obtain  $n$  where  $(s, n) \in \text{td-set } t 0$ 
  by (auto simp add: typ-tag-le-def)
  from  $\text{td-set-field-names-nonempty } (1)$  [OF this] show ?thesis.
qed

```

```

lemma sub-typ-export-uinfo-mono:
assumes  $s-t: s \leq t$ 
shows  $\text{export-uinfo } s \leq \text{export-uinfo } t$ 
using  $s-t$ 
by (meson td-set-export-uinfoD typ-tag-le-def)

```

```

lemma descriptor-not-in-self:  $(\text{TypDesc } \text{algn } st \text{ } nm, n) \notin \text{td-set-struct } st \text{ } m$ 
proof
  assume  $(\text{TypDesc } \text{algn } st \text{ } nm, n) \in \text{td-set-struct } st \text{ } m$ 
  from  $\text{td-set-size}(2)$  [OF this] show False by simp
qed

```

```

lemma field-names-struct-descriptor-empty:  $\text{field-names-struct-u } st (\text{TypDesc } \text{algn } st \text{ } nm) = []$ 
using descriptor-not-in-self non-empty-field-names-u(2)
by blast

```

```

lemma all-field-names-exists-field-names-u:
fixes  $t :: \text{typ-uinfo}$  and
   $st :: \text{typ-uinfo-struct}$  and
   $ts :: \text{typ-uinfo-tuple list}$  and
   $x :: \text{typ-uinfo-tuple}$ 
shows
 $f \in \text{set } (\text{all-field-names } t) \implies \exists s. f \in \text{set } (\text{field-names-u } t \text{ } s)$ 
 $f \in \text{set } (\text{all-field-names-struct } st) \implies \exists s. f \in \text{set } (\text{field-names-struct-u } st \text{ } s)$ 
 $f \in \text{set } (\text{all-field-names-list } ts) \implies \exists s. f \in \text{set } (\text{field-names-list-u } ts \text{ } s)$ 
 $f \in \text{set } (\text{all-field-names-tuple } x) \implies \exists s. f \in \text{set } (\text{field-names-tuple-u } x \text{ } s)$ 
proof (induct t and st and ts and x arbitrary: f and f and f and f)
case  $(\text{TypDesc } \text{algn } st \text{ } nm)$ 
  then show ?case
  proof (cases f = [])
    case True
      with  $\text{TypDesc}$  show ?thesis by auto
    next
      case False
        with  $\text{TypDesc}$  obtain  $s$  where  $s: f \in \text{set } (\text{field-names-struct-u } st \text{ } s)$  by auto
        with field-names-struct-descriptor-empty have  $s \neq \text{TypDesc } \text{algn } st \text{ } nm$  by
auto
        with  $s$  False TypDesc show ?thesis by auto
  qed
next

```

```

    case (TypScalar nat1 nat2 a)
    then show ?case by auto
next
    case (TypAggregate list)
    then show ?case by auto
next
    case Nil-typ-desc
    then show ?case by auto
next
    case (Cons-typ-desc dt-tuple list)
    then show ?case by auto
next
    case (DTuple-typ-desc typ-desc list b)
    then show ?case
      by auto
      (meson empty-all-field-names(1) imageI)
qed

```

theorem *all-field-names-union-field-names-u-conv*: $set (all-field-names t) = (\bigcup s. set (field-names-u t s))$

```

proof
  show  $set (all-field-names t) \subseteq (\bigcup s. set (field-names-u t s))$ 
    using all-field-names-exists-field-names-u
    by auto
next
  show  $(\bigcup s. set (field-names-u t s)) \subseteq set (all-field-names t)$ 
    using field-names-subset-all-field-names(1)
    by auto
qed

```

corollary *all-field-names-union-field-names-export-uinfo-conv*:
 $set (all-field-names (export-uinfo t)) = (\bigcup s. set (field-names t s))$

```

proof –
  have  $(\bigcup s. set (field-names t s)) = (\bigcup s. set (field-names-u (export-uinfo t) s))$ 
    by (simp add: field-names-u-field-names-export-uinfo-conv)
  with all-field-names-union-field-names-u-conv show ?thesis by metis
qed

```

lemma *filter-same-eq*: $(\bigwedge x. x \in set xs \implies P x = Q x) \implies filter P xs = filter Q xs$
 by (*induct xs*) *auto*

lemma *field-lookup-tuple-hd-notin*: $n \neq dt-snd x \implies field-lookup-tuple x (n \# ns) m = None$
 by (*cases x*) *auto*

lemma *field-lookup-list-hd-notin*: $n \notin dt-snd ' set xs \implies field-lookup-list xs (n \# ns) m = None$
 by (*induct xs arbitrary: m*)

(*auto split: option.splits simp add: field-lookup-tuple-hd-notin*)

lemma *list-append-eq-split*: $xs1 = xs2 \implies ys1 = ys2 \implies (xs1 @ ys1) = (xs2 @ ys2)$
by *simp*

lemma *field-names-u-filter-all-field-names-conv*:

fixes $t :: \text{typ-uinfo}$ **and**
 $st :: \text{typ-uinfo-struct}$ **and**
 $ts :: \text{typ-uinfo-tuple list}$ **and**
 $x :: \text{typ-uinfo-tuple}$
shows

wf-desc t \implies
 $(\text{field-names-u } t \ s) = \text{filter } (\lambda f. \exists n. \text{field-lookup } t \ f \ 0 = \text{Some } (s, n)) (\text{all-field-names } t)$
wf-desc-struct st \implies
 $\text{field-names-struct-u } st \ s = \text{filter } (\lambda f. \exists n. \text{field-lookup-struct } st \ f \ 0 = \text{Some } (s, n)) (\text{all-field-names-struct } st)$
wf-desc-list ts \implies
 $\text{field-names-list-u } ts \ s = \text{filter } (\lambda f. \exists n. \text{field-lookup-list } ts \ f \ 0 = \text{Some } (s, n)) (\text{all-field-names-list } ts)$
wf-desc-tuple x \implies
 $\text{field-names-tuple-u } x \ s = \text{filter } (\lambda f. \exists n. \text{field-lookup-tuple } x \ f \ 0 = \text{Some } (s, n)) (\text{all-field-names-tuple } x)$
proof (*induct t and st and ts and x arbitrary: s and s and s and s*)
case (*TypDesc algn st nm*)
then show *?case*
apply (*clarsimp simp add: empty-all-field-names, safe*)
apply (*smt (verit, best) descriptor-not-in-self empty-all-field-names(2)*)
filter-False td-set-struct-field-lookup-structD
by (*metis (no-types, opaque-lifting) field-lookup-struct-empty option.simps(3)*)
next
case (*TypScalar n algn x*)
then show *?case by auto*
next
case (*TypAggregate ts*)
then show *?case by auto*
next
case *Nil-typ-desc*
then show *?case by auto*
next
case (*Cons-typ-desc x xs*)
from *Cons-typ-desc.prem*s **obtain**
 $wf\text{-}x :: \text{wf-desc-tuple } x$ **and**
 $wf\text{-}xs :: \text{wf-desc-list } xs$ **and**
distinct: dt-snd x \notin *dt-snd* ‘*set xs* **by** *clarsimp*
have *eq1*:
 $\text{filter } (\lambda f. \exists n. \text{field-lookup-tuple } x \ f \ 0 = \text{Some } (s, n)) (\text{all-field-names-tuple } x)$
 $=$

```

    filter
      (λf. ∃ n. (case field-lookup-tuple x f 0 of
        None ⇒ field-lookup-list xs f (0 + size-td (dt-fst x))
        | Some x ⇒ Some x) =
        Some (s, n)) (all-field-names-tuple x)
  apply (rule filter-same-eq)
  apply (cases x)
  subgoal for nm t' -
    using distinct
    apply (auto split: option.splits simp add: field-lookup-list-hd-notin)
  done
done
have eq2: filter (λf. ∃ n. field-lookup-list xs f 0 = Some (s, n))
  (all-field-names-list xs) =
  filter
    (λf. ∃ n. (case field-lookup-tuple x f 0 of
      None ⇒ field-lookup-list xs f (0 + size-td (dt-fst x))
      | Some x ⇒ Some x) =
      Some (s, n))
    (all-field-names-list xs)
  apply (rule filter-same-eq)
  apply (cases x)
  subgoal for nm t' -
    using distinct Cons-typ-desc.hyps(2)[OF wf-xs]
    by (smt (verit) all-field-names-exists-field-names-u(3) dt-prj-simps(2)
      field-lookup-list-None field-lookup-list-offsetD field-lookup-offset'(3)
      fl5 hd-Cons-tl mem-Collect-eq option.case(1) option.simps(3) set-filter)
  done

show ?case
  by (simp add: Cons-typ-desc.hyps(1)[OF wf-x] Cons-typ-desc.hyps(2)[OF
wf-xs] eq1 eq2 )
next
  case (DTuple-typ-desc t' nm -)
  then show ?case
    by (auto simp add: filter-map intro: filter-same-eq )
qed

lemma all-field-names-export-uinfo':
  fixes t:: ('a, 'b) typ-info
  and st:: ('a, 'b) typ-info-struct
  and ts:: ('a, 'b) typ-info-tuple list
  and x:: ('a, 'b) typ-info-tuple
  shows
    all-field-names (map-td field-norm (λ-. ()) t) = all-field-names t
    all-field-names-struct (map-td-struct field-norm (λ-. ()) st) = all-field-names-struct
    st
    all-field-names-list (map-td-list field-norm (λ-. ()) ts) = all-field-names-list ts

```

all-field-names-tuple (*map-td-tuple field-norm* ($\lambda\cdot. ()$) *x*) = *all-field-names-tuple x*
by (*induct t and st and ts and x*) *auto*

lemma *all-field-names-export-uinfo*:
all-field-names (*export-uinfo t*) = *all-field-names t*
by (*simp add: export-uinfo-def all-field-names-export-uinfo'*)

lemma *inj* (#)
by (*metis inj-onI list.inject*)

lemma *distinct-map-cons*: *distinct xs* \implies *distinct* (*map* ((#) *ys*) *xs*)
by (*induct xs*) *auto*

lemma *all-field-names-list-conv*: *all-field-names-list xs* =
concat (*map* ($\lambda x. \text{map } ((\#) (dt\text{-snd } x)) ((\text{all-field-names } o \text{ dt-fst}) x)$) *xs*)
apply (*induct xs*)
apply *simp*
subgoal for *x1 xs*
apply (*cases x1*)
apply *auto*
done
done

lemma *distinct-all-field-names*:
fixes *t* :: ('a, 'b) *typ-info* **and**
st :: ('a, 'b) *typ-info-struct* **and**
ts :: ('a, 'b) *typ-info-tuple list* **and**
x :: ('a, 'b) *typ-info-tuple*
shows
wf-desc t \implies *distinct* (*all-field-names t*) **and**
wf-desc-struct st \implies *distinct* (*all-field-names-struct st*) **and**
wf-desc-list ts \implies *distinct* (*all-field-names-list ts*) **and**
wf-desc-tuple x \implies *distinct* (*all-field-names-tuple x*)
proof (*induct t and st and ts and x*)
case (*TypDesc algn st nm*)
then show ?*case*
by *auto* (*metis all-field-names-export-uinfo'(2) empty-all-field-names(2)*)
next
case (*TypScalar n algn x*)
then show ?*case* **by** *auto*
next
case (*TypAggregate ts*)
then show ?*case* **by** *auto*
next
case *Nil-tyt-desc*
then show ?*case* **by** *auto*
next

```

case (Cons-tyt-desc x xs)
then show ?case apply (cases x) by (auto simp add: all-field-names-list-conv)
next
case (DTuple-tyt-desc typ-desc list b)
then show ?case
by (auto simp add: distinct-map-cons)
qed

```

lemma *td-set-field-names-u-nonempty*:

```

(s, n) ∈ td-set t m ⇒ field-names-u t s ≠ []
(s, n) ∈ td-set-struct st m ⇒ field-names-struct-u st s ≠ []
(s, n) ∈ td-set-list ts m ⇒ field-names-list-u ts s ≠ []
(s, n) ∈ td-set-tuple x m ⇒ field-names-tuple-u x s ≠ []
by (induct t and st and ts and x arbitrary: n m and n m and n m and n m)
auto

```

lemma *field-lookup-export-uinfo-Some-rev*:

```

field-lookup (export-uinfo t) f n = Some (s, k) ⇒ ∃ s'. field-lookup t f n = Some
(s', k) ∧ s = export-uinfo s'
by (simp add: export-uinfo-def field-lookup-map map-td-flr-def split: option.splits
prod.splits)

```

lemma *wf-desc-export-uinfo-pres*:

```

fixes t :: ('a, 'b) typ-info and
st :: ('a, 'b) typ-info-struct and
ts :: ('a, 'b) typ-info-tuple list and
x :: ('a, 'b) typ-info-tuple

```

shows

```

wf-desc t ⇒ wf-desc (export-uinfo t)
wf-desc-struct st ⇒ wf-desc-struct (map-td-struct field-norm (λ-. ()) st)
wf-desc-list ts ⇒ wf-desc-list (map-td-list field-norm (λ-. ()) ts)
wf-desc-tuple x ⇒ wf-desc-tuple (map-td-tuple field-norm (λ-. ()) x)
apply (induct t and st and ts and x)
by (auto simp add: export-uinfo-def)
(metis imageI mat4 wf-desc-list.wfd4 wf-desc-map(3))

```

primrec

```

toplevel-field-names :: ('a, 'b) typ-desc ⇒
(field-name) list and
toplevel-field-names-struct :: ('a, 'b) typ-struct ⇒
(field-name) list and
toplevel-field-names-list :: ('a, 'b) typ-tuple list ⇒
(field-name) list and
toplevel-field-names-tuple :: ('a, 'b) typ-tuple ⇒
(field-name) list

```

where

```

toplevel-field-names (TypDesc algn st nm) = toplevel-field-names-struct st

```

| *toplevel-field-names-struct* (*TypScalar m algn d*) = []
| *toplevel-field-names-struct* (*TypAggregate xs*) = *toplevel-field-names-list xs*

| *toplevel-field-names-list* [] = []
| *toplevel-field-names-list* (*x#xs*) = *toplevel-field-names-tuple x @ toplevel-field-names-list xs*

| *toplevel-field-names-tuple* (*DTuple s f d*) = [*f*]

lemma *all-field-names-root*: $\exists xs. \text{all-field-names } t = [] @ xs$
by (*cases t*) *simp*

lemma *toplevel-field-names-all-field-names*:

fixes *t*:: ('a, 'b) *typ-desc*
and *st*:: ('a, 'b) *typ-struct*
and *ts*:: ('a, 'b) *typ-tuple list*
and *x*:: ('a, 'b) *typ-tuple*

shows

$f \in \text{set } (\text{toplevel-field-names } t) \implies [f] \in \text{set } (\text{all-field-names } t)$
 $f \in \text{set } (\text{toplevel-field-names-struct } st) \implies [f] \in \text{set } (\text{all-field-names-struct } st)$
 $f \in \text{set } (\text{toplevel-field-names-list } ts) \implies [f] \in \text{set } (\text{all-field-names-list } ts)$
 $f \in \text{set } (\text{toplevel-field-names-tuple } x) \implies [f] \in \text{set } (\text{all-field-names-tuple } x)$

apply (*induct t and st and ts and x*)

by *auto*

(*metis all-field-names-root append-Cons imageI list.set-intros(1)*)

lemma *append-eq-same-prefixI*: $ys = zs \implies xs @ ys = xs @ zs$
by *simp*

lemma *toplevel-field-names-field-lookup*:

fixes *t*:: ('a, 'b) *typ-info*
and *st*:: ('a, 'b) *typ-info-struct*
and *ts*:: ('a, 'b) *typ-info-tuple list*
and *x*:: ('a, 'b) *typ-info-tuple*

shows

$f \in \text{set } (\text{toplevel-field-names } t) \implies \text{wf-desc } t \implies$
 $\exists s n. \text{field-lookup } t [f] m = \text{Some } (s, m + n)$

$f \in \text{set } (\text{toplevel-field-names-struct } st) \implies \text{wf-desc-struct } st \implies$
 $\exists s n. \text{field-lookup-struct } st [f] m = \text{Some } (s, m + n)$

$f \in \text{set } (\text{toplevel-field-names-list } ts) \implies \text{wf-desc-list } ts \implies$
 $\exists s n. \text{field-lookup-list } ts [f] m = \text{Some } (s, m + n)$

$f \in \text{set } (\text{toplevel-field-names-tuple } x) \implies \text{wf-desc-tuple } x \implies$
 $\exists s n. \text{field-lookup-tuple } x [f] m = \text{Some } (s, m + n)$

proof (*induct t and st and ts and x arbitrary: f m and f m and f m and f m*)
case (*TypDesc algn st nm*)

```

then show ?case by simp
next
case (TypScalar sz algn d)
then show ?case by simp
next
case (TypAggregate ts)
then show ?case by auto
next
case Nil-typ-desc
then show ?case by auto
next
case (Cons-typ-desc x fs)
obtain d nm y where x: x = DTuple d nm y by (cases x) auto
then show ?case using Cons-typ-desc
by auto
(metis (no-types, opaque-lifting) add.assoc)
next
case (DTuple-typ-desc d nm y)
then show ?case by auto
qed

```

lemma *partition-toplevel-field-names:*

```

fixes t:: ('a, 'b) typ-info
and st:: ('a, 'b) typ-info-struct
and ts:: ('a, 'b) typ-info-tuple list
and x:: ('a, 'b) typ-info-tuple

```

shows

$$\text{length } bs = \text{size-td } t \implies \text{aggregate } t \implies \text{wf-desc } t \implies \\ \text{concat } (\text{map } (\lambda f. \text{take } (\text{size-td } (\text{fst } (\text{the } (\text{field-lookup } t [f] n)))) \\ (\text{drop } (\text{snd } (\text{the } (\text{field-lookup } t [f] n)) - n) bs)) \\ (\text{toplevel-field-names } t)) = bs$$

$$\text{length } bs = \text{size-td-struct } st \implies \text{aggregate-struct } st \implies \text{wf-desc-struct } st \implies \\ \text{concat } (\text{map } (\lambda f. \text{take } (\text{size-td } (\text{fst } (\text{the } (\text{field-lookup-struct } st [f] n)))) \\ (\text{drop } (\text{snd } (\text{the } (\text{field-lookup-struct } st [f] n)) - n) bs)) \\ (\text{toplevel-field-names-struct } st)) = bs$$

$$\text{length } bs = \text{size-td-list } ts \implies \text{wf-desc-list } ts \implies \\ \text{concat } (\text{map } (\lambda f. \text{take } (\text{size-td } (\text{fst } (\text{the } (\text{field-lookup-list } ts [f] n)))) \\ (\text{drop } (\text{snd } (\text{the } (\text{field-lookup-list } ts [f] n)) - n) bs)) \\ (\text{toplevel-field-names-list } ts)) = bs$$

$$\text{length } bs = \text{size-td-tuple } x \implies \text{wf-desc-tuple } x \implies \\ \text{concat } (\text{map } (\lambda f. \text{take } (\text{size-td } (\text{fst } (\text{the } (\text{field-lookup-tuple } x [f] n)))) \\ (\text{drop } (\text{snd } (\text{the } (\text{field-lookup-tuple } x [f] n)) - n) bs)) \\ (\text{toplevel-field-names-tuple } x)) = bs$$

proof (*induct t and st and ts and x arbitrary: bs n and bs n and bs n and bs n*)

```

case (TypDesc algn st nm)

```



```

then show ?case by simp
next
case (TypScalar sz algn d)
then show ?case by simp
next
case (TypAggregate ts)
then show ?case by auto
next
case Nil-typ-desc
then show ?case by auto
next
case (Cons-typ-desc x fs)
obtain d nm y where x: x = DTuple d nm y by (cases x) auto
from Cons-typ-desc.prem1 obtain
  lbs: length bs = size-td-tuple x + size-td-list fs and
  wf-desc-x: wf-desc-tuple (DTuple d nm y) and
  nm-notin: nm ∉ dt-snd ' set fs and
  wf-desc-fs: wf-desc-list fs

by (auto simp add: x)

from nm-notin have nm-notin': nm ∉ set (toplevel-field-names-list fs)
proof (induct fs)
  case Nil
  then show ?case by simp
next
  case (Cons f1 fs)
  then show ?case by (cases f1) auto
qed

from lbs have l-take: length (take (size-td d) bs) = size-td d
  by (simp add: x)

from lbs have l-drop: length (drop (size-td d) bs) = size-td-list fs
  by (simp add: x)

from lbs have bs-take-drop: bs = take (size-td d) bs @ drop (size-td d) bs
  by (simp add: x)
note hyp = Cons-typ-desc.hyps(2) [OF l-drop wf-desc-fs, where n = (n + size-td
d)]

show ?case
  apply (simp add: x)
  apply (simp cong: if-cong option.case-cong)
  apply (subst (3) bs-take-drop)
  apply (rule append-eq-same-prefixI)
  apply (subst hyp [symmetric])
  apply (rule arg-cong [where f = concat])

```

```

apply (rule map-cong [OF refl])
using nm-notin'
apply auto
thm toplevel-field-names-field-lookup( $\exists$ )[OF - wf-desc-fs]
apply (frule toplevel-field-names-field-lookup( $\exists$ ) [OF - wf-desc-fs, where  $m =$ 
( $n + \text{size-td } d$ )] )
by auto
(metis add.commute)
next
case (DTuple-typ-desc  $d$   $nm$   $y$ )
then show ?case by auto
qed

```

lemma *toplevel-field-names-export-uinfo'*:

```

fixes  $t::('a, 'b)$  typ-info
and  $st::('a, 'b)$  typ-info-struct
and  $ts::('a, 'b)$  typ-info-tuple list
and  $x::('a, 'b)$  typ-info-tuple
shows
toplevel-field-names (map-td field-norm ( $\lambda.$  ())  $t$ ) = toplevel-field-names  $t$ 
toplevel-field-names-struct (map-td-struct field-norm ( $\lambda.$  ())  $st$ ) = toplevel-field-names-struct
 $st$ 
toplevel-field-names-list (map-td-list field-norm ( $\lambda.$  ())  $ts$ ) = toplevel-field-names-list
 $ts$ 
toplevel-field-names-tuple (map-td-tuple field-norm ( $\lambda.$  ())  $x$ ) = toplevel-field-names-tuple
 $x$ 
by (induct  $t$  and  $st$  and  $ts$  and  $x$ ) auto

```

lemma *toplevel-field-names-export-uinfo*:

```

toplevel-field-names (export-uinfo  $t$ ) = toplevel-field-names  $t$ 
using toplevel-field-names-export-uinfo' by (simp add: export-uinfo-def)

```

lemma (in *xmem-type*) *xmem-type-partition-toplevel-field-names*:

```

assumes aggregate: aggregate (typ-info-t TYPE('a))
assumes lbs: length  $bs = \text{size-of } (TYPE('a))$ 
shows concat (map ( $\lambda f.$  take (size-td (fst (the (field-lookup (typ-info-t TYPE('a))
[f] 0))))
(drop (snd (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))  $bs$ ))
(toplevel-field-names (typ-info-t TYPE('a)))) =  $bs$ 
using partition-toplevel-field-names(1) [OF lbs [simplified size-of-def] aggregate
wf-desc, where  $n=0$ ]
by simp

```

lemma *toplevel-field-names-sum-list-size*:

```

fixes  $t::('a, 'b)$  typ-info
and  $st::('a, 'b)$  typ-info-struct
and  $ts::('a, 'b)$  typ-info-tuple list
and  $x::('a, 'b)$  typ-info-tuple

```

shows

$aggregate\ t \implies wf_desc\ t \implies$
 $sum_list\ (map\ (\lambda f. size_td\ (fst\ (the\ (field_lookup\ t\ [f]\ n))))\ (toplevel_field_names$
 $t)) =$
 $size_td\ t$

$aggregate_struct\ st \implies wf_desc_struct\ st \implies$
 $sum_list\ (map\ (\lambda f. size_td\ (fst\ (the\ (field_lookup_struct\ st\ [f]\ n))))\ (toplevel_field_names_struct$
 $st)) =$
 $size_td_struct\ st$

$wf_desc_list\ ts \implies$
 $sum_list\ (map\ (\lambda f. size_td\ (fst\ (the\ (field_lookup_list\ ts\ [f]\ n))))\ (toplevel_field_names_list$
 $ts)) =$
 $size_td_list\ ts$

$wf_desc\ t \implies$
 $sum_list\ (map\ (\lambda f. size_td\ (fst\ (the\ (field_lookup_tuple\ x\ [f]\ n))))\ (toplevel_field_names_tuple$
 $x)) =$
 $size_td_tuple\ x$

proof (*induct t and st and ts and x arbitrary: n and n and n and n*)

case (*TypDesc algn st nm*)

then show *?case by simp*

next

case (*TypScalar sz algn d*)

then show *?case by simp*

next

case (*TypAggregate ts*)

then show *?case by auto*

next

case *Nil-typ-desc*

then show *?case by auto*

next

case (*Cons-typ-desc x fs*)

obtain *d nm y where x: x = DTuple d nm y by (cases x) auto*

from *Cons-typ-desc.prem*s **obtain**

wf-desc-x: wf-desc-tuple (DTuple d nm y) and

nm-notin: nm \notin dt-snd ' set fs and

wf-desc-fs: wf-desc-list fs

by (*auto simp add: x*)

from *nm-notin have nm-notin': nm \notin set (toplevel-field-names-list fs)*

proof (*induct fs*)

case *Nil*

then show *?case by simp*

next

case (*Cons f1 fs*)

```

    then show ?case by (cases f1) auto
qed

note hyp = Cons-typ-desc.hyps(2) [OF wf-desc-fs, where n = (n + size-td d)]

show ?case
  apply (simp add: x)
  apply (simp cong: if-cong option.case-cong)
  using nm-notin' hyp
  by (smt (verit, best) map-eq-conv option.simps(4))
next
case (DTuple-typ-desc d nm y)
  then show ?case by auto
qed

lemma toplevel-field-names-sum-list-offset:
  fixes t:: ('a, 'b) typ-info
  and st:: ('a, 'b) typ-info-struct
  and ts:: ('a, 'b) typ-info-tuple list
  and x:: ('a, 'b) typ-info-tuple
shows
  aggregate t  $\implies$  wf-desc t  $\implies$   $i < \text{length} (\text{toplevel-field-names } t) \implies$ 
    sum-list (map ( $\lambda i.$  size-td (fst (the (field-lookup t [(toplevel-field-names t) ! i]
n)))) [0.. $i$ ]) =
    (snd (the (field-lookup t [(toplevel-field-names t) ! i] n)) - n)

  aggregate-struct st  $\implies$  wf-desc-struct st  $\implies$   $i < \text{length} (\text{toplevel-field-names-struct }
st) \implies$ 
    sum-list (map ( $\lambda i.$  size-td (fst (the (field-lookup-struct st [(toplevel-field-names-struct
st) ! i] n)))) [0.. $i$ ]) =
    (snd (the (field-lookup-struct st [(toplevel-field-names-struct st) ! i] n)) - n)

  wf-desc-list ts  $\implies$   $i < \text{length} (\text{toplevel-field-names-list } ts) \implies$ 
    sum-list (map ( $\lambda i.$  size-td (fst (the (field-lookup-list ts [(toplevel-field-names-list
ts) ! i] n)))) [0.. $i$ ]) =
    (snd (the (field-lookup-list ts [(toplevel-field-names-list ts) ! i] n)) - n)

  wf-desc-tuple x  $\implies$   $i < \text{length} (\text{toplevel-field-names-tuple } x) \implies$ 
    sum-list (map ( $\lambda i.$  size-td (fst (the (field-lookup-tuple x [(toplevel-field-names-tuple
x) ! i] n)))) [0.. $i$ ]) =
    (snd (the (field-lookup-tuple x [(toplevel-field-names-tuple x) ! i] n)) - n)

proof (induct t and st and ts and x arbitrary: n i and n i and n i and n i)
case (TypDesc algn st nm)
  then show ?case by simp
next
case (TypScalar sz algn d)
  then show ?case by simp

```

```

next
  case (TypAggregate ts)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
  obtain d nm y where x: x = DTuple d nm y by (cases x) auto

  from Cons-typ-desc.prem1 obtain
    wf-desc-x: wf-desc-tuple (DTuple d nm y) and
    nm-notin: nm ∉ dt-snd ' set fs and
    wf-desc-fs: wf-desc-list fs and
    i-bound: i < Suc (length (toplevel-field-names-list fs))
    by (auto simp add: x)

  from nm-notin have nm-notin': nm ∉ set (toplevel-field-names-list fs)
  proof (induct fs)
    case Nil
    then show ?case by simp
  next
    case (Cons f1 fs)
    then show ?case by (cases f1) auto
  qed

  note hyp = Cons-typ-desc.hyps(2) [OF wf-desc-fs, where n= (n + size-td d)]

  from nm-notin' i-bound have nm-eq-conv: nm = (nm # toplevel-field-names-list
  fs) ! i ↔ i = 0
  apply (cases i)
  apply simp
  apply (simp)
  using nth-mem by blast

  show ?case
  proof (cases i)
    case 0
    then show ?thesis by (simp add: x)
  next
    case (Suc i')
    with i-bound have i'-bound: i' < length (toplevel-field-names-list fs) by simp
    have split: [0..<Suc i'] = 0 # [1..<Suc i']
    by (simp add: upt-rec)

    have sum-eq:

```

```

(∑ i←[0..<i']. size-td
  (fst (the (case if nm = toplevel-field-names-list fs ! i
            then Some (d, n) else None of
            None ⇒
            field-lookup-list fs
            [toplevel-field-names-list fs ! i] (n + size-td d)
            | Some x2 ⇒ Some x2)))) =
(∑ i←[0..<i']. size-td
  (fst (the (field-lookup-list fs [toplevel-field-names-list fs ! i]
            (n + size-td d))))))
apply (rule arg-cong [where f=sum-list])
apply (rule map-cong [OF refl])
using nm-notin' i'-bound
apply auto
done

show ?thesis
apply (simp add: x)
apply (clarsimp cong: if-cong option.case-cong simp add: nm-eq-conv)
apply (subst Suc)
apply (subst split)
apply (simp only: list.map sum-list.Cons)
apply (subst sum-list-Suc-index-shift)
apply (simp add: Suc)
apply (simp cong: if-cong option.case-cong)
apply (subst sum-eq)
apply (subst hyp [OF i'-bound])
using toplevel-field-names-field-lookup(3)
  [where f=toplevel-field-names-list fs ! i' and ts = fs and m=(n + size-td
d), OF - wf-desc-fs] i'-bound
apply clarsimp
done
qed
next
  case (DTuple-typ-desc d nm y)
  then show ?case by auto
qed

lemma sum-list-upt-map-nth-conv: sum-list (map (λi. g (xs ! i)) [0..<length xs])
= sum-list (map g xs)
proof -
  from map-nth have eq1: map (!! xs) [0..<length xs] = xs .
  from map-map [where f=g and g=(!! xs)]
  have eq2: map g (map (!! xs) [0..<length xs]) = map (g ∘ (!! xs)) [0..<length
xs] .
  from eq1 eq2
  show ?thesis
  by (simp add: comp-def)
qed

```

lemma *toplevel-field-names-empty-tyt-info*: *toplevel-field-names (empty-tyt-info algn tn) = []*
by (*simp add: empty-tyt-info-def*)

lemma *toplevel-field-names-no-padding-empty-tyt-info*:
filter (Not o padding-field-name) (toplevel-field-names (empty-tyt-info algn tn))
= []
by (*simp add: empty-tyt-info-def*)

lemma *toplevel-field-names-list-append* [*simp*]:
toplevel-field-names-list (xs @ ys) = toplevel-field-names-list xs @ toplevel-field-names-list ys
by (*induct xs*) *auto*

lemma *toplevel-field-names-extend-ti*:
fixes
t :: 'a xtyp-info and
st :: 'a xtyp-info-struct and
ts :: 'a xtyp-info-tuple list and
x :: 'a xtyp-info-tuple
shows
toplevel-field-names (extend-ti t s n fn d) = toplevel-field-names t @ [fn]
toplevel-field-names-struct (extend-ti-struct st s fn d) = toplevel-field-names-struct st @ [fn]
toplevel-field-names-list ts = toplevel-field-names-list ts
toplevel-field-names-tuple x = toplevel-field-names-tuple x
by (*induct t and st and ts and x*) *auto*

lemma *toplevel-field-names-adjust-ti'*:
fixes
t :: 'a xtyp-info and
st :: 'a xtyp-info-struct and
ts :: 'a xtyp-info-tuple list and
x :: 'a xtyp-info-tuple
shows
toplevel-field-names (map-td (λn algn. update-desc acc upd) (update-desc acc upd) t) =
toplevel-field-names t
toplevel-field-names-struct (map-td-struct (λn algn. update-desc acc upd) (update-desc acc upd) st) =
toplevel-field-names-struct st
toplevel-field-names-list (map-td-list (λn algn. update-desc acc upd) (update-desc acc upd) ts) =
toplevel-field-names-list ts
toplevel-field-names-tuple (map-td-tuple (λn algn. update-desc acc upd) (update-desc acc upd) x) =
toplevel-field-names-tuple x

by (*induct t and st and ts and x*) (*auto*)

lemma *toplevel-field-names-adjust-ti*:

toplevel-field-names (adjust-ti t acc upd) = toplevel-field-names t

by (*simp add: toplevel-field-names-adjust-ti' adjust-ti-def*)

lemma *padding-field-name-pad*: *padding-field-name (foldl (@) "\pad-" xs)*

by (*auto simp add: padding-field-name-def foldl-conv-concat*)

lemma *toplevel-field-names-no-padding-ti-pad-combine*:

filter (Not o padding-field-name) (toplevel-field-names (ti-pad-combine n t)) =

filter (Not o padding-field-name) (toplevel-field-names t)

by (*simp add: ti-pad-combine-def Let-def toplevel-field-names-extend-ti(1) padding-field-name-pad*)

lemma *toplevel-field-names-ti-pad-combine*:

(toplevel-field-names (ti-pad-combine n t)) =

toplevel-field-names t @ [foldl (@) "\pad-" (CompoundCTypes.field-names-list

t)]

by (*simp add: ti-pad-combine-def Let-def toplevel-field-names-extend-ti(1) padding-field-name-pad*)

lemma *toplevel-field-names-ti-typ-combine*:

toplevel-field-names (ti-typ-combine t-b acc upd algn fn t) = toplevel-field-names

t @ [fn]

by (*simp add: ti-typ-combine-def toplevel-field-names-extend-ti*)

lemma *toplevel-field-names-no-padding-ti-typ-combine*:

¬ padding-field-name fn ⇒

filter (Not o padding-field-name) (toplevel-field-names (ti-typ-combine t-b acc upd algn fn t)) =

filter (Not o padding-field-name) (toplevel-field-names t) @ [fn]

by (*simp add: toplevel-field-names-ti-typ-combine toplevel-field-names-extend-ti*)

lemma *toplevel-field-names-no-padding-ti-typ-pad-combine*:

¬ padding-field-name fn ⇒

filter (Not o padding-field-name) (toplevel-field-names (ti-typ-pad-combine t-b acc upd algn fn t)) =

filter (Not o padding-field-name) (toplevel-field-names t) @ [fn]

by (*simp add: ti-typ-pad-combine-def toplevel-field-names-ti-typ-combine toplevel-field-names-no-padding-ti-pad-combine Let-def*)

lemma *toplevel-field-names-ti-typ-pad-combine*:

toplevel-field-names (ti-typ-pad-combine (t-b:: 'b itself) acc upd algn fn t) =

toplevel-field-names t @ (

if 0 < padup (max (2 ^ algn) (align-of TYPE('b::c-type))) (size-td t) then

[foldl (@) "\pad-" (CompoundCTypes.field-names-list t), fn]

else

[fn])

by (*simp add: ti-typ-pad-combine-def toplevel-field-names-ti-typ-combine toplevel-field-names-no-padding-ti-pa*
Let-def
toplevel-field-names-ti-pad-combine)

lemma *toplevel-field-names-map-align: toplevel-field-names (map-align n t) = toplevel-field-names*
t
by (*cases t*) (*simp add: map-align-def*)

lemma *toplevel-field-names-no-padding-final-pad:*
filter (Not o padding-field-name) (toplevel-field-names (final-pad n t))
= filter (Not o padding-field-name) (toplevel-field-names t)
by (*simp add: final-pad-def Let-def toplevel-field-names-map-align toplevel-field-names-no-padding-ti-pad-comb*)

lemma *toplevel-field-names-final-pad:*
(toplevel-field-names (final-pad n t))
=
toplevel-field-names t @ (
if 0 < padup (2 ^ max n (align-td t)) (size-td t) then
[foldl (@) "!pad-" (CompoundCTypes.field-names-list t)]
else [])
by (*simp add: final-pad-def Let-def toplevel-field-names-map-align toplevel-field-names-ti-pad-combine*)

lemmas *toplevel-field-names-no-padding-combinator-simps =*
toplevel-field-names-no-padding-empty-typ-info
toplevel-field-names-no-padding-final-pad
toplevel-field-names-no-padding-ti-typ-pad-combine
toplevel-field-names-no-padding-ti-typ-combine

lemmas *toplevel-field-names-combinator-simps =*
toplevel-field-names-empty-typ-info
toplevel-field-names-final-pad
toplevel-field-names-ti-typ-pad-combine
toplevel-field-names-ti-typ-combine

lemma *fold-filter-out-id:*
assumes *filter-out-id: $\bigwedge i v. i < \text{length } xs \implies \neg P (xs ! i) \implies f (xs ! i) v = v$*
shows *fold f xs = fold f (filter P xs)*
using *filter-out-id*
apply (*induct xs*)
apply *simp*
apply (*clarsimp simp add: comp-def fun-eq-iff, safe*)
apply (*metis Suc-mono nth-Cons-Suc*)
by (*metis not-less-eq nth-Cons-0 nth-Cons-Suc zero-less-Suc*)

context *xmem-type*
begin

lemma *xmem-type-toplevel-field-names-sum-list-size:*

assumes *aggregate*: *aggregate* (*typ-info-t* *TYPE('a)*)
shows *sum-list*
 $(\text{map } (\lambda f. \text{size-td } (\text{fst } (\text{the } (\text{field-lookup } (\text{typ-info-t } \text{TYPE}('a)) [f] n))))$
 $(\text{toplevel-field-names } (\text{typ-info-t } \text{TYPE}('a))) =$
 $\text{size-of } \text{TYPE}('a)$
using *toplevel-field-names-sum-list-size(1)* [*OF aggregate wf-desc*] **by** (*simp add:*
size-of-def)

lemma *xmem-type-toplevel-field-names-sum-list-offset*:
assumes *aggregate*: *aggregate* (*typ-info-t* *TYPE('a)*)
assumes *i-bound*: $i < \text{length } (\text{toplevel-field-names } (\text{typ-info-t } \text{TYPE}('a)))$
shows
 $\text{sum-list } (\text{map } (\lambda i. \text{size-td } (\text{fst } (\text{the } (\text{field-lookup } (\text{typ-info-t } \text{TYPE}('a))$
 $[(\text{toplevel-field-names } (\text{typ-info-t } \text{TYPE}('a))) ! i] 0))))$
 $[0..<i]) =$
 $(\text{snd } (\text{the } (\text{field-lookup } (\text{typ-info-t } \text{TYPE}('a)) [(\text{toplevel-field-names } (\text{typ-info-t } \text{TYPE}('a))) ! i] 0))))$
using *toplevel-field-names-sum-list-offset(1)* [*OF aggregate wf-desc i-bound, where*
n=0]
by *simp*

lemma *xmem-type-toplevel-field-names-field-lookup*:
assumes *f*: $f \in \text{set } (\text{toplevel-field-names } (\text{typ-info-t } \text{TYPE}('a)))$
shows $\exists s n. \text{field-lookup } (\text{typ-info-t } \text{TYPE}('a)) [f] 0 = \text{Some } (s, n)$
using *toplevel-field-names-field-lookup(1)* [*OF f wf-desc, of 0*] **by** *simp*

lemma (*in c-type*) *field-lookup-ty-uinfo-t-Some*:
 $\text{field-lookup } (\text{typ-info-t } \text{TYPE}('a)) f m = \text{Some } (s, n) \implies$
 $\text{field-lookup } (\text{typ-uinfo-t } \text{TYPE}('a)) f m = \text{Some } (\text{export-uinfo } s, n)$
by (*simp add: typ-uinfo-t-def field-lookup-export-uinfo-Some*)

lemma *toplevel-field-names-field-lookup-offset-conv*:
assumes *f*: $f \in \text{set } (\text{toplevel-field-names } (\text{typ-info-t } \text{TYPE}('a)))$
shows $\text{snd } (\text{the } (\text{field-lookup } (\text{typ-uinfo-t } \text{TYPE}('a)) [f] 0)) =$
 $\text{snd } (\text{the } (\text{field-lookup } (\text{typ-info-t } \text{TYPE}('a)) [f] 0))$
using *xmem-type-toplevel-field-names-field-lookup* [*OF f*]
by (*auto simp add: field-lookup-ty-uinfo-t-Some*)

lemma *heap-update-list-fold-toplevel-field-names*:
fixes *p*::*'a ptr*
assumes *aggregate*: *aggregate* (*typ-info-t* *TYPE('a)*)
assumes *cgrd*: *c-guard p*
shows
 $\text{heap-update-list } (\text{ptr-val } p) (\text{to-bytes } x (\text{heap-list } h (\text{size-of } \text{TYPE}('a)) (\text{ptr-val } p)))$
 $h =$
 fold
 $(\lambda f h. \text{heap-update-list } (\&(p \rightarrow [f])))$

```

      (access-ti
        (fst (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))
          x
        (heap-list h (size-td (fst (the (field-lookup (typ-info-t TYPE('a)) [f]
0)))) (&(p→[f]))))
          h)
    (toplevel-field-names (typ-info-t TYPE('a))) h (is ?LHS = fold ?F ?fs h)
proof -

define fld::nat ⇒ field-name where
  fld = (λi. topLevel-field-names (typ-info-t TYPE('a)) ! i)
define sz:: nat ⇒ nat where
  sz = (λi. size-td (fst (the (field-lookup (typ-info-t TYPE('a)) [fld i] 0))))
define off:: nat ⇒ nat where
  off = (λi. snd (the (field-lookup (typ-info-t TYPE('a)) [fld i] 0)))
define v:: nat ⇒ byte list ⇒ byte list where
  v = (λi. (access-ti
    (fst (the (field-lookup (typ-info-t TYPE('a)) [fld i] 0)))
      x))

have lbs: length (to-bytes x (heap-list h (size-of TYPE('a)) (ptr-val p))) = size-of
TYPE('a)
by (simp add: local.lense.access-result-size to-bytes-def)
{
  fix i::nat
  assume bound: i < length (toplevel-field-names (typ-info-t TYPE('a)))
  hence topLevel-field-names (typ-info-t TYPE('a)) ! i ∈ set (toplevel-field-names
(typ-info-t TYPE('a)))
  by simp
  from topLevel-field-names-field-lookup-offset-conv [OF this]

  have snd (the (field-lookup (typ-uinfo-t TYPE('a))
    [toplevel-field-names (typ-info-t TYPE('a)) ! i] 0)) =
    snd (the (field-lookup (typ-info-t TYPE('a))
    [toplevel-field-names (typ-info-t TYPE('a)) ! i] 0)) .
} note field-lookup-conv = this

have fs: ?fs = map fld [0..<length(toplevel-field-names (typ-info-t TYPE('a)))]
by (simp add: fld-def map-nth)

from c-guard-no-wrap' [OF cgrd]
have no-overflow: unat (ptr-val p) + size-of TYPE('a) ≤ addr-card .
have fold-conv:
  fold ?F ?fs h =
  fold
    (λi h. heap-update-list (ptr-val p + word-of-nat (off i))
      (v i (heap-list h (sz i) (ptr-val p + word-of-nat (off i)))) h)
    [0..<length (toplevel-field-names (typ-info-t TYPE('a)))] h
  apply (subst fs)

```

```

apply (subst fold-map)
apply (rule fold-cong [OF refl refl])
apply (simp add: fld-def off-def sz-def v-def field-lvalue-def field-offset-def
field-offset-untyped-def)
apply (simp add: field-lookup-conv)
done
note partition= heap-update-list-fold-partition [where a= ptr-val p and sz=sz
and off=off and v=v and h=h
and bs = to-bytes x (heap-list h (size-of TYPE('a)) (ptr-val p)) and
m = length (toplevel-field-names (typ-info-t TYPE('a)))
]

{
fix i
assume i-bound: i < length (toplevel-field-names (typ-info-t TYPE('a)))
from i-bound
have f: toplevel-field-names (typ-info-t TYPE('a)) ! i ∈ set (toplevel-field-names
(typ-info-t TYPE('a)))
by simp
from toplevel-field-names-field-lookup(1) [OF f, where m= 0, OF wf-desc]
obtain s n where
fl: field-lookup (typ-info-t TYPE('a))
[toplevel-field-names (typ-info-t TYPE('a)) ! i] 0 =
Some (s, n) by auto

have lh: length (heap-list h (size-of TYPE('a)) (ptr-val p)) = size-of TYPE('a)
by simp
have v i (take (sz i)
(drop (off i)
(heap-list h
(length (to-bytes x (heap-list h (size-of TYPE('a)) (ptr-val p))))
(ptr-val p)))) =
take (sz i) (drop (off i) (to-bytes x (heap-list h (size-of TYPE('a)) (ptr-val
p))))))
apply (simp add: off-def sz-def v-def fld-def lbs)
apply (simp add: fl)
using mem-type-field-lookup-access-ti-take-drop [OF fl lh, where v=x ]
apply (simp add: to-bytes-def)
done

} note v-focus = this
show ?thesis
apply (subst fold-conv)
apply (rule partition)
subgoal
using no-overflow
by (simp add: local.lense.access-result-size to-bytes-def)
subgoal
apply (subst xmem-type-partition-toplevel-field-names [OF aggregate lbs, sym-

```

```

metric])
  apply (subst fs)
  apply (subst map-map)
  apply (rule arg-cong [where f=concat])
  apply (rule map-cong [OF refl])
  apply (simp add: fld-def off-def sz-def)
  done
subgoal
  apply (subst lbs)
  apply (subst xmem-type-toplevel-field-names-sum-list-size [OF aggregate, sym-
metric])
  apply (simp add: sz-def)
  apply (subst fs)
  apply (simp add: fld-def comp-def)
  done
subgoal for i
  apply (simp add: off-def sz-def)
  apply (simp add: fld-def)
  using xmem-type-toplevel-field-names-sum-list-offset [OF aggregate, where i
= i]
  apply simp
  done
subgoal for i
  by (rule v-focus)
done
qed

```

lemma *heap-update-list-padding-fold-toplevel-field-names:*

```

fixes p::'a ptr
assumes aggregate: aggregate (typ-info-t TYPE('a))
assumes cgrd: c-guard p
assumes lbs: length bs = size-of TYPE('a)
shows
heap-update-list (ptr-val p) (to-bytes x bs) h =
fold
  (λf h. heap-update-list (&(p→[f]))
    (access-ti
      (fst (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))
        x
      (take (size-td (fst (the (field-lookup (typ-info-t TYPE('a)) [f] 0))))
        (drop ((snd (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))) bs)))
        h)
    (toplevel-field-names (typ-info-t TYPE('a))) h (is ?LHS = fold ?F ?fs h)

```

proof –

define *fld::nat ⇒ field-name* **where**

fld = (λ*i*. toplevel-field-names (typ-info-t TYPE('a)) ! *i*)

define *sz:: nat ⇒ nat* **where**

sz = (λ*i*. size-td (fst (the (field-lookup (typ-info-t TYPE('a)) [fld *i*] 0))))

```

define off:: nat ⇒ nat where
  off = (λi. snd (the (field-lookup (typ-info-t TYPE('a)) [fld i] 0)))
define v:: nat ⇒ byte list ⇒ byte list where
  v = (λi. (access-ti
            (fst (the (field-lookup (typ-info-t TYPE('a)) [fld i] 0)))
            x))

have bs': length (to-bytes x bs) = size-of TYPE('a)
by (simp add: local.lense.access-result-size to-bytes-def)
{
  fix i::nat
  assume bound: i < length (toplevel-field-names (typ-info-t TYPE('a)))
  hence toplevel-field-names (typ-info-t TYPE('a)) ! i ∈ set (toplevel-field-names
(typ-info-t TYPE('a)))
    by simp
    from toplevel-field-names-field-lookup-offset-conv [OF this]

  have snd (the (field-lookup (typ-uinfo-t TYPE('a))
    [toplevel-field-names (typ-info-t TYPE('a)) ! i] 0)) =
    snd (the (field-lookup (typ-info-t TYPE('a))
    [toplevel-field-names (typ-info-t TYPE('a)) ! i] 0)) .
} note field-lookup-conv = this

have fs: ?fs = map fld [0..<length(toplevel-field-names (typ-info-t TYPE('a)))]
by (simp add: fld-def map-nth)

from c-guard-no-wrap' [OF cgrd]
have no-overflow: unat (ptr-val p) + size-of TYPE('a) ≤ addr-card .
have fold-conv:
  fold ?F ?fs h =
    fold
      (λi h. heap-update-list (ptr-val p + word-of-nat (off i))
        (v i (take (sz i) (drop (off i) bs))) h)
      [0..<length (toplevel-field-names (typ-info-t TYPE('a)))] h
apply (subst fs)
apply (subst fold-map)
apply (rule fold-cong [OF refl refl])
apply (simp add: fld-def off-def sz-def v-def field-lvalue-def field-offset-def
field-offset-untyped-def)
apply (simp add: field-lookup-conv)
done

note partition= heap-update-list-padding-fold-partition [where a= ptr-val p and
sz=sz and off=off and v=v and h=h
and bs = to-bytes x bs and pbs = bs and
  m = length (toplevel-field-names (typ-info-t TYPE('a)))]
}

```

```

fix i
assume i-bound: i < length (toplevel-field-names (typ-info-t TYPE('a)))
from i-bound
have f: toplevel-field-names (typ-info-t TYPE('a)) ! i ∈ set (toplevel-field-names
(typ-info-t TYPE('a)))
  by simp
from toplevel-field-names-field-lookup(1) [OF f, where m= 0, OF wf-desc]
obtain s n where
  fl: field-lookup (typ-info-t TYPE('a))
    [toplevel-field-names (typ-info-t TYPE('a)) ! i] 0 =
    Some (s, n) by auto

have lh: length (heap-list h (size-of TYPE('a)) (ptr-val p)) = size-of TYPE('a)
by simp
have v i (take (sz i) (drop (off i) bs)) =
  take (sz i) (drop (off i) (to-bytes x bs))
apply (simp add: off-def sz-def v-def fld-def lbs)
apply (simp add: fl)
using mem-type-field-lookup-access-ti-take-drop [OF fl lbs, where v=x ]
apply (simp add: to-bytes-def)
done

} note v-focus = this
show ?thesis
apply (subst fold-conv)
apply (rule partition)
subgoal
  using no-overflow
  by (simp add: local.lense.access-result-size to-bytes-def)
subgoal
apply (subst xmem-type-partition-toplevel-field-names [OF aggregate lbs', sym-
metric])
apply (subst fs)
apply (subst map-map)
apply (rule arg-cong [where f=concat])
apply (rule map-cong [OF refl])
apply (simp add: fld-def off-def sz-def)
done
subgoal
apply (subst lbs')
apply (subst xmem-type-toplevel-field-names-sum-list-size [OF aggregate, sym-
metric])
apply (simp add: sz-def)
apply (subst fs)
apply (simp add: fld-def comp-def)
done
subgoal
apply (subst xmem-type-partition-toplevel-field-names [OF aggregate lbs, sym-
metric])

```

```

    apply (subst fs)
    apply (subst map-map)
    apply (rule arg-cong [where f=concat])
    apply (rule map-cong [OF refl])
    apply (simp add: fld-def off-def sz-def)
  done
subgoal
  apply (subst lbs)
  apply (subst xmem-type-toplevel-field-names-sum-list-size [OF aggregate, symmetric])
  apply (simp add: sz-def)
  apply (subst fs)
  apply (simp add: fld-def comp-def)
  done
subgoal for i
  apply (simp add: off-def sz-def)
  apply (simp add: fld-def)
  using xmem-type-toplevel-field-names-sum-list-offset [OF aggregate, where i = i]
  apply simp
  done
subgoal for i
  by (rule v-focus)
done
qed

```

lemma *heap-update-fold-toplevel-field-names:*

```

  fixes p::'a ptr
  assumes aggregate: aggregate (typ-info-t TYPE('a))
  assumes cgrd: c-guard p
  shows
heap-update p x h =
  fold
    (λf h. heap-update-list (&(p→[f]))
      (access-ti
        (fst (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))
          x
        (heap-list h (size-td (fst (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))) (&(p→[f])))))
      h)
    (toplevel-field-names (typ-info-t TYPE('a))) h (is ?LHS = fold ?F ?fs h)
  using heap-update-list-fold-toplevel-field-names [OF aggregate cgrd]
  by (simp only: heap-update-def)

```

lemma *heap-update-padding-fold-toplevel-field-names:*

```

  fixes p::'a ptr
  assumes aggregate: aggregate (typ-info-t TYPE('a))
  assumes cgrd: c-guard p
  assumes lbs: length bs = size-of TYPE('a)

```


shows
heap-update-padding *p x bs h* =
fold
 ($\lambda f h.$ *heap-update-list* (&(p→[f]))
 (*access-ti*
 (*fst* (*the* (*field-lookup* (*typ-info-t* *TYPE('a)*) [f] 0)))
 x
 (*take* (*size-td* (*fst* (*the* (*field-lookup* (*typ-info-t* *TYPE('a)*) [f] 0))))
 (*drop* ((*snd* (*the* (*field-lookup* (*typ-info-t* *TYPE('a)*) [f] 0)))) *bs*)))
 h)
 (*toplevel-field-names* (*typ-info-t* *TYPE('a)*)) *h* (**is** ?*LHS* = *fold* ?*F* ?*fs* *h*)
using *heap-update-list-padding-fold-toplevel-field-names* [*OF aggregate cgrd lbs*]
by (*simp only: heap-update-padding-def*)

lemma *heap-update-fold-toplevel-field-names-no-padding*:

fixes *p::'a ptr*

assumes *aggregate: aggregate* (*typ-info-t* *TYPE('a)*)

assumes *cgrd: c-guard* *p*

shows

heap-update *p x h* =

fold
 ($\lambda f h.$ *heap-update-list* (&(p→[f]))
 (*access-ti*
 (*fst* (*the* (*field-lookup* (*typ-info-t* *TYPE('a)*) [f] 0)))
 x
 (*heap-list* *h* (*size-td* (*fst* (*the* (*field-lookup* (*typ-info-t* *TYPE('a)*) [f]
 0)))) (&(p→[f])))
 h)
 (*filter* (*Not* *o padding-field-name*) (*toplevel-field-names* (*typ-info-t* *TYPE('a)*)))
h (**is** ?*LHS* = *fold* ?*F* ?*filter-fs* *h*)

proof –

{ **fix** *i*

fix *h*

assume *i-bound: i < length* (*toplevel-field-names* (*typ-info-t* *TYPE('a)*))

assume \neg (*Not* *o padding-field-name*)

 (*toplevel-field-names* (*typ-info-t* *TYPE('a)*) ! *i*)

hence *padding: padding-field-name* (*toplevel-field-names* (*typ-info-t* *TYPE('a)*) ! *i*)

by *simp*

from *i-bound* **obtain** *m s* **where**

fl: field-lookup (*typ-info-t* *TYPE('a)*)

 [(*toplevel-field-names* (*typ-info-t* *TYPE('a)*) ! *i*)] 0 = *Some* (*s*, *m*)

by (*meson nth-mem xmem-type-toplevel-field-names-field-lookup*)

from *field-lookup-padding-field-name(1)* [*OF fl padding wf-padding*]

have *is-padding-tag: is-padding-tag* *s* .

```

have ?F (toplevel-field-names (typ-info-t TYPE('a)) ! i) h = h
apply (simp add: fl)
using is-padding-tag
by (clarsimp simp add: is-padding-tag-def padding-tag-def padding-desc-def
heap-update-list-id')
note pad = this

```

```

have fold ?F ?filter-fs = fold ?F (toplevel-field-names (typ-info-t TYPE('a)))
apply (rule fold-filter-out-id [symmetric])
by (rule pad)
with heap-update-fold-toplevel-field-names [OF aggregate cgrd, where x=x and
h=h]
show ?thesis
by simp
qed

```

lemma heap-list-concat-toplevel-field-names:

```

fixes p::'a ptr
assumes aggregate: aggregate (typ-info-t TYPE('a))
assumes cgrd: c-guard p
shows
heap-list h (size-of TYPE('a)) (ptr-val p) =
concat (map (λf. heap-list h (size-td (fst (the (field-lookup (typ-info-t TYPE('a))
[f] 0)))) (&(p→[f])))
(toplevel-field-names (typ-info-t TYPE('a)))) (is ?LHS = concat (map ?F
?fs))

```

proof –

```

define fld::nat ⇒ field-name where
fld = (λi. toplevel-field-names (typ-info-t TYPE('a)) ! i)
define sz:: nat ⇒ nat where
sz = (λi. size-td (fst (the (field-lookup (typ-info-t TYPE('a)) [fld i] 0))))
define off:: nat ⇒ nat where
off = (λi. snd (the (field-lookup (typ-info-t TYPE('a)) [fld i] 0)))

```

```

have fs: ?fs = map fld [0.. $\text{length}(\text{toplevel-field-names (typ-info-t TYPE('a))})]$ 
by (simp add: fld-def map-nth)
from c-guard-no-wrap' [OF cgrd]
have no-overflow: unat (ptr-val p) + size-of TYPE('a) ≤ addr-card .

```

have concat-conv:

```

concat (map ?F ?fs) =
concat (map (λi. (heap-list h (sz i) (ptr-val p + word-of-nat (off i))))
[0.. $\text{length}(\text{toplevel-field-names (typ-info-t TYPE('a))})]$ )
apply (subst fs)
apply (subst list.map-comp)
apply (rule arg-cong [where f=concat])
apply (rule map-cong [OF refl])
apply (simp add: fld-def off-def sz-def field-lvalue-def field-offset-def field-offset-untyped-def)
by (simp add: toplevel-field-names-field-lookup-offset-conv)

```

```

note partition = heap-list-map-partition [where a = ptr-val p, where sz = sz
and off = off and n = size-of TYPE('a) and
  m = length (toplevel-field-names (typ-info-t TYPE('a))), OF no-overflow]
show ?thesis
  apply (subst concat-conv)
  apply (rule partition)
  subgoal
    apply (subst xmem-type-toplevel-field-names-sum-list-size [OF aggregate, symmetric])
    apply (simp add: sz-def)
    apply (subst fs)
    apply (simp add: fld-def comp-def)
    done
  subgoal for i
    apply (simp add: off-def sz-def)
    apply (simp add: fld-def)
    using xmem-type-toplevel-field-names-sum-list-offset [OF aggregate, where i
= i]
    apply simp
    done
  done
qed

```

```

lemma h-val-concat-toplevel-field-names:
  fixes p::'a ptr
  assumes aggregate: aggregate (typ-info-t TYPE('a))
  assumes cgrd: c-guard p
  shows
    h-val h p =
      from-bytes
        (concat (map (λf. heap-list h (size-td (fst (the (field-lookup (typ-info-t
TYPE('a)) [f] 0)))) (&(p→[f])))
          (toplevel-field-names (typ-info-t TYPE('a))))
    using heap-list-concat-toplevel-field-names [OF aggregate cgrd]
    by (simp add: h-val-def)

end

```

```

lemma set-field-names-u-all-field-names-conv:
  set (field-names-u (typ-uinfo-t TYPE('a::mem-type)) t) =
    {f. f ∈ set (all-field-names (typ-info-t TYPE('a))) ∧
      (∃ s. field-ti TYPE('a) f = Some s ∧ export-uinfo s = t)}
  apply standard
  subgoal
    apply (clarsimp simp add: typ-uinfo-t-def)

```

apply (*rule conjI*)
apply (*metis (mono-tags, lifting) all-field-names-export-uinfo field-names-subset-all-field-names(1) in-mono*)
by (*metis field-lookup-field-ti field-names-Some2(1) field-names-u-field-names-export-uinfo-conv(1) wf-desc*)
subgoal
apply *clarsimp*
by (*metis all-field-names-exists-field-names-u(1) all-field-names-export-uinfo field-lookup-field-ti field-names-Some2(1) field-names-u-field-names-export-uinfo-conv(1) fold-tyt-uinfo-t option.inject wf-desc*)
done

lemma *field-names-u-all-field-names-conv*:
 $field_names_u\ (typ_uinfo_t\ TYPE('a::mem_type))\ t =$
 $filter\ (\lambda f.\ (\exists s.\ field_ti\ TYPE('a)\ f = Some\ s \wedge export_uinfo\ s = t))$
 $(all_field_names\ (typ_info_t\ TYPE('a)))$
using *field-names-u-filter-all-field-names-conv(1)*
by (*smt (verit) all-field-names-exists-field-names-u(1) all-field-names-export-uinfo field-lookup-field-ti field-lookup-uinfo-Some-rev field-ti-def filter-same-eq fold-tyt-uinfo-t mem-Collect-eq option.sel set-field-names-u-all-field-names-conv set-filter wf-desc-tyt-tag*)

lemma *set-field-names-all-field-names-conv*:
 $set\ (field_names\ (typ_info_t\ TYPE('a::mem_type))\ t) =$
 $\{f.\ f \in set\ (all_field_names\ (typ_info_t\ TYPE('a))) \wedge$
 $(\exists s.\ field_ti\ TYPE('a)\ f = Some\ s \wedge export_uinfo\ s = t)\}$
using *set-field-names-u-all-field-names-conv [of t]*
unfolding *typ-uinfo-t-def*
by (*simp add: field-names-u-field-names-export-uinfo-conv(1)*)

lemma *field-names-all-field-names-conv*:
 $field_names\ (typ_info_t\ TYPE('a::mem_type))\ t =$
 $filter\ (\lambda f.\ \exists s.\ field_ti\ TYPE('a)\ f = Some\ s \wedge export_uinfo\ s = t)\ (all_field_names$
 $(typ_info_t\ TYPE('a)))$
using *field-names-u-all-field-names-conv [of t]*
unfolding *typ-uinfo-t-def*
by (*simp add: field-names-u-field-names-export-uinfo-conv(1)*)

lemma *field-lookup-qualified-padding-field-name*:
fixes
 $t :: ('a, 'b)\ typ_info$ **and**
 $st :: ('a, 'b)\ typ_info_struct$ **and**
 $ts :: ('a, 'b)\ typ_info_tuple\ list$ **and**
 $x :: ('a, 'b)\ typ_info_tuple$
shows
 $field_lookup\ t\ f\ n = Some\ (s,\ m) \implies qualified_padding_field_name\ f \implies wf_padding$
 $t \implies$
 $is_padding_tag\ s$
 $field_lookup_struct\ st\ f\ n = Some\ (s,\ m) \implies qualified_padding_field_name\ f \implies$

```

wf-padding-struct st  $\implies$ 
  is-padding-tag s
field-lookup-list ts f n = Some (s, m)  $\implies$  qualified-padding-field-name f  $\implies$  wf-padding-list
ts  $\implies$ 
  is-padding-tag s
field-lookup-tuple x f n = Some (s, m)  $\implies$  qualified-padding-field-name f  $\implies$ 
wf-padding-tuple x  $\implies$ 
  is-padding-tag s
proof (induct t and st and ts and x arbitrary: f n m and f n m and f n m and
f n m)
  case (TypDesc nat typ-struct list)
then show ?case by (auto split: if-split-asm option.splits)
next
case (TypScalar nat1 nat2 a)
  then show ?case by simp
next
  case (TypAggregate list)
  then show ?case by auto
next
case Nil-typ-desc
then show ?case by simp
next
  case (Cons-typ-desc dt-tuple list)
then show ?case by (auto split: if-split-asm option.splits)
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
  apply clarsimp
  by (metis field-lookup-empty last-tl list.exhaust-sel option.distinct(1)
option.inject prod.inject qualified-pading-field-name-cons qualified-pading-field-name-single)
qed

```

lemma all-field-names-empty-typ-info [simp]: all-field-names (empty-typ-info algn n) = []
by (simp add: empty-typ-info-def)

lemma all-field-names-no-padding-empty-typ-info [simp]:
filter (Not o qualified-padding-field-name) (all-field-names (empty-typ-info algn n)) = []
by (simp)

lemma all-field-names-list-append [simp]:
all-field-names-list (xs @ ys) = all-field-names-list xs @ all-field-names-list ys
by (induct xs) auto

lemma all-field-names-extend-ti:
fixes
t :: 'a xtyp-info **and**

```

st :: 'a xtyp-info-struct and
ts :: 'a xtyp-info-tuple list and
x :: 'a xtyp-info-tuple
shows
all-field-names (extend-ti t s n fn d) = all-field-names t @ (map ((#) fn) (all-field-names
s))
all-field-names-struct (extend-ti-struct st s fn d) = all-field-names-struct st @
(map ((#) fn) (all-field-names s))
all-field-names-list ts = all-field-names-list ts
all-field-names-tuple x = all-field-names-tuple x
by (induct t and st and ts and x) auto

```

lemma *all-field-names-adjust-ti'*:

```

fixes
t :: 'a xtyp-info and
st :: 'a xtyp-info-struct and
ts :: 'a xtyp-info-tuple list and
x :: 'a xtyp-info-tuple
shows
all-field-names (map-td (λn algn. update-desc acc upd) (update-desc acc upd) t) =
all-field-names t
all-field-names-struct (map-td-struct (λn algn. update-desc acc upd) (update-desc
acc upd) st) =
all-field-names-struct st
all-field-names-list (map-td-list (λn algn. update-desc acc upd) (update-desc acc
upd) ts) =
all-field-names-list ts
all-field-names-tuple (map-td-tuple (λn algn. update-desc acc upd) (update-desc acc
upd) x) =
all-field-names-tuple x
by (induct t and st and ts and x) (auto)

```

lemma *all-field-names-adjust-ti[simp]*:

```

all-field-names (adjust-ti t acc upd) = all-field-names t
by (simp add: all-field-names-adjust-ti' adjust-ti-def)

```

lemma *all-field-names-no-padding-ti-pad-combine*:

```

filter (Not o qualified-padding-field-name) (all-field-names (ti-pad-combine n t))
=
filter (Not o qualified-padding-field-name) (all-field-names t)
by (simp add: ti-pad-combine-def Let-def all-field-names-extend-ti(1) padding-field-name-pad)

```

lemma *all-field-names-ti-typ-combine*:

```

all-field-names (ti-typ-combine (t-b::'b::c-type itself) acc upd algn fn t) =
all-field-names t @ (map ((#) fn) (all-field-names (typ-info-t TYPE('b))))
by (simp add: ti-typ-combine-def all-field-names-extend-ti)

```

lemma *all-field-names-no-padding-ti-typ-combine*:

```

assumes not-padding: ¬ padding-field-name fn

```

shows *filter (Not o qualified-padding-field-name) (all-field-names (ti-typ-combine (t-b::'b::c-type itself) acc upd algn fn t)) = filter (Not o qualified-padding-field-name) (all-field-names t) @ (map ((#) fn) (filter (Not o qualified-padding-field-name) (all-field-names (typ-info-t TYPE('b))))))*

proof –

from *not-padding have eq: Not o qualified-padding-field-name o (#) fn = Not o qualified-padding-field-name*

apply –

apply *(rule ext)*

apply *(auto simp add: neq-Nil-conv)*

done

from *not-padding show ?thesis*

apply *(simp add: all-field-names-ti-typ-combine all-field-names-extend-ti)*

apply *(simp add: filter-map eq)*

done

qed

lemma *all-field-names-no-padding-ti-typ-pad-combine:*

assumes *not-padding: ¬ padding-field-name fn*

shows *filter (Not o qualified-padding-field-name) (all-field-names (ti-typ-pad-combine (t-b::'b::c-type itself) acc upd algn fn t)) =*

filter (Not o qualified-padding-field-name) (all-field-names t) @

(map ((#) fn) (filter (Not o qualified-padding-field-name) (all-field-names (typ-info-t TYPE('b))))))

by *(simp add: ti-typ-pad-combine-def Let-def*

all-field-names-no-padding-ti-pad-combine all-field-names-no-padding-ti-typ-combine [OF not-padding])

lemma *all-field-names-map-align[simp]: all-field-names (map-align n t) = all-field-names t*

by *(cases t) (simp add: map-align-def)*

lemma *all-field-names-no-padding-final-pad:*

filter (Not o qualified-padding-field-name) (all-field-names (final-pad n t))

= filter (Not o qualified-padding-field-name) (all-field-names t)

by *(simp add: final-pad-def Let-def all-field-names-no-padding-ti-pad-combine)*

lemmas *all-field-names-filter-no-padding-combinator-simps =*

all-field-names-no-padding-empty-typ-info

all-field-names-no-padding-final-pad

all-field-names-no-padding-ti-typ-pad-combine

all-field-names-no-padding-ti-typ-combine

lemma *all-field-names-array-tag-n: all-field-names ((array-tag-n n)::('a::c-type,'b::finite) array xtyp-info) =*

□ #

concat (map (λi. (map ((#) (replicate i CHR "1")) (all-field-names (typ-info-t

```

TYPE('a)))) [0..<n])
  apply (induct n)
  apply (simp add: atn-base)
  apply (auto simp add: atn-rec all-field-names-ti-typ-combine)
done

```

lemma *all-field-names-array*:

```

all-field-names (typ-info-t TYPE('a::c-type['b::finite])) =
  [] #
  concat (map (λi. (map ((#) (replicate i CHR "1")) (all-field-names (typ-info-t
TYPE('a)))) [0..<CARD('b)]))
  by (simp add: typ-info-array array-tag-def all-field-names-array-tag-n)

```

lemma *not-padding-field-name-replicate-1*[simp]: *padding-field-name* (replicate n CHR "1") = False
 by (cases n) auto

lemma *non-empty-not-padding-field-conv*: (x ≠ [] → ¬ *padding-field-name* (last x)) ↔ ¬ *qualified-padding-field-name* x
 by (cases x) auto

named-theorems *all-field-names-no-padding* and *set-all-field-names-no-padding*

definition *all-field-names-no-padding* :: ('a, 'b) typ-desc ⇒ *qualified-field-name* list
 where
all-field-names-no-padding t = filter (Not o *qualified-padding-field-name*) (all-field-names t)

lemma *all-field-names-no-padding-combinator-simps*:

```

all-field-names-no-padding (empty-typ-info algn nm) = []
all-field-names-no-padding (final-pad n t) = all-field-names-no-padding t
¬ padding-field-name fn ⇒
all-field-names-no-padding (ti-typ-pad-combine (t-b::'b::c-type itself) acc upd algn
fn t) =
  all-field-names-no-padding t @
  map ((#) fn)
  (all-field-names-no-padding (typ-info-t TYPE('b::c-type)))
¬ padding-field-name fn ⇒
all-field-names-no-padding (ti-typ-combine (t-b::'b::c-type itself) acc upd algn fn
t) =
  all-field-names-no-padding t @
  map ((#) fn)
  (all-field-names-no-padding (typ-info-t TYPE('b::c-type)))
by (simp-all add: all-field-names-no-padding-def all-field-names-filter-no-padding-combinator-simps)

```

lemma *all-field-names-filter-no-padding-array*:

```

filter (Not o qualified-padding-field-name) (all-field-names (typ-info-t TYPE('a::c-type['b::finite])))
=

```



```

[] #
concat (map (λi. (map ((#) (replicate i CHR "1"))
  (filter (Not o qualified-padding-field-name) (all-field-names (typ-info-t TYPE('a))))))
[0..<CARD('b)])
apply (simp add: all-field-names-array)
apply (simp add: filter-concat)
apply (simp add: comp-def filter-map add: non-empty-not-padding-field-conv )
done

```

lemma *all-field-names-no-padding-array*[*all-field-names-no-padding*]:
all-field-names-no-padding (typ-info-t TYPE('a::c-type['b::finite])) =
[] #
concat (map (λi. (map ((#) (replicate i CHR "1"))
 (all-field-names-no-padding (typ-info-t TYPE('a)))) [0..<CARD('b)])
by (simp add: all-field-names-no-padding-def all-field-names-filter-no-padding-array)

lemma *set-all-field-names-no-padding-array*[*set-all-field-names-no-padding*]:
set (all-field-names-no-padding (typ-info-t TYPE('a::c-type['b::finite]))) =
insert []
(⋃ x∈{0..<CARD('b)}.
 (#) (replicate x CHR "1") '
 set (all-field-names-no-padding (typ-info-t TYPE('a))))

by (simp add: all-field-names-no-padding-array)

lemma *sub-typ-trans*: $t \leq_{\tau} s \implies s \leq_{\tau} w \implies t \leq_{\tau} w$
by (simp add: sub-typ-def)

lemma *sub-typ-trans-rev*: $s \leq_{\tau} w \implies t \leq_{\tau} s \implies t \leq_{\tau} w$
by (simp add: sub-typ-def)

lemma *element-typ-le-array-typ*: $\text{typ-uinfo-t TYPE('a::mem-type)} \leq \text{typ-uinfo-t TYPE('a['b::finite])}$

proof –

```

have 0 < CARD('b) by simp
from field-lookup-array [OF this, of 0, simplified]
have field-lookup (typ-info-t TYPE('a['b])) [[]] 0 =
  Some (adjust-ti (typ-info-t TYPE('a)) (λx. x.[0]) (λx f. Arrays.update f 0 x),
0) .
from td-set-field-lookupD [OF this]
have (adjust-ti (typ-info-t TYPE('a)) (λx. x.[0]) (λx f. Arrays.update f 0 x), 0)
  ∈ td-set (typ-info-t TYPE('a['b])) 0.
from td-set-export-uinfoD [OF this]
show ?thesis
by (auto simp add: export-uinfo-adjust-ti typ-uinfo-t-def typ-tag-le-def)

```

qed

lemma *element-typ-subtyp-array-typ*: $TYPE('a::mem-type) \leq_{\tau} TYPE('a['b::finite])$
by (*simp add: sub-typ-def element-typ-le-array-typ*)

lemma *field-lookup-sub-typ*:
assumes *fl*: *field-lookup* (*typ-info-t* $TYPE('a::c-type)$) *f* 0 = *Some* (*s*, *m*)
assumes *match*: *export-uinfo* *s* = *export-uinfo* (*typ-info-t* $TYPE('b::c-type)$)
shows $TYPE('b) \leq_{\tau} TYPE('a)$
using *match* *td-set-export-uinfoD* [*OF* *td-set-field-lookupD* [*OF* *fl*]]
by (*auto simp add: sub-typ-def typ-tag-le-def typ-uinfo-t-def*)

lemma *field-lookup-sub-typ'*:
assumes *fl*: *field-lookup* (*typ-info-t* $TYPE('a::c-type)$) *f* 0 \equiv *Some* (*adjust-ti*
(*typ-info-t* $TYPE('b::mem-type)$) *acc* *upd*, *n*)
assumes *fg-cons*: *fg-cons* *acc* *upd*
shows $TYPE('b) \leq_{\tau} TYPE('a)$
apply (*rule* *field-lookup-sub-typ* [*of* *f*])
apply (*simp add: fl*)
using *fg-cons*
by (*simp add: export-uinfo-adjust-ti*)

lemma *all-field-names-no-padding-word*[*all-field-names-no-padding*]:
all-field-names-no-padding (*typ-info-t* ($TYPE('a::len8\ word)$)) = $\{\}\{\}$
by (*simp add: all-field-names-no-padding-def*)

lemma *set-all-field-names-no-padding-word*[*set-all-field-names-no-padding*]:
set (*all-field-names-no-padding* (*typ-info-t* ($TYPE('a::len8\ word)$))) = $\{\}\{\}$
by (*simp add: all-field-names-no-padding-def*)

lemma *all-field-names-no-padding-ptr*[*all-field-names-no-padding*]:
all-field-names-no-padding (*typ-info-t* ($TYPE('a::c-type\ ptr)$)) = $\{\}\{\}$
by (*simp add: all-field-names-no-padding-def*)

lemma *set-all-field-names-no-padding-ptr*[*set-all-field-names-no-padding*]:
set (*all-field-names-no-padding* (*typ-info-t* ($TYPE('a::c-type\ ptr)$))) = $\{\}\{\}$
by (*simp add: all-field-names-no-padding-def*)

definition *field-names-no-padding*::('a, 'b) *typ-info* \Rightarrow *typ-uinfo* \Rightarrow *qualified-field-name*
list

where *field-names-no-padding* *t* *s* = *filter* (*Not* *o* *qualified-padding-field-name*)
(*field-names* *t* *s*)

lemma *set-field-names-no-padding-all-field-names-no-padding-conv*:
set (*field-names-no-padding* (*typ-info-t* $TYPE('a::mem-type)$) *t*) =
 $\{f \in \text{set } (\text{all-field-names-no-padding } (\text{typ-info-t } TYPE('a)))$
 $\exists s\ n. \text{field-lookup } (\text{typ-info-t } TYPE('a))\ f\ 0 = \text{Some } (s, n) \wedge \text{export-uinfo } s =$
 $t\}$
using *set-field-names-all-field-names-conv* [**where** 'a='a **and** t=t]

by (*auto simp add: field-ti-def all-field-names-no-padding-def field-names-no-padding-def split: option.split*)

lemma *field-names-no-padding-all-field-names-no-padding-conv:*

field-names-no-padding (typ-info-t TYPE('a::mem-type)) t =
filter (λf. ∃ s n. field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n) ∧ export-uintf s = t)
(all-field-names-no-padding (typ-info-t TYPE('a)))
using *field-names-all-field-names-conv [where 'a='a and t=t]*
by (*auto simp add: field-ti-def all-field-names-no-padding-def field-names-no-padding-def intro: filter-same-eq split: option.split*)

lemma *subset-all-field-names-no-padding-all-field-names:*

set (all-field-names-no-padding t) ⊆ set (all-field-names t)
by (*simp add: all-field-names-no-padding-def*)

lemma *all-field-names-typ-uintf-t-conv:*

all-field-names (typ-info-t (TYPE('a::c-type))) = all-field-names (typ-uintf-t (TYPE('a::c-type)))
by (*simp add: all-field-names-export-uintf typ-uintf-t-def*)

lemma *all-field-names-no-padding-typ-uintf-t-conv:*

all-field-names-no-padding (typ-info-t (TYPE('a::c-type))) = all-field-names-no-padding (typ-uintf-t (TYPE('a::c-type)))
by (*simp add: all-field-names-no-padding-def all-field-names-typ-uintf-t-conv*)

lemma *update-ti-to-bytes-p[simp]:*

update-ti (typ-info-t TYPE('a::xmem-type)) (to-bytes-p (v::'a)) w = v
apply (*subst field-lookup-update-ti-from-bytes-field-conv[OF field-lookup-empty , where xbs=replicate (size-of TYPE('a)) 0 and v=v]*)
unfolding *typ-uintf-t-def* **apply** (*rule refl*)
apply (*simp add: size-of-def*)
apply (*simp add: to-bytes-p-def to-bytes-def*)
apply (*subst field-lookup-update-ti-from-bytes-field-conv[symmetric, where vf=v, OF field-lookup-empty]*)
apply (*simp-all add: size-of-def typ-uintf-t-def*)
apply (*simp add: to-bytes-p-def to-bytes-def*)
apply (*intro lense.update-access*)
done

context *mem-type*

begin

lemma *mem-type-access-ti-super-update-bs:*

assumes *fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)*
assumes *lbs: length bs = size-of TYPE('a)*
assumes *lbs': length bs' = size-td s*
shows *access-ti (typ-info-t TYPE('a))*
(update-ti s (access-ti₀ s w) v) (super-update-bs bs' bs n) =
super-update-bs (access-ti s w bs') (access-ti (typ-info-t TYPE('a)) v bs) n

proof –
have *field-lookup* (*typ-info-t* *TYPE('a)*) *f* *0* = *Some* (*s*, *0* + *n*) **using** *fl* **by** *simp*
from *access-ti-super-update-bs-of-wf*(*1*)[*OF this lbs' lbs[unfolding size-of-def]*]
show *?thesis* **by** *simp*
qed
end

lemma *update-ti-undefined*[*simp*]:
assumes *NO-MATCH* *undefined* *w* **assumes** *bs*: *length* *bs* = *size-of* *TYPE('a)*
shows *update-ti* (*typ-info-t* *TYPE('a::xmem-type)*) *bs* *w* =
update-ti (*typ-info-t* *TYPE('a)*) *bs* *undefined*
by (*simp* *add*: *bs* *size-of-def* *upd-rf* *update-ti-update-ti-t*)

11.28 *heap-upd* and *heap-upd-list*

11.29 *heap-upd*

lemma *heap-upd-id*: *heap-upd* (*p*::*'a::xmem-type* *ptr*) *id* = *id*
by (*simp* *add*: *heap-upd-def* *fun-eq-iff* *xmem-type-class*.*heap-update-id*)

lemma *heap-upd-const*: *heap-upd* *p* (λ -. *x*) = *heap-update* *p* *x*
by (*simp* *add*: *heap-upd-def*[*abs-def*])

lemma *heap-upd-comp*: *heap-upd* (*p*::*'a::xmem-type* *ptr*) (*f* \circ *g*) = *heap-upd* *p* *f* \circ
heap-upd *p* *g*
by (*simp* *add*: *heap-upd-def* *fun-eq-iff* *h-val-heap-update* *heap-update-collapse*)

lemma *hrs-mem-update-heap-upd*:
hrs-mem-update (*heap-upd* *p* *g*) *h* = *hrs-mem-update* (*heap-update* *p* (*g* (*h-val*
(*hrs-mem* *h*) *p*))) *h*
by (*cases* *h*) (*simp* *add*: *hrs-mem-update-def* *heap-upd-def* *hrs-mem-def*)

lemma *heap-update-eq-heap-upd-list*:
fixes *p* :: *'a::mem-type* *ptr*
shows *heap-update* *p* *x* =
heap-upd-list (*size-of* *TYPE('a)*) (*ptr-val* *p*) (*access-ti* (*typ-info-t* *TYPE('a)*) *x*)
by (*simp* *add*: *heap-update-def* *fun-eq-iff* *heap-upd-list-def* *to-bytes-def*)

lemma *heap-upd-list-id*[*simp*]: *heap-upd-list* *n* *p* *id* = *id*
by (*simp* *add*: *heap-upd-list-def* *fun-eq-iff* *heap-update-list-id'*)

lemma *heap-upd-list-access-ti-typ-info-t*[*simp*]:
sz = *size-of* *TYPE('a)* \implies
heap-upd-list *sz* *p* (*access-ti* (*typ-info-t* *TYPE('a::xmem-type)*) *v*) =
heap-update (*PTR('a)* *p*) *v*
by (*simp* *add*: *heap-update-def* *heap-upd-list-def* *fun-eq-iff* *to-bytes-def*)

lemma *heap-list-heap-upd-list*:
 $n \leq \text{addr-card} \implies \text{length } xs = n \implies (\bigwedge xs. \text{length } xs = n \implies \text{length } (f \text{ } xs) = n)$
 \implies
 $\text{heap-list } (\text{heap-upd-list } n \text{ } p \text{ } f \text{ } h) \text{ } n \text{ } p = f \text{ } (\text{heap-list } h \text{ } n \text{ } p)$
unfolding *heap-upd-list-def*
by (*subst heap-list-heap-update-list-id simp-all*)

lemma *heap-upd-list-comp*:
assumes $n \leq \text{addr-card}$ $\text{length } xs = n$
assumes $f: \bigwedge xs. \text{length } xs = n \implies \text{length } (f \text{ } xs) = n$
assumes $g: \bigwedge xs. \text{length } xs = n \implies \text{length } (g \text{ } xs) = n$
shows $\text{heap-upd-list } n \text{ } p \text{ } (f \circ g) = \text{heap-upd-list } n \text{ } p \text{ } f \circ \text{heap-upd-list } n \text{ } p \text{ } g$
using *assms*
by (*simp add: fun-eq-iff heap-upd-list-def heap-list-heap-update-list-id heap-update-list-overwrite*)

lemma *heap-update-list-append*:
fixes $v :: \text{word8}$
shows $\text{heap-update-list } s \text{ } (xs \text{ } @ \text{ } ys) \text{ } hp = \text{heap-update-list } (s + \text{of-nat } (\text{length } xs)) \text{ } ys \text{ } (\text{heap-update-list } s \text{ } xs \text{ } hp)$
proof (*induct xs arbitrary: ys rule: rev-induct*)
case *Nil*
show *?case* **by** *simp*
next
case (*snoc v' vs'*)
show *?case*
apply (*simp add: snoc.hyps field-simps*)
apply (*rule arg-cong [where f = heap-update-list (1 + (s + of-nat (length vs')) ys]*)
apply (*rule ext*)
apply *simp*
done
qed

lemma *heap-update-list-super-update-bs*:
 $\text{length } bs + n \leq \text{length } bs' \implies \text{length } bs' \leq \text{addr-card} \implies$
 $\text{heap-update-list } (p + \text{of-nat } n) \text{ } bs \text{ } (\text{heap-update-list } p \text{ } bs' \text{ } h) =$
 $\text{heap-update-list } p \text{ } (\text{super-update-bs } bs \text{ } bs' \text{ } n) \text{ } h$
apply (*subst super-update-bs-take-drop[symmetric, of n length bs bs']*)
unfolding *super-update-bs-def heap-update-list-append*
apply *simp-all*
apply (*subst heap-update-list-commute*)
subgoal
apply (*subst (2) add-0-right[symmetric]*)
unfolding *intvl-disj-offset add.assoc*
apply (*cases length bs + n = length bs'*)
apply *simp*
apply (*intro intvl-disj-left*)
apply (*simp-all add: addr-card-def card-word unat-of-nat-eq*)

```

done
apply (subst heap-update-list-overwrite)
apply simp-all
done

lemma update-ti-adjust-ti:
  fixes t::'a xtyp-info
  and st::'a xtyp-info-struct
  and ts::'a xtyp-info-tuple list
  and x::'a xtyp-info-tuple
  assumes fg-cons: fg-cons f g
  shows
    update-ti (adjust-ti t (f::'b ⇒ 'a) (g::'a ⇒ 'b ⇒ 'b)) bs v = g (update-ti t bs (f v)) v
    update-ti-struct (map-td-struct (λn algn d. update-desc f g d) (update-desc f g) st) bs v = g (update-ti-struct st bs (f v)) v
    update-ti-list (map-td-list (λn algn d. update-desc f g d) (update-desc f g) ts) bs v = g (update-ti-list ts bs (f v)) v
    update-ti-tuple (map-td-tuple (λn algn d. update-desc f g d) (update-desc f g) x) bs v = g (update-ti-tuple x bs (f v)) v
  unfolding adjust-ti-def
proof (induct t and st and ts and x arbitrary: v bs and v bs and v bs and v bs)
  case (TypDesc nat typ-struct list)
  then show ?case by auto
next
  case (TypScalar nat1 nat2 a)
  then show ?case by (auto simp add: update-desc-def)
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case using fg-cons by (auto simp add: fg-cons-def)
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case apply (cases dt-tuple) using fg-cons by (auto simp add: fg-cons-def)
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by auto
qed

lemma field-ti-field-lookupE:
   $\llbracket \text{field-ti } \text{TYPE}('a :: \text{c-type}) f = \text{Some } t; \bigwedge n. \llbracket \text{field-lookup } (\text{typ-info-t } \text{TYPE}('a)) f 0 = \text{Some } (t, n) \rrbracket \implies P \rrbracket \implies P$ 
  unfolding field-ti-def
  by (clarsimp split: option.splits)

```

lemma *field-ti-append-field-lookup*:
 $field-ti\ TYPE('a::wf-type)\ f = Some\ u \implies field-lookup\ u\ g\ l = Some\ (v, k) \implies$
 $field-ti\ TYPE('a)\ (f\ @\ g) = Some\ v$
by (*auto simp: field-ti-def field-lookup-append split: option.splits*)

lemma *field-tiD*:
 $field-ti\ TYPE('a::mem-type)\ f = Some\ t \implies$
 $field-lookup\ (typ-info-t\ TYPE('a))\ f\ 0 = Some\ (t, field-offset\ TYPE('a)\ f)$
by (*metis field-lookup-offset-eq field-ti-field-lookupE*)

lemma *wf-fd-field-lookup-mem-type*: $field-lookup\ (typ-info-t\ (TYPE('a::mem-type)))$
 $f\ m = Some\ (s, n) \implies wf-fd\ s$
apply (*erule wf-fd-field-lookupD*)
by *simp*

lemma *wf-fd-field-ti-mem-type*: $field-ti\ TYPE('a::mem-type)\ f = Some\ s \implies wf-fd$
 s
unfolding *field-ti-def*
by (*auto simp add: wf-fd-field-lookup-mem-type split: option.splits prod.splits*)

lemma *field-lookup-offset-non-zero*:
 $NO-MATCH\ 0\ m \implies field-lookup\ t\ f\ 0 = Some\ (t', n) \implies field-lookup\ t\ f\ m =$
 $Some\ (t', m + n)$
by (*simp add: field-lookup-offset' [where m'=0]*)

lemma *field-lookup-append-Some*:
assumes *wf: wf-desc t*
shows $field-lookup\ t\ (f@g)\ n = Some\ (s, m) \implies$
 $\exists\ w\ k. field-lookup\ t\ f\ n = Some\ (w, k) \wedge field-lookup\ w\ g\ k = Some\ (s, m)$
using *wf*
proof (*induct n \equiv length (f @ g) arbitrary: f g n s m t*)
case *0*
then show *?case by auto*
next
case (*Suc n*)
show *?case*
by (*metis Suc.premis(1) Suc.premis(2)*
 $field-lookup-prefix-None''(1)\ field-lookup-prefix-Some''(1)\ not-Some-eq-tuple$)
qed

11.30 *merge-ti*

definition *merge-ti* :: $('a\ field-desc, 'b)\ typ-desc \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ **where**
 $merge-ti\ t\ a\ b = update-ti\ t\ (access-ti_0\ t\ a)\ b$

lemma *merge-ti-adjust-ti[simp]*:
 $fg-cons\ g\ s \implies merge-ti\ (adjust-ti\ (typ-info-t\ TYPE('a::xmem-type))\ g\ s) = (\lambda a.$
 $s\ (g\ a))$
by (*simp add: fun-eq-iff merge-ti-def update-ti-adjust-ti*)

lemma *is-scene-merge-ti*:
assumes t : *field-ti* *TYPE*('a::*xmem-type*) $f = \text{Some } t$ **shows** *is-scene* (*merge-ti* t)
proof –
interpret *padding-lense* *access-ti* t *update-ti* t *size-td* t
using t [*THEN field-tiD*] **by** (*rule field-lookup-padding-lense*)

show *?thesis*
proof
show *merge-ti* t (*merge-ti* t a b) $c = \text{merge-ti } t$ a c **for** a b c
by (*simp-all add: merge-ti-def access-ti₀-def*) (*metis access-update update-access*)
qed (*simp-all add: merge-ti-def access-ti₀-def update-access double-update*)
qed

lemma *merge-ti-update-ti-disj*:
assumes $*$: *field-ti* *TYPE*('a::*xmem-type*) $f = \text{Some } t$ *field-ti* *TYPE*('a::*xmem-type*)
 $g = \text{Some } u$
and f - g : *disj-fn* f g
assumes bs : *length* $bs = \text{size-td } u$
shows *merge-ti* t x (*update-ti* u bs y) = *update-ti* u bs (*merge-ti* t x y)
proof –
have eq [*simp*]: $\bigwedge ts$ us x . *length* $ts = \text{size-td } t \implies \text{length } us = \text{size-td } u \implies$
update-ti t ts (*update-ti* u us x) = *update-ti* u us (*update-ti* t ts x)
using *fu-commutes-lookup-disjD*[*OF* $*$ [*THEN field-tiD*] f - g]
by (*simp add: wf-lf-fdp fu-commutes-def update-ti-t-def split: if-splits*)
have [*simp*]: *wf-fd* t *wf-fd* u
using $*$ [*THEN wf-fd-field-ti-mem-type*] **by** *auto*
show *?thesis*
by (*simp add: merge-ti-def access-ti₀-def length-fa-ti* bs)
qed

lemma *disjnt-scene-merge-ti*:
assumes $*$: *field-ti* *TYPE*('a) $f = \text{Some } t$ *field-ti* *TYPE*('a::*xmem-type*) $g =$
 $\text{Some } u$
and f - g : *disj-fn* f g
shows *disjnt-scene* (*merge-ti* t) (*merge-ti* u)
apply (*clarsimp simp: disjnt-scene-def*)
apply (*subst* (2 3) *merge-ti-def*)
apply (*rule merge-ti-update-ti-disj*[*OF assms*])
apply (*simp add:* $*$ [*THEN wf-fd-field-ti-mem-type*] *length-fa-ti access-ti₀-def*)
done

lemma *access-ti-merge-ti-sub*:
assumes r : *field-ti* *TYPE*('a::*xmem-type*) $r = \text{Some } t$ **and** t : *field-lookup* t f 0
 $= \text{Some } (v, n)$
and bs : *length* $bs = \text{size-td } v$
shows *access-ti* v (*merge-ti* t x y) $bs = \text{access-ti } v$ x bs
proof –


```

let ?xbs = super-update-bs bs (replicate (size-td t) 0) n
let ?x-y = update-ti t (access-ti t x (replicate (size-td t) 0)) y
have wf-t: wf-fd t wf-desc t wf-size-desc t
  using wf-fd-field-ti-mem-type[OF r] r
    field-lookup-wf-desc-pres(1)[of typ-info-t TYPE('a) r 0]
    field-lookup-wf-size-desc-pres(1)[of typ-info-t TYPE('a) r 0]
  by (auto elim: field-ti-field-lookupE)

have length-xbs: length ?xbs = size-td t
  using field-lookup-offset-size'[OF t] by (subst length-super-update-bs) (simp-all
add: bs)
have n-t: n ≤ length (replicate (size-td t) 0)
  using field-lookup-offset-size'[OF t] by auto
have eq: access-ti v x bs =
  take (size-td v) (drop n (access-ti t x (super-update-bs bs (replicate (size-td t)
0) n)))
  for x
  using field-lookup-access-ti-take-drop[OF t wf-t length-xbs]
  unfolding take-drop-super-update-bs[OF bs n-t] .

from wf-fd-consD[OF wf-t(1)]
have eq-ac:
  length bs = size-td t ⇒ length bs' = size-td t ⇒
  access-ti t (update-ti t bs v) bs' = access-ti t (update-ti t bs v') bs'
  length bs = size-td t ⇒ update-ti t (access-ti t v bs) v = v
  for bs bs' v v'
  by (auto simp: fd-cons-def fd-cons-desc-def fd-cons-update-access-def
fd-cons-access-update-def update-ti-t-def length-fa-ti wf-t)

have [simp]: access-ti v ?x-y bs = access-ti v x bs unfolding eq
  by (subst eq-ac(1)[of - - x]) (simp-all add: wf-t length-fa-ti length-xbs eq-ac)
show ?thesis
  by (simp add: merge-ti-def fd-cons-access-update-def access-ti0-def)
qed

lemma access-ti-merge-ti-disj:
assumes f: field-ti TYPE('a::xmem-type) f = Some t
assumes g: field-ti TYPE('a::xmem-type) g = Some u
  and f-g: disj-fn f g
  and bs: length bs = size-td u
shows access-ti u (merge-ti t x y) bs = access-ti u y bs
apply (subst is-scene.idem[symmetric, OF is-scene-merge-ti, OF g, of y])
apply (subst disjnt-sceneD[OF disjnt-scene-merge-ti, OF f g f-g])
apply (rule access-ti-merge-ti-sub[OF g field-lookup-empty bs])
done

lemma merge-ti-update-ti-sub:
assumes r: field-ti TYPE('a::xmem-type) r = Some t
  and t: field-lookup t f 0 = Some (v, n)

```

and bs : $\text{length } bs = \text{size-td } v$
shows $\text{merge-ti } t \ x \ (\text{update-ti } v \ bs \ y) = \text{merge-ti } t \ x \ y$
proof –
have $wf-t$: $wf-fd \ t \ wf-desc \ t \ wf-size-desc \ t$
using $wf-fd-field-ti-mem-type[OF \ r] \ r$
 $\text{field-lookup-wf-desc-pres}(1)[\text{of } typ\text{-info-}t \ TYPE('a) \ r \ 0]$
 $\text{field-lookup-wf-size-desc-pres}(1)[\text{of } typ\text{-info-}t \ TYPE('a) \ r \ 0]$
by ($\text{auto elim: field-ti-field-lookupE}$)

from $wf-fd-consD[OF \ wf-t(1)]$
have $eq-uu$:
 $\text{length } bs = \text{size-td } t \implies \text{length } bs' = \text{size-td } t \implies$
 $\text{update-ti } t \ bs \ (\text{update-ti } t \ bs' \ v) = \text{update-ti } t \ bs \ v$
for $bs \ bs' \ v$
by ($\text{auto simp: fd-cons-def fd-cons-desc-def fd-cons-double-update-def update-ti-t-def}$
 $\text{length-fa-ti wf-t split: if-splits}$)

from $wf-fd-consD[OF \ wf-t(1)]$
have $eq-ua$:
 $\text{length } bs = \text{size-td } t \implies \text{update-ti } t \ (\text{access-ti } t \ v \ bs) \ v = v$
for $bs \ v$
by ($\text{auto simp: fd-cons-def fd-cons-desc-def fd-cons-update-access-def update-ti-t-def}$
 $\text{length-fa-ti wf-t split: if-splits}$)

have $n-v-t$: $n + \text{size-td } v \leq \text{size-td } t$
using $\text{field-lookup-offset-size}[OF \ t]$ **by** simp
with $fi-fu-consistentD[OF \ t \ wf-t(1)]$ **have** $eq-u-sub$:
 $\text{length } bs' = \text{size-td } v \implies \text{length } bs = \text{size-td } t \implies$
 $\text{update-ti } t \ (\text{super-update-bs } bs' \ bs \ n) \ y =$
 $\text{update-ti } v \ bs' \ (\text{update-ti } t \ bs \ y)$ **for** $bs' \ y \ bs$
by ($\text{simp add: update-ti-t-def}$)

have $\text{update-ti } v \ bs \ y =$
 $\text{update-ti } v \ bs \ (\text{update-ti } t \ (\text{access-ti } t \ y \ (\text{replicate } (\text{size-td } t) \ 0)) \ y)$
by (subst eq-ua) (simp-all add: bs)
also have $\dots = \text{update-ti } t \ (\text{super-update-bs } bs \ (\text{access-ti } t \ y \ (\text{replicate } (\text{size-td}$
 $t) \ 0)) \ n) \ y$
by (subst eq-u-sub) ($\text{simp-all add: bs length-fa-ti wf-t}$)
finally show $?thesis$
using $bs \ n-v-t$ **by** ($\text{simp add: access-ti}_0\text{-def length-fa-ti wf-t eq-uu merge-ti-def}$)
qed

lemma $\text{merge-ti-merge-ti-sub1}$:
assumes t : $\text{field-ti } TYPE('a::xmem\text{-type}) \ f = \text{Some } t$
assumes u : $\text{field-ti } TYPE('a) \ (f \ @ \ g) = \text{Some } u$
shows $\text{merge-ti } t \ a \ (\text{merge-ti } u \ a \ b) = \text{merge-ti } t \ a \ b$
proof –
obtain $n \ m$ **where** n : $\text{field-lookup } (typ\text{-info-}t \ TYPE('a)) \ f \ 0 = \text{Some } (t, n)$
and m : $\text{field-lookup } (typ\text{-info-}t \ TYPE('a)) \ (f \ @ \ g) \ 0 = \text{Some } (u, m)$

```

    using t u by (auto elim!: field-ti-field-lookupE)
  from n field-lookup-append-Some[OF - m] have u': field-lookup t g 0 = Some (u,
m - n)
    by (simp add: CTypes.field-lookup-offset2)
  have [simp]: fd-cons u
    using fd-cons fd-consistentD m by blast
  show ?thesis
    by (subst (2) merge-ti-def)
      (simp add: merge-ti-update-ti-sub[OF t u'] fd-cons-length-p)
qed

```

lemma *merge-ti-merge-ti-sub2*:

```

  assumes t: field-ti TYPE('a::xmem-type) f = Some t
  assumes u: field-ti TYPE('a) (f @ g) = Some u
  shows merge-ti u a (merge-ti t a b) = merge-ti t a b
proof -
  obtain n m where n: field-lookup (typ-info-t TYPE('a)) f 0 = Some (t, n)
    and m: field-lookup (typ-info-t TYPE('a)) (f@g) 0 = Some (u, m)
    using t u by (auto elim!: field-ti-field-lookupE)
  from n field-lookup-append-Some[OF - m] have u': field-lookup t g 0 = Some (u,
m - n)
    by (simp add: CTypes.field-lookup-offset2)
  have [simp]: fd-cons u
    using fd-cons fd-consistentD m by blast

```

```

interpret u: is-scene merge-ti u
  using u by (rule is-scene-merge-ti)

```

```

have eq: access-ti0 u a = access-ti0 u (merge-ti t a b)
  by (simp add: access-ti-merge-ti-sub[OF t u'] access-ti0-def)
show ?thesis
  apply (subst (1) merge-ti-def)
  apply (subst eq)
  apply (subst (1) merge-ti-def[symmetric])
  apply (rule u.idem)
done

```

qed

lemma *comm-scene-merge-ti-sub*:

```

  assumes *: field-ti TYPE('a::xmem-type) f = Some t field-ti TYPE('a) (f @ g)
= Some u
  shows comm-scene (merge-ti t) (merge-ti u)
  by (clarsimp simp: comm-scene-def merge-ti-merge-ti-sub1[OF *] merge-ti-merge-ti-sub2[OF
*])

```

lemma *comm-scene-merge-ti*:

```

  assumes *: field-ti TYPE('a::xmem-type) f = Some t field-ti TYPE('a) g =
Some u
  shows comm-scene (merge-ti t) (merge-ti u)

```

proof *cases*
assume *disj-fn f g*
from *disjnt-scene-merge-ti[OF * this]* **show** *?thesis*
by *blast*
next
assume \neg *disj-fn f g*
with *** **show** *?thesis*
by (*auto simp: disj-fn-def less-eq-list-def prefix-def intro: comm-scene-merge-ti-sub*)
qed

11.30.1 *merge-ti-list*

definition *merge-ti-list* **where**

merge-ti-list ts a = fold ($\lambda t. \text{merge-ti } t \ a$) ts

lemma *is-scene-merge-ti-list*:

*list-all ($\lambda u. \exists f. \text{field-ti TYPE('a::xmem-type)} f = \text{Some } u$) ts \implies
*is-scene (merge-ti-list ts)**

unfolding *merge-ti-list-def*

apply (*intro is-scene-fold'*)

by (*auto simp: list-all-iff pairwise-def is-scene-merge-ti*)
(meson comm-scene-merge-ti)

lemma *merge-ti-list-nil[simp]*: *merge-ti-list [] = ($\lambda a. \text{id}$)*

by (*simp add: merge-ti-list-def fun-eq-iff*)

lemma *merge-ti-list-cons[simp]*:

merge-ti-list (t # ts) a = merge-ti-list ts a \circ merge-ti t a
by (*simp add: merge-ti-list-def*)

lemma *merge-ti-list-append[simp]*:

merge-ti-list (ts @ ts') x y = merge-ti-list ts' x (merge-ti-list ts x y)
by (*simp add: merge-ti-list-def*)

lemma *access-ti-merge-ti-list*:

assumes *ts: list-all ($\lambda(f, u). \text{field-ti TYPE('a::xmem-type)} f = \text{Some } u$) ts*
distinct-prop disj-fn (map fst ts)

and *r-t: (r, t) \in set ts* **and** *v: field-lookup t f 0 = Some (v, n)*

and *bs: length bs = size-td v*

shows *access-ti v (merge-ti-list (map snd ts) x y) bs = access-ti v x bs*
using *ts r-t*

proof (*induction ts rule: rev-induct*)

case (*snoc r-t ts*)

from *this(2,3)* **have** ***:

list-all ($\lambda(f, u). \text{field-ti TYPE('a)} f = \text{Some } u$) ts

distinct-prop disj-fn (map fst ts)

and *disj: $\forall t \in \text{set } ts. \text{disj-fn (fst } t) (fst \ r-t)$*

and ***:* *field-ti TYPE('a) (fst r-t) = Some (snd r-t)*

by (*auto simp: distinct-prop-append*)

```

show ?case
proof cases
  assume [simp]: r-t = (r, t)
  with ** have ft-r: field-ti TYPE('a) r = Some t by simp
  show ?thesis
    by (simp add: access-ti-merge-ti-sub[OF ft-r v bs])
next
  assume r-t ≠ (r, t)
  with snoc.prem have r-t: (r, t) ∈ set ts by auto
  with * have field-ti TYPE('a) r = Some t
    by (auto simp: list-all-iff)
  from field-ti-append-field-lookup[OF this v]
  have r-f: field-ti TYPE('a) (r @ f) = Some v .
  from r-t disj
  have disj-fn (fst r-t) r by (auto simp: disj-fn-def)
  then have disj: disj-fn (fst r-t) (r @ f)
    by (rule disj-fn-append-right)
  from snoc.IH[OF * r-t] show ?thesis
    by (simp add: access-ti-merge-ti-disj[OF ** r-f disj bs])
qed
qed simp

lemma merge-ti-list-update-ti:
  assumes ts: list-all (λ(f, u). field-ti TYPE('a::xmem-type) f = Some u) ts (f-t,
t) ∈ set ts
  and disj: distinct-prop disj-fn (map fst ts)
  and t: field-lookup t f 0 = Some (v, n)
  and bs: length bs = size-td v
  shows merge-ti-list (map snd ts) x (update-ti v bs y) = merge-ti-list (map snd
ts) x y
  using ts disj
proof (induction ts arbitrary: y)
  case (Cons u' ts)
  then obtain t' f' where ts[simp]: list-all (λ(f, u). field-ti TYPE('a) f = Some
u) ts
    and f': field-ti TYPE('a) f' = Some t'
    and t': t = t' ∨ (t ≠ t' ∧ (f-t, t) ∈ set ts) and u': u' = (f', t')
    and f'-ts: ∀ x ∈ set ts. disj-fn f' (fst x)
    and [simp]: distinct-prop disj-fn (map fst ts)
    by auto

  from Cons have f-t: field-ti TYPE('a) f-t = Some t
    by (auto simp: list-all-iff)

  show ?case using t'
  proof (elim disjE conjE)
    assume t-eq: t = t'
    have merge-ti t' x (update-ti v bs y) = merge-ti t' x y

```

```

    using f' t[unfolded t-eq] bs by (rule merge-ti-update-ti-sub)
  then show ?thesis
    by (simp add: u')
next
assume ne: t ≠ t' and mem: (f-t, t) ∈ set ts
with f'-ts have disj-fn f' f-t by auto
then have disj: disj-fn f' (f-t @ f)
  by (rule disj-fn-append-right)
have merge-ti t' x (update-ti v bs y) = update-ti v bs (merge-ti t' x y)
using f' field-ti-append-field-lookup[OF f-t t] disj bs by (rule merge-ti-update-ti-disj)
then show ?thesis
  by (simp add: Cons mem u')
qed
qed simp

lemma heap-update-eq-fold-subfields:
  assumes ts: list-all (λ(f, u). field-ti TYPE('a::xmem-type) f = Some u) ts
  shows heap-update p x =
    fold (λ(f, u). heap-upd-list (size-td u) &(p→f) (access-ti u x)) ts ∘
    heap-upd-list (size-of TYPE('a)) (ptr-val p)
    (access-ti (typ-info-t TYPE('a)) (merge-ti-list (map snd ts) y x))
proof -
  have snd-ts: list-all (λu. ∃f. field-ti TYPE('a::xmem-type) f = Some u) (map
  snd ts)
  using ts unfolding list.pred-map by (rule list.pred-mono-strong) auto
  then interpret merge-ti-list: is-scene merge-ti-list (map snd ts)
  by (rule is-scene-merge-ti-list)

  have heap-upd-list (size-of TYPE('a)) (ptr-val p)
    (access-ti (typ-info-t TYPE('a)) (merge-ti-list (map snd ts) x z)) =
    fold (λ(f, u). heap-upd-list (size-td u) &(p→f) (access-ti u x)) ts ∘
    heap-upd-list (size-of TYPE('a)) (ptr-val p) (access-ti (typ-info-t TYPE('a))
  z) for z
  using ts
  proof (induction ts arbitrary: z)
    case (Cons t ts)
    then obtain f u where ts: list-all (λa. case a of (f, u) ⇒ field-ti TYPE('a) f
  = Some u) ts
    and f: field-ti TYPE('a) f = Some u and t: t = (f, u)
    by (cases t) auto

    have wf-u: wf-fd u using f by (rule wf-fd-field-ti-mem-type)
    have sz[arith]: size-td u + field-offset TYPE('a) f ≤ size-of TYPE('a)
    using f[THEN field-tiD, THEN field-lookup-offset-size] by (simp add: size-of-def)

    interpret padding-lense access-ti u update-ti u size-td u
    by (rule field-lookup-padding-lense[OF field-tiD, OF f])
    from field-access-eq-padding1[unfolded eq-padding-def]
    have access-ti-access-ti:

```

```

    length bs = size-td u  $\implies$  access-ti u v (access-ti u w bs) = access-ti u v bs for
v w bs
    by auto

have heap-upd-list (size-of TYPE('a)) (ptr-val p)
  (access-ti (typ-info-t TYPE('a)) (update-ti u (access-ti0 u x) z)) =
  heap-upd-list (size-td u) &(p $\rightarrow$ f) (access-ti u x)  $\circ$ 
  heap-upd-list (size-of TYPE('a)) (ptr-val p) (access-ti (typ-info-t TYPE('a))
z)
apply (rule ext)
subgoal for h unfolding heap-upd-list-def
  apply (simp add: field-lvalue-def)
  apply (subst heap-list-update-list)
  subgoal by (simp add: lense.access-result-size)
  subgoal by (simp add: lense.access-result-size)
  apply (subst heap-update-list-super-update-bs)
  subgoal by (simp add: lense.access-result-size length-fa-ti wf-u)
subgoal using max-size[where 'a='a, arith] by (simp add: lense.access-result-size)
  apply (subst mem-type-field-lookup-access-ti-take-drop[symmetric])
  apply (rule f[THEN field-tiD])
  apply simp
  apply (subst access-ti-access-ti)
  apply simp
  apply (subst mem-type-access-ti-super-update-bs[symmetric, OF field-tiD,
OF f])
  apply (simp-all add: super-update-bs-take-drop)
  done
done
then show ?case
  unfolding fold-Cons t by (simp add: Cons.IH[OF ts] merge-ti-def)
qed simp
then show ?thesis
  apply (subst heap-update-eq-heap-upd-list)
  apply (subst merge-ti-list.idem[symmetric])
  apply (subst merge-ti-list.right[symmetric])
  apply (simp del: merge-ti-list.right)
  done
qed
end

```

Chapter 12

More Building Blocks for our C-Language Model

```
theory CLanguage
  imports
    CProof
    Lens
begin
```

12.1 addr bounds

```
lemma addr-card-eq: addr-card = 2LENGTH(addr-bitsize)
  by (simp add: addr-card-def card-word)
```

```
lemma size-of-bnd: size-of TYPE('a::mem-type) < 2LENGTH(addr-bitsize)
  by (rule less-le-trans[OF max-size]) (simp add: addr-card-eq)
```

```
lemma size-of-mem-type[simp]: size-of TYPE('c::mem-type) ≠ 0
  by simp
```

```
lemma addr-card-len-of-conv: addr-card = 2len-of TYPE(addr-bitsize)
  by (simp add: addr-card)
```

```
lemma intvl-split:
```

```
[[ n ≥ a ] ⇒ { p :: ('a :: len) word ..+ n } = { p ..+ a } ∪ { p + of-nat a ..+
(n - a) }
```

```
apply (rule set-eqI, rule iffI)
  apply (clarsimp simp: intvl-def not-less)
  subgoal for k
    apply (rule exI[where x=k])
    apply clarsimp
    apply (rule classical)
    apply (drule-tac x=k - a in spec)
```



```

  apply (clarsimp simp: not-less)
  apply (metis diff-less-mono not-less)
done
subgoal for x
  apply (clarsimp simp: intvl-def not-less)
  apply (rule exI[where x=unat (x - p)])
  apply clarsimp
  apply (erule disjE)
  apply clarsimp
  apply (metis le-unat-voi less-or-eq-imp-le not-less order-trans)
  apply clarsimp
  apply (metis le-def le-eq-less-or-eq le-unat-voi less-diff-conv
    add commute of-nat-add)
done
done

```

12.2 More Heap Typing

primrec

```

  htd-upd :: addr ⇒ typ-slice list ⇒ heap-tyt-desc ⇒ heap-tyt-desc
where
  htd-upd p [] d = d
| htd-upd p (x#xs) d = htd-upd (p+1) xs (d(p := (True, x)))

```

definition (in *c-type*) *ptr-force-type* :: 'a ptr ⇒ heap-tyt-desc ⇒ heap-tyt-desc
where

```

  ptr-force-type p ≡ htd-upd (ptr-val p) (typ-slices TYPE('a))

```

definition *ptr-force-types* :: 'a::c-type ptr list ⇒ heap-tyt-desc ⇒ heap-tyt-desc
where

```

  ptr-force-types = fold ptr-force-type

```

definition *ptr-force-free* :: addr ⇒ nat ⇒ heap-tyt-desc ⇒ heap-tyt-desc **where**
 ptr-force-free p b = ptr-force-types (map (λn. PTR(8 word) p +_p n) (map of-nat [0..²b]))

definition *ptr-u* :: 'a::c-type ptr ⇒ (addr × typ-uinfo) **where**
 ptr-u p = (ptr-val p, typ-uinfo-t TYPE('a))

abbreviation *ptr-span-u* ≡ (λ(a, t). {a ..+ size-td t})

definition *typ-slices-u* :: typ-uinfo ⇒ typ-slice list **where**
 typ-slices-u t = map (λn. list-map (typ-slice-t t n)) [0..^{size-td} t]

definition *ptr-force-type-u* :: typ-uinfo ⇒ addr ⇒ heap-tyt-desc ⇒ heap-tyt-desc
where

```

  ptr-force-type-u t a ≡ htd-upd a (typ-slices-u t)

```

lemma *heap-update-list-id*: $\text{heap-update-list } x \ [] = (\lambda x. x)$
by *auto*

lemma *to-bytes-word8*:
 $\text{to-bytes } (v :: \text{word8}) \ xs = [v]$
by (*simp add: to-bytes-def typ-info-word word-rsplit-same*)

lemma *heap-update-heap-update-list*:
 $\llbracket \text{ptr-val } p = q + (\text{of-nat } (\text{length } l)); \text{Suc } (\text{length } l) < \text{addr-card} \rrbracket \implies$
 $\text{heap-update } (p :: \text{word8 } \text{ptr}) \ v \ (\text{heap-update-list } q \ l \ s) = (\text{heap-update-list } q \ (l$
 $\text{@ } [v]) \ s)$
by (*metis to-bytes-word8 heap-update-def heap-update-list-concat-fold*)

lemma *htd-upd-empty[simp]*: $\text{htd-upd } p \ [] = \text{id}$
by (*simp add: fun-eq-iff*)

lemma *htd-upd-append*:
 $\text{htd-upd } p \ (xs \text{ @ } ys) = \text{htd-upd } (p + \text{of-nat } (\text{length } xs)) \ ys \circ \text{htd-upd } p \ xs$
by (*induction xs arbitrary: p*) (*simp-all add: fun-eq-iff ac-simps*)

lemma *htd-upd-singleton[simp]*: $\text{htd-upd } p \ [x] = \text{upd-fun } p \ (\lambda h. (\text{True}, x))$
by (*simp add: fun-eq-iff upd-fun-def*)

lemma *intvl-Suc-eq*: $\{p \ ..+ \text{Suc } n\} = \text{insert } p \ \{p + 1 \ ..+ n\}$
using *intvl-split[of 1 Suc n p]* **by** (*auto simp add: intvl-def*)

lemma *htd-upd-disj*: $p \notin \{p' \ ..+ \text{length } v\} \implies \text{htd-upd } p' \ v \ h \ p = h \ p$
by (*induction v arbitrary: p' h*)
(auto simp add: intvl-Suc-eq fun-upd-other simp del: fun-upd-apply)

lemma *htd-upd-head*:
 $xs \neq [] \implies \text{length } xs \leq \text{addr-card} \implies \text{htd-upd } p \ xs \ s \ p = (\text{True}, \text{hd } xs)$
using *intvl-Suc-nmem'[of length xs p] htd-upd-disj[of p p + 1 tl xs -]*
by (*cases xs*) (*simp-all add: addr-card-eq del: intvl-Suc-nmem'*)

lemma *htd-upd-at*:
 $i < \text{length } xs \implies \text{length } xs \leq \text{addr-card} \implies \text{htd-upd } p \ xs \ s \ (p + \text{of-nat } i) =$
 $(\text{True}, xs \ ! \ i)$
proof (*induction i arbitrary: p xs s*)
case 0 **with** *htd-upd-head[of xs p s]* **show** *?case* **by** (*simp add: hd-conv-nth*)
next
case (*Suc n*)
from *Suc.prem*s **show** *?case* **by** (*cases xs*) (*simp-all add: Suc.IH flip: add.assoc*)
qed

lemma *ptr-force-type-disj*:
 $p \notin \text{ptr-span } (p' :: 'a :: \text{mem-type } \text{ptr}) \implies \text{ptr-force-type } p' \ h \ p = h \ p$
unfolding *ptr-force-type-def*
by (*intro htd-upd-disj*) *simp-all*

lemma *ptr-force-types-disj*:
fixes $xs :: 'a::mem\text{-}type\ ptr\ list$
assumes $\bigwedge x. x \in set\ xs \implies i \notin ptr\text{-}span\ x$
shows $ptr\text{-}force\text{-}types\ xs\ h\ i = h\ i$
by (*use* *assms* **in** $\langle induction\ xs\ rule: rev\text{-}induct \rangle$)
(auto simp: ptr-force-types-def ptr-force-type-disj)

12.2.1 Heap type tag and valid simple footprint

datatype *heap-typ-contents* =
HeapType typ-uinfo
| *HeapFootprint*
| *HeapEmpty*

definition

heap-type-tag :: $heap\text{-}typ\text{-}desc \Rightarrow addr \Rightarrow heap\text{-}typ\text{-}contents$
where
heap-type-tag $d\ a \equiv$
(if $fst\ (d\ a) = False \vee (\forall x. (snd\ (d\ a))\ x = None) \vee (\forall x. (snd\ (d\ a))\ x \neq$
None) *then*
HeapEmpty
else
case $(snd\ (d\ a))\ (GREATEST\ x.\ snd\ (d\ a)\ x \neq None)$ *of*
Some $(-, False) \Rightarrow HeapFootprint$
| *Some* $(x, True) \Rightarrow HeapType\ x$
| *None* $\Rightarrow HeapEmpty$)

definition

valid-simple-footprint :: $heap\text{-}typ\text{-}desc \Rightarrow addr \Rightarrow typ\text{-}uinfo \Rightarrow bool$
where
valid-simple-footprint $d\ x\ t \equiv$
heap-type-tag $d\ x = HeapType\ t \wedge$
 $(\forall y. y \in \{x + 1..+ (size\text{-}td\ t) - Suc\ 0\} \longrightarrow heap\text{-}type\text{-}tag\ d\ y = HeapFootprint)$

lemma *valid-simple-footprint-size-td*:

assumes *valid*: *valid-simple-footprint* $d\ x\ t$
shows $size\text{-}td\ t \leq addr\text{-}card$

proof (*cases* $size\text{-}td\ t \leq addr\text{-}card$)

case *True*
then show *?thesis* **by** *simp*

next

case *False*
from *valid* **have** *heap-type-tag* $d\ x = HeapType\ t$
by (*simp add: valid-simple-footprint-def*)

moreover
from $False$ **have** $x \in \{x + 1..+ (size\text{-}td\ t) - Suc\ 0\}$
apply (*clarsimp simp add: intvl-def*)
apply (*rule exI [where $x = addr\text{-}card - Suc\ 0$]*)
by (*metis (mono-tags, opaque-lifting) One-nat-def Suc-pred' diff-less-mono*
diff-zero
neq0-conv not-less not-less-eq-eq of-nat-1 of-nat-Suc of-nat-addr-card unatSuc
unat-minus-abs zero-diff zero-neq-one)
with *valid* **have** $heap\text{-}type\text{-}tag\ d\ x = HeapFootprint$ **by** (*auto simp add: valid-simple-footprint-def*)
ultimately have $False$ **by** *simp*
thus *?thesis ..*
qed

lemma *valid-simple-footprintI*:
 $\llbracket heap\text{-}type\text{-}tag\ d\ x = HeapType\ t; \bigwedge y. y \in \{x + 1..+(size\text{-}td\ t) - Suc\ 0\} \implies heap\text{-}type\text{-}tag\ d\ y = HeapFootprint \rrbracket$
 $\implies valid\text{-}simple\text{-}footprint\ d\ x\ t$
by (*clarsimp simp: valid-simple-footprint-def*)

lemma *valid-simple-footprintD*:
 $valid\text{-}simple\text{-}footprint\ d\ x\ t \implies heap\text{-}type\text{-}tag\ d\ x = HeapType\ t$
by (*simp add: valid-simple-footprint-def*)

lemma *valid-simple-footprintD2*:
 $\llbracket valid\text{-}simple\text{-}footprint\ d\ x\ t; y \in \{x + 1..+(size\text{-}td\ t) - Suc\ 0\} \rrbracket \implies heap\text{-}type\text{-}tag\ d\ y = HeapFootprint$
by (*simp add: valid-simple-footprint-def*)

lemma *typ-slices-not-empty*:
 $typ\text{-}slices\ (x::('a::\{mem\text{-}type\}\ itself)) \neq []$
apply (*clarsimp simp: typ-slices-def*)
done

lemma *last-tyl-slice-t*:
 $(last\ (typ\text{-}slice\text{-}t\ t\ 0)) = (t, True)$
apply (*cases t*)
apply *clarsimp*
done

lemma *last-tyl-slice-t-non-zero*:
 $k \neq 0 \implies (last\ (typ\text{-}slice\text{-}t\ t\ k)) = (t, False)$
apply (*cases t*)
apply *clarsimp*
done

lemma *if-eqI*:
 $\llbracket a \implies x = z; \neg a \implies y = z \rrbracket \implies (if\ a\ then\ x\ else\ y) = z$
by *simp*

lemma *heap-type-tag-ptr-retyp*:
 $snd (s (ptr\text{-}val\ t)) = Map.empty \implies$
 $heap\text{-}type\text{-}tag (ptr\text{-}retyp (t :: 'a::mem\text{-}type\ ptr) s) (ptr\text{-}val\ t) = HeapType$
 $(typ\text{-}uinfo\text{-}t\ TYPE('a))$
apply (*unfold ptr-retyp-def heap-type-tag-def*)
apply (*subst htd-update-list-index, fastforce, fastforce*)
apply (*rule if-eqI*)
apply *clarsimp*
apply (*erule disjE*)
apply (*erule-tac x=0 in allE*)
apply *clarsimp*
apply (*erule-tac x=length (typ-slice-t (typ-uinfo-t TYPE('a)) 0) in allE*)
apply (*clarsimp simp: list-map-eq*)
apply (*clarsimp simp: list-map-eq last-conv-nth [simplified, symmetric] last-typ-slice-t*
split: option.splits if-split-asm prod.splits)
done

lemma *not-snd-last-typ-slice-t*:
 $k \neq 0 \implies \neg snd (last (typ\text{-}slice\text{-}t\ z\ k))$
by (*cases z, clarsimp*)

lemma *heap-type-tag-ptr-retyp-rest*:
 $\llbracket snd (s (ptr\text{-}val\ t + k)) = Map.empty; 0 < k; unat\ k < size\text{-}td (typ\text{-}uinfo\text{-}t$
 $TYPE('a)) \rrbracket \implies$
 $heap\text{-}type\text{-}tag (ptr\text{-}retyp (t :: 'a::mem\text{-}type\ ptr) s) (ptr\text{-}val\ t + k) = Heap\text{-}$
 $Footprint$
apply (*unfold ptr-retyp-def heap-type-tag-def*)
apply (*subst htd-update-list-index, simp, clarsimp,*
metis intvlI size-of-def word-unat.Rep-inverse)
apply (*rule if-eqI*)
apply *clarsimp*
apply (*erule disjE*)
apply (*erule-tac x=0 in allE*)
apply (*clarsimp simp: size-of-def*)
apply (*erule-tac x=length (typ-slice-t (typ-uinfo-t TYPE('a)) (unat k)) in allE*)
apply (*clarsimp simp: size-of-def list-map-eq*)
apply (*clarsimp simp: list-map-eq last-conv-nth [simplified, symmetric] size-of-def*
split: option.splits if-split-asm prod.splits bool.splits)
apply (*metis surj-pair*)
apply (*subst (asm) (2) surjective-pairing*)
apply (*subst (asm) not-snd-last-typ-slice-t*)
apply *clarsimp*
apply *unat-arith*
apply *simp*
done

lemma *typ-slices-addr-card [simp]*:
 $length (typ\text{-}slices (x::('a::\{mem\text{-}type\} itself))) < addr\text{-}card$

```

apply (clarsimp simp: typ-slices-def)
done

lemma unat-less-impl-less:
  unat a < unat b  $\implies$  a < b
by unat-arith

lemma valid-simple-footprint-ptr-retyp:
   $\llbracket \forall k < \text{size-td } (\text{typ-uinfo-t } \text{TYPE}('a)). \text{snd } (s \text{ (ptr-val } t + \text{of-nat } k)) =$ 
  Map.empty;
   $1 \leq \text{size-td } (\text{typ-uinfo-t } \text{TYPE}('a));$ 
   $\text{size-td } (\text{typ-uinfo-t } \text{TYPE}('a)) < \text{addr-card} \rrbracket$ 
   $\implies \text{valid-simple-footprint } (\text{ptr-retyp } (t :: 'a::\text{mem-type ptr}) s) (\text{ptr-val } t)$ 
  (typ-uinfo-t TYPE('a))
apply (clarsimp simp: valid-simple-footprint-def)
apply (rule conjI)
apply (subst heap-type-tag-ptr-retyp)
apply (erule allE [where x=0])
apply clarsimp
apply clarsimp
apply (clarsimp simp: intvl-def)
subgoal for k
apply (erule-tac x=k + 1 in allE)
apply (erule impE)
apply (metis One-nat-def less-diff-conv)
apply (subst add.assoc, subst heap-type-tag-ptr-retyp-rest)
apply clarsimp
apply (cases 1 + of-nat k = (0 :: addr))
apply (metis add.left-neutral intvlI intvl-Suc-nmem size-of-def)
apply unat-arith
apply clarsimp
apply (metis lt-size-of-unat-simps size-of-def Suc-eq-plus1 One-nat-def less-diff-conv
of-nat-Suc)
apply simp
done
done

lemma heap-type-tag-cong: s p = s' p  $\implies$  heap-type-tag s p = heap-type-tag s' p
by (simp add: heap-type-tag-def cong: if-cong)

lemma heap-type-tag:
assumes eq: h p = (f, list-map l)
shows heap-type-tag h p =
  (if  $\neg f \vee l = []$  then HeapEmpty else
   case last l of (x, b)  $\implies$  if b then HeapType x else HeapFootprint)
unfolding heap-type-tag-def eq
by (auto simp add: list-map-def)
  (auto simp: split-beta' last-conv-nth list-map-eq simp flip: list-map-def)

```

lemma *valid-simple-footprint-cong-state*:
assumes t : wf-size-desc t
assumes eq : $\bigwedge p'. p' \in \{p \text{ ..+size-td } t\} \implies s \ p' = s' \ p'$
shows $valid-simple-footprint \ s \ p \ t \longleftrightarrow valid-simple-footprint \ s' \ p \ t$
unfolding *valid-simple-footprint-def*
using eq wf-size-desc-gt(1)[OF t]
using *intvl-split*[of 1 size-td t p]
by (*intro arg-cong2*[**where** $f=(\wedge)$] *all-cong arg-cong*[**where** $f=\lambda x. x = _$] *heap-type-tag-cong*)
(*auto simp: intvl-def*)

lemma *heap-type-tag-ptr-force-type-HeapType*:
fixes $x :: 'a::mem-type \ ptr$
shows $heap-type-tag \ (ptr-force-type \ x \ s) \ (ptr-val \ x) = HeapType \ (typ-uinfo-t \ TYPE('a))$
by (*subst heap-type-tag*)
(*auto simp: ptr-force-type-def htd-upd-head typ-slices-not-empty max-size*[*THEN less-imp-le*]
hd-conv-nth last-typ-slice-t)

lemma *heap-type-tag-ptr-force-type-HeapFootprint*:
fixes $p :: 'a::mem-type \ ptr$
shows $p' \in \{ptr-val \ p + 1 \text{ ..+ size-of } TYPE('a) - Suc \ 0\} \implies$
 $heap-type-tag \ (ptr-force-type \ p \ s) \ p' = HeapFootprint$
unfolding *intvl-def*
apply (*clarsimp simp: less-diff-conv add.assoc ptr-force-type-def simp flip: of-nat-Suc*)
subgoal **premises** $prems$ **for** k
using $prems(1)$
apply (*subst heap-type-tag*)
apply (*subst htd-upd-at*)
apply (*simp-all add: max-size*[*THEN less-imp-le*])
apply (*simp add: last-typ-slice-t split-beta' not-snd-last-typ-slice-t*)
done
done

lemma *valid-simple-footprint-ptr-force-type-iff*:
fixes $p :: 'a::mem-type \ ptr$
assumes t : wf-size-desc t
shows $valid-simple-footprint \ (ptr-force-type \ p \ s) \ a \ t \longleftrightarrow$
 $(valid-simple-footprint \ s \ a \ t \wedge disjnt \ \{a \text{ ..+ size-td } t\} \ (ptr-span \ p)) \vee$
 $(t = typ-uinfo-t \ TYPE('a) \wedge p = Ptr \ a)$
proof *cases*
assume $disjnt$: $disjnt \ \{a \text{ ..+ size-td } t\} \ (ptr-span \ p)$
moreover **have** $p \neq Ptr \ a$
using $disjnt$ [*unfolded disjnt-iff, THEN spec, of a*] t [*THEN wf-size-desc-gt(1)*]
by (*cases p*) (*auto simp: intvl-self*)
moreover **have** $valid-simple-footprint \ (ptr-force-type \ p \ s) \ a \ t = valid-simple-footprint$
 $s \ a \ t$
using $t \ disjnt$
by (*intro valid-simple-footprint-cong-state ptr-force-type-disj*) (*simp-all add:*

```

disjnt-iff)
  ultimately show ?thesis
    by simp
next
assume ndisjnt:  $\neg$  disjnt  $\{a \dots + \text{size-td } t\}$  (ptr-span p)
from intvl-inter[OF this[unfolded disjnt-def]]
consider a = ptr-val p
  | a  $\in$   $\{\text{ptr-val } p + 1 \dots + \text{size-of TYPE('a)} - 1\}$ 
  | ptr-val p  $\in$   $\{a + 1 \dots + \text{size-td } t - \text{Suc } 0\}$  ptr-val p  $\neq$  a
  by (auto dest: intvl-neq-start)
then show ?thesis
proof cases
case 1
have valid-simple-footprint (ptr-force-type p s) (ptr-val p) (typ-uinfo-t TYPE('a))
  by (auto simp: valid-simple-footprint-def heap-type-tag-ptr-force-type-HeapType
    heap-type-tag-ptr-force-type-HeapFootprint size-of-tag)
with 1 valid-simple-footprintD[of ptr-force-type p s a t] ndisjnt
show ?thesis
  by (auto simp: heap-type-tag-ptr-force-type-HeapType)
next
case 2 with valid-simple-footprintD[of ptr-force-type p s a t] show ?thesis
  by (auto simp: heap-type-tag-ptr-force-type-HeapFootprint ndisjnt)
next
case 3 with valid-simple-footprintD2[of ptr-force-type p s a t ptr-val p] show
?thesis
  by (auto simp add: heap-type-tag-ptr-force-type-HeapType ndisjnt)
qed
qed

```

```

lemma valid-simple-footprint-fold-ptr-force-type-iff:
  fixes ps :: 'a::mem-type ptr list
  assumes [simp]: wf-size-desc t
  shows distinct-prop ( $\lambda p1 p2. \text{disjnt (ptr-span } p1) (\text{ptr-span } p2)$ ) ps  $\implies$ 
    valid-simple-footprint (fold ptr-force-type ps s) a t  $\longleftrightarrow$ 
    (valid-simple-footprint s a t  $\wedge$  disjnt  $\{a \dots + \text{size-td } t\}$  ( $\bigcup_{p \in \text{set } ps. \text{ptr-span } p}$ ))  $\vee$ 
    (t = typ-uinfo-t TYPE('a)  $\wedge$  Ptr a  $\in$  set ps)
  by (induction ps arbitrary: s)
    (auto simp: valid-simple-footprint-ptr-force-type-iff size-of-tag distinct-prop-append)

```

12.3 Pointers to local (stack) variables

definition *stack-typ-info* t = (stack-byte-name \notin td-names t)

lemma *stack-typ-info-export-uinfo*[simp]: *stack-typ-info* (export-uinfo t) = *stack-typ-info* t
 by (simp add: stack-typ-info-def)

lemma *stack-typ-info-td-set*:


```

assumes stack-typ: stack-typ-info t
assumes t':  $(t', n) \in \text{td-set } t \ 0$ 
shows typ-name t'  $\neq$  stack-byte-name
proof (cases typ-name t' = pad-typ-name)
  case True
    then show ?thesis by (simp add: stack-byte-name-def pad-typ-name-def)
  next
    case False
      from td-set-td-names [OF t' False]
      have typ-name t'  $\in$  td-names t .
      with stack-typ show ?thesis
        by (auto simp add: stack-typ-info-def)
qed

```

```

lemma stack-typ-info-td-set-stack-byte:
  assumes stack-typ: stack-typ-info t
  assumes t':  $(t', n) \in \text{td-set } t \ 0$ 
  shows  $t' \neq \text{typ-uinfo-t } \text{TYPE}(\text{stack-byte})$ 
  using stack-typ-info-td-set [OF stack-typ t']
  apply (cases t')
  apply (simp add: typ-uinfo-t-def typ-info-stack-byte)
  done

```

```

class stack-type = c-type +
  assumes stack-typ-info: stack-typ-info (typ-info-t TYPE('a))
begin
lemma stack-typ-uinfo: stack-typ-info (typ-uinfo-t TYPE('a))
  using stack-typ-info
  by (simp add: typ-uinfo-t-def)

```

```

lemma no-stack-byte [simp]:  $\text{typ-uinfo-t } \text{TYPE}('a) \neq \text{typ-uinfo-t } \text{TYPE}(\text{stack-byte})$ 
  by (metis order.refl stack-typ-info-td-set-stack-byte stack-typ-uinfo typ-tag-le-def)

```

end

```

lemma stack-typ-info-no-stack-byte:
   $\text{stack-typ-info } t \implies t \neq \text{typ-uinfo-t } \text{TYPE}(\text{stack-byte})$ 
  using stack-typ-info-def typ-tag-le-def stack-typ-info-td-set-stack-byte by fastforce

```

```

lemma stack-typ-info-empty-typ-info:
   $nm \neq \text{stack-byte-name} \implies \text{stack-typ-info} (\text{empty-typ-info } \text{algn } nm)$ 
  by (simp add: empty-typ-info-def stack-typ-info-def typ-info-stack-byte)

```

```

lemma td-set-list-to-set:
   $(t, m) \in \text{td-set-list } xs \ n \implies (\exists x \ k. x \in \text{set } xs \wedge (t, k) \in \text{td-set } (\text{dt-fst } x) \ 0)$ 
  apply (induct xs arbitrary: m n)

```

apply *simp*
by (*metis Un-iff dt-tuple.collapse insertCI list.set(2) td-set-list.simps(2)*
td-set-offset-0-conv' ts5)

lemma *td-names-list-to-set*:
 $nm \in \text{td-names-list } xs \implies (\exists x. x \in \text{set } xs \wedge nm \in \text{td-names } (dt\text{-fst } x))$
apply (*induct xs arbitrary*:)
apply *simp*
by (*metis Un-iff dt-tuple.collapse insert-iff list.set(2) td-names-list.simps(2) tnm5*)

lemma *set-to-td-names-list*:
 $x \in \text{set } xs \implies nm \in \text{td-names } (dt\text{-fst } x) \implies nm \in \text{td-names-list } xs$
apply (*induct xs arbitrary*:)
apply *simp*
by (*metis Un-iff dt-tuple.collapse insert-iff list.set(2) td-names-list.simps(2) tnm5*)

lemma *stack-typ-info-TypAggregateI*:
assumes *xs*: $\bigwedge x. x \in \text{set } xs \implies \text{stack-typ-info } (dt\text{-fst } x)$
assumes *nm*: $nm \neq \text{stack-byte-name}$
shows *stack-typ-info* (*TypDesc algn* (*TypAggregate* (*xs*)) *nm*)
using *xs nm*
apply (*auto simp add: stack-typ-info-def stack-byte-name-def dest: td-names-list-to-set*)
done

lemma *TypAggregate-not-stack-byte*:
 $\text{TypDesc algn } (\text{TypAggregate } xs) nm \neq \text{typ-uinfo-t TYPE}(\text{stack-byte})$
by (*auto simp add: typ-info-stack-byte typ-uinfo-t-def*)

lemma *stack-typ-info-TypAggregateD*:
assumes *aggr*: $\text{stack-typ-info } (\text{TypDesc algn } (\text{TypAggregate } (xs))) nm$
assumes *x*: $x \in \text{set } xs$
shows *stack-typ-info* (*dt-fst x*)
using *aggr x*
by (*auto simp add: stack-typ-info-def stack-byte-name-def dest:set-to-td-names-list*)

lemma *stack-typ-info-extend-ti*:
 $\text{stack-typ-info } s \implies \text{stack-typ-info } t \implies$
 $\text{stack-typ-info } (\text{extend-ti } s t \text{ algn fn } d)$
apply (*cases s*)
by (*simp add: stack-typ-info-def*)

lemma *stack-typ-info-ti-pad-combine*:
 $\text{stack-typ-info } t \implies \text{stack-typ-info } (\text{ti-pad-combine } n t)$
apply (*simp add: ti-pad-combine-def Let-def*)
apply (*rule stack-typ-info-extend-ti*)
apply *assumption*
apply (*auto simp add: stack-typ-info-def typ-uinfo-t-def typ-info-stack-byte*)

done

lemma *stack-typ-info-export-uinfo-adjust-ti'*:
 shows *stack-typ-info (adjust-ti (typ-info-t TYPE('b::stack-type)) acc upd)*
proof –
 from *stack-typ-info*
 have *stack-typ-info (typ-info-t TYPE('b))*.
 then show *?thesis*
 by (*simp add: stack-typ-info-def*)
qed

lemma *stack-typ-info-export-uinfo-adjust-ti*:
 shows *stack-typ-info (typ-info-t (TYPE('b))) \implies stack-typ-info (adjust-ti (typ-info-t TYPE('b::c-type)) acc upd)*
 by (*simp add: stack-typ-info-def*)

lemma *stack-typ-info-ti-typ-combine'*:
 stack-typ-info t \implies
 stack-typ-info (ti-typ-combine TYPE('b::stack-type) acc upd algn nm t)
 apply (*simp add: ti-typ-combine-def*)
 apply (*rule stack-typ-info-extend-ti*)
 apply *assumption*
 apply (*rule stack-typ-info-export-uinfo-adjust-ti'*)
done

lemma *stack-typ-info-ti-typ-combine*:
 stack-typ-info (typ-info-t (TYPE('b))) \implies stack-typ-info t \implies
 stack-typ-info (ti-typ-combine TYPE('b::c-type) acc upd algn nm t)
 apply (*simp add: ti-typ-combine-def*)
 apply (*rule stack-typ-info-extend-ti*)
 apply *assumption*
 apply (*simp add: stack-typ-info-export-uinfo-adjust-ti*)
done

lemma *stack-typ-info-ti-typ-pad-combine'*:
 stack-typ-info t \implies
 stack-typ-info (ti-typ-pad-combine TYPE('b::stack-type) acc upd algn nm t)
 by (*auto simp add: ti-typ-pad-combine-def Let-def stack-typ-info-ti-typ-combine'*
stack-typ-info-ti-pad-combine)

lemma *stack-typ-info-ti-typ-pad-combine*:
 stack-typ-info (typ-info-t (TYPE('b))) \implies stack-typ-info t \implies
 stack-typ-info (ti-typ-pad-combine TYPE('b::c-type) acc upd algn nm t)
 by (*auto simp add: ti-typ-pad-combine-def Let-def stack-typ-info-ti-typ-combine*
stack-typ-info-ti-pad-combine)

lemma *stack-typ-info-map-align*:
 stack-typ-info t \implies stack-typ-info (map-align algn t)

```

by (cases t) (auto simp add: stack-typ-info-def)

lemma stack-typ-info-final-pad: stack-typ-info t  $\implies$ 
  stack-typ-info (final-pad algn t)
by (auto simp add: final-pad-def Let-def stack-typ-info-ti-pad-combine stack-typ-info-map-align)

lemmas stack-typ-info-intros =
  stack-typ-info-empty-typ-info
  stack-typ-info-ti-typ-pad-combine
  stack-typ-info-final-pad
  stack-typ-info

named-theorems stack-typ-info

definition valid-root-footprint :: heap-typ-desc  $\Rightarrow$  addr  $\Rightarrow$  typ-uinfo  $\Rightarrow$  bool where
  valid-root-footprint d x t  $\equiv$ 
    let n = size-td t in
      0 < n  $\wedge$  ( $\forall y. y < n \longrightarrow$ 
        snd (d (x + of-nat y)) = list-map (typ-slice-t t y)  $\wedge$  fst (d (x +
of-nat y)))

lemma valid-root-footprint-valid-footprint: valid-root-footprint d x t  $\implies$  valid-footprint
d x t
by (auto simp add: valid-root-footprint-def valid-footprint-def Let-def)

lemma valid-root-footprint-valid-footprint-dom-conv:
  valid-root-footprint d a t
 $\longleftrightarrow$ 
  (valid-footprint d a t  $\wedge$ 
    ( $\forall n. n < \text{size-td } t \longrightarrow \text{dom (snd (d (a + \text{of-nat } n)))} = \{0..<\text{length (typ-slice-t }
t n)\}$ ))
apply (clarsimp simp add: valid-footprint-def valid-root-footprint-def Let-def)
apply (intro iffI conjI)
subgoal by auto
subgoal by auto
subgoal by auto
subgoal by (metis list-map-eq not-None-eq dom-list-map map-le-same-dom-eq)
by (metis dom-list-map map-le-same-dom-eq)

lemma valid-root-footprint-dom-typing:
  valid-root-footprint d a t  $\implies$  n < size-td t  $\implies$ 
  dom (snd (d (a + of-nat n))) = {0..<length (typ-slice-t t n)}
by (simp add: valid-root-footprint-valid-footprint-dom-conv)

lemma valid-root-footprint-typing-conv:
fixes p::'a::c-type ptr
assumes valid: valid-root-footprint d (ptr-val p) (typ-uinfo-t TYPE('a))
assumes n: n < size-of (TYPE('a))

```

shows d ($\text{ptr-val } p + \text{of-nat } n$) = (True , $\text{list-map } (\text{typ-slice-t } (\text{typ-uinfo-t } \text{TYPE}('a)))$
 n)
using $\text{valid } n$
by ($\text{simp add: valid-root-footprint-def Let-def size-of-def prod-eq-iff}$)

definition

$\text{root-ptr-valid} :: \text{heap-typ-desc} \Rightarrow 'a::\text{c-type ptr} \Rightarrow \text{bool}$

where

$\text{root-ptr-valid } d \ p \equiv$
 $\text{valid-root-footprint } d$ ($\text{ptr-val } (p::'a \ \text{ptr})$) ($\text{typ-uinfo-t } \text{TYPE}('a)$) \wedge
 $\text{c-guard } p$

lemma $\text{root-ptr-valid-c-guard}$: $\text{root-ptr-valid } d \ p \implies \text{c-guard } p$

by ($\text{simp add: root-ptr-valid-def}$)

lemma $\text{root-ptr-valid-typing-conv}$:

fixes $p::'a::\text{c-type ptr}$

assumes valid : $\text{root-ptr-valid } d \ p$

assumes n : $n < \text{size-of } (\text{TYPE}('a))$

shows d ($\text{ptr-val } p + \text{of-nat } n$) = (True , $\text{list-map } (\text{typ-slice-t } (\text{typ-uinfo-t } \text{TYPE}('a)))$
 n)
using valid

by ($\text{simp add: root-ptr-valid-def valid-root-footprint-typing-conv [OF - n]}$)

lemma $\text{root-ptr-valid-h-t-valid}$: $\text{root-ptr-valid } d \ p \implies d, \text{c-guard} \models_t p$

by ($\text{simp add: h-t-valid-def root-ptr-valid-def valid-root-footprint-valid-footprint-dom-conv}$)

lemma $\text{valid-root-footprint-cong-state}$:

assumes t : $\text{wf-size-desc } t$

assumes eq : $\bigwedge p'. p' \in \{p \ \dots + \text{size-td } t\} \implies s \ p' = s' \ p'$

shows $\text{valid-root-footprint } s \ p \ t \longleftrightarrow \text{valid-root-footprint } s' \ p \ t$

unfolding $\text{valid-root-footprint-def Let-def}$

using $\text{eq wf-size-desc-gt}(1)[\text{OF } t]$

using $\text{intvl-split}[of \ 1 \ \text{size-td } t \ p]$

by (metis intvlI)

lemma (**in** mem-type) $\text{valid-root-foot-print-ptr-force-type}$:

$\text{valid-root-footprint}$

($\text{ptr-force-type } (p::'a \ \text{ptr}) \ s$) ($\text{ptr-val } p$) ($\text{typ-uinfo-t } (\text{TYPE}('a))$)

by ($\text{simp add: htd-upd-at local.ptr-force-type-def}$

$\text{local.size-of-fold order-less-imp-le valid-root-footprint-def}$)

lemma $\text{list-map-greatest-last}$: $xs \neq [] \implies \text{last } xs = v \implies \text{list-map } xs \ (\text{GREATEST}$

$k. \exists v. \text{list-map } xs \ k = \text{Some } v) = \text{Some } v$

proof ($\text{induct } n == \text{length } xs \ \text{arbitrary: } xs$)

case 0

then show $?case$ **by** simp

next

```

case (Suc n xs)
show ?case
proof (cases xs)
  case Nil
  then show ?thesis using Suc by simp
next
  case (Cons x xs')
  have xs: xs = x#xs' by fact
  show ?thesis
  proof (cases xs')
    case Nil

    have *: (GREATEST k::nat.  $\exists v'. [0 \mapsto v] k = \text{Some } v'$ ) = 0
      by (rule Greatest-equality) (auto simp add: fun-upd-def split: if-split-asm)

    from Nil show ?thesis using Suc.prem1 xs
      by (simp add: list-map-def * fun-upd-def Greatest-equality)
  next
  case (Cons x' xs'')
  from Suc xs Cons obtain n: n = length xs' and non-empty: xs'  $\neq []$  and
  last: last xs' = v
    by simp

  from Suc.hyps (1) [OF n non-empty last]
  have hyp: list-map xs' (GREATEST x.  $\exists v. \text{list-map } xs' x = \text{Some } v$ ) = Some
  v .
  from hyp show ?thesis by (simp add: xs Cons add: list-map-eq split:
  if-split-asm)
  qed
  qed
qed

```

```

lemma valid-root-footprint-valid-simple-footprint:
  assumes valid-td: size-td t  $\leq$  addr-card
  shows valid-root-footprint d x t  $\implies$  valid-simple-footprint d x t
  apply (clarsimp simp add: valid-root-footprint-def valid-simple-footprint-def Let-def)
proof -
  assume sz: 0 < size-td t
  assume root[rule-format]:  $\forall y < \text{size-td } t. \text{snd } (d (x + \text{word-of-nat } y)) = \text{list-map}$ 
  (typ-slice-t t y)  $\wedge$  fst (d (x + word-of-nat y))
  from root [of 0, simplified] sz obtain
  d-x: d x = (True, list-map (typ-slice-t t 0))
  by (cases d x) auto

  have x-HeapType: heap-type-tag d x = HeapType t
  proof -
  from last-typ-slice-t have last (typ-slice-t t 0) = (t, True) .

```

```

from list-map-greatest-last [OF typ-slice-t-not-empty this]
have *: list-map (typ-slice-t t 0) (GREATEST k. ∃ a b. list-map (typ-slice-t t 0) k = Some (a,b) = Some (t, True))
  by auto
from d-x show ?thesis
  apply (clarsimp simp add: heap-type-tag-def *, intro conjI)
  using * apply blast
  by (metis less-irrefl list-map-eq option.simps(3))
qed

show heap-type-tag d x = HeapType t ∧ (∀ y. y ∈ {x + 1..+size-td t - Suc 0} → heap-type-tag d y = HeapFootprint)
proof
  from x-HeapType
  show heap-type-tag d x = HeapType t .
next
  show  $\forall y. y \in \{x + 1..+size-td t - Suc 0\} \longrightarrow heap-type-tag d y = HeapFootprint$ 
proof (standard+)
  fix y
  assume y-bounds: y ∈ {x + 1..+size-td t - Suc 0}
  with sz obtain off where y-off: y = x + word-of-nat off and off: off < size-td t
  by (meson intvlD intvl-plus-sub-Suc)
  from root [OF off] y-off have d-y: d y = (True, list-map (typ-slice-t t off))
  by (cases d (x + word-of-nat off)) auto

  have  $x \notin \{x + 1..+size-td t - Suc 0\}$ 
  using intvl-Suc-nmem'' [OF valid-td] [simplified addr-card-len-of-conv] by blast
  with y-bounds y-off have off-non-zero: off ≠ 0 by (cases off) auto

  from last-typ-slice-t-non-zero [OF off-non-zero] have last (typ-slice-t t off) = (t, False) .
  from list-map-greatest-last [OF typ-slice-t-not-empty this]
  have *: list-map (typ-slice-t t off) (GREATEST k. ∃ a b. list-map (typ-slice-t t off) k = Some (a,b) = Some (t, False))
    by auto
  from d-y show heap-type-tag d y = HeapFootprint
  apply (clarsimp simp add: heap-type-tag-def *, intro conjI)
  using * apply blast
  by (metis less-irrefl list-map-eq option.simps(3))
qed
qed
qed

```

lemma *valid-root-footprint-valid-simple-footprint-typ-uinfo-t:*

assumes *valid-root*: *valid-root-footprint* *d x* (*typ-uinfo-t* *TYPE('a::mem-type)*)
shows *valid-simple-footprint* *d x* (*typ-uinfo-t* *TYPE('a::mem-type)*)
apply (*rule valid-root-footprint-valid-simple-footprint* [*OF - valid-root*])
by (*metis less-imp-le max-size size-of-def typ-uinfo-size*)

lemma *first-in-intvl*:

$b \neq 0 \implies a \in \{a ..+ b\}$
by (*force simp: intvl-def*)

lemma *list-map-comono*:

assumes *s*: *list-map* *m* \subseteq_m *list-map* *n*
shows $m \leq n$
using *s*
proof (*induct m arbitrary: n rule: rev-induct*)
case *Nil* **thus** ?*case* **unfolding** *list-map-def* **by** *simp*
next
case (*snoc* *x xs*)

from *snoc.prem*s **have**

sm : [*length* *xs* \mapsto *x*] ++ *list-map* *xs* \subseteq_m *list-map* *n*
unfolding *list-map-def* **by** *simp*

hence *xsn*: $xs \leq n$

by (*rule snoc.hyps* [*OF map-add-le-mapE*])

have *list-map* *n* (*length* *xs*) = *Some* *x* **using** *sm*

by (*simp add: map-le-def list-map-def merge-dom2 set-zip*)

hence *length* *xs* < *length* *n* **and** $x = n ! \text{length } xs$

by (*auto simp add: list-map-eq split: if-split-asm*)

thus $xs @ [x] \leq n$ **using** *xsn*

by (*simp add: append-one-prefix less-eq-list-def*)

qed

lemma *valid-root-footprint-overlap-sub-ty*:

assumes *valid-root-x*: *valid-root-footprint* *d x t*
assumes *valid-y*: *valid-footprint* *d y s*
assumes *overlap*: $\{x ..+ \text{size-td } t\} \cap \{y ..+ \text{size-td } s\} \neq \{\}$
shows $s \leq t$
using *valid-root-x* *overlap* *valid-y*
by (*auto simp add: valid-root-footprint-def valid-footprint-def Let-def typ-tag-le-def*)
(*metis intvlD list-map-comono typ-slice-sub typ-tag-le-def*)

lemma *valid-root-footprint-type-neq*:

\llbracket *valid-root-footprint* *d p s*;
valid-root-footprint *d q t*;
 $s \neq t \rrbracket \implies$
 $p \notin \{q ..+ (\text{size-td } t)\}$

by (*metis antisym-conv disjoint-iff first-in-intvl neq0-conv valid-root-footprint-def*
valid-root-footprint-overlap-sub-typ valid-root-footprint-valid-footprint)

lemma *valid-root-footprint-ptr-force-type-iff*:

fixes $p :: 'a::\text{mem-type ptr}$

assumes $t: \text{wf-size-desc } t$

shows $\text{valid-root-footprint } (\text{ptr-force-type } p \ s) \ a \ t \longleftrightarrow$

$(\text{valid-root-footprint } s \ a \ t \wedge \text{disjnt } \{a \ ..+ \ \text{size-td } t\} \ (\text{ptr-span } p)) \vee$

$(t = \text{typ-uinfo-t TYPE}'a) \wedge p = \text{Ptr } a)$

proof *cases*

assume $\text{disjnt}: \text{disjnt } \{a \ ..+ \ \text{size-td } t\} \ (\text{ptr-span } p)$

moreover **have** $p \neq \text{Ptr } a$

using $\text{disjnt}[\text{unfolded disjoint-iff}, \text{ THEN spec}, \text{ of } a] \ t[\text{THEN wf-size-desc-gt}(1)]$

by (*cases p*) (*auto simp: intvl-self*)

moreover **have** $\text{valid-root-footprint } (\text{ptr-force-type } p \ s) \ a \ t = \text{valid-root-footprint}$
 $s \ a \ t$

using $t \ \text{disjnt}$

by (*intro valid-root-footprint-cong-state ptr-force-type-disj*) (*simp-all add: disjoint-iff*)

ultimately show *?thesis*

by *simp*

next

assume $\text{ndisjnt}: \neg \text{disjnt } \{a \ ..+ \ \text{size-td } t\} \ (\text{ptr-span } p)$

show *?thesis*

using ndisjnt

by (*metis (no-types) valid-root-footprint-valid-simple-footprint-typ-uinfo-t disjoint-def*

valid-root-foot-print-ptr-force-type intvl-inter ptr-val.ptr-val-def size-of-tag t

valid-simple-footprint-ptr-force-type-iff valid-root-footprint-type-neq)

qed

lemma *valid-root-footprint-fold-ptr-force-type-iff*:

fixes $ps :: 'a::\text{mem-type ptr list}$

assumes [*simp*]: $\text{wf-size-desc } t$

shows $\text{distinct-prop } (\lambda p1 \ p2. \ \text{disjnt } (\text{ptr-span } p1) \ (\text{ptr-span } p2)) \ ps \implies$

$\text{valid-root-footprint } (\text{fold } \text{ptr-force-type } ps \ s) \ a \ t \longleftrightarrow$

$(\text{valid-root-footprint } s \ a \ t \wedge \text{disjnt } \{a \ ..+ \ \text{size-td } t\} \ (\bigcup_{p \in \text{set } ps} \ \text{ptr-span } p))$

\vee

$(t = \text{typ-uinfo-t TYPE}'a) \wedge \text{Ptr } a \in \text{set } ps)$

by (*induction ps arbitrary: s*)

(*auto simp: valid-root-footprint-ptr-force-type-iff size-of-tag distinct-prop-append*)

lemma *valid-simple-footprint-neq*:

assumes $\text{valid-p}: \text{valid-simple-footprint } d \ p \ s$

and $\text{valid-q}: \text{valid-simple-footprint } d \ q \ t$

and $\text{neq}: p \neq q$

shows $p \notin \{q \ ..+ \ (\text{size-td } t)\}$

proof –

have *heap-type-p*: *heap-type-tag* *d p* = *HeapType s*
apply (*metis valid-p valid-simple-footprint-def*)
done

have *heap-type-q*: *heap-type-tag* *d q* = *HeapType t*
apply (*metis valid-q valid-simple-footprint-def*)
done

have *heap-type-q-footprint*:
 $\bigwedge x. x \in \{(q + 1)..+(size-td\ t - Suc\ 0)\} \implies \text{heap-type-tag } d\ x = \text{HeapFootprint}$
apply (*insert valid-q*)
apply (*simp add: valid-simple-footprint-def*)
done

show *?thesis*
using *heap-type-q-footprint heap-type-p neq*
intvl-neq-start heap-type-q
by (*metis heap-tyt-contents.simps(2)*)
qed

lemma *valid-simple-footprint-type-neq*:
 $\llbracket \text{valid-simple-footprint } d\ p\ s;$
 $\text{valid-simple-footprint } d\ q\ t;$
 $s \neq t \rrbracket \implies$
 $p \notin \{q..+(size-td\ t)\}$
apply (*subgoal-tac p \neq q*)
apply (*rule valid-simple-footprint-neq, simp-all*)[1]
apply (*clarsimp simp: valid-simple-footprint-def*)
done

lemma *valid-simple-footprint-neq-disjoint*:
 $\llbracket \text{valid-simple-footprint } d\ p\ s;$
 $\text{valid-simple-footprint } d\ q\ t;$
 $p \neq q \rrbracket \implies$
 $\{p..+(size-td\ s)\} \cap \{q..+(size-td\ t)\} = \{\}$
apply (*rule ccontr*)
apply (*fastforce simp: valid-simple-footprint-neq dest!: intvl-inter*)
done

lemma *valid-simple-footprint-type-neq-disjoint*:
 $\llbracket \text{valid-simple-footprint } d\ p\ s;$
 $\text{valid-simple-footprint } d\ q\ t;$
 $s \neq t \rrbracket \implies$
 $\{p..+(size-td\ s)\} \cap \{q..+(size-td\ t)\} = \{\}$
apply (*subgoal-tac p \neq q*)
apply (*rule valid-simple-footprint-neq-disjoint, simp-all*)[1]
apply (*clarsimp simp: valid-simple-footprint-def*)
done

lemma *valid-simple-footprint-disjnt-or-eq*:
valid-simple-footprint d $a1$ $t1$ \implies *valid-simple-footprint* d $a2$ $t2$ \implies
 $\text{disjnt } \{a1 \text{ ..+ size-td } t1\} \{a2 \text{ ..+ size-td } t2\} \vee (a1 = a2 \wedge t1 = t2)$
using *valid-simple-footprint-type-neq-disjoint*[of d $a1$ $t1$ $a2$ $t2$]
using *valid-simple-footprint-neq-disjoint*[of d $a1$ $t1$ $a2$ $t2$]
by (*auto simp: disjnt-def*)

lemma *valid-root-footprint-type-neq-disjoint*:
 $\llbracket \text{valid-root-footprint } d \text{ } p \text{ } s;$
 $\text{valid-root-footprint } d \text{ } q \text{ } t;$
 $s \neq t \rrbracket \implies$
 $\{p \text{ ..+ (size-td } s)\} \cap \{q \text{ ..+ (size-td } t)\} = \{\}$
by (*metis intvl-inter valid-root-footprint-type-neq*)

lemma *valid-root-footprint-neq*:
assumes *valid-p*: *valid-root-footprint* d p s
and *valid-q*: *valid-root-footprint* d q t
and *neq*: $p \neq q$
shows $p \notin \{q \text{ ..+ (size-td } t)\}$
proof
assume $p \in \{q \text{ ..+ size-td } t\}$
then have $\neg t \leq s$
by (*metis (no-types, opaque-lifting) less-irrefl less-le-trans neq valid-footprint-sub2*
 $\text{valid-p valid-q valid-root-footprint-valid-footprint}$)
moreover
from p **have** $\{p \text{ ..+ size-td } s\} \cap \{q \text{ ..+ size-td } t\} \neq \{\}$
by (*metis disjoint-iff first-in-intvl less-irrefl valid-p valid-root-footprint-def*)
from *valid-root-footprint-overlap-sub-typ* [*OF* *valid-p* *valid-root-footprint-valid-footprint*
[*OF* *valid-q*] *this*]
have $t \leq s$.
ultimately show *False*
by *blast*
qed

lemma *valid-root-footprint-neq-disjoint*:
 $\llbracket \text{valid-root-footprint } d \text{ } p \text{ } s;$ *valid-root-footprint* d q t ; $p \neq q \rrbracket \implies$
 $\{p \text{ ..+ (size-td } s)\} \cap \{q \text{ ..+ (size-td } t)\} = \{\}$
by (*metis intvl-inter valid-root-footprint-neq*)

lemma *valid-root-footprint-disjnt-or-eq*:
valid-root-footprint d $a1$ $t1$ \implies *valid-root-footprint* d $a2$ $t2$ \implies
 $\text{disjnt } \{a1 \text{ ..+ size-td } t1\} \{a2 \text{ ..+ size-td } t2\} \vee (a1 = a2 \wedge t1 = t2)$
using *valid-root-footprint-type-neq-disjoint*[of d $a1$ $t1$ $a2$ $t2$]
using *valid-root-footprint-neq-disjoint*[of d $a1$ $t1$ $a2$ $t2$]
by (*auto simp: disjnt-def*)

definition *ptr-aligned-u* :: *typ-uinfo* \Rightarrow *addr* \Rightarrow *bool* **where**

$ptr\text{-aligned}\text{-}u\ t\ a \equiv 2^{\wedge}(align\text{-}td\ t)\ dvd\ unat\ a$

lemma *ptr-aligned-ptr-aligned-u-conv*:

fixes $p::'a::c\text{-type}\ ptr$

shows $ptr\text{-aligned}\ p = ptr\text{-aligned}\text{-}u\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ (ptr\text{-val}\ p)$

by (*simp add: ptr-aligned-def ptr-aligned-u-def align-of-def typ-uinfo-t-def*)

definition *c-null-guard-u* :: $typ\text{-}uinfo \Rightarrow addr \Rightarrow bool$ **where**

$c\text{-null}\text{-}guard\text{-}u\ t\ a \equiv 0 \notin \{a..+size\text{-}td\ t\}$

lemma *c-null-guard-c-null-guard-u-conv*:

fixes $p::'a::c\text{-type}\ ptr$

shows $c\text{-null}\text{-}guard\ p = c\text{-null}\text{-}guard\text{-}u\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ (ptr\text{-val}\ p)$

by (*simp add: c-null-guard-def c-null-guard-u-def size-of-def*)

definition *c-guard-u* :: $typ\text{-}uinfo \Rightarrow addr \Rightarrow bool$ **where**

$c\text{-guard}\text{-}u\ t\ a \equiv ptr\text{-aligned}\text{-}u\ t\ a \wedge c\text{-null}\text{-}guard\text{-}u\ t\ a$

lemma *c-guard-c-guard-u-conv*:

fixes $p::'a::c\text{-type}\ ptr$

shows $c\text{-guard}\ p = c\text{-guard}\text{-}u\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ (ptr\text{-val}\ p)$

by (*simp add: ptr-aligned-ptr-aligned-u-conv c-guard-def c-guard-u-def c-null-guard-c-null-guard-u-conv*)

definition

root-ptr-valid-u :: $typ\text{-}uinfo \Rightarrow heap\text{-}typ\text{-}desc \Rightarrow addr \Rightarrow bool$ **where**

$root\text{-ptr}\text{-valid}\text{-}u\ t\ d\ a \equiv valid\text{-root}\text{-footprint}\ d\ a\ t \wedge c\text{-guard}\text{-}u\ t\ a$

definition

cvalid-u :: $typ\text{-}uinfo \Rightarrow heap\text{-}typ\text{-}desc \Rightarrow addr \Rightarrow bool$ **where**

$cvalid\text{-}u\ t\ d\ a \equiv valid\text{-footprint}\ d\ a\ t \wedge c\text{-guard}\text{-}u\ t\ a$

lemma *root-ptr-valid-root-ptr-valid-u-conv*:

fixes $p::'a::c\text{-type}\ ptr$

shows $root\text{-ptr}\text{-valid}\ d\ p = root\text{-ptr}\text{-valid}\text{-}u\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ d\ (ptr\text{-val}\ p)$

by (*simp add: root-ptr-valid-def root-ptr-valid-u-def c-guard-c-guard-u-conv*)

lemma *root-ptr-valid-ptr-force-type*:

$c\text{-guard}\ p \Longrightarrow root\text{-ptr}\text{-valid}\ (ptr\text{-force}\text{-type}\ p\ s)\ (p::'a::mem\text{-type}\ ptr)$

by (*simp add: root-ptr-valid-root-ptr-valid-u-conv root-ptr-valid-u-def c-guard-c-guard-u-conv valid-root-foot-print-ptr-force-type*)

lemma *cvalid-cvalid-u-conv*:

fixes $p::'a::c\text{-type}\ ptr$

shows $d \models_t p = cvalid\text{-}u\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ d\ (ptr\text{-val}\ p)$

by (*simp add: h-t-valid-def cvalid-u-def c-guard-c-guard-u-conv*)

lemma *root-ptr-valid-u-cvalid-u*: $root\text{-ptr}\text{-valid}\text{-}u\ t\ d\ a \Longrightarrow cvalid\text{-}u\ t\ d\ a$

by (*simp add: root-ptr-valid-u-def cvalid-u-def valid-root-footprint-valid-footprint*)

lemma *fold-root-ptr-valid-u*:

$root\text{-}ptr\text{-}valid\text{-}u$ ($typ\text{-}uinfo\text{-}t$ $TYPE('a::c\text{-}type)$) d $a = root\text{-}ptr\text{-}valid$ d (PTR ($'a$) a)

by ($simp$ add : $root\text{-}ptr\text{-}valid\text{-}root\text{-}ptr\text{-}valid\text{-}u\text{-}conv$)

lemma $ptr\text{-}force\text{-}type\text{-}eq\text{-}ptr\text{-}force\text{-}type\text{-}u$:

$ptr\text{-}force\text{-}type$ ($p :: 'a::c\text{-}type$ ptr) = $ptr\text{-}force\text{-}type\text{-}u$ ($typ\text{-}uinfo\text{-}t$ $TYPE('a)$) ($ptr\text{-}val$ p)

by ($simp$ add : $ptr\text{-}force\text{-}type\text{-}def$ $ptr\text{-}force\text{-}type\text{-}u\text{-}def$ $typ\text{-}slices\text{-}def$ $typ\text{-}slices\text{-}u\text{-}def$ $size\text{-}of\text{-}def$)

lemma $typ\text{-}slices\text{-}u\text{-}length$ [$simp$]: $length$ ($typ\text{-}slices\text{-}u$ t) = $size\text{-}td$ t

by ($simp$ add : $typ\text{-}slices\text{-}u\text{-}def$)

lemma $typ\text{-}slices\text{-}u\text{-}index$ [$simp$]:

$n < size\text{-}td$ $t \implies typ\text{-}slices\text{-}u$ t ! $n = list\text{-}map$ ($typ\text{-}slice\text{-}t$ t n)

by ($simp$ add : $typ\text{-}slices\text{-}u\text{-}def$)

lemma $valid\text{-}root\text{-}footprint\text{-}ptr\text{-}force\text{-}type\text{-}u$:

$wf\text{-}size\text{-}desc$ $t \implies size\text{-}td$ $t < addr\text{-}card \implies$

$valid\text{-}root\text{-}footprint$ ($ptr\text{-}force\text{-}type\text{-}u$ t a h) a t

by ($simp$ add : $valid\text{-}root\text{-}footprint\text{-}def$ $ptr\text{-}force\text{-}type\text{-}u\text{-}def$ $Let\text{-}def$ $wf\text{-}size\text{-}desc\text{-}gt$ $htd\text{-}upd\text{-}at$)

lemma $valid\text{-}root\text{-}footprint\text{-}ptr\text{-}force\text{-}type\text{-}u\text{-}size$:

$wf\text{-}size\text{-}desc$ $t \implies size\text{-}td$ $t < addr\text{-}card \implies$

$valid\text{-}root\text{-}footprint$ ($ptr\text{-}force\text{-}type\text{-}u$ t a h) a t

by ($simp$ add : $valid\text{-}root\text{-}footprint\text{-}def$ $ptr\text{-}force\text{-}type\text{-}u\text{-}def$ $Let\text{-}def$ $wf\text{-}size\text{-}desc\text{-}gt$ $htd\text{-}upd\text{-}at$)

definition

$stack\text{-}alloc::$ $addr$ $set \implies 'a::$ $mem\text{-}type$ $itself \implies heap\text{-}typ\text{-}desc \implies ('a$ $ptr \times heap\text{-}typ\text{-}desc)$ set **where**

$\langle stack\text{-}alloc$ \mathcal{S} T $d = \{$

$(p::'a$ $ptr, d')$.

$ptr\text{-}span$ $p \subseteq \mathcal{S} \wedge$

$typ\text{-}uinfo\text{-}t$ ($TYPE('a)$) $\neq typ\text{-}uinfo\text{-}t$ ($TYPE(stack\text{-}byte)$) \wedge

$(\forall a \in ptr\text{-}span$ $p. root\text{-}ptr\text{-}valid$ d (PTR ($stack\text{-}byte$) a)) \wedge

$ptr\text{-}aligned$ $p \wedge$

$d' = ptr\text{-}force\text{-}type$ p d

$\}\rangle$

definition

$stack\text{-}allocs::$ $nat \implies addr$ $set \implies 'a::$ $mem\text{-}type$ $itself \implies heap\text{-}typ\text{-}desc \implies ('a$ $ptr \times heap\text{-}typ\text{-}desc)$ set **where**

$\langle stack\text{-}allocs$ n \mathcal{S} T $d = \{$

$(p::'a$ $ptr, d')$.

$0 < n \wedge$

$(\forall i < n. \text{ptr-span } (p +_p \text{ int } i) \subseteq \mathcal{S}) \wedge$
 $\text{typ-uinfo-t } (TYPE('a)) \neq \text{typ-uinfo-t } (TYPE(\text{stack-byte})) \wedge$
 $(\forall a \in \{\text{ptr-val } p ..+ n * \text{size-of } TYPE('a)\} . \text{root-ptr-valid } d \text{ (PTR } (\text{stack-byte})$
 $a)) \wedge$
 $\text{ptr-aligned } p \wedge$
 $d' = \text{fold } (\lambda i. \text{ptr-force-type } (p +_p \text{ int } i)) [0..<n] d$
 \rangle

lemma *stack-alloc-stack-allocs-conv*: *stack-alloc = stack-allocs 1*
by (*simp add: stack-alloc-def stack-allocs-def fun-eq-iff*)

lemma *htd-update-list-other*:

assumes *bound*: *length xs < addr-card*
assumes *notin*: $x \notin \{p..+length\ xs\}$
shows *htd-update-list* $p\ xs\ d\ x = d\ x$
using *bound notin*

proof (*induct xs arbitrary: p d x*)

case *Nil*

then show *?case by simp*

next

case (*Cons x1 xs*)

from *Cons.prem*s **obtain**

$Suc\ (length\ xs) < addr-card$ **and**

$bound'$: $length\ xs < addr-card$ **and**

$x \notin \{p..+Suc\ (length\ xs)\}$ **and**

$notin'$: $x \notin \{(p + 1)..+(length\ xs)\}$ **and**

neg : $x \neq p$

apply *clarsimp*

by (*metis One-nat-def add-diff-cancel-left' intvl-plus-sub-Suc intvl-self*

plus-1-eq-Suc zero-less-Suc)

note *hyp* = *Cons.hyps* [*OF bound' notin'*, **where** $d = (d(p := (True, snd\ (d\ p)$

$++\ x1))$)]

show *?case by (simp add: hyp fun-upd-other neg)*

qed

lemma *dom-typ-slice-t-stack-byte*:

$dom\ (\text{list-map } (\text{typ-slice-t } (\text{typ-uinfo-t } TYPE(\text{stack-byte}))\ n)) = \{0\}$

by (*simp add: typ-info-stack-byte typ-uinfo-t-def*)

lemma *htd-update-list-same'*:

$\llbracket 0 < unat\ k; unat\ k \leq addr-card - length\ v \rrbracket \implies \text{htd-update-list } (p + k)\ v\ h\ p$

$= h\ p$

apply (*insert htd-update-list-same* [**where** $v=v$ **and** $p=p$ **and** $h=h$ **and** $k=unat$

k])

apply *clarsimp*

done

lemma *dom-htd-update-list*:

```

assumes xs-bound:  $\text{length } xs < \text{addr-card}$ 
assumes n-bound:  $n < \text{length } xs$ 
shows  $\text{dom } (\text{snd } (\text{htd-update-list } a \text{ } xs \text{ } d \text{ } (a + \text{word-of-nat } n))) =$ 
 $\text{dom } (\text{snd } (d \text{ } (a + \text{word-of-nat } n))) \cup \text{dom } (xs \text{ } ! \text{ } n)$ 
using xs-bound n-bound
proof (induct xs arbitrary: n a d)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  from Cons.prems obtain
    Suc ( $\text{length } xs < \text{addr-card}$  and
    lxs:  $\text{length } xs < \text{addr-card}$  and
    n:  $n < \text{Suc } (\text{length } xs)$ )
  by auto
  note hyp = Cons.hyps [OF lxs]
  show ?case
  proof (cases n)
    case 0
    show ?thesis
    apply (simp add: 0)
    by (smt (verit, ccfv-threshold) One-nat-def add-cancel-left-right add-diff-cancel-left'
 $\text{dom-map-add fun-upd-same htd-update-list-same' le-eq-less-or-eq le-iff-add}$ 
 $\text{less-Suc-eq-0-disj lxs nat-neq-iff snd-conv sup-commute unsigned-1 zero-order(1)}$ )
  next
  case (Suc m)
  have conv:  $a + 1 + \text{word-of-nat } m = a + (1 + \text{word-of-nat } m)$ 
  by simp

  from Suc n have m-xs:  $m < \text{length } xs$  by simp
  note hyp = hyp [where  $a = a + 1$  and  $d = (d(a := (\text{True}, \text{snd } (d \text{ } a) ++ x)))$ ]
and  $n = m$ , OF m-xs, simplified conv]
  show ?thesis
  apply (simp add: Suc)
  apply (simp add: hyp)
  by (smt (verit, best) add-cancel-left-right add-diff-cancel-left' add-leE fun-upd-other
 $\text{less-natE linorder-not-less lxs m-xs one-plus-x-zero plus-1-eq-Suc}$ )
qed
qed

lemma dom-ptr-retyp:
  fixes p::'a::mem-type ptr
  assumes n:  $n < \text{size-of TYPE('a)}$ 
  shows  $\text{dom } (\text{snd } (\text{ptr-retyp } p \text{ } d \text{ } (\text{ptr-val } p + \text{of-nat } n))) =$ 
 $\text{dom } (\text{snd } (d \text{ } (\text{ptr-val } p + \text{of-nat } n))) \cup$ 
 $\{0..<\text{length } (\text{typ-slice-t } (\text{typ-uinfo-t } \text{TYPE('a)}) \text{ } n)\}$ 
proof –

```

```

have sz-bound: size-of  $TYPE('a)$  < addr-card by simp
hence l-slices: length (typ-slices  $TYPE('a)$ ) < addr-card by simp
with sz-bound n have n':  $n < \text{length} (\text{typ-slices } TYPE('a))$  by simp

from n'
have dom-conv:  $\text{dom} (\text{typ-slices } TYPE('a) ! n) = \{0..<\text{length} (\text{typ-slice-t } (\text{typ-ufinfo-t } TYPE('a)) n)\}$ 
by (simp)
show ?thesis
apply (simp add: ptr-retyp-def size-of-def )
apply (simp add: dom-htd-update-list [OF l-slices n', where  $a = \text{ptr-val } p$  and  $d = d$ ] dom-conv)
done
qed

lemma length-typ-slice-t:  $0 < \text{length} (\text{typ-slice-t } t \ n)$ 
by (cases t) auto

lemma valid-root-footprint-retyp-stack':
fixes p::'a::mem-type ptr
assumes stack:  $\forall a \in \text{ptr-span } p. \text{valid-root-footprint } d \ a \ (\text{typ-ufinfo-t } TYPE(\text{stack-byte}))$ 
shows  $\text{valid-root-footprint}(\text{ptr-retyp } p \ d) (\text{ptr-val } p) (\text{typ-ufinfo-t } TYPE('a))$ 
proof –
{
fix n
assume n-bound:  $n < \text{size-td} (\text{typ-ufinfo-t } TYPE('a))$ 
have  $\text{dom} (\text{snd} (\text{ptr-retyp } p \ d) (\text{ptr-val } p + \text{word-of-nat } n)) = \{0..<\text{length} (\text{typ-slice-t } (\text{typ-ufinfo-t } TYPE('a)) n)\}$ 
proof –
from n-bound have n-sz:  $n < \text{size-of} (TYPE('a))$  by (simp add: size-of-def)
from n-sz
have  $\text{ptr-val } p + \text{word-of-nat } n \in \text{ptr-span } p$ 
by (simp add: intvlI)
note conv = valid-root-footprint-dom-typing [OF stack [rule-format, OF this],
where  $n=0$ , simplified ]
have dom-eq:  $\text{dom} (\text{snd} (d (\text{ptr-val } p + \text{word-of-nat } n))) = \{0\}$ 
by (simp add: conv typ-ufinfo-t-def typ-ufinfo-stack-byte)

from length-typ-slice-t have l:  $0 < \text{length} (\text{typ-slice-t } (\text{typ-ufinfo-t } TYPE('a))$ 
n) by simp
show ?thesis
apply (simp add: dom-ptr-retyp [OF n-sz] dom-eq )
using l atLeastLessThan-iff by blast
qed
} note dom-conv = this
show ?thesis
by (simp add: valid-root-footprint-valid-footprint-dom-conv ptr-retyp-valid-footprint
dom-conv)
qed

```



```

lemma (in mem-type) ptr-force-type-valid-footprint:
  valid-footprint (ptr-force-type p d) (ptr-val (p::'a ptr)) (typ-uinfo-t TYPE('a))
  using valid-root-foot-print-ptr-force-type valid-root-footprint-valid-footprint by
  blast

lemma ptr-force-type-valid-footprint:
  valid-footprint (ptr-force-type p d) (ptr-val (p::'a::mem-type ptr)) (typ-uinfo-t
  TYPE('a))
  using valid-root-foot-print-ptr-force-type valid-root-footprint-valid-footprint by
  blast

lemma valid-root-footprint-retyp-stack:
  fixes p::'a::mem-type ptr
  assumes stack:  $\forall a \in \text{ptr-span } p. \text{valid-root-footprint } d \ a \ (\text{typ-uinfo-t } \text{TYPE}(\text{stack-byte}))$ 
  shows valid-root-footprint(ptr-force-type p d) (ptr-val p) (typ-uinfo-t TYPE('a))
proof -
  {
    fix n
    assume n-bound:  $n < \text{size-td } (\text{typ-info-t } \text{TYPE}('a))$ 
    have dom (snd (ptr-force-type p d (ptr-val p + word-of-nat n))) =
      {0.. $\text{length } (\text{typ-slice-t } (\text{typ-uinfo-t } \text{TYPE}('a)) \ n)$ }
    proof -
      from n-bound have n-sz:  $n < \text{size-of } (\text{TYPE}('a))$  by (simp add: size-of-def)
      from n-sz
      have ptr-val p + word-of-nat n  $\in \text{ptr-span } p$ 
      by (simp add: intervalI)
      note conv = valid-root-footprint-dom-typing [OF stack [rule-format, OF this],
where n=0, simplified ]
      have dom-eq:  $\text{dom } (\text{snd } (d \ (\text{ptr-val } p + \text{word-of-nat } n))) = \{0\}$ 
      by (simp add: conv typ-uinfo-t-def typ-info-stack-byte)

      from length-typ-slice-t have l:  $0 < \text{length } (\text{typ-slice-t } (\text{typ-uinfo-t } \text{TYPE}('a))$ 
n) by simp
      show ?thesis
      by (simp add: n-sz size-of-tag valid-root-foot-print-ptr-force-type
        valid-root-footprint-dom-typing)
    qed
  } note dom-conv = this
  show ?thesis
  by (simp add: valid-root-footprint-valid-footprint-dom-conv ptr-force-type-valid-footprint
    dom-conv)
qed

lemma root-ptr-valid-retyp-stack':
  fixes p::'a::mem-type ptr
  assumes stack:  $\forall a \in \text{ptr-span } p. \text{root-ptr-valid } d \ (\text{PTR } (\text{stack-byte}) \ a)$ 
  assumes aligned: ptr-aligned p

```

shows *root-ptr-valid* (*ptr-retyp* *p* *d*) *p*
proof –
from *stack*
have *stack'*: $\forall a \in \text{ptr-span } p. \text{valid-root-footprint } d \ a \ (\text{typ-uinfo-t } \text{TYPE}(\text{stack-byte}))$
by (*simp add: root-ptr-valid-def*)
from *stack aligned* **have** *c-guard*: *c-guard* *p*
apply (*simp add: root-ptr-valid-def c-guard-def c-null-guard-def*)
using *first-in-intvl* **by** *blast*

from *valid-root-footprint-retyp-stack'* [*OF stack'*]
have *valid-root-footprint* (*ptr-retyp* *p* *d*) (*ptr-val* *p*) (*typ-uinfo-t* *TYPE('a)*),
with *c-guard* **show** *?thesis*
by (*simp add: root-ptr-valid-def*)
qed

lemma *root-ptr-valid-retyp-stack*:
fixes *p*::*'a*::*mem-type* *ptr*
assumes *stack*: $\forall a \in \text{ptr-span } p. \text{root-ptr-valid } d \ (\text{PTR } (\text{stack-byte}) \ a)$
assumes *aligned*: *ptr-aligned* *p*
shows *root-ptr-valid* (*ptr-force-type* *p* *d*) *p*
proof –
from *stack*
have *stack'*: $\forall a \in \text{ptr-span } p. \text{valid-root-footprint } d \ a \ (\text{typ-uinfo-t } \text{TYPE}(\text{stack-byte}))$
by (*simp add: root-ptr-valid-def*)
from *stack aligned* **have** *c-guard*: *c-guard* *p*
apply (*simp add: root-ptr-valid-def c-guard-def c-null-guard-def*)
using *first-in-intvl* **by** *blast*

from *valid-root-footprint-retyp-stack* [*OF stack'*]
have *valid-root-footprint* (*ptr-force-type* *p* *d*) (*ptr-val* *p*) (*typ-uinfo-t* *TYPE('a)*),
with *c-guard* **show** *?thesis*
by (*simp add: root-ptr-valid-def*)
qed

lemma *fold-ptr-retyp-other*:
fixes *p*::*'a*::*mem-type* *ptr*
assumes *a-notin*: $a \notin \{\text{ptr-val } p \ ..+ \ n * \text{size-of } \text{TYPE}('a)\}$
shows (*fold* ($\lambda i. \text{ptr-retyp } (p \ +_p \ \text{int } i)$) [$0 \ ..<n$] *d*) $a = d \ a$
using *a-notin*
proof (*induct* *n* *arbitrary*: *d*)
case *0*
then **show** *?case* **by** *simp*
next
case (*Suc* *n*)
from *Suc.prem*s **have** *a-notin*: $a \notin \{\text{ptr-val } p \ ..+ \ \text{Suc } n * \text{size-of } \text{TYPE}('a)\}$.
from *a-notin*
have *a-notin-n*: $a \notin \{\text{ptr-val } p \ ..+ \ n * \text{size-of } \text{TYPE}('a)\}$
by (*metis* *add-leD2* *intvlD* *intvlI* *linorder-not-less* *mult commute* *mult-Suc-right*)
from *a-notin*

```

have a-notin-Suc:  $a \notin \text{ptr-span } (p +_p \text{ int } n)$ 
  apply (clarsimp simp add: intvl-def ptr-add-def)
  by (metis (no-types, opaque-lifting) add.commute add-less-cancel-right mult.commute of-nat-add of-nat-mult)

```

```

have fold-split: ( $\text{fold } (\lambda i. \text{ptr-retyp } (p +_p \text{ int } i)) ([0..<\text{Suc } n]) d$ ) =
  ( $\text{fold } (\lambda i. \text{ptr-retyp } (p +_p \text{ int } i)) ([0..<n] @ [n]) d$ )
  apply (simp only: Many-More.split-upt-on-n [where n=n])
  apply simp
  done
show ?case
  apply (simp only: fold-split)
  apply (simp only: fold-append)
  apply (simp)
  apply (subst (2) Suc.hyps [OF a-notin-n, symmetric])
  apply (rule ptr-retyp-d [OF a-notin-Suc])
  done

```

qed

```

lemma (in mem-type) ptr-force-type-d:
   $x \notin \{\text{ptr-val } (p::'a \text{ ptr})..+ \text{size-of TYPE('a)}\} \implies$ 
   $\text{ptr-force-type } p \ d \ x = d \ x$ 
  by (simp add: htd-upd-disj local.ptr-force-type-def)

```

```

lemma fold-ptr-force-type-other:
  fixes  $p::'a::\text{mem-type ptr}$ 
  assumes a-notin:  $a \notin \{\text{ptr-val } p ..+ n * \text{size-of TYPE('a)}\}$ 
  shows ( $\text{fold } (\lambda i. \text{ptr-force-type } (p +_p \text{ int } i)) [0..<n] d$ )  $a = d \ a$ 
  using a-notin
proof (induct n arbitrary: d)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prems have a-notin:  $a \notin \{\text{ptr-val } p..+ \text{Suc } n * \text{size-of TYPE('a)}\}$  .
  from a-notin
  have a-notin-n:  $a \notin \{\text{ptr-val } p..+ n * \text{size-of TYPE('a)}\}$ 
  by (metis add-leD2 intvlD intvlI linorder-not-less mult.commute mult-Suc-right)
  from a-notin
  have a-notin-Suc:  $a \notin \text{ptr-span } (p +_p \text{ int } n)$ 
  apply (clarsimp simp add: intvl-def ptr-add-def)
  by (metis (no-types, opaque-lifting) add.commute add-less-cancel-right mult.commute of-nat-add of-nat-mult)

```

```

have fold-split: ( $\text{fold } (\lambda i. \text{ptr-force-type } (p +_p \text{ int } i)) ([0..<\text{Suc } n]) d$ ) =
  ( $\text{fold } (\lambda i. \text{ptr-force-type } (p +_p \text{ int } i)) ([0..<n] @ [n]) d$ )
  apply (simp only: Many-More.split-upt-on-n [where n=n])
  apply simp
  done

```

```

show ?case
  apply (simp only: fold-split)
  apply (simp only: fold-append)
  apply (simp)
  apply (subst (2) Suc.hyps [OF a-notin-n, symmetric])
  apply (rule ptr-force-type-d [OF a-notin-Suc])
  done
qed

lemma root-ptr-valid-domain:
  fixes p::'a::mem-type ptr
  assumes root-ptr-valid d p
  assumes  $\bigwedge a. a \in \text{ptr-span } p \implies d' a = d a$ 
  shows root-ptr-valid d' p
  by (metis (no-types, lifting) assms(1) assms(2) root-ptr-valid-def s-footprintD
s-footprintI2 size-of-tag valid-root-footprint-def)

lemma root-ptr-valid-domain':
  fixes p::'a::mem-type ptr
  assumes  $\bigwedge a. a \in \text{ptr-span } p \implies d' a = d a$ 
  shows root-ptr-valid d' p = root-ptr-valid d p
  by (metis assms root-ptr-valid-domain)

lemma root-ptr-valid-range-not-NULL:
  root-ptr-valid htd (p :: ('a :: c-type) ptr)
   $\implies 0 \notin \{\text{ptr-val } p \text{ ..+ size-of TYPE('a)}\}$ 
  apply (clarsimp simp: root-ptr-valid-def)
  apply (metis c-guard-def c-null-guard-def)
  done

lemma intvl-no-overflow-nat':
  assumes no-overflow: unat a + b  $\leq 2 \wedge \text{len-of TYPE('a::len)}$ 
  shows ( $x \in \{(a :: 'a \text{ word}) \text{ ..+ } b\} = (\text{unat } a \leq \text{unat } x \wedge \text{unat } x < (\text{unat } a + b))$ )
  apply (simp add: intvl-def)
  using no-overflow
  apply (intro iffI conjI)
  apply (smt (verit) add.commute nat-add-left-cancel-less nat-le-iff-add trans-less-add1
unat-of-nat-len word-arith-nat-add)
  apply (smt (verit) add.commute nat-add-left-cancel-less nat-le-iff-add trans-less-add1
unat-of-nat-len word-arith-nat-add)
  by (metis nat-add-left-cancel-less nat-le-iff-add of-nat-add unat-eq-of-nat unat-lt2p)

lemma intvl-no-overflow-nat:
  assumes no-overflow: unat a + b  $\leq \text{addr-card}$ 
  shows ( $x \in \{(a :: \text{addr-bitsize word}) \text{ ..+ } b\} = (\text{unat } a \leq \text{unat } x \wedge \text{unat } x < (\text{unat } a + b))$ )
  using no-overflow unfolding addr-card-def using intvl-no-overflow-nat' by
(metis card-word)

```

lemma *intvl-no-overflow-nat-conv*:
assumes *no-overflow*: $\text{unat } a + b \leq \text{addr-card}$
shows $\{(a :: \text{addr-bitsize word}) ..+ b\} = \{x. (\text{unat } a \leq \text{unat } x \wedge \text{unat } x < (\text{unat } a + b))\}$
using *intvl-no-overflow-nat [OF no-overflow]* **by** *blast*

lemma *zero-not-in-intvl-no-overflow*:
 $0 \notin \{a :: 'a::\text{len word} ..+ b\} \implies \text{unat } a + b \leq 2 \wedge \text{len-of TYPE}('a)$
apply (*rule ccontr*)
apply (*simp add: intvl-def not-le*)
apply (*drule-tac x=2 ^ len-of TYPE('a) - unat a in spec*)
apply (*clarsimp simp: not-less*)
by (*smt (verit) Nat.le-diff-conv2 add commute add.left-neutral add-uminus-conv-diff*

cancel-comm-monoid-add-class.diff-cancel diff-zero linorder-not-less
nat-less-le of-nat-numeral semiring-1-class.of-nat-power unat-0
unat-lt2p unat-minus' word-arith-nat-add word-exp-length-eq-0

lemma *root-ptr-valid-last-byte-no-overflow*:
root-ptr-valid htd (p :: ('a :: c-type) ptr)
 $\implies \text{unat } (\text{ptr-val } p) + \text{size-of TYPE}('a) \leq 2 \wedge \text{len-of TYPE}(\text{addr-bitsize})$
by (*metis c-guard-def c-null-guard-def root-ptr-valid-def*
zero-not-in-intvl-no-overflow)

lemma *root-ptr-valid-retyps-stack'*:
fixes *p::'a::mem-type ptr*
assumes *stack*: $\forall a \in \{\text{ptr-val } p ..+ n * \text{size-of TYPE}('a)\}. \text{root-ptr-valid } d (\text{PTR}(\text{stack-byte}) a)$
assumes *aligned*: *ptr-aligned p*
assumes *i-bound*: $i < n$
shows *root-ptr-valid* (*fold* ($\lambda i. \text{ptr-retyp } (p +_p \text{int } i)$) $[0..<n]$ *d*) ($p +_p \text{int } i$)
using *stack i-bound*
proof (*induct n arbitrary: d*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)

have *stack*: $\forall a \in \{\text{ptr-val } p ..+ \text{Suc } n * \text{size-of TYPE}('a)\}. \text{root-ptr-valid } d (\text{PTR}(\text{stack-byte}) a)$ **by** *fact*
from *stack* **have** *stack-n*: $\forall a \in \{\text{ptr-val } p ..+ n * \text{size-of TYPE}('a)\}. \text{root-ptr-valid } d (\text{PTR}(\text{stack-byte}) a)$
by (*metis add commute intvlD intvlI mult-Suc trans-less-add1*)
from *stack*
have *stack-Suc*: $\forall a \in \text{ptr-span } (p +_p \text{int } n). \text{root-ptr-valid } d (\text{PTR}(\text{stack-byte}) a)$
apply (*clarsimp simp add: intvl-def ptr-add-def*)
by (*metis (no-types, opaque-lifting) add.assoc add commute add-less-cancel-right*)

```

of-nat-add of-nat-mult)

from stack have no-overflow:  $0 \notin \{\text{ptr-val } p..+ \text{Suc } n * \text{size-of } \text{TYPE}('a)\}$ 
apply (clarsimp simp add: intvl-def)
by (metis c-guard-NULL-simp root-ptr-valid-def)

from stack have no-overflow':  $\forall a \in \{\text{ptr-val } p..+ \text{Suc } n * \text{size-of } \text{TYPE}('a)\}. 0$ 
 $\notin \text{ptr-span } (\text{PTR}(\text{stack-byte}) a)$ 
apply (clarsimp simp add: intvl-def)
by (metis c-guard-NULL-simp root-ptr-valid-def)

have i-bound:  $i < \text{Suc } n$  by fact

have fold-split:  $(\text{fold } (\lambda i. \text{ptr-retyp } (p +_p \text{int } i)) ([0..<\text{Suc } n]) d) =$ 
 $(\text{fold } (\lambda i. \text{ptr-retyp } (p +_p \text{int } i)) ([0..<n] @ [n]) d)$ 
apply (simp only: Many-More.split-upt-on-n [where n=n])
apply simp
done

have sz-a:  $0 < \text{size-of } (\text{TYPE}('a))$ 
by simp
hence sz-bound:  $\text{size-of } \text{TYPE}(\text{stack-byte}) \leq \text{size-of } \text{TYPE}('a)$ 
using size-of-stack-byte(1) by linarith

from stack
have bound3:  $\text{unat } (\text{ptr-val } p) + n * \text{size-of } (\text{TYPE}('a)) < \text{addr-card}$ 
by (smt (verit, ccfv-SIG) add.commute add.left-neutral add-less-cancel-right
c-guard-NULL-simp len-of-addr-card less-le mult.commute mult-Suc-right not-less
root-ptr-valid-def stack-n sz-a zero-not-in-intvl-no-overflow)
from bound3 have unat-dist1:  $\text{unat } (\text{ptr-val } (p +_p \text{int } n)) = \text{unat } (\text{ptr-val } p) +$ 
 $n * \text{size-of } (\text{TYPE}('a))$ 
by (smt (verit, ccfv-threshold) Abs-fnat-hom-add Abs-fnat-hom-mult CTypes-
Defs.ptr-add-def len-of-addr-card
of-int-of-nat-eq of-nat-inverse ptr-val.ptr-val-def word-unat.Rep-inverse')
show ?case
proof (cases i=n)
case True
show ?thesis
apply (simp only: fold-split)
apply (simp only: fold-append)
apply (simp add: True)
apply (rule root-ptr-valid-retyp-stack')
subgoal
apply clarify
apply (rule root-ptr-valid-domain)
apply (rule stack-Suc [rule-format])
apply assumption
apply (rule fold-ptr-retyp-other)

```

```

    apply (subst (asm) intvl-no-overflow-nat-conv)
  subgoal
    apply (simp add: unat-dist1)
    using bound3
    by (metis c-guard-NULL-simp len-of-addr-card root-ptr-valid-def stack-Suc
    unat-dist1 zero-not-in-intvl-no-overflow)
  apply (subst (asm) intvl-no-overflow-nat-conv)
  subgoal
    apply simp
    using bound3
    by (metis Suc-leI len-of-addr-card unat-lt2p)
  apply (subst intvl-no-overflow-nat-conv)
  subgoal
    using bound3
    by (simp add: mult.commute)
  subgoal using bound3 by (simp add: mult.commute unat-dist1)
  done
  subgoal
    using aligned ptr-aligned-plus by blast
  done
next
  case False
  with i-bound have i-bound-n: i < n by simp
  from no-overflow have bound1: unat (ptr-val p + word-of-nat n * word-of-nat
  (size-of TYPE('a))) < addr-card
    by (metis len-of-addr-card unsigned-less)
  from Suc.hyps [OF stack-n i-bound-n]
  have hyp: root-ptr-valid (fold (λi. ptr-retyp (p +p int i)) [0..n] d) (p +p int
  i) .
  from bound3 i-bound-n have unat-dist2: unat (ptr-val (p +p int i)) = unat
  (ptr-val p) + i * size-of (TYPE('a))
    by (smt (verit, ccfv-threshold) Abs-fnat-hom-add Abs-fnat-hom-mult CTypes-
  Defs.ptr-add-def add.commute
    add-less-cancel-right len-of-addr-card mult-strict-right-mono nat-less-le
  of-int-of-nat-eq of-nat-inverse
    order-less-le-trans ptr-val.ptr-val-def sz-nzero word-unat.Rep-inverse')
  show ?thesis
    apply (simp only: fold-split)
    apply (simp only: fold-append)
    apply (simp)
    apply (rule root-ptr-valid-domain)
    apply (rule hyp)
    apply (rule ptr-retyp-d)

  apply (subst intvl-no-overflow-nat-conv)
  subgoal
    using bound1
    by (metis c-guard-NULL-simp len-of-addr-card root-ptr-valid-def stack-Suc
  zero-not-in-intvl-no-overflow)

```

```

subgoal for a
  apply (subst (asm) intvl-no-overflow-nat-conv)
  subgoal using bound3 i-bound-n
    apply (simp add: ptr-add-def)
    by (metis (no-types, lifting) CTypesDefs.ptr-add-def hyp len-of-addr-card
of-int-of-nat-eq
      ptr-val.ptr-val-def root-ptr-valid-last-byte-no-overflow)
  subgoal using i-bound-n
    by (auto simp add: unat-dist1 unat-dist2 dest!: le-less-trans)
    (metis add.commute le-def less-le-mult-nat mem-type-class.mem-type-simps(3))
  done
done
qed
qed

```

```

lemma root-ptr-valid-retyps-stack:
  fixes p::'a::mem-type ptr
  assumes stack:  $\forall a \in \{\text{ptr-val } p \dots + n * \text{size-of } \text{TYPE}('a)\}$ . root-ptr-valid d (PTR
(stack-byte) a)
  assumes aligned: ptr-aligned p
  assumes i-bound:  $i < n$ 
  shows root-ptr-valid (fold ( $\lambda i. \text{ptr-force-type } (p +_p \text{int } i)$ ) [ $0..<n$ ] d) ( $p +_p \text{int}$ 
i)
  using stack i-bound
proof (induct n arbitrary: d)
  case 0
  then show ?case by simp
next
  case (Suc n)

```

```

  have stack:  $\forall a \in \{\text{ptr-val } p \dots + \text{Suc } n * \text{size-of } \text{TYPE}('a)\}$ . root-ptr-valid d (PTR(stack-byte)
a) by fact
  from stack have stack-n:  $\forall a \in \{\text{ptr-val } p \dots + n * \text{size-of } \text{TYPE}('a)\}$ . root-ptr-valid
d (PTR(stack-byte) a)
  by (metis add.commute intvlD intvlI mult-Suc trans-less-add1)
  from stack
  have stack-Suc:  $\forall a \in \text{ptr-span } (p +_p \text{int } n)$ . root-ptr-valid d (PTR(stack-byte)
a)
  apply (clarsimp simp add: intvl-def ptr-add-def)
  by (metis (no-types, opaque-lifting) add.assoc add.commute add-less-cancel-right
of-nat-add of-nat-mult)

```

```

from stack have no-overflow:  $0 \notin \{\text{ptr-val } p \dots + \text{Suc } n * \text{size-of } \text{TYPE}('a)\}$ 
  apply (clarsimp simp add: intvl-def)
  by (metis c-guard-NULL-simp root-ptr-valid-def)

```

```

from stack have no-overflow':  $\forall a \in \{\text{ptr-val } p \dots + \text{Suc } n * \text{size-of } \text{TYPE}('a)\}$ .  $0$ 
 $\notin \text{ptr-span } (\text{PTR}(\text{stack-byte}) a)$ 

```



```

apply (clarsimp simp add: intvl-def)
by (metis c-guard-NULL-simp root-ptr-valid-def)

have i-bound:  $i < \text{Suc } n$  by fact

have fold-split: (fold ( $\lambda i.$  ptr-force-type ( $p +_p$  int  $i$ )) ( $[0..<\text{Suc } n]$ )  $d$ ) =
  (fold ( $\lambda i.$  ptr-force-type ( $p +_p$  int  $i$ )) ( $[0..<n]$  @  $[n]$ )  $d$ )
apply (simp only: Many-More.split-upt-on-n [where  $n=n$ ])
apply simp
done

have sz-a:  $0 < \text{size-of } (\text{TYPE}'a)$ 
by simp
hence sz-bound:  $\text{size-of } \text{TYPE}(\text{stack-byte}) \leq \text{size-of } \text{TYPE}'a$ 
using size-of-stack-byte(1) by linarith

from stack
have bound3:  $\text{unat } (\text{ptr-val } p) + n * \text{size-of } (\text{TYPE}'a) < \text{addr-card}$ 
by (smt (verit, ccfv-SIG) add.commute add.left-neutral add-less-cancel-right
c-guard-NULL-simp len-of-addr-card less-le mult.commute mult-Suc-right not-less
root-ptr-valid-def stack-n sz-a zero-not-in-intvl-no-overflow)
from bound3 have unat-dist1:  $\text{unat } (\text{ptr-val } (p +_p \text{int } n)) = \text{unat } (\text{ptr-val } p) +$ 
 $n * \text{size-of } (\text{TYPE}'a)$ 
by (smt (verit, ccfv-threshold) Abs-fnat-hom-add Abs-fnat-hom-mult CTypes-
Defs.ptr-add-def len-of-addr-card
of-int-of-nat-eq of-nat-inverse ptr-val.ptr-val-def word-unat.Rep-inverse')
show ?case
proof (cases  $i=n$ )
case True
show ?thesis
apply (simp only: fold-split)
apply (simp only: fold-append)
apply (simp add: True)
apply (rule root-ptr-valid-retyp-stack)
subgoal
apply clarify
apply (rule root-ptr-valid-domain)
apply (rule stack-Suc [rule-format])
apply assumption
apply (rule fold-ptr-force-type-other)
apply (subst (asm) intvl-no-overflow-nat-conv)
subgoal
apply (simp add: unat-dist1)
using bound3
by (metis c-guard-NULL-simp len-of-addr-card root-ptr-valid-def stack-Suc
unat-dist1 zero-not-in-intvl-no-overflow)
apply (subst (asm) intvl-no-overflow-nat-conv)
subgoal
apply simp

```

```

    using bound3
    by (metis Suc-leI len-of-addr-card unat-lt2p)
  apply (subst intvl-no-overflow-nat-conv)
  subgoal
    using bound3
    by (simp add: mult.commute)
  subgoal using bound3 by (simp add: mult.commute unat-dist1)
  done
  subgoal
    using aligned ptr-aligned-plus by blast
  done
next
  case False
  with i-bound have i-bound-n:  $i < n$  by simp
  from no-overflow have bound1:  $\text{unat}(\text{ptr-val } p + \text{word-of-nat } n * \text{word-of-nat}(\text{size-of TYPE('a)))) < \text{addr-card}$ 
    by (metis len-of-addr-card unsigned-less)
  from Suc.hyps [OF stack-n i-bound-n]
  have hyp:  $\text{root-ptr-valid}(\text{fold}(\lambda i. \text{ptr-force-type}(p +_p \text{int } i)) [0..<n] d)(p +_p \text{int } i)$ .
  from bound3 i-bound-n have unat-dist2:  $\text{unat}(\text{ptr-val}(p +_p \text{int } i)) = \text{unat}(\text{ptr-val } p) + i * \text{size-of}(\text{TYPE('a)})$ 
    by (smt (verit, ccv-threshold) Abs-fnat-hom-add Abs-fnat-hom-mult CTypes-
      Defs.ptr-add-def add.commute
      add-less-cancel-right len-of-addr-card mult-strict-right-mono nat-less-le
      of-int-of-nat-eq of-nat-inverse
      order-less-le-trans ptr-val.ptr-val-def sz-nzero word-unat.Rep-inverse')
  show ?thesis
    apply (simp only: fold-split)
    apply (simp only: fold-append)
    apply (simp)
    apply (rule root-ptr-valid-domain)
    apply (rule hyp)
    apply (rule ptr-force-type-d)

  apply (subst intvl-no-overflow-nat-conv)
  subgoal
    using bound1
    by (metis c-guard-NULL-simp len-of-addr-card root-ptr-valid-def stack-Suc
      zero-not-in-intvl-no-overflow)
  subgoal for a
    apply (subst (asm) intvl-no-overflow-nat-conv)
    subgoal using bound3 i-bound-n
      apply (simp add: ptr-add-def)
      by (metis (no-types, lifting) CTypesDefs.ptr-add-def hyp len-of-addr-card
        of-int-of-nat-eq
        ptr-val.ptr-val-def root-ptr-valid-last-byte-no-overflow)
    subgoal using i-bound-n
      by (auto simp add: unat-dist1 unat-dist2 dest!: le-less-trans)

```

```

      (metis add.commute le-def less-le-mult-nat mem-type-class.mem-type-simps(3))
    done
  done
qed
qed

```

definition *stack-free* :: *heap-typ-desc* \Rightarrow *addr set* **where**
stack-free *d* = {*a*. *root-ptr-valid* *d* (*PTR* (*stack-byte*) *a*)}

lemma *stack-alloc-cases* [*consumes 1*]:

fixes *p::'a::mem-type ptr*

assumes *stack-alloc*: (*p*, *d'*) \in *stack-alloc* \mathcal{S} *T* *d*

assumes *dest*:

$\llbracket \text{typ-uinfo-t } (TYPE('a)) \neq \text{typ-uinfo-t } (TYPE(\text{stack-byte}));$

$\text{ptr-span } p \subseteq \mathcal{S};$

$\forall a \in \text{ptr-span } p. \text{root-ptr-valid } d (\text{PTR } (\text{stack-byte}) a);$

$\text{ptr-aligned } p; \text{c-guard } p; \text{root-ptr-valid } d' p;$

$d' = \text{ptr-force-type } p \ d \rrbracket \Longrightarrow P$

shows *P*

using *dest root-ptr-valid-retyp-stack stack-alloc root-ptr-valid-c-guard*

by (*auto simp add: stack-alloc-def*)

lemma *stack-allocs-cases* [*consumes 1*]:

fixes *p::'a::mem-type ptr*

assumes *stack-alloc*: (*p*, *d'*) \in *stack-allocs* *n* \mathcal{S} *T* *d*

assumes *dest*:

$\llbracket 0 < n; 0 \notin \{\text{ptr-val } p \dots + n * \text{size-of } TYPE('a)\}; \text{unat } (\text{ptr-val } p) + n * \text{size-of } TYPE('a) \leq \text{addr-card};$

$\text{typ-uinfo-t } (TYPE('a)) \neq \text{typ-uinfo-t } (TYPE(\text{stack-byte}));$

$\forall i < n. \text{ptr-span } (p +_p \text{int } i) \subseteq \mathcal{S};$

$\forall a \in \{\text{ptr-val } p \dots + n * \text{size-of } TYPE('a)\}. \text{root-ptr-valid } d (\text{PTR } (\text{stack-byte}) a);$

$\{\text{ptr-val } p \dots + n * \text{size-of } TYPE('a)\} \subseteq \text{stack-free } d;$

$\text{ptr-aligned } p; \text{c-guard } p; \text{root-ptr-valid } d' p;$

$\forall i < n. \text{ptr-aligned } (p +_p \text{int } i); \forall i < n. \text{c-guard } (p +_p \text{int } i); \forall i < n.$

$\text{root-ptr-valid } d' (p +_p \text{int } i);$

$d' = \text{fold } (\lambda i. \text{ptr-force-type } (p +_p \text{int } i)) [0..<n] \ d \rrbracket \Longrightarrow P$

shows *P*

proof –

from *stack-alloc* **obtain**

bound: $0 < n$ **and**

\mathcal{S} : ($\forall i < n. \text{ptr-span } (p +_p \text{int } i) \subseteq \mathcal{S}$) **and**

non-stack-byte: $\text{typ-uinfo-t } (TYPE('a)) \neq \text{typ-uinfo-t } (TYPE(\text{stack-byte}))$ **and**

valid-stack-byte: $\forall a \in \{\text{ptr-val } p \dots + n * \text{size-of } TYPE('a)\}. \text{root-ptr-valid } d (\text{PTR } (\text{stack-byte}) a)$ **and**

stack-free: $\{\text{ptr-val } p \dots + n * \text{size-of } TYPE('a)\} \subseteq \text{stack-free } d$ **and**

aligned-p: *ptr-aligned* *p* **and**

d': $d' = \text{fold } (\lambda i. \text{ptr-force-type } (p +_p \text{int } i)) [0..<n] \ d$

by (*auto simp add: stack-allocs-def stack-free-def*)

```

from valid-stack-byte have no-overflow:  $0 \notin \{\text{ptr-val } p \dots + n * \text{size-of } \text{TYPE}('a)\}$ 
  using c-guard-NULL-simp root-ptr-valid-c-guard by blast
hence no-overflow':  $\text{unat } (\text{ptr-val } p) + n * \text{size-of } \text{TYPE}('a) \leq \text{addr-card}$ 
  by (metis len-of-addr-card zero-not-in-intvl-no-overflow)
note valid-retyps = root-ptr-valid-retyps-stack [OF valid-stack-byte aligned-p]
show ?thesis
  apply (rule dest)
    apply (rule bound)
    apply (rule no-overflow)
    apply (rule no-overflow')
    apply (rule non-stack-byte)
    apply (rule S)
    apply (rule valid-stack-byte)
    apply (rule stack-free)
    apply (rule aligned-p)
subgoal using valid-retyps [OF bound] root-ptr-valid-c-guard d'
  by auto
subgoal using valid-retyps [OF bound] d'
  by auto
subgoal using aligned-p
  by (auto simp add: ptr-aligned-plus)
subgoal using valid-retyps root-ptr-valid-c-guard d'
  by auto
subgoal using valid-retyps d'
  by auto
apply (rule d')
done
qed

```

lemma *stack-allocs-no-overflow*:
assumes *stack-alloc*: $(p, d') \in \text{stack-allocs } n \mathcal{S} (\text{TYPE}('a::\text{mem-type})) d$
shows $\text{unat } (\text{ptr-val } p) + n * \text{size-of } \text{TYPE}('a) \leq \text{addr-card}$
using *stack-allocs-cases* [*OF stack-alloc*] .

lemma *stack-alloc-ptr-force-type*: $(p, d') \in \text{stack-alloc } \mathcal{S} T d \implies$
 $d' = \text{ptr-force-type } p d$
by (*auto simp add: stack-alloc-def*)

lemma *stack-allocs-ptr-force-type*: $(p, d') \in \text{stack-allocs } n \mathcal{S} T d \implies$
 $d' = \text{fold } (\lambda i. \text{ptr-force-type } (p +_p \text{int } i)) [0..<n] d$
by (*auto simp add: stack-allocs-def*)

lemma *stack-allocs-no-stack-byte*: $(p::'a::\text{mem-type } \text{ptr}, d') \in \text{stack-allocs } n \mathcal{S} T d$
 $\implies \text{typ-uinfo-t } (\text{TYPE}('a)) \neq \text{typ-uinfo-t } (\text{TYPE}(\text{stack-byte}))$
by (*erule stack-allocs-cases*)

lemma *stack-allocs-S*: $(p::'a::\text{mem-type } \text{ptr}, d') \in \text{stack-allocs } n \mathcal{S} T d \implies i < n$
 $\implies \text{ptr-span } (p +_p \text{int } i) \subseteq \mathcal{S}$

by (*erule stack-allocs-cases*) *auto*

lemma *ptr-retyp-split*: *ptr-retyp* (*p*::'*a*::*mem-type ptr*) *d a* =
(*if a* ∈ *ptr-span p* then
 (*True*, *snd* (*d a*) ++ *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) (*unat* (*a* -
ptr-val p))))
 else d a)
by (*auto simp add: ptr-retyp-d ptr-retyp-footprint*)

lemma (*in mem-type*) *ptr-force-type-footprint*:
x ∈ {*ptr-val p*..*size-of TYPE*('*a*)} ⇒
 ptr-force-type (*p*::'*a ptr*) *d x* =
 (*True*, *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) (*unat* (*x* - *ptr-val p*))))
apply(*clarsimp simp: ptr-force-type-def*)
by (*metis add-diff-cancel-left' htd-upd-at intvlD less-imp-le local.lt-size-of-unat-simps*

local.max-size local.typ-slices-index local.typ-slices-length word-unat.Rep-inverse)

lemma *ptr-force-type-split*: *ptr-force-type* (*p*::'*a*::*mem-type ptr*) *d a* =
(*if a* ∈ *ptr-span p* then
 (*True*, *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) (*unat* (*a* - *ptr-val p*))))
 else d a)
by (*auto simp add: ptr-force-type-d ptr-force-type-footprint*)

lemma *in-intvl-Suc*: *x* ∈ {*x*..*Suc n*}
by (*simp add: intvl-self*)

definition *zero-heap*:: *heap-mem* **where**
zero-heap = (*λa. zero-class.zero*)

definition *stack-byte-typing*::*heap-typ-desc* **where**
stack-byte-typing = (*λa. ptr-force-type* (*PTR*(*stack-byte*) *a*) *empty-htd a*)

definition *stack-release*:: '*a*::*mem-type ptr* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc* **where**
stack-release p d = *override-on d stack-byte-typing* (*ptr-span p*)

definition *stack-releases*:: *nat* ⇒ '*a*::*mem-type ptr* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc*
where
stack-releases n p d = *override-on d stack-byte-typing* {*ptr-val p* ..+ *n * size-of*
TYPE('*a*)}

lemma *stack-release-stack-releases-conv*: *stack-release* = *stack-releases 1*
by (*auto simp add: stack-release-def stack-releases-def fun-eq-iff*)

lemma *stack-releases-0* [*simp*]: *stack-releases 0 d* = *id*
by (*rule ext*) (*auto simp add: stack-releases-def*)

lemma *stack-release-stack-alloc-inverse*: $(p, d') \in \text{stack-alloc } \mathcal{S} \ T \ d \implies \text{stack-release } p \ d' = d$

```

apply (rule ext)
subgoal for  $x$ 
apply (clarsimp simp add: stack-alloc-def stack-release-def stack-byte-typing-def)
)
apply (cases  $x \in \text{ptr-span } p$ )
apply (simp add: root-ptr-valid-def)
apply (subst ptr-force-type-footprint)
apply (simp add: in-intvl-Suc)
apply (simp add: prod.expand valid-root-footprint-def)
apply (simp add: ptr-force-type-d)
done
done

```

lemma *stack-releases-stack-allocs-inverse*: $(p, d') \in \text{stack-allocs } n \ \mathcal{S} \ T \ d \implies \text{stack-releases } n \ p \ d' = d$

```

apply (rule ext)
subgoal for  $x$ 
apply (clarsimp simp add: stack-allocs-def stack-releases-def stack-byte-typing-def)
)
apply (cases  $x \in \{\text{ptr-val } p \ ..+ \ n * \text{size-of } \text{TYPE}('a)\}$ )
apply (simp add: root-ptr-valid-def)
apply (subst ptr-force-type-footprint)
apply (simp add: in-intvl-Suc)
apply (simp add: prod.expand valid-root-footprint-def)
apply (simp add: fold-ptr-force-type-other)
done
done

```

lemma *sub-typ-stack-byte*:

$$\text{TYPE}('b::\text{c-type}) \leq_{\tau} \text{TYPE}(\text{stack-byte}) \implies \text{typ-uinfo-t } \text{TYPE}('b) = \text{typ-uinfo-t } \text{TYPE}(\text{stack-byte})$$

by (simp add: sub-typ-def typ-uinfo-t-def typ-info-stack-byte typ-tag-le-def)

lemma *root-ptr-valid-not-subtype-disjoint*:

$$\llbracket \text{root-ptr-valid } d \ (p::'a::\text{mem-type } \text{ptr}); \ d \models_t \ (q::'b::\text{mem-type } \text{ptr}); \ \neg \ \text{TYPE}('b) \leq_{\tau} \ \text{TYPE}('a) \rrbracket \implies \text{ptr-span } p \cap \text{ptr-span } q = \{\}$$

by (metis h-t-valid-def root-ptr-valid-def size-of-tag sub-typ-def valid-root-footprint-overlap-sub-typ)

lemma *stack-alloc-disjoint*:

fixes $q::'b::\text{mem-type } \text{ptr}$

assumes *stack-alloc*: $(p, d') \in \text{stack-alloc } \mathcal{S} \ (\text{TYPE}('a::\text{mem-type})) \ d$

assumes *no-stack*: $\text{typ-uinfo-t } (\text{TYPE}('b)) \neq \text{typ-uinfo-t } (\text{TYPE}(\text{stack-byte}))$

assumes *typed*: $d \models_t q$

shows $\text{ptr-span } p \cap \text{ptr-span } q = \{\}$

proof –
from *stack-alloc* **obtain**
 $\text{typ-ufinfo-t } (TYPE('a)) \neq \text{typ-ufinfo-t } (TYPE(\text{stack-byte}))$ **and**
 $\text{stack-bytes: } \forall a \in \text{ptr-span } p. \text{root-ptr-valid } d \text{ (PTR (stack-byte) } a)$
by (*cases rule: stack-alloc-cases*) *auto*
from *no-stack* **have** $\text{no-sub-typ: } \neg TYPE('b) \leq_{\tau} TYPE(\text{stack-byte})$ **by** (*metis*
sub-typ-stack-byte)
{
fix a
assume $a: a \in \text{ptr-span } p$
have $a \notin \text{ptr-span } q$
proof –
from *stack-bytes* [*rule-format*, *OF a*] **have** $\text{root-ptr-valid } d \text{ (PTR (stack-byte) } a)$.
from *root-ptr-valid-not-subtype-disjoint* [*OF this typed no-sub-typ*] **show**
?thesis
by (*simp add: disjoint-iff first-in-intvl*)
qed
}
then show $\text{ptr-span } p \cap \text{ptr-span } q = \{\}$
by *auto*
qed

lemma *stack-allocs-disjoint*:

fixes $q::'b::\text{mem-type ptr}$
assumes *stack-alloc*: $(p, d') \in \text{stack-allocs } n \mathcal{S} (TYPE('a::\text{mem-type})) d$
assumes *no-stack*: $\text{typ-ufinfo-t } (TYPE('b)) \neq \text{typ-ufinfo-t } (TYPE(\text{stack-byte}))$
assumes *typed*: $d \models_t q$
shows $\{\text{ptr-val } p \text{ ..+ } n * \text{size-of } TYPE('a)\} \cap \text{ptr-span } q = \{\}$
proof –
from *stack-alloc* **obtain**
 $\text{typ-ufinfo-t } (TYPE('a)) \neq \text{typ-ufinfo-t } (TYPE(\text{stack-byte}))$ **and**
 $\text{stack-bytes: } \forall a \in \{\text{ptr-val } p \text{ ..+ } n * \text{size-of } TYPE('a)\}. \text{root-ptr-valid } d \text{ (PTR (stack-byte) } a)$
by (*cases rule: stack-allocs-cases*) *auto*
from *no-stack* **have** $\text{no-sub-typ: } \neg TYPE('b) \leq_{\tau} TYPE(\text{stack-byte})$ **by** (*metis*
sub-typ-stack-byte)
{
fix a
assume $a: a \in \{\text{ptr-val } p \text{ ..+ } n * \text{size-of } TYPE('a)\}$
have $a \notin \text{ptr-span } q$
proof –
from *stack-bytes* [*rule-format*, *OF a*] **have** $\text{root-ptr-valid } d \text{ (PTR (stack-byte) } a)$.
from *root-ptr-valid-not-subtype-disjoint* [*OF this typed no-sub-typ*] **show**
?thesis
by (*simp add: disjoint-iff first-in-intvl*)
qed
}
}

then show $\{ptr\text{-val } p \text{ ..+ } n * \text{size-of } TYPE('a)\} \cap ptr\text{-span } q = \{\}$
by auto
qed

lemma *stack-allocs-contained*:

assumes *stack-alloc*: $(p, d') \in \text{stack-allocs } n \mathcal{S} (TYPE('a::\text{mem-type})) d$
assumes *i*: $i < n$
shows $ptr\text{-span } (p +_p \text{int } i) \subseteq \{ptr\text{-val } p \text{ ..+ } n * \text{size-of } TYPE('a)\}$
using *i stack-allocs-no-overflow* [*OF stack-alloc*]
apply (*clarsimp simp add: intvl-def ptr-add-def*)
by (*metis (no-types, opaque-lifting) mult.commute nat-index-bound of-nat-add of-nat-mult*)

lemma *stack-allocs-disjoint'*:

fixes *q::'b::mem-type ptr*
assumes *stack-alloc*: $(p, d') \in \text{stack-allocs } n \mathcal{S} (TYPE('a::\text{mem-type})) d$
assumes *no-stack*: $\text{typ-uinfo-t } (TYPE('b)) \neq \text{typ-uinfo-t } (TYPE(\text{stack-byte}))$
assumes *typed*: $d \models_t q$
assumes *i*: $i < n$
shows $ptr\text{-span } (p +_p \text{int } i) \cap ptr\text{-span } q = \{\}$
using *stack-allocs-disjoint* [*OF stack-alloc no-stack typed*] *i stack-allocs-contained*
[*OF stack-alloc*]
by blast

lemma *ptr-retyp-valid-footprint-disjoint2*:

$\llbracket \text{valid-footprint } (ptr\text{-retyp } (q::'b::\text{mem-type } ptr) d) p s; \{p \text{ ..+ } \text{size-td } s\} \cap \{ptr\text{-val } q \text{ ..+ } \text{size-of } TYPE('b)\} = \{\} \rrbracket$
 $\implies \text{valid-footprint } d p s$
apply (*clarsimp simp: valid-footprint-def Let-def*)
apply (*drule spec, drule (1) mp*)
apply (*subgoal-tac p + of-nat y \in \{p \text{ ..+ } \text{size-td } s\}*)
apply (*subst (asm) ptr-retyp-d*)
apply *clarsimp*
apply *fast*
apply (*clarsimp simp add: ptr-retyp-d-eq-fst split: if-split-asm*)
apply *fast*
apply (*erule intvlI*)
done

lemma *ptr-force-type-valid-footprint-disjoint2*:

$\llbracket \text{valid-footprint } (ptr\text{-force-type } (q::'b::\text{mem-type } ptr) d) p s; \{p \text{ ..+ } \text{size-td } s\} \cap \{ptr\text{-val } q \text{ ..+ } \text{size-of } TYPE('b)\} = \{\} \rrbracket$
 $\implies \text{valid-footprint } d p s$
by (*simp add: disjoint-iff intvlI ptr-force-type-d valid-footprint-def*)

lemma *ptr-span-no-overflow-split-last-disjoint*:

fixes *p::'a::mem-type ptr*

assumes *no-overflow*: $0 \notin \{\text{ptr-val } p..+\text{Suc } n * \text{size-of } \text{TYPE}('a)\}$
shows $\text{ptr-span } (p +_p \text{int } n) \cap \{\text{ptr-val } p..+n * \text{size-of } \text{TYPE}('a)\} = \{\}$
proof –
have *sz-a*: $0 < \text{size-of } (\text{TYPE}('a))$
by *simp*
from *no-overflow sz-a*
have *bound-Suc*: $\text{unat } (\text{ptr-val } p) + \text{Suc } n * \text{size-of } (\text{TYPE}('a)) \leq \text{addr-card}$
by (*metis len-of-addr-card zero-not-in-intvl-no-overflow*)
with *sz-a*
have *bound-n*: $\text{unat } (\text{ptr-val } p) + n * \text{size-of } (\text{TYPE}('a)) < \text{addr-card}$
by (*metis lessI mult-less-cancel2 nat-add-left-cancel-less order-less-le-trans*)
show *?thesis*
apply (*subst intvl-no-overflow-nat-conv*)
subgoal using *bound-Suc*
apply (*simp add: ptr-add-def*)
by (*smt (verit, best) Abs-fnat-hom-add Abs-fnat-hom-mult add.commute add.left-commute bound-n len-of-addr-card mult.commute of-nat-inverse word-unat.Rep-inverse'*)
apply (*subst intvl-no-overflow-nat-conv*)
subgoal using *bound-n by simp*
subgoal using *bound-n bound-Suc*
by *auto (smt (verit) Abs-fnat-hom-add Abs-fnat-hom-mult CTypesDefs.ptr-add-def bound-n len-of-addr-card mult-of-nat-commute not-le of-int-of-nat-eq ptr-val.ptr-val-def unat-of-nat-len word-unat.Rep-inverse')*
done
qed

lemma *ptr-span-no-overflow-indexes-disjoint*:
fixes *p::'a::mem-type ptr*
assumes *no-overflow*: $0 \notin \{\text{ptr-val } p..+n * \text{size-of } \text{TYPE}('a)\}$
assumes *i*: $i < n$
assumes *j*: $j < n$
assumes *neq*: $i \neq j$
shows $\text{ptr-span } (p +_p \text{int } i) \cap \text{ptr-span } (p +_p \text{int } j) = \{\}$
proof –
have *sz-a*: $0 < \text{size-of } (\text{TYPE}('a))$
by *simp*
from *no-overflow sz-a*
have *bound-n*: $\text{unat } (\text{ptr-val } p) + n * \text{size-of } (\text{TYPE}('a)) \leq \text{addr-card}$
by (*metis len-of-addr-card zero-not-in-intvl-no-overflow*)
from *bound-n i* **have** *bound-i*: $\text{unat } (\text{ptr-val } p) + i * \text{size-of } (\text{TYPE}('a)) < \text{addr-card}$
by (*metis linorder-not-le mult-le-cancel2 nat-add-left-cancel-le order-less-le-trans sz-nzero*)
from *bound-n j* **have** *bound-j*: $\text{unat } (\text{ptr-val } p) + j * \text{size-of } (\text{TYPE}('a)) < \text{addr-card}$
by (*metis linorder-not-le mult-le-cancel2 nat-add-left-cancel-le order-less-le-trans sz-nzero*)

```

show ?thesis
  apply (subst intvl-no-overflow-nat-conv)
  subgoal using bound-i i bound-n
    apply (simp add: ptr-add-def)
    by (smt (verit, ccfv-SIG) len-of-addr-card less-le-mult mult-strict-right-mono
      of-nat-add of-nat-inverse of-nat-le-iff of-nat-less-iff of-nat-mult word-unat.Rep-inverse)
  apply (subst intvl-no-overflow-nat-conv)
  subgoal using bound-j j bound-n
    apply (simp add: ptr-add-def)
    by (smt (verit, ccfv-SIG) len-of-addr-card less-le-mult mult-strict-right-mono
      of-nat-add of-nat-inverse of-nat-le-iff of-nat-less-iff of-nat-mult word-unat.Rep-inverse)
  subgoal using bound-i bound-j i j neq
    by (auto simp add: ptr-add-def)
      (smt (verit, ccfv-SIG) len-of-addr-card less-le-mult mult-strict-right-mono
        of-nat-add of-nat-eq-iff of-nat-inverse of-nat-less-iff of-nat-mult
          order-le-less-trans word-unat.Rep-inverse)
  done
qed

```

```

lemma array-to-index-span:
  assumes x-in:  $x \in \{\text{ptr-val } (p::'a::\text{mem-type ptr})..+ n * \text{size-of TYPE('a)}\}$ 
  shows  $\exists i. i < n \wedge x \in \text{ptr-span } (p +_p \text{ int } i)$ 
  using x-in
  apply (clarsimp simp add: intvl-def ptr-add-def)
  subgoal for k
    apply (rule exI [where  $x=k \text{ div size-of TYPE('a)}$ ])
    by (metis (no-types, opaque-lifting) less-mult-imp-div-less mod-div-decomp
      mod-less-divisor of-nat-add of-nat-mult sz-nzero)
  done

```

```

lemma array-to-index-span-exact:
  assumes x-in:  $x \in \{\text{ptr-val } (p::'a::\text{mem-type ptr})..+ n * \text{size-of TYPE('a)}\}$ 
  shows  $x \in \text{ptr-span } (p +_p \text{ int } ((\text{unat } (x - \text{ptr-val } p)) \text{ div size-of TYPE('a)}))$ 
  using x-in
  apply (clarsimp simp add: intvl-def ptr-add-def)
  by (metis (no-types, opaque-lifting) mod-div-decomp mod-less-divisor of-nat-add
    of-nat-mult
      sz-nzero word-unat.Rep-inverse)

```

```

lemma array-index-span-conv:
   $\{\text{ptr-val } (p::'a::\text{mem-type ptr})..+ n * \text{size-of TYPE('a)}\} = (\bigcup_{i < n. \text{ptr-span } (p +_p \text{ int } i)}$ 
  apply standard
  subgoal
    apply clarsimp
    using array-to-index-span by blast
  subgoal
    apply clarsimp
    by (smt (verit, del-insts) CTypesDefs.ptr-add-def add.assoc intvlD intvlI mult.commute

```

nat-index-bound of-int-of-nat-eq of-nat-add of-nat-mult ptr-val.ptr-val-def)
done

lemma *fold-ptr-retyp-footprint*:

fixes $p: 'a::\text{mem-type ptr}$

assumes *no-overflow*: $0 \notin \{\text{ptr-val } p..+ n * \text{size-of TYPE('a)}\}$

assumes $i: i < n$

assumes $x: x \in \text{ptr-span } (p +_p \text{int } i)$

shows $\text{fold } (\lambda i. \text{ptr-retyp } (p +_p \text{int } i)) [0..<n] d x =$

$(\text{True, snd } (d x) ++ \text{list-map } (\text{typ-slice-t } (\text{typ-uinfo-t TYPE('a)) } (\text{unat } (x$
 $- \text{ptr-val } (p +_p \text{int } i))))))$

using *no-overflow i*

proof (*induct n arbitrary: d*)

case 0

then show *?case by simp*

next

case (*Suc n*)

have *no-overflow*: $0 \notin \{\text{ptr-val } p..+ \text{Suc } n * \text{size-of TYPE('a)}\}$ **by fact**

hence *no-overflow-n*: $0 \notin \{\text{ptr-val } p..+ n * \text{size-of TYPE('a)}\}$

by (*meson intvl-start-le lessI mult-less-cancel2 order-less-le subsetD sz-nzero*)

from *ptr-span-no-overflow-split-last-disjoint [OF no-overflow]*

have *disj*: $\text{ptr-span } (p +_p \text{int } n) \cap \{\text{ptr-val } p..+ n * \text{size-of TYPE('a)}\} = \{\}$.

have $i: i < \text{Suc } n$ **by fact**

show *?case*

proof (*cases i = n*)

case *True*

with *disj x* **have** *x-notin-n*: $x \notin \{\text{ptr-val } p..+ n * \text{size-of TYPE('a)}\}$ **by auto**

from *True x* **have** $x: x \in \text{ptr-span } (p +_p \text{int } n)$ **by simp**

show *?thesis*

apply *simp*

using x

apply (*simp add: ptr-retyp-split*)

apply (*subst fold-ptr-retyp-other [OF x-notin-n]*)

apply (*simp add: True*)

done

next

case *False*

with x *ptr-span-no-overflow-indexes-disjoint [OF no-overflow i, of n]*

have *x-notin-n*: $x \notin \text{ptr-span } (p +_p \text{int } n)$ **by blast**

from *False i* **have** $i < n$ **by simp**

note *hyp = Suc.hyps [OF no-overflow-n this]*

show *?thesis*

apply *simp*

using *x-notin-n*

apply (*simp add: ptr-retyp-split*)

apply (*simp add: hyp*)

done

qed
qed

lemma *ptr-retyp-idem*:

ptr-retyp *p* (*ptr-retyp* (*p*::'*a*::*mem-type ptr*) *d*) = *ptr-retyp* *p* *d*
apply (*rule ext*)
apply (*clarsimp simp add: ptr-retyp-split*)
by (*metis (no-types, lifting) map-add-assoc map-add-dom-eq*)

lemma *fold-ptr-retyp-d-empty*:

fixes *p*::'*a*::*mem-type ptr*
assumes *no-overflow*: $0 \notin \{\text{ptr-val } p..+ n * \text{size-of TYPE('a)}\}$
assumes *i*: $i < n$
assumes *x*: $x \in \text{ptr-span } (p +_p \text{ int } i)$
shows *fold* ($\lambda i. \text{ptr-retyp } (p +_p \text{ int } i)$) [$0..<n$] *d* *x* =
 $(\text{True}, \text{snd } (d \ x) ++ \text{snd } (\text{ptr-retyp } (p +_p \text{ int } i) \ \text{empty-htd } x))$
apply (*simp add: fold-ptr-retyp-footprint [OF no-overflow i x]*)
apply (*simp add: ptr-retyp-footprint [OF x]*)
done

lemma *fold-ptr-retyp-eq-fst*:

assumes *no-overflow*: $0 \notin \{\text{ptr-val } p..+ n * \text{size-of TYPE('a)}\}$
shows*fst* (*fold* ($\lambda i. \text{ptr-retyp } (p +_p \text{ int } i)$) [$0..<n$] *d* *x*) =
 $(\text{if } x \in \{\text{ptr-val } (p::'\text{a}::\text{mem-type ptr})..+ n * \text{size-of TYPE('a)}\} \text{ then True}$
 $\text{else fst } (d \ x))$
proof (*cases* $x \in \{\text{ptr-val } (p::'\text{a}::\text{mem-type ptr})..+ n * \text{size-of TYPE('a)}\}$)
case *True*
from *array-to-index-span [OF True]*
obtain *i* **where** *i*: $i < n$ **and** *x-in*: $x \in \text{ptr-span } (p +_p \text{ int } i)$
by *blast*

from *fold-ptr-retyp-footprint [OF no-overflow i x-in]* *True*
show *?thesis* **by** *simp*
next
case *False*
from *fold-ptr-retyp-other [OF False]* *False* **show** *?thesis* **by** *simp*
qed

lemma *fold-ptr-retyp-valid-footprint-disjoint2*:

assumes *no-overflow*: $0 \notin \{\text{ptr-val } q..+ n * \text{size-of TYPE('b)}\}$
shows $\llbracket \text{valid-footprint } (\text{fold } (\lambda i. \text{ptr-retyp } ((q::'\text{b}::\text{mem-type ptr}) +_p \text{ int } i)) [0..<n]$
 $d) \ p \ s;$
 $\{\text{p}..+\text{size-td } s\} \cap \{\text{ptr-val } q ..+ n * \text{size-of TYPE('b)}\} = \{\}$ \rrbracket
 $\implies \text{valid-footprint } d \ p \ s$
apply (*clarsimp simp: valid-footprint-def Let-def*)
apply (*drule spec, drule (1) mp*)
apply (*subgoal-tac p + of-nat y ∈ {p..+size-td s}*)
apply (*subst (asm) fold-ptr-retyp-other*)

apply *clarsimp*
apply *fast*
apply (*clarsimp simp add: fold-ptr-retyp-eq-fst [OF no-overflow] split: if-split-asm*)
apply *fast*
apply (*erule intvlI*)
done

lemma *ptr-retyp-disjoint2*:
 $\llbracket \text{ptr-retyp } (p::'a::\text{mem-type ptr}) \ d, g \models_t q; \text{ptr-span } p \cap \text{ptr-span } q = \{\} \rrbracket$
 $\implies d, g \models_t (q::'b::\text{mem-type ptr})$
apply(*clarsimp simp: h-t-valid-def*)
apply(*erule ptr-retyp-valid-footprint-disjoint2*)
apply(*simp add: size-of-def*)
apply *fast*
done

lemma *fold-ptr-retyp-disjoint2*:
fixes *p::'a::mem-type ptr*
assumes *no-overflow*: $0 \notin \{\text{ptr-val } p..+ n * \text{size-of TYPE('a)}\}$
shows $\llbracket \text{fold } (\lambda i. \text{ptr-retyp } (p +_p \text{int } i)) \ [0..<n] \ d, g \models_t q; \{\text{ptr-val } p..+ n * \text{size-of TYPE('a)}\} \cap \text{ptr-span } q = \{\} \rrbracket$
 $\implies d, g \models_t (q::'b::\text{mem-type ptr})$
apply(*clarsimp simp: h-t-valid-def*)
apply(*erule fold-ptr-retyp-valid-footprint-disjoint2 [OF no-overflow]*)
apply(*simp add: size-of-def*)
apply *fast*
done

lemma *ptr-retyp-disjoint-iff*:
 $\{\text{ptr-val } p..+ \text{size-of TYPE('a)}\} \cap \{\text{ptr-val } q..+ \text{size-of TYPE('b)}\} = \{\}$
 $\implies \text{ptr-retyp } (p::'a::\text{mem-type ptr}) \ d, g \models_t q = d, g \models_t (q::'b::\text{mem-type ptr})$
apply *standard*
apply (*erule (1) ptr-retyp-disjoint2*)
apply (*erule (1) ptr-retyp-disjoint*)
done

lemma (**in** *mem-type*) *ptr-force-type-valid-footprint-disjoint*:
 $\llbracket \text{valid-footprint } d \ p \ s; \{\text{p}..+ \text{size-td } s\} \cap \{\text{ptr-val } q..+ \text{size-of TYPE('a)}\} = \{\} \rrbracket$
 $\implies \text{valid-footprint } (\text{ptr-force-type } (q::'a \ \text{ptr}) \ d) \ p \ s$
apply(*clarsimp simp: valid-footprint-def Let-def*)
apply((*subst ptr-force-type-d; clarsimp*), *fastforce intro: intvlI*)
done

lemma *ptr-force-type-disjoint*:
 $\llbracket d, g \models_t (q::'b::\text{mem-type ptr}); \{\text{ptr-val } p..+ \text{size-of TYPE('a)}\} \cap \{\text{ptr-val } q..+ \text{size-of TYPE('b)}\} = \{\} \rrbracket \implies$
 $\text{ptr-force-type } (p::'a::\text{mem-type ptr}) \ d, g \models_t q$

```

apply(clarsimp simp: h-t-valid-def)
apply(erule ptr-force-type-valid-footprint-disjoint)
apply(fastforce simp: size-of-def)
done

lemma ptr-force-type-disjoint2:
   $\llbracket \text{ptr-force-type } (p::'a::\text{mem-type ptr}) \ d, g \models_t q; \text{ptr-span } p \cap \text{ptr-span } q = \{\} \rrbracket$ 
   $\implies d, g \models_t (q::'b::\text{mem-type ptr})$ 
apply(clarsimp simp: h-t-valid-def)
apply(erule ptr-force-type-valid-footprint-disjoint2)
apply(simp add: size-of-def)
apply fast
done

lemma ptr-force-type-disjoint-iff:
   $\{\text{ptr-val } p..+\text{size-of TYPE('a)}\} \cap \{\text{ptr-val } q..+\text{size-of TYPE('b)}\} = \{\}$ 
   $\implies \text{ptr-force-type } (p::'a::\text{mem-type ptr}) \ d, g \models_t q = d, g \models_t (q::'b::\text{mem-type ptr})$ 
apply standard
apply (erule (1) ptr-force-type-disjoint2)
apply (erule (1) ptr-force-type-disjoint)
done

lemma fold-ptr-force-type-valid-footprint-disjoint2:
  assumes no-overflow:  $0 \notin \{\text{ptr-val } q..+ n * \text{size-of TYPE('b)}\}$ 
  shows  $\llbracket \text{valid-footprint } (\text{fold } (\lambda i. \text{ptr-force-type } ((q::'b::\text{mem-type ptr}) +_p \text{int } i)) \ [0..<n] \ d) \ p \ s; \{\text{ptr-val } q..+ n * \text{size-of TYPE('b)}\} = \{\} \rrbracket$ 
   $\implies \text{valid-footprint } d \ p \ s$ 
apply(clarsimp simp: valid-footprint-def Let-def)
apply (drule spec, drule (1) mp)
apply(subgoal-tac p + of-nat y  $\in$  {p..+size-td s})
apply (subst (asm) fold-ptr-force-type-other)
apply clarsimp
apply fast
apply (simp add: disjoint-iff fold-ptr-force-type-other)
apply (erule intvlI)
done

lemma fold-ptr-force-type-disjoint2:
  fixes p::'a::mem-type ptr
  assumes no-overflow:  $0 \notin \{\text{ptr-val } p..+ n * \text{size-of TYPE('a)}\}$ 
shows  $\llbracket \text{fold } (\lambda i. \text{ptr-force-type } (p +_p \text{int } i)) \ [0..<n] \ d, g \models_t q; \{\text{ptr-val } p..+ n * \text{size-of TYPE('a)}\} \cap \text{ptr-span } q = \{\} \rrbracket$ 
   $\implies d, g \models_t (q::'b::\text{mem-type ptr})$ 
apply(clarsimp simp: h-t-valid-def)
apply(erule fold-ptr-force-type-valid-footprint-disjoint2 [OF no-overflow])
apply(simp add: size-of-def)

```

apply *fast*
done

lemma *fold-ptr-retyp-valid-footprint-disjoint:*

$\llbracket \text{valid-footprint } d \ p \ s; \{p..+size-td \ s\} \cap \{\text{ptr-val } q \ ..+ \ n * \text{size-of } TYPE('b)\} = \{\} \rrbracket$
 $\implies \text{valid-footprint } (\text{fold } (\lambda i. \text{ptr-retyp } ((q::'b:: \text{mem-type } \text{ptr}) +_p \text{int } i)) \ [0..<n]$
 $d) \ p \ s$
apply(*clarsimp simp: valid-footprint-def Let-def*)
apply((*subst fold-ptr-retyp-other; clarsimp*), *fastforce intro: intvlI*)
done

lemma *fold-ptr-force-type-valid-footprint-disjoint:*

$\llbracket \text{valid-footprint } d \ p \ s; \{p..+size-td \ s\} \cap \{\text{ptr-val } q \ ..+ \ n * \text{size-of } TYPE('b)\} = \{\} \rrbracket$
 $\implies \text{valid-footprint } (\text{fold } (\lambda i. \text{ptr-force-type } ((q::'b:: \text{mem-type } \text{ptr}) +_p \text{int } i))$
 $[0..<n] \ d) \ p \ s$
apply(*clarsimp simp: valid-footprint-def Let-def*)
apply((*subst fold-ptr-force-type-other; clarsimp*), *fastforce intro: intvlI*)
done

lemma *fold-ptr-retyp-disjoint:*

fixes $p::'a::\text{mem-type } \text{ptr}$
shows $\llbracket d, g \models_t (q::'b::\text{mem-type } \text{ptr}); \{\text{ptr-val } p..+ \ n * \text{size-of } TYPE('a)\} \cap$
 $\text{ptr-span } q = \{\} \rrbracket \implies$
 $\text{fold } (\lambda i. \text{ptr-retyp } (p +_p \text{int } i)) \ [0..<n] \ d, g \models_t q$
apply(*clarsimp simp: h-t-valid-def*)
apply(*erule fold-ptr-retyp-valid-footprint-disjoint*)
apply(*fastforce simp: size-of-def*)
done

lemma *fold-ptr-force-type-disjoint:*

fixes $p::'a::\text{mem-type } \text{ptr}$
shows $\llbracket d, g \models_t (q::'b::\text{mem-type } \text{ptr}); \{\text{ptr-val } p..+ \ n * \text{size-of } TYPE('a)\} \cap$
 $\text{ptr-span } q = \{\} \rrbracket \implies$
 $\text{fold } (\lambda i. \text{ptr-force-type } (p +_p \text{int } i)) \ [0..<n] \ d, g \models_t q$
apply(*clarsimp simp: h-t-valid-def*)
apply(*erule fold-ptr-force-type-valid-footprint-disjoint*)
apply(*fastforce simp: size-of-def*)
done

lemma *fold-ptr-retyp-disjoint-iff:*

fixes $p::'a::\text{mem-type } \text{ptr}$
assumes *no-overflow*: $0 \notin \{\text{ptr-val } p..+ \ n * \text{size-of } TYPE('a)\}$
shows $\{\text{ptr-val } p..+ \ n * \text{size-of } TYPE('a)\} \cap \text{ptr-span } q = \{\}$
 $\implies \text{fold } (\lambda i. \text{ptr-retyp } (p +_p \text{int } i)) \ [0..<n] \ d, g \models_t q = d, g \models_t (q::'b::\text{mem-type}$
 $\text{ptr})$
apply *standard*
apply (*erule (1) fold-ptr-retyp-disjoint2 [OF no-overflow]*)

apply (erule (1) fold-*ptr-retyp-disjoint*)
done

lemma *fold-*ptr-force-type-disjoint-iff**:

fixes $p::'a::\text{mem-type ptr}$
assumes *no-overflow*: $0 \notin \{\text{ptr-val } p..+ n * \text{size-of TYPE('a)}\}$
shows $\{\text{ptr-val } p..+ n * \text{size-of TYPE('a)}\} \cap \text{ptr-span } q = \{\}$
 $\implies \text{fold } (\lambda i. \text{ptr-force-type } (p +_p \text{int } i)) [0..<n] d, g \models_t q = d, g \models_t (q::'b::\text{mem-type ptr})$
apply *standard*
apply (erule (1) *fold-*ptr-force-type-disjoint2** [OF *no-overflow*])
apply (erule (1) *fold-*ptr-force-type-disjoint**)
done

lemma *stack-alloc-preserves-typing*:

fixes $q::'b::\text{mem-type ptr}$
assumes *stack-alloc*: $(p, d') \in \text{stack-alloc } \mathcal{S} (\text{TYPE('a::mem-type)}) d$
assumes *no-stack*: $\text{typ-uinfo-t } (\text{TYPE('b)}) \neq \text{typ-uinfo-t } (\text{TYPE}(\text{stack-byte}))$
assumes *typed*: $d \models_t q$
shows $d' \models_t q$
proof –
from *stack-alloc obtain*
 $d': d' = \text{ptr-force-type } p d$
by (cases rule: *stack-alloc-cases*) *auto*

from *stack-alloc-disjoint* [OF *stack-alloc no-stack typed*]
have $\text{ptr-span } p \cap \text{ptr-span } q = \{\}$.
from *ptr-force-type-disjoint-iff* [OF *this, where d=d and g=c-guard*] *typed*
show *?thesis*
by (*simp add: d'*)
qed

lemma *stack-allocs-preserves-typing*:

fixes $q::'b::\text{mem-type ptr}$
assumes *stack-alloc*: $(p, d') \in \text{stack-allocs } n \mathcal{S} (\text{TYPE('a::mem-type)}) d$
assumes *no-stack*: $\text{typ-uinfo-t } (\text{TYPE('b)}) \neq \text{typ-uinfo-t } (\text{TYPE}(\text{stack-byte}))$
assumes *typed*: $d \models_t q$
shows $d' \models_t q$
proof –
from *stack-alloc obtain*
 $d': d' = \text{fold } (\lambda i. \text{ptr-force-type } (p +_p \text{int } i)) [0..<n] d$ **and**
no-overflow: $0 \notin \{\text{ptr-val } p ..+ n * \text{size-of TYPE('a)}\}$
by (cases rule: *stack-allocs-cases*) *auto*

from *stack-allocs-disjoint* [OF *stack-alloc no-stack typed*]
have $\{\text{ptr-val } p..+ n * \text{size-of TYPE('a)}\} \cap \text{ptr-span } q = \{\}$.
from *fold-*ptr-force-type-disjoint-iff** [OF *no-overflow this, where d=d and g=c-guard*]
typed show ?thesis
by (*simp add: d'*)

qed

lemma *h-t-valid-valid-footprint*: $d, g \models_t p \implies \text{valid-footprint } d \text{ (ptr-val (p::'a::c-type ptr)) (typ-uinfo-t TYPE('a))}$
by (*simp add: h-t-valid-def*)

lemma *stack-alloc-preserves-root-ptr-valid*:

fixes $q::'b::\text{mem-type ptr}$

assumes *stack-alloc*: $(p, d') \in \text{stack-alloc } \mathcal{S} \text{ (TYPE('a::mem-type)) } d$

assumes *no-stack*: $\text{typ-uinfo-t (TYPE('b))} \neq \text{typ-uinfo-t (TYPE(stack-byte))}$

assumes *typed*: *root-ptr-valid* $d \ q$

shows *root-ptr-valid* $d' \ q$

proof –

from *stack-alloc* **obtain**

d' : $d' = \text{ptr-force-type } p \ d$

by (*cases rule: stack-alloc-cases*) *auto*

from *typed* **have** *typed-q*: $d \models_t q$

by (*simp add: root-ptr-valid-h-t-valid*)

from *stack-alloc-disjoint* [*OF stack-alloc no-stack this*]

have *disj*: $\text{ptr-span } p \cap \text{ptr-span } q = \{\}$.

from *stack-alloc-preserves-typing* [*OF stack-alloc no-stack typed-q*]

have *typed'*: $d' \models_t q$.

hence *valid-fp*: *valid-footprint* $d' \text{ (ptr-val } q \text{) (typ-uinfo-t TYPE('b))}$

by (*simp add: h-t-valid-valid-footprint*)

show *?thesis*

apply (*simp add: root-ptr-valid-def valid-root-footprint-valid-footprint-dom-conv valid-fp*)

h-t-valid-c-guard [*OF typed'*]

apply (*simp add: d'*)

by (*metis (no-types, lifting) disj disjoint-iff intvlI ptr-force-type-d root-ptr-valid-def*

size-of-def typ-uinfo-size typed valid-root-footprint-dom-typing)

qed

lemma *stack-allocs-preserves-root-ptr-valid*:

fixes $q::'b::\text{mem-type ptr}$

assumes *stack-alloc*: $(p, d') \in \text{stack-allocs } n \ \mathcal{S} \text{ (TYPE('a::mem-type)) } d$

assumes *no-stack*: $\text{typ-uinfo-t (TYPE('b))} \neq \text{typ-uinfo-t (TYPE(stack-byte))}$

assumes *typed*: *root-ptr-valid* $d \ q$

shows *root-ptr-valid* $d' \ q$

proof –

from *stack-alloc* **obtain**

d' : $d' = \text{fold } (\lambda i. \text{ptr-force-type } (p +_p \text{ int } i)) \ [0..<n] \ d$ **and**

no-overflow: $0 \notin \{\text{ptr-val } p \ .. + n * \text{size-of TYPE('a)}\}$

by (*cases rule: stack-allocs-cases*) *auto*

from *typed* **have** *typed-q*: $d \models_t q$
by (*simp add: root-ptr-valid-h-t-valid*)

from *stack-allocs-disjoint* [*OF stack-alloc no-stack this*]
have *disj*: $\{ptr\text{-val } p..+n * \text{size-of } TYPE('a)\} \cap ptr\text{-span } q = \{\}$.
from *stack-allocs-preserves-typing* [*OF stack-alloc no-stack typed-q*]
have *typed'*: $d' \models_t q$.
hence *valid-fp*: *valid-footprint* d' (*ptr-val* q) (*typ-uinfo-t* $TYPE('b)$)
by (*simp add: h-t-valid-valid-footprint*)

show *?thesis*
apply (*simp add: root-ptr-valid-def valid-root-footprint-valid-footprint-dom-conv*
valid-fp
h-t-valid-c-guard [*OF typed'*])
apply (*simp add: d'*)
using *disj fold-ptr-force-type-other*
by (*smt (verit) d' disjoint-iff dom-list-map root-ptr-valid-def s-footprintD*
s-footprintI2 size-of-def stack-alloc stack-allocs-cases typ-uinfo-size typed
valid-root-footprint-def)
qed

lemma *stack-alloc-root-ptr-valid-new-cases*:
fixes $q::'b::mem\text{-type } ptr$
assumes *stack-alloc*: $(p, d') \in \text{stack-alloc } \mathcal{S} (TYPE('a::mem\text{-type})) d$
assumes *root-ptr-valid* $d' q$
shows $(ptr\text{-val } p = ptr\text{-val } q \wedge typ\text{-uinfo-t } (TYPE('b)) = typ\text{-uinfo-t } (TYPE('a)))$
 \vee *root-ptr-valid* $d q$
by (*metis Ptr-ptr-val assms(2) ptr.inject root-ptr-valid-def stack-alloc stack-alloc-cases*

valid-root-footprint-ptr-force-type-iff wf-size-desc-typ-uinfo-t-simp)

lemma *valid-root-footprints-no-overlap*:
assumes *valid-root-footprint* $d a1 t1$
assumes *valid-root-footprint* $d a2 t2$
assumes $t1 \neq t2$
shows $\{a1 ..+ \text{size-td } t1\} \cap \{a2 ..+ \text{size-td } t2\} = \{\}$
using *assms(1) assms(2) assms(3) valid-root-footprint-type-neq-disjoint* **by** *pres-*
burger

lemma *root-ptr-valid-neq-disjoint*:
 \llbracket *root-ptr-valid* $d (p::'a::c\text{-type } ptr)$;
root-ptr-valid $d (q::'b::c\text{-type } ptr)$;
 $ptr\text{-val } p \neq ptr\text{-val } q \rrbracket \implies$
 $\{ptr\text{-val } p..+ \text{size-of } TYPE('a)\} \cap$
 $\{ptr\text{-val } q..+ \text{size-of } TYPE('b)\} = \{\}$
apply (*clarsimp simp only: size-of-tag [symmetric]*)
by (*metis boolean-algebra.conj-ac(2) h-t-valid-valid-footprint intvl-inter*
order-antisym-conv root-ptr-valid-def root-ptr-valid-h-t-valid
valid-footprint-neq-nmem valid-root-footprint-overlap-sub-typ)

lemma *root-ptr-valid-same-type-neq-disjoint*:
 $root\text{-}ptr\text{-}valid\ d\ p \implies root\text{-}ptr\text{-}valid\ d\ q \implies p \neq q = (ptr\text{-}span\ p \cap ptr\text{-}span\ q = \{\})$
apply *standard*
apply (*simp add: root-ptr-valid-neq-disjoint*)
by (*metis empty-iff inf.idem intvlI root-ptr-valid-def size-of-tag valid-root-footprint-def*)

lemma *cvalid-same-type-neq-disjoint*:
 $d \models_t p \implies d \models_t q \implies p \neq q = (ptr\text{-}span\ p \cap ptr\text{-}span\ q = \{\})$
apply *standard*
apply (*simp add: h-t-valid-neq-disjoint peer-typ-not-field-of*)
by (*metis disjoint-iff h-t-valid-valid-footprint intvl-self size-of-tag valid-footprint-def*)

lemma *root-ptr-valid-type-neq-disjoint*:
 $\llbracket root\text{-}ptr\text{-}valid\ d\ (p::'a::c\text{-}type\ ptr);$
 $root\text{-}ptr\text{-}valid\ d\ (q::'b::c\text{-}type\ ptr);$
 $typ\text{-}uinfo\text{-}t\ TYPE('a) \neq typ\text{-}uinfo\text{-}t\ TYPE('b) \rrbracket \implies$
 $\{ptr\text{-}val\ p..+size\text{-}of\ TYPE('a)\} \cap$
 $\{ptr\text{-}val\ q..+size\text{-}of\ TYPE('b)\} = \{\}$
apply (*subgoal-tac ptr-val p \neq ptr-val q*)
apply (*rule root-ptr-valid-neq-disjoint, auto*)[1]
by (*metis disjoint-iff intvlI root-ptr-valid-def sub-tag-antisym*
valid-footprint-def valid-root-footprint-overlap-sub-typ
valid-root-footprint-valid-footprint-dom-conv)

lemma *valid-root-footprints-cases*:
assumes *valid-root-footprint d a1 t1*
assumes *valid-root-footprint d a2 t2*
shows $(t1 = t2 \wedge a1 = a2) \vee (\{a1 ..+ size\text{-}td\ t1\} \cap \{a2 ..+ size\text{-}td\ t2\} = \{\})$
using *assms valid-root-footprint-neq-disjoint valid-root-footprint-type-neq-disjoint*
by *blast*

lemma *root-ptr-valid-cases*:
fixes $p::'a::mem\text{-}type\ ptr$
fixes $q::'b::mem\text{-}type\ ptr$
assumes *root-p: root-ptr-valid d p*
assumes *root-q: root-ptr-valid d q*
shows $(ptr\text{-}val\ p = ptr\text{-}val\ q \wedge typ\text{-}uinfo\text{-}t\ (TYPE('a)) = typ\text{-}uinfo\text{-}t\ (TYPE('b)))$
 \vee
 $(ptr\text{-}span\ p \cap ptr\text{-}span\ q) = \{\}$
using *root-p root-ptr-valid-neq-disjoint root-ptr-valid-type-neq-disjoint root-q* **by**
blast

lemma *root-ptr-valid-casesE* [*consumes 2*]:
fixes $p::'a::mem\text{-}type\ ptr$
fixes $q::'b::mem\text{-}type\ ptr$
assumes *root-p: root-ptr-valid d p*

assumes *root-q*: *root-ptr-valid* *d* *q*
assumes *same*: (*ptr-val* *q* = *ptr-val* *p* \wedge *typ-uinfo-t* (*TYPE*('a')) = *typ-uinfo-t* (*TYPE*('b'))) \implies *P*
assumes *disj*: *ptr-span* *p* \cap *ptr-span* *q* = {} \implies *P*
shows *P*
using *root-ptr-valid-cases* [*OF* *root-p* *root-q*] *same disj* **by** *auto*

lemma *stack-allocs-root-ptr-valid-new-cases*:

fixes *q*::'b::mem-type *ptr*
assumes *stack-alloc*: (*p*, *d'*) \in *stack-allocs* *n* \mathcal{S} (*TYPE*('a::mem-type')) *d*
assumes *root-ptr-valid* *d'* *q*
shows ($\exists i < n$. *ptr-val* *q* = *ptr-val* (*p* +_{*p*} *int* *i*) \wedge *typ-uinfo-t* (*TYPE*('b')) = *typ-uinfo-t* (*TYPE*('a'))) \vee *root-ptr-valid* *d* *q*
proof (*cases* {*ptr-val* *p* ..+ *n* * *size-of* *TYPE*('a')} \cap *ptr-span* *q* = {})
case *True*
with *stack-alloc* **show** ?*thesis*
by (*smt* (*verit*, *best*) *assms*(2) *disjoint-iff* *fold-ptr-force-type-other* *root-ptr-valid-domain'* *stack-allocs-cases*)
next
case *False*
with *stack-alloc* *array-to-index-span* **show** ?*thesis*
by (*smt* (*verit*) *assms*(2) *disjoint-iff* *root-ptr-valid-neq-disjoint* *root-ptr-valid-type-neq-disjoint* *stack-allocs-cases*)
qed

lemma *stack-alloc-root-ptr-valid-same*:

fixes *q*::'b::mem-type *ptr*
assumes *stack-alloc*: (*p*, *d'*) \in *stack-alloc* \mathcal{S} (*TYPE*('a::mem-type')) *d*
assumes *addr-eq*: *ptr-val* *p* = *ptr-val* *q*
assumes *match*: *typ-uinfo-t* (*TYPE*('b')) = *typ-uinfo-t* (*TYPE*('a'))
shows *root-ptr-valid* *d'* *q*
proof (*cases* *root-ptr-valid* *d'* *q*)
case *True*
then **show** ?*thesis* **by** *simp*
next
case *False*
from *stack-alloc* **have** *root-ptr-valid* *d'* *p* **by** (*rule* *stack-alloc-cases*)
with *addr-eq* *match* **show** ?*thesis*
apply (*clarsimp* *simp* *add*: *root-ptr-valid-def*)
apply (*simp* *add*: *c-guard-def* *c-null-guard-def* *ptr-aligned-def*)
apply (*clarsimp* *simp* *add*: *align-of-def* *align-td-uinfo*[*symmetric*] *size-of-def*)
by (*metis* *typ-uinfo-size*)
qed

lemma *stack-allocs-root-ptr-valid-same*:

fixes *q*::'b::mem-type *ptr*
assumes *stack-alloc*: (*p*, *d'*) \in *stack-allocs* *n* \mathcal{S} (*TYPE*('a::mem-type')) *d*
assumes *i*: *i* < *n*
assumes *addr-eq*: *ptr-val* *q* = *ptr-val* (*p* +_{*p*} *int* *i*)

```

    assumes match: typ-uinto-t (TYPE('b)) = typ-uinto-t (TYPE('a))
    shows root-ptr-valid d' q
  proof (cases root-ptr-valid d' q)
    case True
    then show ?thesis by simp
  next
    case False
    from stack-alloc have root-ptr-valid d' (p +p int i)
    apply (rule stack-allocs-cases)
    using i
    by auto

  with addr-eq match show ?thesis
  apply (clarsimp simp add: root-ptr-valid-def )
  apply (simp add: c-guard-def c-null-guard-def ptr-aligned-def)
  apply (clarsimp simp add: align-of-def align-td-uinto[symmetric] size-of-def )
  by (metis typ-uinto-size)
qed

```

```

lemma stack-alloc-root-ptr-valid-other:
  fixes q::'b::mem-type ptr
  assumes stack-alloc: (p, d') ∈ stack-alloc S (TYPE('a::mem-type)) d
  assumes valid-d: root-ptr-valid d q
  assumes non-stack: typ-uinto-t (TYPE('b)) ≠ typ-uinto-t (TYPE(stack-byte))
  shows root-ptr-valid d' q
  proof (cases root-ptr-valid d' q)
    case True
    then show ?thesis by simp
  next
    case False
    from stack-alloc
    show ?thesis
    apply (rule stack-alloc-cases)
    using False valid-d non-stack
    using stack-alloc stack-alloc-preserves-root-ptr-valid by blast
  qed

```

```

lemma stack-allocs-root-ptr-valid-other:
  fixes q::'b::mem-type ptr
  assumes stack-alloc: (p, d') ∈ stack-allocs n S (TYPE('a::mem-type)) d
  assumes valid-d: root-ptr-valid d q
  assumes non-stack: typ-uinto-t (TYPE('b)) ≠ typ-uinto-t (TYPE(stack-byte))
  shows root-ptr-valid d' q
  proof (cases root-ptr-valid d' q)
    case True
    then show ?thesis by simp
  next
    case False
    from stack-alloc

```

show *?thesis*
apply (*rule stack-allocs-cases*)
using *False valid-d non-stack*
using *stack-alloc stack-allocs-preserves-root-ptr-valid* **by** *blast*
qed

lemma *stack-alloc-root-ptr-valid-cases:*

fixes $q::'b::\text{mem-type ptr}$
assumes *stack-alloc:* $(p, d') \in \text{stack-alloc } \mathcal{S} (\text{TYPE}('a::\text{mem-type})) d$
assumes *non-stack-byte:* $\text{typ-uinto-t } (\text{TYPE}('b)) \neq \text{typ-uinto-t } (\text{TYPE}(\text{stack-byte}))$
shows *root-ptr-valid* $d' q \longleftrightarrow$
 $(\text{ptr-val } p = \text{ptr-val } q \wedge \text{typ-uinto-t } (\text{TYPE}('b)) = \text{typ-uinto-t } (\text{TYPE}('a))) \vee$
 $\text{root-ptr-valid } d q$

using *stack-alloc non-stack-byte*
stack-alloc-root-ptr-valid-new-cases stack-alloc-root-ptr-valid-other stack-alloc-root-ptr-valid-same
by *blast*

lemma *stack-allocs-root-ptr-valid-cases:*

fixes $q::'b::\text{mem-type ptr}$
assumes *stack-alloc:* $(p, d') \in \text{stack-allocs } n \mathcal{S} (\text{TYPE}('a::\text{mem-type})) d$
assumes *non-stack-byte:* $\text{typ-uinto-t } (\text{TYPE}('b)) \neq \text{typ-uinto-t } (\text{TYPE}(\text{stack-byte}))$
shows *root-ptr-valid* $d' q \longleftrightarrow$
 $(\exists i < n. \text{ptr-val } q = \text{ptr-val } (p +_p \text{int } i) \wedge \text{typ-uinto-t } (\text{TYPE}('b)) = \text{typ-uinto-t}$
 $(\text{TYPE}('a))) \vee$
 $\text{root-ptr-valid } d q$

using *stack-alloc non-stack-byte*
stack-allocs-root-ptr-valid-new-cases stack-allocs-root-ptr-valid-other stack-allocs-root-ptr-valid-same
by *metis*

lemma *stack-alloc-root-ptr-valid-same-type-cases:*

assumes *stack-alloc:* $(p, d') \in \text{stack-alloc } \mathcal{S} (\text{TYPE}('a::\text{mem-type})) d$
shows *root-ptr-valid* $d' q \longleftrightarrow p = q \vee \text{root-ptr-valid } d q$
by (*metis ptr-val-inj stack-alloc stack-alloc-cases stack-alloc-root-ptr-valid-cases*)

lemma *stack-allocs-root-ptr-valid-same-type-cases:*

assumes *stack-alloc:* $(p, d') \in \text{stack-allocs } n \mathcal{S} (\text{TYPE}('a::\text{mem-type})) d$
shows *root-ptr-valid* $d' q \longleftrightarrow (\exists i < n. q = (p +_p \text{int } i) \vee \text{root-ptr-valid } d q)$
using *stack-alloc stack-allocs-cases stack-allocs-root-ptr-valid-cases*
by (*metis ptr-val-inj*)

lemma *root-ptr-valid-valid-root-footprint:*

root-ptr-valid $d (p::'a \text{ ptr}) \implies \text{valid-root-footprint } d (\text{ptr-val } p) (\text{typ-uinto-t } \text{TYPE}('a::\text{c-type}))$
by (*simp add: root-ptr-valid-def*)

definition

$allocated\text{-}ptr :: addr\ set \Rightarrow 'a :: mem\text{-}type\ itself \Rightarrow heap\text{-}typ\text{-}desc \Rightarrow heap\text{-}typ\text{-}desc$
 $\Rightarrow 'a\ ptr\ \mathbf{where}$
 $\langle allocated\text{-}ptr\ \mathcal{S}\ T\ d\ d' = (THE\ p.\ (p,\ d') \in stack\text{-}alloc\ \mathcal{S}\ TYPE('a)\ d) \rangle$

definition

$allocated\text{-}ptrs :: nat \Rightarrow addr\ set \Rightarrow 'a :: mem\text{-}type\ itself \Rightarrow heap\text{-}typ\text{-}desc \Rightarrow heap\text{-}typ\text{-}desc$
 $\Rightarrow 'a\ ptr\ \mathbf{where}$
 $\langle allocated\text{-}ptrs\ n\ \mathcal{S}\ T\ d\ d' = (THE\ p.\ (p,\ d') \in stack\text{-}allocs\ n\ \mathcal{S}\ TYPE('a)\ d) \rangle$

lemma $allocated\text{-}ptr\text{-}allocated\text{-}ptrs\text{-}def$: $allocated\text{-}ptr = allocated\text{-}ptrs\ 1$

by ($simp\ add$: $stack\text{-}alloc\text{-}stack\text{-}allocs\text{-}conv\ allocated\text{-}ptr\text{-}def\ allocated\text{-}ptrs\text{-}def\ fun\text{-}eq\text{-}iff$)

abbreviation ($input$)

$cptr\text{-}type :: ('a :: c\text{-}type)\ ptr \Rightarrow 'a\ itself$

where

$cptr\text{-}type\ p \equiv TYPE('a)$

lemma $h\text{-}t\text{-}valid\text{-}guard\text{-}subst$:

$\llbracket d, g \models_t p; g' p \rrbracket \Longrightarrow d, g' \models_t p$

apply ($simp\ add$: $h\text{-}t\text{-}valid\text{-}def$)

done

lemma $h\text{-}t\text{-}valid\text{-}ptr\text{-}retype\text{-}eq$:

$\neg\ cptr\text{-}type\ p <_{\tau}\ cptr\text{-}type\ p' \Longrightarrow h\text{-}t\text{-}valid\ (ptr\text{-}retype\ p\ td)\ g\ p'$

$= (if\ ptr\text{-}span\ (p :: 'a :: mem\text{-}type\ ptr) \cap ptr\text{-}span\ (p' :: 'b :: mem\text{-}type\ ptr) = \{\}\ then$
 $h\text{-}t\text{-}valid\ td\ g\ p'$

$else\ field\text{-}of\text{-}t\ p' p \wedge g\ p')$

apply ($clarsimp\ simp$: $ptr\text{-}retype\text{-}disjoint\text{-}iff\ split$: $if\text{-}split$)

apply ($cases\ g\ p'$)

apply ($rule\ iffI$)

apply ($rule\ ccontr$, $drule\ h\text{-}t\text{-}valid\text{-}neq\text{-}disjoint$, $rule\ ptr\text{-}retype\text{-}h\text{-}t\text{-}valid$, $simp+$)

apply ($simp\ add$: $Int\text{-}commute$)

apply ($clarsimp\ simp$: $field\text{-}of\text{-}t\text{-}def\ field\text{-}of\text{-}def$)

apply ($drule\ sub\text{-}h\text{-}t\text{-}valid[\mathbf{where}\ p=p, rotated]$, $rule\ ptr\text{-}retype\text{-}h\text{-}t\text{-}valid$, $simp$,
 $simp$)

apply ($erule(1)\ h\text{-}t\text{-}valid\text{-}guard\text{-}subst$)

apply ($simp\ add$: $h\text{-}t\text{-}valid\text{-}def$)

done

lemma $field\text{-}of\text{-}t\text{-}refl$:

$field\text{-}of\text{-}t\ p\ p' = (p = p')$

apply ($safe$, $simp\text{-}all\ add$: $field\text{-}of\text{-}t\text{-}def$)

apply ($simp\ add$: $field\text{-}of\text{-}def$)

apply ($drule\ td\text{-}set\text{-}size\text{-}lte$)

apply ($simp\ add$: $unat\text{-}eq\text{-}0$)

done

lemma $ptr\text{-}retype\text{-}same\text{-}cleared\text{-}region$:

fixes $p :: 'a :: mem\text{-}type\ ptr$ **and** $p' :: 'a :: mem\text{-}type\ ptr$

assumes $ht: ptr\text{-}retype\ p\ td, g \models_t p'$
shows $p = p' \vee \{ptr\text{-}val\ p..+ size\text{-}of\ TYPE('a)\} \cap \{ptr\text{-}val\ p'..+ size\text{-}of\ TYPE('a)\}$
 $= \{\}$
using ht
by ($simp\ add: h\text{-}t\text{-}valid\ ptr\text{-}retype\ eq[\mathbf{where}\ p=p\ \mathbf{and}\ p'=p']\ field\text{-}of\text{-}t\text{-}refl$
 $split: if\text{-}split\text{-}asm$)

lemma (in $mem\text{-}type$) $ptr\text{-}force\text{-}type\text{-}h\text{-}t\text{-}valid$:
 $g\ p \implies ptr\text{-}force\text{-}type\ p\ d, g \models_t (p::'a\ ptr)$
by ($simp\ add: h\text{-}t\text{-}valid\ def\ ptr\text{-}force\text{-}type\ valid\ footprint$)

lemma $h\text{-}t\text{-}valid\ ptr\text{-}force\text{-}type\ eq$:
 $\neg\ cptr\text{-}type\ p <_{\tau}\ cptr\text{-}type\ p' \implies h\text{-}t\text{-}valid\ (ptr\text{-}force\text{-}type\ p\ td)\ g\ p'$
 $= (if\ ptr\text{-}span\ (p::'a::mem\text{-}type\ ptr) \cap ptr\text{-}span\ (p'::'b::mem\text{-}type\ ptr) = \{\}\ then$
 $h\text{-}t\text{-}valid\ td\ g\ p'$
 $\quad else\ field\text{-}of\text{-}t\ p'\ p \wedge g\ p')$
apply ($clarsimp\ simp: ptr\text{-}force\text{-}type\ disjoint\text{-}iff\ split: if\text{-}split$)
apply ($cases\ g\ p'$)
apply ($rule\ iffI$)
apply ($rule\ ccontr, drule\ h\text{-}t\text{-}valid\ neg\ disjoint, rule\ ptr\text{-}force\text{-}type\ h\text{-}t\text{-}valid,$
 $simp+$)
apply ($simp\ add: Int\text{-}commute$)
apply ($clarsimp\ simp: field\text{-}of\text{-}t\ def\ field\text{-}of\text{-}def$)
apply ($drule\ sub\text{-}h\text{-}t\text{-}valid[\mathbf{where}\ p=p, rotated], rule\ ptr\text{-}force\text{-}type\ h\text{-}t\text{-}valid,$
 $simp, simp$)
apply ($erule(1)\ h\text{-}t\text{-}valid\ guard\ subst$)
apply ($simp\ add: h\text{-}t\text{-}valid\ def$)
done

lemma $ptr\text{-}force\text{-}type\ same\ cleared\ region$:
fixes $p :: 'a :: mem\text{-}type\ ptr\ \mathbf{and}\ p' :: 'a :: mem\text{-}type\ ptr$
assumes $ht: ptr\text{-}force\text{-}type\ p\ td, g \models_t p'$
shows $p = p' \vee \{ptr\text{-}val\ p..+ size\text{-}of\ TYPE('a)\} \cap \{ptr\text{-}val\ p'..+ size\text{-}of\ TYPE('a)\}$
 $= \{\}$
using ht
by ($simp\ add: h\text{-}t\text{-}valid\ ptr\text{-}force\text{-}type\ eq[\mathbf{where}\ p=p\ \mathbf{and}\ p'=p']\ field\text{-}of\text{-}t\text{-}refl$
 $split: if\text{-}split\text{-}asm$)

lemma $stack\text{-}alloc\ unique$:
assumes $p: (p, d') \in stack\text{-}alloc\ \mathcal{S}\ (TYPE('a::mem\text{-}type))\ d$
assumes $q: (q, d') \in stack\text{-}alloc\ \mathcal{S}\ (TYPE('a::mem\text{-}type))\ d$
shows $p = q$
proof –
from p **obtain** $p\text{-}props$:
 $typ\text{-}uinfo\text{-}t\ (TYPE('a)) \neq typ\text{-}uinfo\text{-}t\ (TYPE(stack\text{-}byte))$
 $\forall a \in ptr\text{-}span\ p. root\text{-}ptr\text{-}valid\ d\ (PTR\ (stack\text{-}byte)\ a)$
 $ptr\text{-}aligned\ p$
 $c\text{-}guard\ p$
 $root\text{-}ptr\text{-}valid\ d'\ p$

$d' = \text{ptr-force-type } p \ d$
by (cases rule: stack-alloc-cases) auto

from q **obtain** q -props:
 $\text{typ-uinto-t } (\text{TYPE}('a)) \neq \text{typ-uinto-t } (\text{TYPE}(\text{stack-byte}))$
 $\forall a \in \text{ptr-span } q. \text{root-ptr-valid } d \ (\text{PTR } (\text{stack-byte}) \ a)$
 $\text{ptr-aligned } q$
 $c\text{-guard } q$
 $\text{root-ptr-valid } d' \ q$ **and**
 $d' = \text{ptr-force-type } q \ d$
by (cases rule: stack-alloc-cases) auto

show ?thesis
proof (cases ptr-val $p = \text{ptr-val } q$)
 case True
 then show ?thesis **by** simp
next
 case False
 with p -props q -props **show** ?thesis
 by (metis disj-ptr-span-ptr-neq[unfolded disjnt-def] ptr-force-type-disjoint2
 ptr-force-type-same-cleared-region q root-ptr-valid-h-t-valid stack-alloc-disjoint)
qed
qed

lemma stack-allocs-unique:
assumes $p: (p, d') \in \text{stack-allocs } n \ \mathcal{S} \ (\text{TYPE}('a::\text{mem-type})) \ d$
assumes $q: (q, d') \in \text{stack-allocs } n \ \mathcal{S} \ (\text{TYPE}('a::\text{mem-type})) \ d$
shows $p = q$
proof –
from p **obtain** p -props:
 $0 \notin \{\text{ptr-val } p \ ..+ \ n * \text{size-of } \text{TYPE}('a)\}$
 $\text{typ-uinto-t } (\text{TYPE}('a)) \neq \text{typ-uinto-t } (\text{TYPE}(\text{stack-byte}))$
 $\forall a \in \{\text{ptr-val } p \ ..+ \ n * \text{size-of } \text{TYPE}('a)\}. \text{root-ptr-valid } d \ (\text{PTR } (\text{stack-byte}) \ a)$
 $\text{ptr-aligned } p \ c\text{-guard } p \ \text{root-ptr-valid } d' \ p$
 $d' = \text{fold } (\lambda i. \text{ptr-force-type } (p \ +_p \ \text{int } i)) \ [0..<n] \ d$
by (cases rule: stack-allocs-cases) auto

from q **obtain** q -props:
 $0 < n$
 $0 \notin \{\text{ptr-val } q \ ..+ \ n * \text{size-of } \text{TYPE}('a)\}$
 $\text{typ-uinto-t } (\text{TYPE}('a)) \neq \text{typ-uinto-t } (\text{TYPE}(\text{stack-byte}))$
 $\forall a \in \{\text{ptr-val } q \ ..+ \ n * \text{size-of } \text{TYPE}('a)\}. \text{root-ptr-valid } d \ (\text{PTR } (\text{stack-byte}) \ a)$
 $\text{ptr-aligned } q \ c\text{-guard } q \ \text{root-ptr-valid } d' \ q$
 $d' = \text{fold } (\lambda i. \text{ptr-force-type } (q \ +_p \ \text{int } i)) \ [0..<n] \ d$
by (cases rule: stack-allocs-cases) auto

```

show ?thesis
proof (cases ptr-val p = ptr-val q)
  case True
    then show ?thesis by simp
  next
    case False
      with p-props q-props show ?thesis
      using disj-ptr-span-ptr-neq fold-ptr-retyp-disjoint2 ptr-force-type-same-cleared-region

        q p root-ptr-valid-h-t-valid stack-allocs-disjoint
      by (smt (verit, ccfv-threshold) disjoint-iff fold-ptr-force-type-disjoint2 inf.idem

        inf.order-iff intvl-inter intvl-no-overflow-nat len-of-addr-card mem-type-self

        order-antisym-conv root-ptr-valid-cases stack-allocs-contained
        stack-allocs-ptr-force-type zero-not-in-intvl-no-overflow)
    qed
qed

lemma stack-alloc-allocated-ptr:
  (p, d') ∈ stack-alloc S TYPE('a) d ⇒ allocated-ptr S TYPE('a::mem-type) d
  d' = p
  apply (simp add: allocated-ptr-def)
  apply (rule theI2)
  apply (assumption)
  apply (erule (1) stack-alloc-unique)
  apply (erule (1) stack-alloc-unique)
  done

lemma stack-allocs-allocated-ptrs:
  (p, d') ∈ stack-allocs n S TYPE('a) d ⇒ allocated-ptrs n S TYPE('a::mem-type)
  d d' = p
  apply (simp add: allocated-ptrs-def)
  apply (rule theI2)
  apply (assumption)
  apply (erule (1) stack-allocs-unique)
  apply (erule (1) stack-allocs-unique)
  done

lemma null-not-stack-free: 0 ∉ stack-free d
  by (simp add: root-ptr-valid-def stack-free-def)

lemma stack-alloc-stack-subset-stack-free:
  (p, d') ∈ stack-alloc S TYPE('a::mem-type) d ⇒
  ptr-span p ⊆ stack-free d
  by (metis mem-Collect-eq stack-alloc-cases stack-free-def subsetI)

```

lemma *stack-allocs-stack-subset-stack-free'*:
 $(p, d') \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a::\text{mem-type}) \ d \implies i < n \implies$
 $\text{ptr-span } (p +_p \text{int } i) \subseteq \text{stack-free } d$
using *stack-allocs-cases*
by (*smt* (*verit*, *best*) *mem-Collect-eq stack-allocs-contained stack-free-def subsetD subsetI*)

lemma *stack-allocs-stack-subset-stack-free*:
 $(p, d') \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a::\text{mem-type}) \ d \implies$
 $\{\text{ptr-val } p \ ..+ \ n * \text{size-of } \text{TYPE}('a)\} \subseteq \text{stack-free } d$
by (*metis mem-Collect-eq stack-allocs-cases stack-free-def subsetI*)

lemma *stack-alloc-stack-free-mono*:
assumes *sub*: $\text{stack-free } d1 \subseteq \text{stack-free } d2$
assumes *alloc-d1*: $(p, d1') \in \text{stack-alloc } \mathcal{S} \ \text{TYPE}('a::\text{mem-type}) \ d1$
shows $\exists d2'. (p, d2') \in \text{stack-alloc } \mathcal{S} \ \text{TYPE}('a) \ d2$
by (*smt* (*verit*, *del-insts*) *Collect-mono-iff alloc-d1 case-prodI mem-Collect-eq stack-alloc-cases stack-alloc-def stack-free-def sub*)

lemma *stack-allocs-stack-free-mono*:
assumes *sub*: $\text{stack-free } d1 \subseteq \text{stack-free } d2$
assumes *alloc-d1*: $(p, d1') \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a::\text{mem-type}) \ d1$
shows $\exists d2'. (p, d2') \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a) \ d2$
using *alloc-d1*
apply (*rule stack-allocs-cases*)
using *sub stack-allocs-stack-subset-stack-free'* [*OF alloc-d1*]
apply (*auto simp add: stack-allocs-def stack-free-def*)
done

lemma *stack-alloc-stack-free-eq*:
assumes *sub*: $\text{stack-free } d1 = \text{stack-free } d2$
assumes *alloc-d1*: $(p, d1') \in \text{stack-alloc } \mathcal{S} \ \text{TYPE}('a::\text{mem-type}) \ d1$
shows $\exists d2'. (p, d2') \in \text{stack-alloc } \mathcal{S} \ \text{TYPE}('a) \ d2$
using *stack-alloc-stack-free-mono* [*OF - alloc-d1*] *sub* **by** *blast*

lemma *stack-allocs-stack-free-eq*:
assumes *sub*: $\text{stack-free } d1 = \text{stack-free } d2$
assumes *alloc-d1*: $(p, d1') \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a::\text{mem-type}) \ d1$
shows $\exists d2'. (p, d2') \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a) \ d2$
using *stack-allocs-stack-free-mono* [*OF - alloc-d1*] *sub* **by** *blast*

lemma *fresh-ptr-stack-free-disjunct*:
 $(p, d') \in \text{stack-alloc } \mathcal{S} \ \text{TYPE}('a::\text{mem-type}) \ d \implies \text{ptr-span } p \cap \text{stack-free } d' =$
 $\{\}$
by (*smt* (*verit*, *best*) *disjoint-iff mem-Collect-eq mem-type-self ptr-val.ptr-val-def root-ptr-valid-type-neq-disjoint stack-alloc-cases stack-free-def*)

lemma *fresh-ptrs-stack-free-disjunct'*:
 $(p, d') \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a::\text{mem-type}) \ d \implies i < n \implies \text{ptr-span } (p +_p \text{int } i) \cap \text{stack-free } d' = \{\}$
using *stack-allocs-cases*
by (*smt (verit, ccfv-threshold) disjoint-iff mem-Collect-eq mem-type-self ptr-val.ptr-val-def root-ptr-valid-casesE stack-free-def*)

lemma *fresh-ptrs-stack-free-disjunct*:
 $(p, d') \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a::\text{mem-type}) \ d \implies \{\text{ptr-val } p \text{ ..+ } n * \text{size-of } \text{TYPE}('a) \} \cap \text{stack-free } d' = \{\}$
apply (*simp add: array-index-span-conv*)
using *fresh-ptrs-stack-free-disjunct'*
apply *blast*
done

lemma *stack-allocs-neq*: $(p, d') \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a::\text{mem-type}) \ d \implies d \neq d'$
by (*meson basic-trans-rules(31) disjoint-iff fresh-ptrs-stack-free-disjunct mem-type-self stack-allocs-cases stack-allocs-contained*)

lemma *stack-free-stack-alloc*:
assumes $p: (p, d') \in \text{stack-alloc } \mathcal{S} \ \text{TYPE}('a::\text{mem-type}) \ d$
shows $\text{stack-free } d' = \text{stack-free } d - \text{ptr-span } p$
proof –
from p **obtain**
not-sb: $\text{typ-uinfo-t } (\text{TYPE}('a)) \neq \text{typ-uinfo-t } (\text{TYPE}(\text{stack-byte}))$ **and**
sb: $\forall a \in \text{ptr-span } p. \text{root-ptr-valid } d \ (\text{PTR } (\text{stack-byte}) \ a)$ **and**
ptr-aligned p **and**
c-guard: $c\text{-guard } p$ **and**
valid-d': $\text{root-ptr-valid } d' \ p$ **and**
d': $d' = \text{ptr-force-type } p \ d$
by (*cases rule: stack-alloc-cases*) *auto*

from $c\text{-guard } \text{valid-d}' \ \text{not-sb}$
have *not-stack*: $\forall a \in \text{ptr-span } p. \neg \text{root-ptr-valid } d' \ (\text{PTR } (\text{stack-byte}) \ a)$
apply (*simp add: d'*)
by (*metis IntI empty-iff mem-type-self ptr-force-type-h-t-valid ptr-val.ptr-val-def*
 $\text{root-ptr-valid-not-subtype-disjoint sub-typ-stack-byte}$)
with *fresh-ptr-stack-free-disjunct* [*OF* p] *stack-alloc-stack-subset-stack-free* [*OF* p]

show *?thesis*
apply *safe*
subgoal
using *not-sb d' not-stack*
by (*metis mem-Collect-eq p stack-alloc-root-ptr-valid-new-cases stack-free-def*)
subgoal *by auto*
subgoal

using *sb d' not-stack*
by (*simp add: ptr-force-type-d root-ptr-valid-def stack-free-def valid-root-footprint-def*)
done
qed

lemma *stack-free-stack-allocs:*

assumes $p: (p, d') \in \text{stack-allocs } n \mathcal{S} \text{ TYPE}('a::\text{mem-type}) d$
shows $\text{stack-free } d' = \text{stack-free } d - \{\text{ptr-val } p \dots + n * \text{size-of } \text{TYPE}('a)\}$
proof –
from p **obtain**
not-sb: typ-winfo-t (TYPE('a)) \neq typ-winfo-t (TYPE(stack-byte)) **and**
*sb: $\forall a \in \{\text{ptr-val } p \dots + n * \text{size-of } \text{TYPE}('a)\}. \text{root-ptr-valid } d \text{ (PTR (stack-byte)) } a$* **and**

aligned: $\forall i < n. \text{ptr-aligned } (p +_p \text{int } i)$ **and**
c-guard: $\forall i < n. \text{c-guard } (p +_p \text{int } i)$ **and**
root-valid: $\forall i < n. \text{root-ptr-valid } d' (p +_p \text{int } i)$ **and**
d': $d' = \text{fold } (\lambda i. \text{ptr-force-type } (p +_p \text{int } i)) [0..<n] d$
by (*cases rule: stack-allocs-cases*) *auto*

from *c-guard root-valid not-sb*
have *not-stack: $\forall a \in \{\text{ptr-val } p \dots + n * \text{size-of } \text{TYPE}('a)\}. \neg \text{root-ptr-valid } d' \text{ (PTR (stack-byte)) } a$*
apply (*simp add: d' array-index-span-conv*)
by (*smt (verit, best) disjoint-iff lessThan-iff mem-type-self ptr-val.ptr-val-def root-ptr-valid-casesE*)

with *fresh-ptrs-stack-free-disjunct [OF p] stack-allocs-stack-subset-stack-free [OF p]*

show *?thesis*

apply *safe*

subgoal

using *not-sb d' not-stack*

by (*metis mem-Collect-eq p stack-allocs-root-ptr-valid-new-cases stack-free-def*)

subgoal by *auto*

subgoal

using *sb d' not-stack*

by (*simp add: fold-ptr-force-type-other root-ptr-valid-def stack-free-def valid-root-footprint-def*)

done

qed

lemma *stack-release-other: $x \notin \text{ptr-span } p \implies \text{stack-release } p \ d \ x = d \ x$*

by (*simp add: stack-release-def ptr-retyp-d*)

lemma *stack-releases-other:*

fixes $p::'a::\text{mem-type } ptr$

shows $x \notin \{\text{ptr-val } p \dots + n * \text{size-of } \text{TYPE}('a::\text{mem-type})\} \implies \text{stack-releases } n \ p \ d \ x = d \ x$

by (*simp add: stack-releases-def fold-ptr-retyp-other*)

lemma *in-ptr-span-itself*: $x \in \text{ptr-span } (\text{PTR}('a::\text{mem-type}) x)$
by (*metis mem-type-self ptr-val.ptr-val-def*)

lemma *stack-byte-typing-footprint*:
stack-byte-typing $x = (\text{True}, \text{list-map } (\text{typ-slice-t } (\text{typ-uinfo-t } \text{TYPE}(\text{stack-byte})))$
 $0))$
apply (*simp add: stack-release-def stack-byte-typing-def*)
apply (*subst ptr-force-type-footprint*)
apply (*rule in-ptr-span-itself*)
apply *simp*
done

lemma *stack-release-footprint*: $x \in \text{ptr-span } p \implies$
stack-release $p d x = (\text{True}, \text{list-map } (\text{typ-slice-t } (\text{typ-uinfo-t } \text{TYPE}(\text{stack-byte})))$
 $0))$
apply (*simp add: stack-release-def stack-byte-typing-def*)
apply (*subst ptr-force-type-footprint*)
apply (*rule in-ptr-span-itself*)
apply *simp*
done

lemma *stack-releases-footprint*:
fixes $p::'a::\text{mem-type } ptr$
shows $x \in \{\text{ptr-val } p \text{ ..+ } n * \text{size-of } \text{TYPE}('a::\text{mem-type})\} \implies$
stack-releases $n p d x = (\text{True}, \text{list-map } (\text{typ-slice-t } (\text{typ-uinfo-t } \text{TYPE}(\text{stack-byte})))$
 $0))$
apply (*simp add: stack-releases-def stack-byte-typing-def*)
apply (*subst ptr-force-type-footprint*)
apply (*rule in-ptr-span-itself*)
apply *simp*
done

lemma *stack-byte-typing-valid-root-footprint*:
valid-root-footprint *stack-byte-typing* $x (\text{typ-uinfo-t } \text{TYPE}(\text{stack-byte}))$
using *stack-byte-typing-footprint valid-root-footprint-def* **by** (*fastforce*)

lemma *stack-release-valid-root-footprint*: $x \in \text{ptr-span } p \implies$
valid-root-footprint (*stack-release* $p d$) $x (\text{typ-uinfo-t } \text{TYPE}(\text{stack-byte}))$
using *stack-release-footprint valid-root-footprint-def* **by** (*fastforce*)

lemma *stack-releases-valid-root-footprint*:
fixes $p::'a::\text{mem-type } ptr$
shows $x \in \{\text{ptr-val } p \text{ ..+ } n * \text{size-of } \text{TYPE}('a::\text{mem-type})\} \implies$
valid-root-footprint (*stack-releases* $n p d$) $x (\text{typ-uinfo-t } \text{TYPE}(\text{stack-byte}))$
using *stack-releases-footprint valid-root-footprint-def* **by** *fastforce*

```

lemma stack-release-root-ptr-valid1:
  fixes p::'a::mem-type ptr
  fixes q::'b::mem-type ptr
  assumes non-stack-p: typ-uinto-t TYPE('a) ≠ typ-uinto-t TYPE(stack-byte)
  assumes non-stack-q: typ-uinto-t TYPE('b) ≠ typ-uinto-t TYPE(stack-byte)
  assumes root-q: root-ptr-valid (stack-release p d) q
  shows ptr-span p ∩ ptr-span q = {} ∧ root-ptr-valid d q
  apply (rule conjI)
  using assms
  apply (smt (verit, ccfv-threshold) disjoint-iff root-ptr-valid-valid-root-footprint
size-of-tag
stack-release-valid-root-footprint valid-root-footprint-type-neq)
  by (smt (verit, best) intvlI non-stack-q root-ptr-valid-def root-q stack-release-other

stack-release-valid-root-footprint valid-root-footprint-def valid-root-footprint-type-neq)

```

```

lemma stack-releases-root-ptr-valid1:
  fixes p::'a::mem-type ptr
  fixes q::'b::mem-type ptr
  assumes non-stack-p: typ-uinto-t TYPE('a) ≠ typ-uinto-t TYPE(stack-byte)
  assumes non-stack-q: typ-uinto-t TYPE('b) ≠ typ-uinto-t TYPE(stack-byte)
  assumes root-q: root-ptr-valid (stack-releases n p d) q
  shows {ptr-val p ..+ n * size-of TYPE('a::mem-type)} ∩ ptr-span q = {} ∧
root-ptr-valid d q
  apply (rule context-conjI)
  subgoal
  using assms
  by (smt (verit, best) disjoint-iff root-ptr-valid-def size-of-tag stack-releases-valid-root-footprint
valid-root-footprint-type-neq)
  subgoal
  using assms
  by (metis (full-types) disjoint-iff root-ptr-valid-domain' stack-releases-other)
  done

```

```

lemma stack-release-root-ptr-valid2:
  fixes p::'a::mem-type ptr
  fixes q::'b::mem-type ptr
  assumes disj: ptr-span p ∩ ptr-span q = {}
  assumes valid-q: root-ptr-valid d q
  shows root-ptr-valid (stack-release p d) q
  using assms
  by (smt (verit, ccfv-threshold) disjoint-iff intvlI root-ptr-valid-def size-of-def
stack-release-other typ-uinto-size valid-root-footprint-def)

```

```

lemma stack-releases-root-ptr-valid2:
  fixes p::'a::mem-type ptr
  fixes q::'b::mem-type ptr
  assumes disj: {ptr-val p ..+ n * size-of TYPE('a::mem-type)} ∩ ptr-span q =
{}

```

assumes *valid-q*: *root-ptr-valid* *d* *q*
shows *root-ptr-valid* (*stack-releases* *n* *p* *d*) *q*
using *assms*
by (*metis* (*full-types*) *disjoint-iff* *root-ptr-valid-domain'* *stack-releases-other*)

lemma *stack-release-root-ptr-valid-cases*:
fixes *p*::*'a::mem-type ptr*
fixes *q*::*'b::mem-type ptr*
assumes *non-stack-p*: *typ-uinto-t* *TYPE('a)* \neq *typ-uinto-t* *TYPE(stack-byte)*
assumes *non-stack-q*: *typ-uinto-t* *TYPE('b)* \neq *typ-uinto-t* *TYPE(stack-byte)*
shows *root-ptr-valid* (*stack-release* *p* *d*) *q* \longleftrightarrow *ptr-span* *p* \cap *ptr-span* *q* = $\{\}$ \wedge
root-ptr-valid *d* *q*
using *assms* *stack-release-root-ptr-valid1* *stack-release-root-ptr-valid2* **by** *blast*

lemma *stack-releases-root-ptr-valid-cases*:
fixes *p*::*'a::mem-type ptr*
fixes *q*::*'b::mem-type ptr*
assumes *non-stack-p*: *typ-uinto-t* *TYPE('a)* \neq *typ-uinto-t* *TYPE(stack-byte)*
assumes *non-stack-q*: *typ-uinto-t* *TYPE('b)* \neq *typ-uinto-t* *TYPE(stack-byte)*
shows *root-ptr-valid* (*stack-releases* *n* *p* *d*) *q* \longleftrightarrow $\{\text{ptr-val } p \text{ ..+ } n * \text{size-of } \text{TYPE('a::mem-type)}\} \cap \text{ptr-span } q = \{\} \wedge \text{root-ptr-valid } d \text{ } q$
using *assms* *stack-releases-root-ptr-valid1* *stack-releases-root-ptr-valid2* **by** *blast*

lemma *stack-release-root-ptr-valid-same-type-cases*:
fixes *p*::*'a::mem-type ptr*
assumes *cvalid-p*: *d* \models_t *p*
assumes *non-stack-p*: *typ-uinto-t* *TYPE('a)* \neq *typ-uinto-t* *TYPE(stack-byte)*
shows *root-ptr-valid* (*stack-release* *p* *d*) *q* \longleftrightarrow *p* \neq *q* \wedge *root-ptr-valid* *d* *q*
using *non-stack-p* *cvalid-p* *root-ptr-valid-h-t-valid* *cvalid-same-type-neq-disjoint*
stack-release-root-ptr-valid-cases
by *fastforce*

lemma *stack-releases-root-ptr-valid-same-type-cases*:
fixes *p*::*'a::mem-type ptr*
assumes *cvalid-p*: $\bigwedge i. i < n \implies d \models_t (p +_p \text{int } i)$
assumes *non-stack-p*: *typ-uinto-t* *TYPE('a)* \neq *typ-uinto-t* *TYPE(stack-byte)*
shows *root-ptr-valid* (*stack-releases* *n* *p* *d*) *q* \longleftrightarrow $(\forall i < n. q \neq p +_p (\text{int } i)) \wedge$
root-ptr-valid *d* *q*
apply *standard*
subgoal
using *non-stack-p* *cvalid-p* *cvalid-same-type-neq-disjoint* *stack-releases-root-ptr-valid-cases*
array-index-span-conv
by (*smt* (*verit*) *Abs-fnat-hom-mult* *CTypesDefs.ptr-add-def* *disjoint-iff-not-equal*
ex-in-conv
in-ptr-span-itself *intvl* *intvl-empty* *mult-less-cancel2* *mult-zero-left* *not-gr-zero*
of-int-of-nat-eq *ptr-add-0-id* *root-ptr-valid-domain* *semiring-1-class.of-nat-0*
stack-releases-other)

subgoal
using *non-stack-p cvalid-p root-ptr-valid-same-type-neq-disjoint stack-releases-root-ptr-valid-cases array-index-span-conv*
by (*smt (verit, best) array-to-index-span cvalid-same-type-neq-disjoint disjoint-iff root-ptr-valid-h-t-valid*)
done

lemma *ptr-aligned-stack-byte[simp]: ptr-aligned (PTR(stack-byte) x)*
by (*simp add: ptr-aligned-def*)

lemma *c-null-guard-cast-stack-byte:*
 $x \in \text{ptr-span } (p::'a::\text{mem-type } ptr) \implies \text{c-null-guard } p \implies$
 $\text{c-null-guard } (PTR \text{ (stack-byte) } x)$
apply (*clarsimp simp add: c-null-guard-def*)
using *intvl-Suc* **by** *blast*

lemma *c-guard-cast-stack-byte:*
 $x \in \text{ptr-span } (p::'a::\text{mem-type } ptr) \implies \text{c-guard } p \implies$
 $\text{c-guard } (PTR \text{ (stack-byte) } x)$
by (*clarsimp simp add: c-guard-def c-null-guard-cast-stack-byte*)

lemma *stack-heap-typing-root-ptr-valid-footprint: c-guard (p::stack-byte ptr) \implies*
root-ptr-valid stack-byte-typing p
by (*simp add: root-ptr-valid-def stack-byte-typing-valid-root-footprint c-guard-cast-stack-byte*)

lemma *stack-release-root-ptr-valid-footprint: $x \in \text{ptr-span } p \implies \text{c-guard } p \implies$*
root-ptr-valid (stack-release p d) (PTR (stack-byte) x)
by (*simp add: root-ptr-valid-def stack-release-valid-root-footprint c-guard-cast-stack-byte*)

lemma *stack-releases-root-ptr-valid-footprint:*
fixes $p::'a::\text{mem-type } ptr$
shows $x \in \{\text{ptr-val } p \dots n * \text{size-of } TYPE('a::\text{mem-type})\} \implies \forall i < n. \text{c-guard}$
 $(p +_p \text{int } i) \implies$
 $\text{root-ptr-valid } (\text{stack-releases } n \ p \ d) \ (PTR \text{ (stack-byte) } x)$
apply (*simp add: root-ptr-valid-def stack-releases-valid-root-footprint c-guard-cast-stack-byte*)
using *array-to-index-span c-guard-cast-stack-byte* **by** *blast*

lemma *stack-alloc-other:*
 $(p, d') \in \text{stack-alloc } \mathcal{S} \ TYPE('a::\text{mem-type}) \ d \implies x \notin \text{ptr-span } p \implies$
 $d' \ x = \ d \ x$
using *ptr-force-type-d stack-alloc-cases* **by** *blast*

lemma *stack-allocs-other:*
 $(p, d') \in \text{stack-allocs } n \ \mathcal{S} \ TYPE('a::\text{mem-type}) \ d \implies x \notin \{\text{ptr-val } p \dots n * \text{size-of } TYPE('a::\text{mem-type})\} \implies$
 $d' \ x = \ d \ x$
using *fold-ptr-force-type-other stack-allocs-cases* **by** *blast*

lemma *stack-free-stack-release-mono*:

shows $stack\text{-}free\ d \subseteq stack\text{-}free\ (stack\text{-}release\ p\ d)$

by (*smt* (*verit*) *Abs-fnat-hom-0 One-nat-def add.right-neutral less-Suc0 mem-Collect-eq root-ptr-valid-def size-of-stack-byte(2) stack-release-other stack-release-valid-root-footprint stack-free-def subsetI typ-uinfo-size valid-root-footprint-def*)

lemma *stack-free-stack-release-mono'*:

$stack\text{-}free\ d1 \subseteq stack\text{-}free\ d2 \implies stack\text{-}free\ (stack\text{-}release\ p\ d1) \subseteq stack\text{-}free\ (stack\text{-}release\ p\ d2)$

by (*smt* (*verit*) *Collect-mono-iff One-nat-def less-Suc0 root-ptr-valid-def size-of-stack-byte(3) stack-free-def stack-release-footprint stack-release-other valid-root-footprint-def*)

lemma *stack-free-stack-releases-mono*:

shows $stack\text{-}free\ d \subseteq stack\text{-}free\ (stack\text{-}releases\ n\ p\ d)$

by (*smt* (*verit*) *Abs-fnat-hom-0 One-nat-def add.right-neutral less-Suc0 mem-Collect-eq root-ptr-valid-def size-of-stack-byte(2) stack-releases-other stack-releases-valid-root-footprint stack-free-def subsetI typ-uinfo-size valid-root-footprint-def*)

lemma *stack-free-stack-releases-mono'*:

$stack\text{-}free\ d1 \subseteq stack\text{-}free\ d2 \implies stack\text{-}free\ (stack\text{-}releases\ n\ p\ d1) \subseteq stack\text{-}free\ (stack\text{-}releases\ n\ p\ d2)$

apply (*clarsimp simp add: stack-releases-def stack-free-def root-ptr-valid-def valid-root-footprint-def*)
by (*smt* (*verit*, *best*) *Collect-mono-iff override-on-apply-in override-on-apply-notin*)

lemma *stack-free-ptr-span-stack-release*:

$c\text{-null-guard}\ p \implies ptr\text{-}span\ p \subseteq stack\text{-}free\ (stack\text{-}release\ p\ d)$

by (*simp add: c-guard-def c-null-guard-cast-stack-byte root-ptr-valid-def stack-release-valid-root-footprint stack-free-def subsetI*)

lemma *stack-free-ptr-span-stack-releases*:

fixes $p::'a::mem\text{-}type\ ptr$

shows $(\bigwedge i. i < n \implies c\text{-null-guard}\ (p +_p\ int\ i)) \implies$

$\{ptr\text{-}val\ p \ ..+ n * size\text{-}of\ TYPE('a::mem\text{-}type)\} \subseteq stack\text{-}free\ (stack\text{-}releases\ n\ p\ d)$

by (*fastforce simp add: c-guard-def c-null-guard-cast-stack-byte root-ptr-valid-def*

stack-releases-valid-root-footprint stack-free-def subsetI array-index-span-conv)

lemma *stack-free-stack-release*:

assumes *c-null-guard: c-null-guard p*

shows $stack\text{-}free\ (stack\text{-}release\ p\ d) = ptr\text{-}span\ p \cup stack\text{-}free\ d$

proof

show $stack\text{-}free\ (stack\text{-}release\ p\ d) \subseteq ptr\text{-}span\ p \cup stack\text{-}free\ d$

proof

fix x

assume $x\text{-in}: x \in stack\text{-}free\ (stack\text{-}release\ p\ d)$

show $x \in ptr\text{-}span\ p \cup stack\text{-}free\ d$

```

proof (cases x ∈ ptr-span p)
  case True
  then show ?thesis
  by simp
next
  case False
  with c-null-guard x-in show ?thesis
  by (simp add: root-ptr-valid-def stack-release-other stack-free-def
    valid-root-footprint-def)
  qed
qed
next
  show ptr-span p ∪ stack-free d ⊆ stack-free (stack-release p d)
  using c-null-guard stack-free-ptr-span-stack-release stack-free-stack-release-mono
  by blast
qed

lemma stack-free-stack-releases:
  fixes p::'a::mem-type ptr
  assumes c-null-guard:  $\bigwedge i. i < n \implies c\text{-null-guard } (p +_p \text{int } i)$ 
  shows stack-free (stack-releases n p d) = {ptr-val p ..+ n * size-of TYPE('a::mem-type)}
  ∪ stack-free d
proof
  show stack-free (stack-releases n p d) ⊆ {ptr-val p ..+ n * size-of TYPE('a::mem-type)}
  ∪ stack-free d
  proof
  fix x
  assume x-in: x ∈ stack-free (stack-releases n p d)
  show x ∈ {ptr-val p ..+ n * size-of TYPE('a::mem-type)} ∪ stack-free d
  proof (cases x ∈ {ptr-val p ..+ n * size-of TYPE('a::mem-type)})
  case True
  then show ?thesis
  by simp
  next
  case False
  with c-null-guard x-in show ?thesis
  by (simp add: root-ptr-valid-def stack-releases-other stack-free-def
    valid-root-footprint-def)
  qed
qed
next
  show {ptr-val p ..+ n * size-of TYPE('a::mem-type)} ∪ stack-free d ⊆ stack-free
  (stack-releases n p d)
  using c-null-guard stack-free-ptr-span-stack-releases stack-free-stack-releases-mono
  by blast
qed

```

definition On-Exit:: ('s, 'p, 'f) com ⇒ ('s, 'p, 'f) com ⇒ ('s, 'p, 'f) com **where**
 On-Exit c cleanup = Seq (Catch c (Seq cleanup Throw)) cleanup

```

locale heap-typing-state =
  lense htd htd-upd
  for
  htd:: 's ⇒ heap-typ-desc and
  htd-upd:: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 's ⇒ 's

locale heap-mem-state =
  lense hmem hmem-upd
  for
  hmem:: 's ⇒ heap-mem and
  hmem-upd:: (heap-mem ⇒ heap-mem) ⇒ 's ⇒ 's

locale heap-state =
  typing: heap-typing-state htd htd-upd + heap: heap-mem-state hmem hmem-upd
  for
  htd:: 's ⇒ heap-typ-desc and
  htd-upd:: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 's ⇒ 's and
  hmem:: 's ⇒ heap-mem and
  hmem-upd:: (heap-mem ⇒ heap-mem) ⇒ 's ⇒ 's +
  assumes heap-commute: hmem-upd f (htd-upd g s) = htd-upd g (hmem-upd f s)

begin
lemma htd-hmem-upd [simp]: htd (hmem-upd f s) = htd s
  by (metis heap-commute typing.get-upd typing.upd-get)

lemma hmem-htd-upd [simp]: hmem (htd-upd f s) = hmem s
  by (metis heap.get-upd heap.upd-get heap-commute)
end

locale heap-state-global =
  heap-state htd htd-upd hmem hmem-upd + lense glob glob-upd
  for
  htd:: 's ⇒ heap-typ-desc and
  htd-upd:: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 's ⇒ 's and
  hmem:: 's ⇒ heap-mem and
  hmem-upd:: (heap-mem ⇒ heap-mem) ⇒ 's ⇒ 's and
  glob:: 's ⇒ 'a and
  glob-upd:: ('a ⇒ 'a) ⇒ 's ⇒ 's +
  assumes glob-htd-commute:  $\bigwedge g h. \text{glob-upd } g (\text{htd-upd } h s) = \text{htd-upd } h (\text{glob-upd } g s)$ 
  assumes glob-hmem-commute:  $\bigwedge g h. \text{glob-upd } g (\text{hmem-upd } h s) = \text{hmem-upd } h (\text{glob-upd } g s)$ 

locale heap-raw-state =
  lense t-hrs t-hrs-update
  for
  t-hrs :: 's ⇒ heap-raw-state and
  t-hrs-update:: (heap-raw-state ⇒ heap-raw-state) ⇒ 's ⇒ 's

```

```

begin
sublocale heap-state
   $\lambda s. (hrs\text{-}htd (t\text{-}hrs\ s)) \lambda upd. (t\text{-}hrs\text{-}update (hrs\text{-}htd\text{-}update\ upd))$ 
   $\lambda s. (hrs\text{-}mem (t\text{-}hrs\ s)) \lambda upd. (t\text{-}hrs\text{-}update (hrs\text{-}mem\text{-}update\ upd))$ 
  apply (unfold-locales)
    apply (simp-all add: hrs-htd-update hrs-htd-update-comp hrs-mem-update
hrs-mem-update-comp)
    apply (metis hrs-htd-def hrs-htd-update hrs-mem-def hrs-mem-htd-update prod.collapse
upd-same)
    apply (metis hrs-htd-def hrs-htd-mem-update hrs-mem-def hrs-mem-update prod.expand
upd-same)
    using hrs-update-commute by force
end

```

```

locale heap-raw-state-global =
  heap-raw-state t-hrs t-hrs-update + lense glob glob-upd
  for
    t-hrs :: 's  $\Rightarrow$  heap-raw-state and
    t-hrs-update:: (heap-raw-state  $\Rightarrow$  heap-raw-state)  $\Rightarrow$  's  $\Rightarrow$  's and
    glob:: 's  $\Rightarrow$  'a and
    glob-upd:: ('a  $\Rightarrow$  'a)  $\Rightarrow$  's  $\Rightarrow$  's +
  assumes glob-heap-commute:  $\bigwedge g\ h. glob\text{-}upd\ g (t\text{-}hrs\text{-}update\ h\ s) = t\text{-}hrs\text{-}update\ h$ 
(glob-upd g s)
begin
sublocale heap-state-global
   $\lambda s. (hrs\text{-}htd (t\text{-}hrs\ s)) \lambda upd. (t\text{-}hrs\text{-}update (hrs\text{-}htd\text{-}update\ upd))$ 
   $\lambda s. (hrs\text{-}mem (t\text{-}hrs\ s)) \lambda upd. (t\text{-}hrs\text{-}update (hrs\text{-}mem\text{-}update\ upd))$ 
  glob glob-upd
  apply (unfold-locales)
  using glob-heap-commute
  apply simp
  using glob-heap-commute
  apply simp
  done
end

```

```

locale stack-heap-state = heap-state htd htd-upd hmem hmem-upd
for
  htd:: 's  $\Rightarrow$  heap-tyr-desc and
  htd-upd:: (heap-tyr-desc  $\Rightarrow$  heap-tyr-desc)  $\Rightarrow$  's  $\Rightarrow$  's and
  hmem:: 's  $\Rightarrow$  heap-mem and
  hmem-upd:: (heap-mem  $\Rightarrow$  heap-mem)  $\Rightarrow$  's  $\Rightarrow$  's +
  fixes S::addr set
begin

```

```

definition With-Fresh-Stack-Ptr:: nat  $\Rightarrow$  ('s  $\Rightarrow$  'a list set)  $\Rightarrow$  (('a::mem-type ptr)

```

```

⇒ ('s, 'p, strictc-errortype) com) ⇒
  ('s, 'p, strictc-errortype) com where
  ‹ With-Fresh-Stack-Ptr n init c =
    Guard StackOverflow ({s. stack-allocs n S TYPE('a) (htd s) ≠ {} ∧ (∃ vs. vs ∈
    init s ∧ length vs = n)})
    (DynCom (λs0.
      Spec {(s, t). ∃ p d' vs bs.
        (p, d') ∈ stack-allocs n S TYPE('a) (htd s) ∧
        vs ∈ init s ∧ length vs = n ∧ length bs = n * size-of TYPE('a) ∧
        t = hmem-upd (fold (λi. heap-update-padding (p +p int i) (vs!i) (take (size-of
        TYPE('a)) (drop (i * size-of TYPE('a)) bs))) [0..n])
        (htd-upd (λ-. d') s)});;
      DynCom (λs1.
        On-Exit
          (c (allocated-ptrs n S TYPE('a) (htd s0) (htd s1)))
          (Spec {(s, t). ∃ bs. length bs = n * size-of TYPE('a) ∧
            t = hmem-upd (heap-update-list (ptr-val (allocated-ptrs n S TYPE('a) (htd
            s0) (htd s1))) bs)
            (htd-upd (stack-releases n ((allocated-ptrs n S TYPE('a) (htd s0) (htd
            s1)))) s)}))
        ›
  ›

```

ML ‹

```

structure With-Fresh-Stack-Ptr =
struct

```

```

  type data = {match: term -> {n:term, init:term, c:term, ct: term, instantiate:
  {n:term, init:term, c:term} -> term},
    cterm-match: cterm -> {n:cterm, init:cterm, c:cterm, ct: cterm,
  instantiate: {n:cterm, init:cterm, c:cterm} -> cterm}}

```

```

  fun map-match f ({match, cterm-match}:data) = {match = f match, cterm-match
  = cterm-match}:data
  fun map-cterm-match f ({match, cterm-match}:data) = {match = match, cterm-match
  = f cterm-match}:data
  fun merge-match (f, g) = Utils.fast-merge (fn (f, g) => Utils.first-match [f, g])
  (f, g)

```

```

structure Data = Generic-Data (
  type T = data
  val empty = {match = fn - => raise Match, cterm-match = fn - => raise
  Match};
  val merge = Utils.fast-merge (fn ({match = f1, cterm-match = g1}, {match =
  f2, cterm-match = g2}) =>
    {match = merge-match (f1, f2), cterm-match = merge-match (g1, g2)});
)

```

```

fun match ctxt = #match (Data.get (Context.Proof ctxt))

```

```

fun cterm-match ctxt = #cterm-match (Data.get (Context.Proof ctxt))

fun add-match match = Data.map (map-match (Utils.add-match match))
fun add-cterm-match cterm-match = Data.map (map-cterm-match (Utils.add-match
cterm-match))

end
>

declaration <
fn phi => fn context =>
  let
    fun match t = @{morph-match (fo) <With-Fresh-Stack-Ptr ?n ?init ?c>} phi
  (Context.theory-of context) t
    handle Pattern.MATCH => raise Match
    fun cterm-match ct = @{cterm-morph-match (fo) <With-Fresh-Stack-Ptr ?n
?init ?c>} phi ct
    handle Pattern.MATCH => raise Match
  in
    context
    |> With-Fresh-Stack-Ptr.add-match match
    |> With-Fresh-Stack-Ptr.add-cterm-match cterm-match
  end
  end
>

end

locale stack-heap-raw-state = heap-raw-state t-hrs t-hrs-update
for
  t-hrs :: 's ⇒ heap-raw-state and
  t-hrs-update:: (heap-raw-state ⇒ heap-raw-state) ⇒ 's ⇒ 's +
fixes S::addr set
begin
  sublocale stack-heap-state
    λs. hrs-htd (t-hrs s) λupd. t-hrs-update (hrs-htd-update upd)
    λs. hrs-mem (t-hrs s) λupd. t-hrs-update (hrs-mem-update upd)
    S
  by unfold-locales
end

locale globals-stack-heap-state = stack-heap-state htd htd-upd hmem hmem-upd S
for
  htd:: 's ⇒ heap-tyr-desc and
  htd-upd:: (heap-tyr-desc ⇒ heap-tyr-desc) ⇒ 's ⇒ 's and
  hmem:: 's ⇒ heap-mem and
  hmem-upd:: (heap-mem ⇒ heap-mem) ⇒ 's ⇒ 's and
  S::addr set +

```

```

fixes  $\mathcal{G}::\text{addr set}$ 

locale globals-stack-heap-raw-state = stack-heap-raw-state t-hrs t-hrs-update  $\mathcal{S}$ 
for
  t-hrs :: 's  $\Rightarrow$  heap-raw-state and
  t-hrs-update:: (heap-raw-state  $\Rightarrow$  heap-raw-state)  $\Rightarrow$  's  $\Rightarrow$  's and
   $\mathcal{S}::\text{addr set} +$ 
  fixes  $\mathcal{G}::\text{addr set}$ 
begin
sublocale globals-stack-heap-state
   $\lambda s.$  hrs-htd (t-hrs s)  $\lambda upd.$  t-hrs-update (hrs-htd-update upd)
   $\lambda s.$  hrs-mem (t-hrs s)  $\lambda upd.$  t-hrs-update (hrs-mem-update upd)
   $\mathcal{S}$   $\mathcal{G}$ 
  by unfold-locales
end

```

12.4 Misc derived language elements

definition

```

creturn :: (('e c-exntype  $\Rightarrow$  'e c-exntype)  $\Rightarrow$  ('g, 'l, 'e, 'x) state-scheme  $\Rightarrow$  ('g, 'l,
'e, 'x) state-scheme)
   $\Rightarrow$  (('a  $\Rightarrow$  'a)  $\Rightarrow$  ('g, 'l, 'e, 'x) state-scheme  $\Rightarrow$  ('g, 'l, 'e, 'x) state-scheme)
   $\Rightarrow$  (('g, 'l, 'e, 'x) state-scheme  $\Rightarrow$  'a)  $\Rightarrow$  (('g, 'l, 'e, 'x) state-scheme, 'p, 'f) com

```

where

```

creturn rtu xfu v  $\equiv$  (Basic ( $\lambda s.$  xfu ( $\lambda.$  v s) s);; (Basic (rtu ( $\lambda.$  Return)));;
THROW)

```

definition

```

creturn-void :: (('e c-exntype  $\Rightarrow$  'e c-exntype)  $\Rightarrow$  ('g, 'l, 'e, 'x) state-scheme
 $\Rightarrow$  ('g, 'l, 'e, 'x) state-scheme)  $\Rightarrow$  (('g, 'l, 'e, 'x) state-scheme, 'p, 'f) com

```

where

```

creturn-void rtu  $\equiv$  (Basic (rtu ( $\lambda.$  Return)));; THROW

```

definition

```

cexit :: (('g, 'l, 'e, 'x) state-scheme  $\Rightarrow$  ('g, 'l, 'e, 'x) state-scheme)  $\Rightarrow$  (('g, 'l, 'e,
'x) state-scheme, 'p, 'f) com

```

where

```

cexit xfu  $\equiv$  (Basic xfu);; THROW

```

definition

```

cbreak :: (('e c-exntype  $\Rightarrow$  'e c-exntype)  $\Rightarrow$  ('g, 'l, 'e, 'x) state-scheme
 $\Rightarrow$  ('g, 'l, 'e, 'x) state-scheme)  $\Rightarrow$  (('g, 'l, 'e, 'x) state-scheme, 'p, 'f) com

```

where

```

cbreak rtu  $\equiv$  (Basic (rtu ( $\lambda.$  Break)));; THROW

```

definition

ccatchbrk :: (('g, 'l, 'e, 'x) state-scheme ⇒ 'e c-exntype) ⇒ (('g, 'l, 'e, 'x) state-scheme, 'p, 'f) com

where

ccatchbrk *rt* ≡ *Cond* {*s*. *rt s* = *Break*} *SKIP THROW*

definition

cgoto :: *string* ⇒ (('e c-exntype ⇒ 'e c-exntype) ⇒ ('g, 'l, 'e, 'x) state-scheme ⇒ ('g, 'l, 'e, 'x) state-scheme) ⇒ (('g, 'l, 'e, 'x) state-scheme, 'p, 'f) com

where

cgoto *l* *rtu* ≡ (*Basic* (*rtu* (λ-. *Goto* *l*)); *THROW*)

definition

ccatchgoto :: *string* ⇒ (('g, 'l, 'e, 'x) state-scheme ⇒ 'e c-exntype) ⇒ (('g, 'l, 'e, 'x) state-scheme, 'p, 'f) com

where

ccatchgoto *l* *rt* ≡ *Cond* {*s*. *rt s* = *Goto* *l*} *SKIP THROW*

definition

ccatchreturn :: (('g, 'l, 'e, 'x) state-scheme ⇒ 'e c-exntype) ⇒ (('g, 'l, 'e, 'x) state-scheme, 'p, 'f) com

where

ccatchreturn *rt* ≡ *Cond* {*s*. *is-local* (*rt s*)} *SKIP THROW*

definition

cchaos :: ('b ⇒ 'a ⇒ 'a) ⇒ ('a, 'c, 'd) com

where

cchaos *upd* ≡ *Spec* { (*s0*, *s*) . ∃ *v*. *s* = *upd* *v* *s0* }

definition

guarded-spec-body *F* *R* = *Guard* *F* (*fst* ' *R*) (*Spec* *R*)

end

theory *UMM*

imports

Padding-Equivalence

CLanguage

begin

instantiation *word* :: (*len8*) *stack-type*

begin

instance

by *intro-classes* (*simp* *add*: *typ-info-word* *stack-tyt-info-def* *stack-byte-name-def*)

end

```

instantiation ptr :: (c-type) stack-type
begin
instance
  by intro-classes (simp add: typ-info-ptr stack-typ-info-def stack-byte-name-def)
end

lemma list-neq-witness:  $x \in \text{set } ys \implies x \notin \text{set } xs \implies xs \neq ys$ 
  by blast

lemma stack-typ-info-array-tag-n:
  stack-typ-info ((array-tag-n n)::('a::{stack-type}, 'b::finite) array xtyp-info)
  apply (induct n)
  subgoal
    apply (simp add: stack-typ-info-def atn-base stack-byte-name-def)
    apply (rule list-neq-witness [where x=CHR "r"])
    apply simp
    apply simp
    done
  apply (simp add: atn-rec)
  apply (rule stack-typ-info-ti-typ-combine')
  apply assumption
  done

instantiation array ::(stack-type, finite) stack-type
begin
instance
  by intro-classes (simp add: typ-info-array array-tag-def stack-typ-info-array-tag-n)
end

lemma max-non-zero-unfold: NO-MATCH 0 a  $\implies$  NO-MATCH 0 b  $\implies$  max a b
= (if a  $\leq$  b then b else a)
  by (simp add: max-def)

lemma eq-comp:
  assumes eq1: field-update (component-desc x) bs v  $\equiv$  g
  assumes eq2: field-update (component-desc x) bs w  $\equiv$  g
  shows field-update (component-desc x) bs v = field-update (component-desc x)
bs w
  using eq1 eq2 by simp

lemma word-rcat-single: word-rcat [x] = x
  by (simp add: word-rcat-def bin-rcat-def)

lemma length-word-rsplit-8: length ((word-rsplit (x::8 word)) :: 8 word list) = 1
  by (simp add: word-rsplit-def bin-rsplit-def)
lemma length-word-rsplit-16: length ((word-rsplit (x::16 word)) :: 8 word list) = 2
  by (simp add: word-rsplit-def bin-rsplit-def)
lemma length-word-rsplit-32: length ((word-rsplit (x::32 word)) :: 8 word list) = 4

```

by (*simp add: word-rsplit-def bin-rsplit-def*)
lemma *length-word-rsplit-64*: *length ((word-rsplit (x::64 word)) :: 8 word list) = 8*
by (*simp add: word-rsplit-def bin-rsplit-def*)
lemma *length-word-rsplit-128*: *length ((word-rsplit (x::128 word)) :: 8 word list) = 16*
by (*simp add: word-rsplit-def bin-rsplit-def*)

lemma *length-word-rsplit-signed-8*: *length ((word-rsplit (x::8 signed word)) :: 8 word list) = 1*
by (*simp add: word-rsplit-def bin-rsplit-def*)
lemma *length-word-rsplit-signed-16*: *length ((word-rsplit (x::16 signed word)) :: 8 word list) = 2*
by (*simp add: word-rsplit-def bin-rsplit-def*)
lemma *length-word-rsplit-signed-32*: *length ((word-rsplit (x::32 signed word)) :: 8 word list) = 4*
by (*simp add: word-rsplit-def bin-rsplit-def*)
lemma *length-word-rsplit-signed-64*: *length ((word-rsplit (x::64 signed word)) :: 8 word list) = 8*
by (*simp add: word-rsplit-def bin-rsplit-def*)
lemma *length-word-rsplit-signed-128*: *length ((word-rsplit (x::128 signed word)) :: 8 word list) = 16*
by (*simp add: word-rsplit-def bin-rsplit-def*)

lemmas *length-word-rsplit =*
length-word-rsplit-8
length-word-rsplit-16
length-word-rsplit-32
length-word-rsplit-64
length-word-rsplit-128
length-word-rsplit-signed-8
length-word-rsplit-signed-16
length-word-rsplit-signed-32
length-word-rsplit-signed-64
length-word-rsplit-signed-128

lemmas *wf-component-descs-intros =*
wf-component-descs-empty-typ-info
wf-component-descs-final-pad
wf-xfield.wf-component-descs-ti-typ-combine
wf-xfield.wf-component-descs-ti-typ-pad-combine

lemmas *component-descs-independent-intros =*
component-descs-independent-empty-typ-info
component-descs-independent-final-pad
wf-xfield.component-descs-independent-ti-typ-combine
wf-xfield.component-desc-independent-ti-typ-pad-combine

lemmas *wf-field-descs-intros =*

wf-field-descs-empty-typ-info
wf-field-descs-final-pad
wf-xfield.wf-field-descs-ti-typ-combine
wf-xfield.wf-field-descs-ti-typ-pad-combine

lemmas *contained-field-descs-intros* =
contained-field-descs-empty-typ-info
contained-field-descs-final-pad
contained-field-descs-ti-typ-combine
contained-field-descs-ti-typ-pad-combine

lemmas *field-update-simps* =
size-of-def ti-typ-pad-combine-def empty-typ-info-def ti-typ-combine-def ti-pad-combine-def
final-pad-def padup-def Let-def

lemmas *size-td-simps-arr-fl* =
size-td-simps
size-td-array align-td-array max-def

method *wf-xfield-solver* =
(intro-locales, rule wf-field.intro; simp add: comp-def)

method *try-wf-xfield-solver* **methods** *m* =
(match conclusion in wf-xfield acc upd for acc upd ⇒ wf-xfield-solver | m)

method *wf-component-descs-solver* =
(try-wf-xfield-solver ⟨(rule wf-component-descs-intros)⟩)+

method *field-desc-independent-solver* =
(rule field-desc-independent-PAD-expand,
simp only: aggregate-typ-combinators-simps set-toplevel-field-descs-combinator-simps,
(simp only: insert-union-out)?,
rule field-desc-independent-PAD-collapse,
rule field-desc-independent.intro;
fastforce simp add: fu-commutes-def)

method *component-descs-independent-solver* =
((try-wf-xfield-solver ⟨rule component-descs-independent-intros⟩)+;
field-desc-independent-solver)

method *wf-field-descs-solver* =
(try-wf-xfield-solver ⟨(rule wf-field-descs-intros)⟩)+

method *contained-field-descs-solver* =
(rule contained-field-descs-intros)+

lemma *unat-less-helper'*: $x < \text{of-nat } n \equiv \text{True} \implies \text{unat } x < n$
by (*simp add: unat-less-helper*)

lemma *unat-less-helper-numeral*:
 $x < (\text{numeral } n) \implies \text{unat } x < (\text{numeral } n)$
 $x < 1 \implies \text{unat } x < 1$
by (*simp-all add: unat-less-helper*)

lemma *unat-less-helper-numeral'*:
 $x < (\text{numeral } n) \equiv \text{True} \implies \text{unat } x < (\text{numeral } n)$
 $x < 1 \equiv \text{True} \implies \text{unat } x < 1$
using *unat-less-helper-numeral* **by** *blast+*

lemma *nat-sint-less-helper*:
 $i < s \text{ of-nat } n \implies 0 \leq s \ i \implies (\text{nat } (\text{sint } i)) < n$
by (*simp add: sint-eq-uint unat-less-helper word-sle-msb-le word-sless-msb-less*)

lemma *nat-sint-less-helper'*:
 $i < s \text{ of-nat } n \equiv \text{True} \implies 0 \leq s \ i \equiv \text{True} \implies (\text{nat } (\text{sint } i)) < n$
by (*simp add: nat-sint-less-helper*)

lemma *nat-sint-less-helper-numeral*:
 $i < s (\text{numeral } n) \implies 0 \leq s \ i \implies \text{nat } (\text{sint } i) < (\text{numeral } n)$
 $i < s \ 1 \implies 0 \leq s \ i \implies \text{nat } (\text{sint } i) < 1$
subgoal **by** (*metis nat-uint-eq not-less signed.leD sint-eq-uint unat-less-helper-numeral(1) word-msb-0 word-sle-msb-le*)
by (*metis linorder-not-less nat-code(2) signed.leD signed-0 word-less-1 word-msb-0 word-sle-msb-le zero-less-one*)

lemma *nat-sint-less-helper-numeral'*:
 $i < s (\text{numeral } n) \equiv \text{True} \implies 0 \leq s \ i \equiv \text{True} \implies \text{nat } (\text{sint } i) < (\text{numeral } n)$
 $i < s \ 1 \equiv \text{True} \implies 0 \leq s \ i \equiv \text{True} \implies \text{nat } (\text{sint } i) < 1$
using *nat-sint-less-helper-numeral* **by** *blast+*

lemma *sint-ucast-eq-uint'*:
 $\llbracket \text{LENGTH}(a) < \text{LENGTH}(b) \rrbracket$
 $\implies \text{sint } ((\text{ucast} :: ('a::\text{len word} \Rightarrow 'b::\text{len word})) \ x) = \text{uint } x$
apply (*rule sint-ucast-eq-uint*)
apply (*simp add: is-down-def target-size-def source-size-def*)
done

lemma *sint-ucast-signed-eq-uint*:
 $\text{LENGTH}(a) < \text{LENGTH}(b) \implies \text{sint } (\text{ucast } (x :: 'a :: \text{len word}) :: 'b :: \text{len signed word}) = \text{uint } x$
apply *transfer*
apply (*clarsimp simp add: signed-take-bit-take-bit*)
done

lemma *ucast-unat-sless-helper*:
 $UCAST('a::len \rightarrow 'b::len \text{ signed}) x <_s \text{ of-nat } n \implies$
 $LNGTH('a::len) < LNGTH('b::len) \implies \text{ unat } x < n$
by (*smt* (*verit*, *best*) *Word.of-nat-unat nat-int-comparison(2)* *of-nat-numeral*
sint-ucast-signed-eq-uint unat-lt2p unat-of-nat-eq word-sless-sint-le)

lemma *ucast-unat-sless-helper'*:
 $UCAST('a::len \rightarrow 'b::len \text{ signed}) x <_s \text{ of-nat } n \equiv \text{ True} \implies$
 $LNGTH('a::len) < LNGTH('b::len) \implies \text{ unat } x < n$
using *ucast-unat-sless-helper* **by** *blast*

lemma *ucast-unat-sless-helper-numeral-n*:
 $UCAST('a::len \rightarrow 'b::len \text{ signed}) x <_s (\text{ numeral } n) \implies$
 $LNGTH('a::len) < LNGTH('b::len) \implies \text{ unat } x < (\text{ numeral } n)$
using *ucast-unat-sless-helper*
by (*metis of-nat-numeral*)

lemma *ucast-unat-sless-helper-numeral-1*:
 $UCAST('a::len \rightarrow 'b::len \text{ signed}) x <_s 1 \implies$
 $LNGTH('a::len) < LNGTH('b::len) \implies \text{ unat } x < 1$
using *ucast-unat-sless-helper*
by *force*

lemmas *ucast-unat-sless-helper-numeral =*
ucast-unat-sless-helper-numeral-n
ucast-unat-sless-helper-numeral-1

lemma *ucast-unat-sless-helper-numeral'*:
 $UCAST('a::len \rightarrow 'b::len \text{ signed}) x <_s (\text{ numeral } n) \equiv \text{ True} \implies$
 $LNGTH('a::len) < LNGTH('b::len) \implies \text{ unat } x < (\text{ numeral } n)$
 $UCAST('a::len \rightarrow 'b::len \text{ signed}) x <_s 1 \equiv \text{ True} \implies$
 $LNGTH('a::len) < LNGTH('b::len) \implies \text{ unat } x < 1$
using *ucast-unat-sless-helper-numeral* **by** *blast+*

lemma *ucast-unat-less-helper*:
 $UCAST('a::len \rightarrow 'b::len) x < \text{ of-nat } n \implies$
 $LNGTH('a::len) \leq LNGTH('b::len) \implies \text{ unat } x < n$
by (*metis unat-less-helper unat-ucast-up-simp*)

lemma *ucast-unat-less-helper'*:
 $UCAST('a::len \rightarrow 'b::len) x < \text{ of-nat } n \equiv \text{ True} \implies$
 $LNGTH('a::len) \leq LNGTH('b::len) \implies \text{ unat } x < n$
using *ucast-unat-less-helper* **by** *blast*

lemma *ucast-unat-less-helper-numeral-n*:
 $UCAST('a::len \rightarrow 'b::len) x < (\text{ numeral } n) \implies$
 $LNGTH('a::len) \leq LNGTH('b::len) \implies \text{ unat } x < (\text{ numeral } n)$
using *ucast-unat-less-helper*
by (*metis of-nat-numeral*)

```

lemma ucast-unat-less-helper-numeral-1:
  UCAST('a::len → 'b::len) x < 1 ⇒
  LENGTH('a::len) ≤ LENGTH('b::len) ⇒ unat x < 1
using ucast-unat-less-helper
by force

lemmas ucast-unat-less-helper-numeral =
  ucast-unat-less-helper-numeral-n
  ucast-unat-less-helper-numeral-1

lemma ucast-unat-less-helper-numeral':
  UCAST('a::len → 'b::len) x < (numeral n) ≡ True ⇒
  LENGTH('a::len) ≤ LENGTH('b::len) ⇒ unat x < (numeral n)
  UCAST('a::len → 'b::len) x < 1 ≡ True ⇒
  LENGTH('a::len) ≤ LENGTH('b::len) ⇒ unat x < 1
using ucast-unat-less-helper-numeral by blast+

lemma len-of-less-basic-cases:
  LENGTH(8) < LENGTH(16)
  LENGTH(8) < LENGTH(32)
  LENGTH(8) < LENGTH(64)
  LENGTH(8) < LENGTH(128)
  LENGTH(16) < LENGTH(32)
  LENGTH(16) < LENGTH(64)
  LENGTH(16) < LENGTH(128)
  LENGTH(32) < LENGTH(64)
  LENGTH(32) < LENGTH(128)
  LENGTH(64) < LENGTH(128)
by simp-all

lemma len-of-le-basic-cases:
  LENGTH(8) ≤ LENGTH(16)
  LENGTH(8) ≤ LENGTH(32)
  LENGTH(8) ≤ LENGTH(64)
  LENGTH(8) ≤ LENGTH(128)
  LENGTH(16) ≤ LENGTH(32)
  LENGTH(16) ≤ LENGTH(64)
  LENGTH(16) ≤ LENGTH(128)
  LENGTH(32) ≤ LENGTH(64)
  LENGTH(32) ≤ LENGTH(128)
  LENGTH(64) ≤ LENGTH(128)
by simp-all

ML <

structure UMM-Tools =
struct
fun tactic-from-method (m:Proof.context → Method.method) ctxt thms st =

```

```

(m ctxt thms (ctxt, st)) |> Seq.filter-results |> Seq.map snd

fun tactic-from-src ctxt src =
let
  val (-, tok) = Method.read-closure-input ctxt src
  val m = Method.method ctxt tok
in tactic-from-method m end

val wf-component-descs-tac = tactic-from-src @ {context} <wf-component-descs-solver>
val component-descs-independent-tac = tactic-from-src @ {context} <component-descs-independent-solver>
val wf-field-descs-tac = tactic-from-src @ {context} <wf-field-descs-solver>
val contained-field-descs-tac = tactic-from-src @ {context} <contained-field-descs-solver>

end
>

```

ML <val UMM-ss = simpset-of **context**>

lemma *heap-update-fold-comp-apply* : $\text{heap-update } p \ v \ (g \ z) \equiv (\text{heap-update } p \ v \circ g) \ z$
by (*simp add: comp-apply*)

named-theorems

```

fg-cons-simps and
typ-info-simps and
td-names-simps and
typ-name-simps and
upd-lift-simps and
upd-other-simps and
size-align-simps and
fl-Some-simps and
fl-ti-simps and
sub-typ-simps and
typ-tag-defs and
size-simps and
typ-name-itself and
heap-update-fold-toplevel-fields and
heap-update-fold-toplevel-fields-pointless and
h-val-fields and heap-update-fields and
h-val-unfold

```

named-theorems *field-lookup-prems*

declare

```

size-of-words[size-simps]

```


size-of-swords[*size-simps*]
size-of-array[*size-simps*]
size-of-stack-byte[*size-simps*]
size-of-ptr[*size-simps*]

lemma *field-of-lookup-info*:

fixes $p::'a:: \text{mem-type ptr}$

assumes $\text{field}: \text{field-of off } (\text{typ-uinfo-t TYPE('b::mem-type)}) (\text{typ-uinfo-t TYPE('a)})$

shows $\exists f. f \in \text{set } (\text{field-names-u } (\text{typ-uinfo-t TYPE('a)}) (\text{typ-uinfo-t TYPE('b)}))$

\wedge

$\text{field-lookup } (\text{typ-uinfo-t TYPE('a)}) f 0 = \text{Some } (\text{typ-uinfo-t TYPE('b)}, \text{unat off}) \wedge$

$\text{field-of-t } (\text{PTR('b)} \ \&(p \rightarrow f)) p \wedge$

$\&(p \rightarrow f) = \text{ptr-val } (p::'a \ \text{ptr}) + \text{off} \wedge$

$\{ \&(p \rightarrow f) .. + \text{size-td } (\text{typ-uinfo-t TYPE('b)}) \} \subseteq \text{ptr-span } p$

proof –

have $\text{wf-desc } (\text{typ-info-t TYPE('a)})$ **by** (*rule wf-desc*)

from *td-set-field-lookup* [*OF wf-desc-export-uinfo-pres(1)*] [*OF this*] *field*

obtain f **where** $\text{lookup-u}: \text{field-lookup } (\text{typ-uinfo-t TYPE('a)}) f 0 = \text{Some } (\text{typ-uinfo-t TYPE('b)}, \text{unat off})$

by (*auto simp add: field-of-def typ-uinfo-t-def*)

from *field-lookup-export-uinfo-Some-rev* [*OF lookup-u*] [*simplified typ-uinfo-t-def*]

obtain s' **where** $\text{lookup}: \text{field-lookup } (\text{typ-info-t TYPE('a)}) f 0 = \text{Some } (s', \text{unat off})$ **and**

$\text{exp-s'}: \text{export-uinfo } (\text{typ-info-t TYPE('b)}) = \text{export-uinfo } s'$

by *blast*

from *field-names-SomeD3* [*OF lookup*]

have $f\text{-in}: f \in \text{set } (\text{field-names-u } (\text{typ-uinfo-t TYPE('a)}) (\text{typ-uinfo-t TYPE('b)}))$

by (*simp add: field-names-u-field-names-export-uinfo-conv (1)*] [*symmetric*] *typ-uinfo-t-def exp-s'*)

from *lookup*

have $\text{addr}: \&(p \rightarrow f) = \text{ptr-val } (p::'a \ \text{ptr}) + \text{off}$

by (*simp add: field-lvalue-def*)

from *field-tag-sub* [*OF lookup*]

have $\text{contained}: \{ \&(p \rightarrow f) .. + \text{size-td } (\text{typ-uinfo-t TYPE('b)}) \} \subseteq \text{ptr-span } p$

by (*simp add: exp-s' typ-uinfo-t-def*)

from *field addr*

have $\text{field-of-t } (\text{PTR('b)} \ \&(p \rightarrow f)) p$

by (*simp add: field-of-t-def field-of-def*)

with $\text{contained addr lookup-u f-in}$

show *?thesis*

by *blast*

qed

lemma *sub-typ-field-names-u-nonempty*:

```

assumes  $s-t: s \leq t$ 
shows  $\text{field-names-u } t \ s \neq []$ 
proof –
  from  $s-t$  obtain  $n$  where  $(s, n) \in \text{td-set } t \ 0$ 
    by (auto simp add: typ-tag-le-def)
  from  $\text{td-set-field-names-u-nonempty } (1)$  [OF this] show ?thesis.
qed

definition  $\text{TO-SUC } (n::\text{nat}) \equiv n$ 

simproc-setup  $\text{TO-SUC } (\langle \text{TO-SUC } (\text{numeral } x) \rangle) = \langle$ 
   $\text{fn } \text{phi} \Rightarrow \text{fn } \text{ctxt} \Rightarrow \text{fn } \text{ct} \Rightarrow$ 
     $\text{SOME } (\text{Simplifier.rewrite } (\text{ctxt } \text{addsimps } @\{\text{thms } \text{TO-SUC-def Num.numeral-nat}\})$ 
   $\text{ct})$ 
   $\rangle$ 
declare [simproc del: TO-SUC]

lemma  $\text{array-tag-SUC}$ :
 $\text{array-tag } (t::('a::\text{c-type}, 'b::\text{finite}) \text{ array itself}) = \text{array-tag-n } (\text{TO-SUC } (\text{CARD } ('b)))$ 
  by (simp add: array-tag-def TO-SUC-def)

lemma  $\text{field-lookup-cons}$ :  $\text{field-lookup } t \ [f] \ m = \text{Some } (t', n) \Longrightarrow \text{wf-desc } t \Longrightarrow$ 
   $\text{field-lookup } t \ (f \ \# \ g \ \# \ \text{gs}) \ m = \text{field-lookup } t' \ (g \ \# \ \text{gs}) \ n$ 
  using  $\text{field-lookup-prefix-Some''(1)}$  [rule-format], where  $f = [f]$  and  $g = g \ \# \ \text{gs}$ 
  by auto

lemma  $\text{field-lookup-cons}'$ :
   $\text{field-lookup } (\text{typ-info-t } (\text{TYPE } ('a::\text{mem-type}))) \ [f] \ m = \text{Some } (t', n) \Longrightarrow$ 
   $\text{field-lookup } (\text{typ-info-t } (\text{TYPE } ('a::\text{mem-type}))) \ (f \ \# \ g \ \# \ \text{gs}) \ m = \text{field-lookup } t'$ 
   $(g \ \# \ \text{gs}) \ n$ 
  by (simp add: field-lookup-cons)

lemma  $\text{exists-conj-disj}$ :  $P \Longrightarrow (\exists n. (P \wedge Q \ n) \vee R \ n) = (\exists n. Q \ n \vee R \ n)$ 
  by blast

lemma  $\text{abs-if-eq}$ :
  assumes  $\bigwedge x. b \ x \Longrightarrow f1 \ x = f2 \ x$ 
  assumes  $\bigwedge x. \neg b \ x \Longrightarrow g1 \ x = g2 \ x$ 
  shows  $(\lambda x. \text{if } b \ x \ \text{then } f1 \ x \ \text{else } g1 \ x) = (\lambda x. \text{if } b \ x \ \text{then } f2 \ x \ \text{else } g2 \ x) \longleftrightarrow \text{True}$ 
  using assms
  by metis

lemma  $\text{field-lookup-typ-uinfo-t-Some}$ :
 $\text{field-lookup } (\text{typ-info-t } \text{TYPE } ('a::\text{c-type})) \ f \ m = \text{Some } (s, n) \Longrightarrow$ 
 $\text{field-lookup } (\text{typ-uinfo-t } \text{TYPE } ('a)) \ f \ m = \text{Some } (\text{export-uinfo } s, n)$ 
  by (simp add: typ-uinfo-t-def field-lookup-export-uinfo-Some)

lemma  $\text{adjust-ti-wf-fd-pres}'$ :
  fixes  $t::'a \ \text{xtyp-info}$ 

```

```

and st::'a xtyp-info-struct
and ts::'a xtyp-info-tuple list
and x::'a xtyp-info-tuple
assumes fg-cons: fg-cons acc upd
shows
  wf-fd t  $\implies$ 
    wf-fd (map-td ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) t)
  wf-fd-struct st  $\implies$ 
    wf-fd-struct (map-td-struct ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd)
  st)
  wf-fd-list ts  $\implies$ 
    wf-fd-list (map-td-list ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd) ts)
  wf-fd-tuple x  $\implies$ 
    wf-fd-tuple (map-td-tuple ( $\lambda n$  algn. update-desc acc upd) (update-desc acc upd)
  x)
proof (induct t and st and ts and x)
case (TypDesc algn st nm)
then show ?case by auto
next
case (TypScalar algn st d)
  then show ?case
    apply simp
    apply (simp add: fd-cons-struct-def)
    apply (insert fg-cons)
    apply (simp add: fd-cons-desc-def fg-cons-def)
    apply safe
    subgoal
      by (auto simp add: fd-cons-double-update-def update-desc-def)
      presburger
    subgoal
      by (auto simp add: fd-cons-update-access-def update-desc-def)
      presburger
    subgoal
      by (auto simp add: fd-cons-access-update-def update-desc-def)
    subgoal
      by (auto simp add: fd-cons-length-def update-desc-def)
    done
  next
    case (TypAggregate ts)
    then show ?case by auto
  next
    case Nil-typ-desc
    then show ?case by auto
  next
    case (Cons-typ-desc x fs)
    obtain d nm y where x: x = DTuple d nm y by (cases x) auto

from Cons-typ-desc.prems obtain
  wf-fd-d: wf-fd d and

```

```

wf-fd-fs: wf-fd-list fs and
commutes-d-fs: fu-commutes (update-ti-t d) (update-ti-list-t fs) and
fa-fu-ind-d-fs: fa-fu-ind
  (⟨field-access = access-ti d, field-update = update-ti-t d, field-sz = size-td d⟩
  ⟨field-access = access-ti-list fs, field-update = update-ti-list-t fs,
    field-sz = size-td-list fs⟩)
  (size-td-list fs) (size-td d) and
fa-fu-ind-fs-d: fa-fu-ind
  (⟨field-access = access-ti-list fs, field-update = update-ti-list-t fs,
    field-sz = size-td-list fs⟩
  ⟨field-access = access-ti d, field-update = update-ti-t d, field-sz = size-td d⟩
  (size-td d) (size-td-list fs))
  by (simp add: x)

note hyps = Cons-typ-desc.hyps [simplified x, simplified]
note hyp-d = hyps(1) [OF wf-fd-d]
note hyp-fs = hyps(2) [OF wf-fd-fs]
show ?case
  apply (simp add: x hyp-d hyp-fs)
  apply safe
  subgoal
    using commutes-d-fs fg-cons
    apply (simp add: fu-commutes-def fg-cons-def)
    by auto
    (smt (verit) adjust-ti-def fg-cons field-desc.select-convs(2) field-desc-adjust-ti(3)
field-desc-list-def
  map-td-ext'(3) map-td-extI update-desc-def update-ti-t-adjust-ti wf-fd-d
wf-fd-fs)
  subgoal
    using fa-fu-ind-d-fs fg-cons
    apply (simp add: fa-fu-ind-def fg-cons-def)
    apply clarsimp
    by (smt (verit) ⟨fu-commutes (update-ti-t (map-td (λn algn. update-desc acc
upd) (update-desc acc upd) d)) (update-ti-list-t (map-td-list (λn algn. update-desc
acc upd) (update-desc acc upd) fs))⟩
  fa-fu-v fd-cons-length fd-cons-update-access fu-commutes-def hyp-d map-td-size(1)
wf-fd-consD)
  subgoal
    using fa-fu-ind-fs-d fg-cons
    apply (simp add: fa-fu-ind-def fg-cons-def)
    apply clarsimp
    by (metis adjust-ti-def fg-cons update-ti-t-adjust-ti)
  done
next
  case (DTuple-typ-desc d nm y)
  then show ?case by auto
qed

```

lemma *adjust-ti-wf-fd-pres[simp]*: $fg-cons\ acc\ upd \implies wf-fd\ t \implies wf-fd\ (adjust-ti\ t\ acc\ upd)$

by (*simp add: adjust-ti-wf-fd-pres' adjust-ti-def*)

lemma *neq-td-names-eq-neq-export-uinfo*: $td-names\ (typ-info-t\ t) \neq td-names\ (typ-info-t\ s) \implies export-uinfo\ (typ-info-t\ t) = export-uinfo\ (typ-info-t\ s) \iff False$

by (*metis td-names-export-uinfo*)

lemma *set-field-names-no-padding-all-field-names-no-padding-conv'*:

set (field-names-no-padding (typ-info-t TYPE('a::mem-type)) t) =

Set.filter

($\lambda f. \exists s\ n. field-lookup\ (typ-info-t\ TYPE('a))\ f\ 0 = Some\ (s, n) \wedge export-uinfo\ s = t$)

(set (all-field-names-no-padding (typ-info-t TYPE('a))))

by (*simp add: set-field-names-no-padding-all-field-names-no-padding-conv Set.filter-def*)

lemma *field-names-no-padding-all-field-names-no-padding-conv'*:

field-names-no-padding (typ-info-t TYPE('a::mem-type)) t =

filter

($\lambda f. \exists s\ n. field-lookup\ (typ-info-t\ TYPE('a))\ f\ 0 = Some\ (s, n) \wedge export-uinfo\ s = t$)

(all-field-names-no-padding (typ-info-t TYPE('a)))

by (*simp add: field-names-no-padding-all-field-names-no-padding-conv Set.filter-def*)

lemma *set-filter-insert*: *Set.filter P (insert x S) =*

(if P x then insert x (Set.filter P S) else Set.filter P S)

by (*auto simp add: Set.filter-def*)

lemma *set-filter-cons-image*: *Set.filter P ((#) x ' S) = (#) x ' Set.filter ($\lambda fs. P\ (x\#\#fs)$) S*

by (*auto simp add: Set.filter-def*)

lemma *set-filter-Sup*: *Set.filter P ($\bigcup_{x \in X} S\ x$) = ($\bigcup_{x \in X} Set.filter\ P\ (S\ x)$)*

by (*auto simp add: Set.filter-def*)

lemma *set-filter-empty*: *Set.filter P {} = {}*

by (*auto simp add: Set.filter-def*)

lemma *cons-image-Sup*: *(#) x ' ($\bigcup_{xs \in X} S\ xs$) = ($\bigcup_{xs \in X} ((#) x ' S\ xs)$)*

by (*rule image-UN*)

lemma *set-filter-image-all*:

assumes $\bigwedge x. x < n \implies P\ (f\ x)$

shows *Set.filter P (f ' {0..<n}) = f ' {0..<n}*

using *assms*

by *fastforce*

lemma *set-filter-image-none*:

```

assumes  $\bigwedge x. x < n \implies \neg P (f x)$ 
shows  $Set.filter P (f ` \{0..<n\}) = \{\}$ 
using assms
by fastforce

lemma set-filter-union-distrib:  $Set.filter P (X \cup Y) = Set.filter P X \cup Set.filter P Y$ 
by (auto simp add: Set.filter-def)

lemma sub-typ-refl [simp]:  $TYPE('a) \leq_{\tau} TYPE('a::c-type)$ 
by (simp add: sub-typ-def)

lemma not-sub-typ-via-td-name:
assumes ta:  $typ-name (typ-info-t TYPE('a :: c-type)) \neq pad-typ-name$ 
and tina:  $typ-name (typ-info-t TYPE('a :: c-type)) \notin td-names (typ-info-t TYPE('b :: c-type))$ 
shows  $\neg TYPE('a :: c-type) \leq_{\tau} TYPE('b :: c-type)$ 
using ta tina
apply (clarsimp simp add: sub-typ-def typ-tag-le-def)
apply (drule td-set-td-names)
apply (auto simp add: typ-uinfo-t-def)
done

lemma nat-to-bin-string-eq-to-nat-eq:
assumes eq:  $nat-to-bin-string n = nat-to-bin-string m$ 
shows  $n = m$ 
using eq
proof (induct n arbitrary: m rule: nat-less-induct)
case (1 n)
then obtain eq:  $nat-to-bin-string n = nat-to-bin-string m$  and
  hyp:  $\bigwedge m x. m < n \implies nat-to-bin-string m = nat-to-bin-string x \implies m = x$ 
by auto

show ?case
proof (cases n)
case 0
with eq show ?thesis
  apply(cases m)
  apply (simp add: ntbs)
  apply (simp add: ntbs)
  by (metis list.simps(3))
next
case (Suc n')
note  $n' = this$ 
show ?thesis
proof (cases m)
case 0
with eq show ?thesis
  apply (simp add: n' ntbs)

```

```

    by (meson list.simps(3))
next
case (Suc m')
have le: (Suc n' div 2) < n
  by (simp add: n')

note hyp' = hyp [OF le, of Suc m' div 2]
from eq show ?thesis
  apply (simp add: n' Suc)
  apply (subst (asm) (1 2) ntbs)
  apply clarsimp
  apply (frule hyp')
  apply (clarsimp split: if-split-asm)
  apply arith+
done
qed
qed
qed

lemma nat-to-bin-string-inj [simp]: nat-to-bin-string n = nat-to-bin-string m  $\longleftrightarrow$ 
n = m
  using nat-to-bin-string-eq-to-nat-eq by blast

simproc-setup nat-to-bin-string (⟨nat-to-bin-string (numeral x)⟩) = ⟨
fn phi => fn ctxt => fn ct =>
  SOME (Simplifier.rewrite (ctxt addsimps @ {thms nat-to-bin-string.simps}) ct)
⟩

lemma rewrite-solve-prop:
  assumes rew: (PROP P)  $\equiv$  Trueprop Q
  assumes solve: PROP Trueprop Q
  shows (PROP P)  $\equiv$  Trueprop True
proof -
  from solve have Q = True by simp
  from rew [simplified this]
  show PROP ?thesis
    by simp
qed

lemma trueprop-eq-bool-eq:
  assumes prop-eq: PROP Trueprop P  $\equiv$  PROP Trueprop Q
  shows P = Q
proof (cases P)
  case True
  with prop-eq have Q
    by (simp add: True)
  with True show ?thesis by simp
next
  case False

```

with *prop-eq* **have** $Q = \text{False}$
by (*cases Q*) *auto*
with *False* **show** *?thesis* **by** *simp*
qed

lemmas *rewrite-solve-prop-eq* = *eq-reflection* [*OF trueprop-eq-bool-eq, OF rewrite-solve-prop*]
lemmas *trueprop-eq-bool-meta-eq* = *trueprop-eq-bool-eq* [*THEN eq-reflection*]

lemma *export-uinfo-typ-uinfo-t-match*[*simp*]:
export-uinfo (*typ-info-t* $\text{TYPE}('a)$) = *typ-uinfo-t* ($t::'a::\text{c-type}$ *itself*) = *True*
by (*simp add: typ-uinfo-t-def*)

lemma *export-uinfo-eq-sub-typ-conv*:
export-uinfo (*typ-info-t* $\text{TYPE}('a::\text{c-type})$) = *export-uinfo* (*typ-info-t* $\text{TYPE}('b::\text{c-type})$)
 \longleftrightarrow
 $\text{TYPE}('a) \leq_{\tau} \text{TYPE}('b) \wedge \text{TYPE}('b) \leq_{\tau} \text{TYPE}('a)$
apply *standard*
apply (*simp add: fold-typ-uinfo-t sub-typ-def*)
by (*simp add: dual-order.antisym fold-typ-uinfo-t sub-typ-def*)

lemma *typ-uinfo-eq-sub-typ-conv*:
typ-uinfo-t $\text{TYPE}('a::\text{c-type})$ = *export-uinfo* (*typ-info-t* $\text{TYPE}('b::\text{c-type})$)
 \longleftrightarrow
 $\text{TYPE}('a) \leq_{\tau} \text{TYPE}('b) \wedge \text{TYPE}('b) \leq_{\tau} \text{TYPE}('a)$
export-uinfo (*typ-info-t* $\text{TYPE}('a::\text{c-type})$) = *typ-uinfo-t* $\text{TYPE}('b::\text{c-type})$
 \longleftrightarrow
 $\text{TYPE}('a) \leq_{\tau} \text{TYPE}('b) \wedge \text{TYPE}('b) \leq_{\tau} \text{TYPE}('a)$
typ-uinfo-t $\text{TYPE}('a::\text{c-type})$ = *typ-uinfo-t* $\text{TYPE}('b::\text{c-type})$
 \longleftrightarrow
 $\text{TYPE}('a) \leq_{\tau} \text{TYPE}('b) \wedge \text{TYPE}('b) \leq_{\tau} \text{TYPE}('a)$
by (*simp-all add: typ-uinfo-t-def export-uinfo-eq-sub-typ-conv*)

lemma *array-typ-subtyp-array-typ*:
assumes *typ-uinfo-t* ($\text{TYPE}('a::\text{mem-type})$) = *typ-uinfo-t* ($\text{TYPE}('c::\text{mem-type})$)
shows *typ-uinfo-t* ($\text{TYPE}('a::\text{mem-type}['b::\text{finite}])$) = *typ-uinfo-t* ($\text{TYPE}('c['b::\text{finite}])$)
apply (*simp add: typ-uinfo-array-tag-n-m-eq*)
apply (*simp add: uinfo-array-tag-n-m-def*)
using *assms*
by *simp*

lemma *le-array-typ-intro*:
 $\text{TYPE}('a::\text{mem-type}) \leq_{\tau} \text{TYPE}('c::\text{mem-type}) \implies$
 $\text{TYPE}('c::\text{mem-type}) \leq_{\tau} \text{TYPE}('a::\text{mem-type}) \implies$
 $\text{TYPE}('a::\text{mem-type}['b::\text{finite}]) \leq_{\tau} \text{TYPE}('c::\text{mem-type}['b::\text{finite}])$
using *typ-uinfo-eq-sub-typ-conv array-typ-subtyp-array-typ*
by (*smt (verit)*)

lemma *sub-typ-signed-unsiged*: $\text{TYPE}('a::\text{len8 signed word}) \leq_{\tau} \text{TYPE}('a \text{ word})$
by (*simp add: sub-typ-def typ-uinfo-t-signed-word-word-conv*)

lemma *sub-typ-unsigned-signed*: $TYPE('a \text{ word}) \leq_{\tau} TYPE('a::len8 \text{ signed word})$
by (*simp add: sub-typ-def typ-uinfo-t-signed-word-word-conv*)

lemma *sub-typ-proper-conv*: $TYPE('a::c-type) <_{\tau} TYPE('b::c-type) \longleftrightarrow$
 $typ-uinfo-t \ TYPE('a) \neq typ-uinfo-t \ TYPE('b) \wedge TYPE('a::c-type) \leq_{\tau}$
 $TYPE('b::c-type)$
by (*metis sub-typ-def sub-typ-proper-def typ-tag-lt-def*)

lemma *sub-typ-proper-to-sub-typ*:
 $TYPE('a::c-type) <_{\tau} TYPE('b::c-type) \implies TYPE('a::c-type) \leq_{\tau} TYPE('b::c-type)$
using *sub-typ-proper-conv* **by** *blast*

lemma *to-bytes-p-zero*: $to-bytes-p \ (c-type-class.zero::'a::xmem-type) = replicate \ (size-of$
 $TYPE('a)) \ 0$
by (*simp add: zero-def to-bytes-p-def to-bytes-from-bytes-id*)

lemma *field-lookup-zero*:
assumes *fl*: $field-lookup \ (typ-info-t \ TYPE('a::xmem-type)) \ f \ 0 = Some \ (t, n)$
assumes *match*: $export-uinfo \ t = typ-uinfo-t \ TYPE('b::c-type)$
shows $from-bytes \ (access-ti_0 \ t \ (c-type-class.zero::'a)) = (c-type-class.zero::'b)$
proof –
from *field-access-take-dropD* [*OF fl, simplified*]
have *eq1*: $take \ (size-td \ t) \ (drop \ n \ (to-bytes-p \ (c-type-class.zero::'a))) = access-ti_0$
 $t \ (c-type-class.zero::'a)$ **by** *simp*

from *fl match* **have** *sz*: $size-td \ t = size-of \ TYPE('b)$
by (*simp add: export-size-of*)
from *fl* **have** *le*: $size-of \ TYPE('b) \leq size-of \ TYPE('a) - n$
by (*metis export-size-of field-lookup-offset-size fold-typ-uinfo-t nat-move-sub-le*
sz)

have $take \ (size-td \ t) \ (drop \ n \ (to-bytes-p \ (c-type-class.zero::'a))) = replicate$
 $(size-of \ TYPE('b)) \ 0$
by (*simp add: to-bytes-p-zero sz le*)
with *eq1* **have** *eq2*: $access-ti_0 \ t \ (c-type-class.zero::'a) = replicate \ (size-of \ TYPE('b))$
 0 **by** *simp*
show *?thesis*
apply (*simp add: eq2*)
apply (*simp add: zero-def*)
done

qed

lemma *field-lookup-zero'*:
assumes *fl*: $field-lookup \ (typ-info-t \ TYPE('a::xmem-type)) \ f \ 0 \equiv Some \ (t, n)$
assumes *match*: $export-uinfo \ t = export-uinfo \ (typ-info-t \ TYPE('b::c-type))$
shows $from-bytes \ (access-ti_0 \ t \ (c-type-class.zero::'a)) = (c-type-class.zero::'b)$
using *field-lookup-zero* [*simplified typ-uinfo-t-def, OF - match*] *fl* **by** *blast*

```

lemma array-index-zero:
  assumes i-bound:  $i < \text{CARD}(b)$ 
  shows  $(c\text{-type-class.zero}::('a :: \text{array-outer-max-size})[b :: \text{array-max-count}])).[i]$ 
  =  $(c\text{-type-class.zero}::'a)$ 
proof -
  from field-lookup-array [OF i-bound, where 'a='a ]
  have fl: field-lookup (typ-info-t  $\text{TYPE}(a[b])$ ) [replicate i CHR "1"] 0 =
    Some (adjust-ti (typ-info-t  $\text{TYPE}(a)$ )  $(\lambda x. x.[i])$   $(\lambda x f. \text{Arrays.update } f \ i$ 
x),  $i * \text{size-of } \text{TYPE}(a)$ )
    by simp

  have export-uinfo (adjust-ti (typ-info-t  $\text{TYPE}(a)$ )  $(\lambda x::'a[b]. x.[i])$   $(\lambda x f. \text{Ar-$ 
rays.update } f \ i \ x)) = typ-uinfo-t  $\text{TYPE}(a)$ 
    by (simp add: typ-uinfo-t-def i-bound)

  from field-lookup-zero [OF fl this]
  show ?thesis
    by simp
qed

named-theorems zero-simps and make-zero
end

```

Chapter 13

Packed Types (no implicit padding)

```
theory PackedTypes
imports WordSetup CProof
begin
```

13.1 Underlying definitions for the class axioms

field-access / *field-update* is the identity for packed types

```
definition fa-fu-idem :: 'a field-desc  $\Rightarrow$  nat  $\Rightarrow$  bool where
  fa-fu-idem fd n  $\equiv$ 
     $\forall$  bs bs' v. length bs = n  $\longrightarrow$  length bs' = n  $\longrightarrow$  field-access fd (field-update fd
    bs v) bs' = bs
```

primrec

```
  td-fafu-idem :: ('a field-desc, 'b)typ-desc  $\Rightarrow$  bool and
  td-fafu-idem-struct :: ('a field-desc, 'b) typ-struct  $\Rightarrow$  bool and
  td-fafu-idem-list :: (('a field-desc, 'b) typ-desc, char list, 'b) dt-tuple list  $\Rightarrow$  bool
and
  td-fafu-idem-tuple :: (('a field-desc, 'b) typ-desc, char list, 'b) dt-tuple  $\Rightarrow$  bool
where
  fai0: td-fafu-idem (TypDesc algn ts n) = td-fafu-idem-struct ts

| fai1: td-fafu-idem-struct (TypScalar n algn d) = fa-fu-idem d n
| fai2: td-fafu-idem-struct (TypAggregate ts) = td-fafu-idem-list ts

| fai3: td-fafu-idem-list [] = True
| fai4: td-fafu-idem-list (x#xs) = (td-fafu-idem-tuple x  $\wedge$  td-fafu-idem-list xs)

| fai5: td-fafu-idem-tuple (DTuple x n d) = td-fafu-idem x

lemmas td-fafu-idem-simps = fai0 fai1 fai2 fai3 fai4 fai5
```

field-access is independent of the underlying bytes

definition *fa-heap-indep* :: 'a field-desc \Rightarrow nat \Rightarrow bool **where**

fa-heap-indep fd n \equiv
 \forall bs bs' v. length bs = n \longrightarrow length bs' = n \longrightarrow field-access fd v bs = field-access fd v bs'

primrec

td-fa-hi :: ('a field-desc, 'b) typ-desc \Rightarrow bool **and**
td-fa-hi-struct :: ('a field-desc, 'b) typ-struct \Rightarrow bool **and**
td-fa-hi-list :: (('a field-desc, 'b) typ-desc, char list, 'b) dt-tuple list \Rightarrow bool **and**
td-fa-hi-tuple :: (('a field-desc, 'b) typ-desc, char list, 'b) dt-tuple \Rightarrow bool

where

fahi0: *td-fa-hi* (TypDesc algn ts n) = *td-fa-hi-struct* ts

| *fahi1*: *td-fa-hi-struct* (TypScalar n algn d) = *fa-heap-indep* d n

| *fahi2*: *td-fa-hi-struct* (TypAggregate ts) = *td-fa-hi-list* ts

| *fahi3*: *td-fa-hi-list* [] = True

| *fahi4*: *td-fa-hi-list* (x#xs) = (*td-fa-hi-tuple* x \wedge *td-fa-hi-list* xs)

| *fahi5*: *td-fa-hi-tuple* (DTuple x n d) = *td-fa-hi* x

lemmas *td-fa-hi-simps* = *fahi0 fahi1 fahi2 fahi3 fahi4 fahi5*

13.2 Lemmas about *td-fafu-idem*

lemma *field-lookup-td-fafu-idem*:

shows \bigwedge (s :: ('a field-desc, 'b) typ-desc) f m n.

\llbracket *field-lookup* t f m = Some (s, n); *td-fafu-idem* t $\rrbracket \Longrightarrow$ *td-fafu-idem* s

and \bigwedge (s :: ('a field-desc, 'b) typ-desc) f m n.

\llbracket *field-lookup-struct* st f m = Some (s, n); *td-fafu-idem-struct* st $\rrbracket \Longrightarrow$

td-fafu-idem s

and \bigwedge (s :: ('a field-desc, 'b) typ-desc) f m n.

\llbracket *field-lookup-list* ts f m = Some (s, n); *td-fafu-idem-list* ts $\rrbracket \Longrightarrow$ *td-fafu-idem*

s

and \bigwedge (s :: ('a field-desc, 'b) typ-desc) f m n.

\llbracket *field-lookup-tuple* p f m = Some (s, n); *td-fafu-idem-tuple* p $\rrbracket \Longrightarrow$

td-fafu-idem s

by (*induct* t **and** st **and** ts **and** p) (*auto split: if-split-asm option.splits*)

lemma *field-access-update-same*:

fixes t :: ('a :: mem-type field-desc, 'b) typ-desc **and** st :: ('a field-desc, 'b) typ-struct **and**

ts :: ('a field-desc, 'b) typ-tuple list **and**

p :: ('a field-desc, 'b) typ-tuple

shows \bigwedge (v :: 'a) bs bs'. \llbracket *td-fafu-idem* t; *wf-fd* t; length bs = size-td t; length bs' = size-td t \rrbracket

```

    ⇒ access-ti t (update-ti t bs v) bs' = bs
  and  $\wedge(v :: 'a) bs bs'. \llbracket \text{td-fafu-idem-struct } st; \text{wf-fd-struct } st; \text{length } bs = \text{size-td-struct } st; \text{length } bs' = \text{size-td-struct } st \rrbracket$ 
    ⇒ access-ti-struct st (update-ti-struct st bs v) bs' = bs
  and  $\wedge(v :: 'a) bs bs'. \llbracket \text{td-fafu-idem-list } ts; \text{wf-fd-list } ts; \text{length } bs = \text{size-td-list } ts; \text{length } bs' = \text{size-td-list } ts \rrbracket$ 
    ⇒ access-ti-list ts (update-ti-list ts bs v) bs' = bs
  and  $\wedge(v :: 'a) bs bs'. \llbracket \text{td-fafu-idem-tuple } p; \text{wf-fd-tuple } p; \text{length } bs = \text{size-td-tuple } p; \text{length } bs' = \text{size-td-tuple } p \rrbracket$ 
    ⇒ access-ti-tuple p (update-ti-tuple p bs v) bs' = bs
  proof (induct t and st and ts and p)
  case TypScalar thus ?case by (clarsimp simp: fa-fu-idem-def)
next
  case (Cons-typ-desc p' ts' v bs bs')
  hence fu-commutes (update-ti-tuple-t p') (update-ti-list-t ts') by clarsimp
  moreover
  have update-ti-tuple p' (take (size-td-tuple p') bs) = update-ti-tuple-t p' (take (size-td-tuple p') bs)
  using Cons-typ-desc.prem1 by (simp add: update-ti-tuple-t-def min-ll)
  moreover
  have update-ti-list ts' (drop (size-td-tuple p') bs) = update-ti-list-t ts' (drop (size-td-tuple p') bs)
  using Cons-typ-desc.prem2 by (simp add: update-ti-list-t-def)
  ultimately have updeq:
    (update-ti-tuple p' (take (size-td-tuple p') bs) (update-ti-list ts' (drop (size-td-tuple p') bs) v))
    = (update-ti-list ts' (drop (size-td-tuple p') bs) (update-ti-tuple p' (take (size-td-tuple p') bs) v))
  unfolding fu-commutes-def by simp

  show ?case using Cons-typ-desc.prem3
  by (auto simp add: Cons-typ-desc.hyps) (simp add: updeq Cons-typ-desc.hyps)
qed simp+

```

```

lemma access-ti-tuple-dt-fst:
  access-ti-tuple p v bs = access-ti (dt-fst p) v bs
  by (cases p, simp)

```

```

lemma wf-fd-tuple-dt-fst:
  wf-fd-tuple p = wf-fd (dt-fst p)
  by (cases p, simp)

```

```

lemma field-lookup-offset2:
  assumes fl: (field-lookup t f (m + n) = Some (s, q))
  shows field-lookup t f m = Some (s, q - n)
  proof -
  from fl have le: m + n ≤ q
  by (rule field-lookup-offset-le)

```

hence $q = (m + n) + (q - (m + n))$
by *simp*

hence $\text{field-lookup } t f (m + n) = \text{Some } (s, (m + n) + (q - (m + n)))$ using *fl*
by *simp*

hence $\text{field-lookup } t f m = \text{Some } (s, m + (q - (m + n)))$
by (rule *iffD1* [*OF field-lookup-offset'(1)*])

thus *?thesis* using *le* by *simp*
qed

lemma *field-lookup-offset2-list*:

assumes *fl*: ($\text{field-lookup-list } t f (m + n) = \text{Some } (s, q)$)
shows $\text{field-lookup-list } t f m = \text{Some } (s, q - n)$

proof –

from *fl* have *le*: $m + n \leq q$
by (rule *field-lookup-offset-le*)

hence $q = (m + n) + (q - (m + n))$
by *simp*

hence $\text{field-lookup-list } t f (m + n) = \text{Some } (s, (m + n) + (q - (m + n)))$
using *fl* by *simp*

hence $\text{field-lookup-list } t f m = \text{Some } (s, m + (q - (m + n)))$
by (rule *iffD1* [*OF field-lookup-offset'(3)*])

thus *?thesis* using *le* by *simp*
qed

lemma *field-lookup-offset2-pair*:

assumes *fl*: ($\text{field-lookup-tuple } p f (m + n) = \text{Some } (s, q)$)
shows $\text{field-lookup-tuple } p f m = \text{Some } (s, q - n)$

proof –

from *fl* have *le*: $m + n \leq q$
by (rule *field-lookup-offset-le*)

hence $q = (m + n) + (q - (m + n))$
by *simp*

hence $\text{field-lookup-tuple } p f (m + n) = \text{Some } (s, (m + n) + (q - (m + n)))$
using *fl* by *simp*

hence $\text{field-lookup-tuple } p f m = \text{Some } (s, m + (q - (m + n)))$
by (rule *iffD1* [*OF field-lookup-offset'(4)*])

thus *?thesis* using *le* by *simp*

qed

lemma *field-access-update-nth-inner*:

shows $\bigwedge f (s :: ('a :: \text{mem-type field-desc, 'b}) \text{typ-desc}) n x v bs bs'$.

$\llbracket \text{field-lookup } t f 0 = \text{Some } (s, n); n \leq x; x < n + \text{size-td } s; \text{td-fafu-idem } s; \text{wf-fd } s; \text{wf-fd } t;$

$\text{length } bs = \text{size-td } s; \text{length } bs' = \text{size-td } t \rrbracket$

$\implies \text{access-ti } t (\text{update-ti } s bs v) bs' ! x = bs ! (x - n)$

and $\bigwedge f (s :: ('a :: \text{mem-type field-desc, 'b}) \text{typ-desc}) n x v bs bs'$.

$\llbracket \text{field-lookup-struct } st f 0 = \text{Some } (s, n); n \leq x; x < n + \text{size-td } s; \text{td-fafu-idem } s; \text{wf-fd } s; \text{wf-fd-struct } st;$

$\text{length } bs = \text{size-td } s; \text{length } bs' = \text{size-td-struct } st \rrbracket$

$\implies \text{access-ti-struct } st (\text{update-ti } s bs v) bs' ! x = bs ! (x - n)$

and $\bigwedge f (s :: ('a :: \text{mem-type field-desc, 'b}) \text{typ-desc}) n x v bs bs'$.

$\llbracket \text{field-lookup-list } ts f 0 = \text{Some } (s, n); n \leq x; x < n + \text{size-td } s; \text{td-fafu-idem } s; \text{wf-fd } s; \text{wf-fd-list } ts;$

$\text{length } bs = \text{size-td } s; \text{length } bs' = \text{size-td-list } ts \rrbracket$

$\implies \text{access-ti-list } ts (\text{update-ti } s bs v) bs' ! x = bs ! (x - n)$

and $\bigwedge f (s :: ('a :: \text{mem-type field-desc, 'b}) \text{typ-desc}) n x v bs bs'$.

$\llbracket \text{field-lookup-tuple } p f 0 = \text{Some } (s, n); n \leq x; x < n + \text{size-td } s; \text{td-fafu-idem } s; \text{wf-fd } s; \text{wf-fd-tuple } p;$

$\text{length } bs = \text{size-td } s; \text{length } bs' = \text{size-td-tuple } p \rrbracket$

$\implies \text{access-ti-tuple } p (\text{update-ti } s bs v) bs' ! x = bs ! (x - n)$

proof (*induct t and st and ts and p*)

case (*TypDesc algn typ-struct ls f s n x v bs bs'*)

show *?case*

proof (*cases f = []*)

case *False thus ?thesis using TypDesc by clarsimp*

next

case *True*

thus *?thesis using TypDesc.prem*

by (*simp add: field-access-update-same*)

qed

next

case (*Cons-tydesc p' ts' f s n x v bs bs'*)

have *nlex: n ≤ x and xln: x < n + size-td s*

and *lbs: length bs = size-td s and lbs': length bs' = size-td-list (p' # ts')* **by** *fact+*

from *Cons-tydesc* **have** *wf: wf-fd (dt-fst p')* **and** *wfts: wf-fd-list ts'* **by** (*cases p', auto*)

{

assume *fl: field-lookup-list ts' f (size-td (dt-fst p')) = Some (s, n)*

```

hence mlt: size-td (dt-fst p') ≤ n
  by (rule field-lookup-offset-le)

from fl have fl': field-lookup-list ts' f 0 = Some (s, n - size-td (dt-fst p'))
  by (rule field-lookup-offset2-list [where m = 0, simplified])

hence atl: access-ti-list ts' (update-ti s bs v) (drop (size-td (dt-fst p')) bs') ! (x
- size-td (dt-fst p')) = bs' ! (x - n)
  using mlt nlex xln lbs lbs' wf wfts ⟨td-fafu-idem s⟩ ⟨wf-fd s⟩
  by (simp add: Cons-typ-desc.hyps(2) [OF fl'] size-td-tuple-dt-fst)

from mlt have size-td (dt-fst p') ≤ x
  by (rule order-trans) fact

hence ?case using wf lbs lbs' atl
  by (simp add: nth-append length-fa-ti access-ti-tuple-dt-fst size-td-tuple-dt-fst)
}
moreover
{
note ih = Cons-typ-desc.hyps(1)[simplified access-ti-tuple-dt-fst wf-fd-tuple-dt-fst]

assume fl: field-lookup-tuple p' f 0 = Some (s, n)

hence x < size-td (dt-fst p')
  apply (cases p')
  apply (simp split: if-split-asm)
  apply (drule field-lookup-offset-size')
  apply (rule order-less-le-trans [OF xln])
  apply simp
  done

hence ?case using wf lbs lbs' nlex xln wf wfts ⟨td-fafu-idem s⟩ ⟨wf-fd s⟩
  by (simp add: nth-append length-fa-ti access-ti-tuple-dt-fst size-td-tuple-dt-fst
ih[OF fl])
}
ultimately show ?case using ⟨field-lookup-list (p' # ts') f 0 = Some (s, n)⟩ by
(simp split: option.splits)
qed (clarsimp split: if-split-asm)+

```

13.2.1 *td-fa-hi*

lemma *fa-heap-indepD*:

$\llbracket \text{fa-heap-indep } fd \ n; \text{ length } bs = n; \text{ length } bs' = n \rrbracket \implies$

$\text{field-access } fd \ v \ bs = \text{field-access } fd \ v \ bs'$

unfolding *fa-heap-indep-def*

apply (*drule spec, drule spec, drule spec*)

apply (*drule (1) mp*)

apply (*erule (1) mp*)

done


```

lemma td-fa-hi-heap-independence:
  fixes t::('a::mem-type, 'b) typ-info and
    st::('a::mem-type, 'b) typ-info-struct and
    ts::('a::mem-type, 'b) typ-info-tuple list and
    p::('a::mem-type, 'b) typ-info-tuple

  shows  $\bigwedge(v :: 'a :: \text{mem-type}) h h'. \llbracket \text{td-fa-hi } t; \text{ length } h = \text{size-td } t; \text{ length } h' = \text{size-td } t \rrbracket$ 
     $\implies \text{access-ti } t v h = \text{access-ti } t v h'$ 
  and  $\bigwedge(v :: 'a :: \text{mem-type}) h h'. \llbracket \text{td-fa-hi-struct } st; \text{ length } h = \text{size-td-struct } st; \text{ length } h' = \text{size-td-struct } st \rrbracket$ 
     $\implies \text{access-ti-struct } st v h = \text{access-ti-struct } st v h'$ 
  and  $\bigwedge(v :: 'a :: \text{mem-type}) h h'. \llbracket \text{td-fa-hi-list } ts; \text{ length } h = \text{size-td-list } ts; \text{ length } h' = \text{size-td-list } ts \rrbracket$ 
     $\implies \text{access-ti-list } ts v h = \text{access-ti-list } ts v h'$ 
  and  $\bigwedge(v :: 'a :: \text{mem-type}) h h'. \llbracket \text{td-fa-hi-tuple } p; \text{ length } h = \text{size-td-tuple } p; \text{ length } h' = \text{size-td-tuple } p \rrbracket$ 
     $\implies \text{access-ti-tuple } p v h = \text{access-ti-tuple } p v h'$ 
proof (induct t and st and ts and p)
  case TypDesc
  from TypDesc.prems show ?case
    by (simp) (erule (2) TypDesc.hyps)
next
  case TypScalar
  from TypScalar.prems show ?case
    by simp (erule (2) fa-heap-indepD)
next
  case TypAggregate
  from TypAggregate.prems show ?case
    by (simp) (erule (2) TypAggregate.hyps)
next
  case Nil-typ-desc thus ?case by simp
next
  case Cons-typ-desc
  from Cons-typ-desc.prems show ?case
    apply simp
    apply (erule conjE)
    apply (rule arg-cong2 [where f = (@)])
    apply (erule Cons-typ-desc.hyps; simp)
    apply (erule Cons-typ-desc.hyps; simp)
    done
next
  case DTuple-typ-desc
  from DTuple-typ-desc.prems show ?case
    by simp (erule (2) DTuple-typ-desc.hyps)
qed

```

13.3 Simp rules for deriving packed props from the type combinators

13.3.1 *td-fafu-idem*

lemma *td-fafu-idem-map-align* [*simp*]: *td-fafu-idem (map-align f t) = td-fafu-idem t*

by (*cases t simp*)

lemma *td-fafu-idem-final-pad*:

padup (2 ^ max algn (align-td t)) (size-td t) = 0

\implies *td-fafu-idem (final-pad algn t) = td-fafu-idem t*

unfolding *final-pad-def*

by (*clarsimp simp add: padup-def Let-def*)

lemma *td-fafu-idem-ti-typ-pad-combine*:

fixes *t :: 'a :: c-type itself and s :: ('b :: c-type) xtyp-info*

assumes *pad: padup (max (2 ^ algn) (align-of TYPE('a))) (size-td s) = 0*

shows *td-fafu-idem (ti-typ-pad-combine t xf xfu algn nm s) = td-fafu-idem (ti-typ-combine t xf xfu algn nm s)*

unfolding *ti-typ-pad-combine-def using pad*

by (*clarsimp simp: Let-def*)

lemma *td-fafu-idem-list-append*:

fixes *xs :: 'a :: c-type xtyp-info-tuple list*

shows *td-fafu-idem-list (xs @ ys) = (td-fafu-idem-list xs ^ td-fafu-idem-list ys)*

by (*induct xs simp+*)

lemma *td-fafu-idem-extend-ti*:

fixes *t :: 'a :: c-type xtyp-info*

fixes *s :: 'a :: c-type xtyp-info*

assumes *as: td-fafu-idem s*

and *at: td-fafu-idem t*

shows *td-fafu-idem (extend-ti s t algn nm d) using as at*

apply (*cases s*)

subgoal for *x1 typ-struct xs*

apply (*cases typ-struct; simp add: td-fafu-idem-list-append*)

done

done

lemma *fd-cons-access-updateD*:

\llbracket *fd-cons-access-update d n; length bs = n; length bs' = n* \implies

field-access d (field-update d bs v) bs' = field-access d (field-update d bs v') bs'

unfolding *fd-cons-access-update-def by clarsimp*

lemma *fa-fu-idem-update-desc*:

fixes *a :: 'a field-desc*

assumes *fg: fg-cons xf xfu*

and *fd: fd-cons-struct (TypScalar n n' a)*

shows $fa\text{-}fu\text{-}idem (update\text{-}desc\ xf\ xfu\ a)\ n = fa\text{-}fu\text{-}idem\ a\ n$
proof
assume $asm: fa\text{-}fu\text{-}idem (update\text{-}desc\ xf\ xfu\ a)\ n$

let $?fu = \lambda bs. \text{if length } bs = n \text{ then field-update } a\ bs \text{ else id}$
let $?a' = (\mid \text{field-access} = \text{field-access } a, \text{field-update} = ?fu, \text{field-sz} = n \mid)$

show $fa\text{-}fu\text{-}idem\ a\ n$
unfolding $fa\text{-}fu\text{-}idem\text{-}def$
proof (*intro impI conjI allI*)
fix $bs :: \text{byte list and } bs' :: \text{byte list and } v$
assume $l: \text{length } bs = n \text{ and } l': \text{length } bs' = n$

hence $(\forall v. \text{field-access } a\ (\text{field-update } a\ bs\ (xf\ v))\ bs' = bs)$
 $= (\forall v. \text{field-access } a\ (?fu\ bs\ (xf\ v))\ bs' = bs)$ **by** *simp*

also have $\dots = (\forall v. \text{field-access } a\ (\text{field-update } a\ bs\ v)\ bs' = bs)$ **using** *fd*
apply –
apply (*rule iffI*)
apply (*rule allI*)
apply (*subst (asm) fd-cons-access-updateD [OF - l l', where d = ?a', simplified]*)
apply (*simp add: fd-cons-struct-def fd-cons-desc-def*)
apply (*fastforce simp: l l'*)
apply (*fastforce simp: l l'*)
done

finally show $\text{field-access } a\ (\text{field-update } a\ bs\ v)\ bs' = bs$ **using** *asm fg l l'*
by (*clarsimp simp add: update-desc-def fa-fu-idem-def fg-cons-def*)
qed

next
assume $fa\text{-}fu\text{-}idem\ a\ n$
thus $fa\text{-}fu\text{-}idem (update\text{-}desc\ xf\ xfu\ a)\ n$
unfolding $fa\text{-}fu\text{-}idem\text{-}def\ update\text{-}desc\text{-}def$ **using** *fg*
by (*clarsimp simp add: update-desc-def fa-fu-idem-def fg-cons-def*)
qed

lemma $td\text{-}fafu\text{-}idem\text{-}map\text{-}td\text{-}update\text{-}desc$:
assumes $fg: fg\text{-}cons\ xf\ xfu$
shows $wf\text{-}fd\ t \implies td\text{-}fafu\text{-}idem (map\text{-}td\ (\lambda\ -. \text{update-desc } xf\ xfu)\ (update\text{-}desc\ xf\ xfu)\ t) = td\text{-}fafu\text{-}idem\ t$
and $wf\text{-}fd\text{-}struct\ st \implies td\text{-}fafu\text{-}idem\text{-}struct (map\text{-}td\text{-}struct\ (\lambda\ -. \text{update-desc } xf\ xfu)\ (update\text{-}desc\ xf\ xfu)\ st) = td\text{-}fafu\text{-}idem\text{-}struct\ st$
and $wf\text{-}fd\text{-}list\ ts \implies td\text{-}fafu\text{-}idem\text{-}list (map\text{-}td\text{-}list\ (\lambda\ -. \text{update-desc } xf\ xfu)\ (update\text{-}desc\ xf\ xfu)\ ts) = td\text{-}fafu\text{-}idem\text{-}list\ ts$
and $wf\text{-}fd\text{-}tuple\ p \implies td\text{-}fafu\text{-}idem\text{-}tuple (map\text{-}td\text{-}tuple\ (\lambda\ -. \text{update-desc } xf\ xfu)\ (update\text{-}desc\ xf\ xfu)\ p) = td\text{-}fafu\text{-}idem\text{-}tuple\ p$
by (*induct t and st and ts and p (auto elim!: fa-fu-idem-update-desc [OF fg])*)

lemmas *td-fafu-idem-adjust-ti* = *td-fafu-idem-map-td-update-desc*(1)[*folded adjust-ti-def*]

lemma *td-fafu-idem-ti-typ-combine*:

fixes *s* :: 'b :: c-type *xtyp-info*
assumes *fg*: *fg-cons* *xf* *xfu*
and *tda*: *td-fafu-idem* (*typ-info-t* *TYPE*('a :: mem-type))
and *tds*: *td-fafu-idem* *s*
shows *td-fafu-idem* (*ti-typ-combine* *TYPE*('a :: mem-type) *xf* *xfu* *algn* *nm* *s*)
unfolding *ti-typ-combine-def* **using** *tda* *tds*
apply (*clarsimp simp: Let-def*)
apply (*cases* *s*)
subgoal for *x1 typ-struct* *xs*
apply (*cases typ-struct*)
apply *simp*
apply (*subst td-fafu-idem-adjust-ti* [*OF fg wf-fd*], *assumption*)
apply (*simp add: td-fafu-idem-list-append*)
apply (*subst td-fafu-idem-adjust-ti* [*OF fg wf-fd*], *assumption*)
done
done

lemma *td-fafu-idem-ptr*:

td-fafu-idem (*typ-info-t* *TYPE*('a :: c-type ptr))
apply (*clarsimp simp add: fa-fu-idem-def*)
apply (*subst word-rsplit-rcat-size*)
apply (*clarsimp simp add: size-of-def word-size*)
apply *simp*
done

lemma *td-fafu-idem-word*:

td-fafu-idem (*typ-info-t* *TYPE*('a :: len8 word))
apply(*clarsimp simp: fa-fu-idem-def*)
apply (*subst word-rsplit-rcat-size*)
apply (*insert len8-dv8*)
apply (*clarsimp simp add: size-of-def word-size*)
apply (*subst dvd-div-mult-self; simp*)
apply *simp*
done

lemma *td-fafu-idem-array-n*:

\llbracket *td-fafu-idem* (*typ-info-t* *TYPE*('a)); $n \leq \text{card}$ (*UNIV* :: 'b set) $\rrbracket \implies$
td-fafu-idem (*array-tag-n* *n* :: ('a :: mem-type ['b :: finite]) *xtyp-info*)
by (*induct* *n*; *simp add: array-tag-n.simps empty-typ-info-def*)
(*simp add: td-fafu-idem-ti-typ-combine*)

lemma *td-fafu-idem-array*:

td-fafu-idem (*typ-info-t* *TYPE*('a)) \implies *td-fafu-idem* (*typ-info-t* *TYPE*('a :: mem-type
['b :: finite]))
by (*clarsimp simp: typ-info-array array-tag-def fa-fu-idem-def td-fafu-idem-array-n*)

lemma *td-fafu-idem-empty-typ-info*:
td-fafu-idem (empty-typ-info algn t)
unfolding *empty-typ-info-def*
by *simp*

13.3.2 *td-fa-hi*

lemma *td-fa-hi-final-pad*:
padup (2 ^ max algn (align-td t)) (size-td t) = 0
 \implies *td-fa-hi (final-pad algn t) = td-fa-hi t*
unfolding *final-pad-def*
by (*cases t*) (*clarsimp simp add: padup-def Let-def*)

lemma *td-fa-hi-ti-typ-pad-combine*:
fixes *t :: 'a :: c-type itself and s :: 'b :: c-type xtyp-info*
assumes *pad: padup (max (2 ^ algn) (align-of TYPE('a))) (size-td s) = 0*
shows *td-fa-hi (ti-typ-pad-combine t xf xfu algn nm s) = td-fa-hi (ti-typ-combine t xf xfu algn nm s)*
unfolding *ti-typ-pad-combine-def using pad*
by (*clarsimp simp: Let-def*)

lemma *td-fa-hi-list-append*:
fixes *xs :: 'a :: c-type xtyp-info-tuple list*
shows *td-fa-hi-list (xs @ ys) = (td-fa-hi-list xs ^ td-fa-hi-list ys)*
by (*induct xs*) *simp+*

lemma *td-fa-hi-extend-ti*:
fixes *t :: 'a :: c-type xtyp-info*
assumes *as: td-fa-hi s*
and *at: td-fa-hi t*
shows *td-fa-hi (extend-ti s t algn nm d) using as at*
apply (*cases s*)
subgoal for *x1 typ-struct xs*
by (*cases typ-struct; simp add: td-fa-hi-list-append*)
done

lemma *fa-heap-indep-update-desc*:
fixes *a :: 'a field-desc*
assumes *fg: fg-cons xf xfu*
and *fd: fd-cons-struct (TypScalar n n' a)*
shows *fa-heap-indep (update-desc xf xfu a) n = fa-heap-indep a n*

proof
assume *asm: fa-heap-indep (update-desc xf xfu a) n*

have *xf-xfu: $\bigwedge v v'. xf (xfu v v') = v$ using fg*
unfolding *fg-cons-def*
by *simp*

```

show fa-heap-indep a n
  unfolding fa-heap-indep-def
proof (intro impI conjI allI)
  fix bs :: byte list and bs' :: byte list and v
  assume l: length bs = n and l': length bs' = n
  with asm
  have field-access (update-desc xf xfu a) (xfu v undefined) bs =
    field-access (update-desc xf xfu a) (xfu v undefined) bs'
  by (rule fa-heap-indepD)

  thus field-access a v bs = field-access a v bs'
  unfolding update-desc-def
  by (simp add: xf-xfu)
qed
next
  assume asm: fa-heap-indep a n
  show fa-heap-indep (update-desc xf xfu a) n
  unfolding fa-heap-indep-def update-desc-def
  apply (simp, intro impI conjI allI)
  using asm by (metis fa-heap-indepD)
qed

lemma td-fa-hi-map-td-update-desc:
  assumes fg: fg-cons xf xfu
  shows wf-fd t  $\implies$  td-fa-hi (map-td ( $\lambda$ - . update-desc xf xfu) (update-desc xs
xfu) t) = td-fa-hi t
  and wf-fd-struct st  $\implies$  td-fa-hi-struct (map-td-struct ( $\lambda$ - . update-desc xf xfu)
(update-desc xs xfu) st) = td-fa-hi-struct st
  and wf-fd-list ts  $\implies$  td-fa-hi-list (map-td-list ( $\lambda$ - . update-desc xf xfu) (update-desc
xs xfu) ts) = td-fa-hi-list ts
  and wf-fd-tuple p  $\implies$  td-fa-hi-tuple (map-td-tuple ( $\lambda$ - . update-desc xf xfu)
(update-desc xs xfu) p) = td-fa-hi-tuple p
  by (induct t and st and ts and p) (auto elim!: fa-heap-indep-update-desc [OF
fg])

lemma td-fa-hi-adjust-ti:
  assumes fg: fg-cons xf xfu
  assumes wf: wf-fd t
  shows td-fa-hi (adjust-ti t xf xfu) = td-fa-hi t
  using fg wf
  by (simp add: adjust-ti-def td-fa-hi-map-td-update-desc)

lemma td-fa-hi-ti-typ-combine:
  fixes s :: 'b :: c-type xtyp-info
```

```

apply (cases s)
subgoal for x1 typ-struct xs
  by (cases typ-struct; simp add: td-fa-hi-list-append td-fa-hi-adjust-ti[OF fg wf-fd])
done

```

```

lemma td-fa-hi-ptr:
  td-fa-hi (typ-info-t TYPE('a :: c-type ptr))
  by (clarsimp simp add: fa-heap-indep-def)

```

```

lemma td-fa-hi-word:
  td-fa-hi (typ-info-t TYPE('a :: len8 word))
  by (clarsimp simp add: fa-heap-indep-def)

```

```

lemma td-fa-hi-array-n:
   $\llbracket \text{td-fa-hi (typ-info-t TYPE('a)); } n \leq \text{card (UNIV :: 'b set)} \rrbracket \implies \text{td-fa-hi (array-tag-n } n \text{ :: ('a :: mem-type ['b :: finite]) xtyp-info)}$ 
  by (induct n; simp add: array-tag-n.simps empty-typ-info-def td-fa-hi-ti-typ-combine)

```

```

lemma td-fa-hi-array:
  td-fa-hi (typ-info-t TYPE('a)) \implies td-fa-hi (typ-info-t TYPE('a :: mem-type ['b :: finite]))
  by (clarsimp simp add: typ-info-array array-tag-def fa-fu-idem-def td-fa-hi-array-n)

```

```

lemma td-fa-hi-empty-typ-info:
  td-fa-hi (empty-typ-info algn t)
  unfolding empty-typ-info-def
  by simp

```

13.4 The type class and simp sets

Packed types, with no padding, have the defining property that access is invariant under substitution of the underlying heap and access/update is the identity

```

class packed-type = mem-type +
  assumes td-fafu-idem: td-fafu-idem (typ-info-t TYPE('a))
  assumes td-fa-hi: td-fa-hi (typ-info-t TYPE('a))

```

```

lemmas td-fafu-idem-intro-simps =
  — Axioms
  td-fafu-idem
  — Combinators
  td-fafu-idem-final-pad td-fafu-idem-ti-typ-pad-combine td-fafu-idem-ti-typ-combine
td-fafu-idem-empty-typ-info
  — Constructors
  td-fafu-idem-ptr td-fafu-idem-word td-fafu-idem-array

```

```

lemmas td-fa-hi-intro-simps =

```

— Axioms
td-fa-hi
 — Combinators
td-fa-hi-final-pad td-fa-hi-ti-typ-pad-combine td-fa-hi-ti-typ-combine td-fa-hi-empty-typ-info
 — Constructors
td-fa-hi-ptr td-fa-hi-word td-fa-hi-array

lemma *align-td-wo-align-array'*:
align-td-wo-align (typ-info-t TYPE('a :: c-type['b :: finite])) = align-td-wo-align
(typ-info-t TYPE('a))
by (*simp add: typ-info-array array-tag-def align-td-wo-align-array-tag*)

lemma *align-td-array'*:
align-td (typ-info-t TYPE('a :: c-type['b :: finite])) = align-td (typ-info-t TYPE('a))
by (*simp add: typ-info-array array-tag-def align-td-array-tag*)

lemmas *packed-type-intro-simps =*
td-fafu-idem-intro-simps td-fa-hi-intro-simps align-td-wo-align-array' size-td-simps-3
size-td-array

lemma *access-ti-append'*:
 $\bigwedge list.$
access-ti-list (xs @ ys) t list =
access-ti-list xs t (take (size-td-list xs) list) @
access-ti-list ys t (drop (size-td-list xs) list)
proof(*induct xs*)
case *Nil show ?case by simp*
next
case (*Cons x xs*) **thus** *?case by (simp add: min-def ac-simps drop-take)*
qed

13.5 Instances

Words (of multiple of 8 size) are packed

instantiation *word :: (len8) packed-type*
begin
instance
by (*intro-classes; rule td-fafu-idem-word td-fa-hi-word*)
end

Pointers are always packed

instantiation *ptr :: (c-type)packed-type*
begin
instance
by (*intro-classes; simp add: fa-fu-idem-def word-rsplit-rcat-size word-size fa-heap-indep-def*)
end

Arrays of packed types are in turn packed


```

class array-outer-packed = packed-type + array-outer-max-size
class array-inner-packed = array-outer-packed + array-inner-max-size

instance word :: (len8)array-outer-packed ..
instance word :: (len8)array-inner-packed ..

instance array :: (array-outer-packed, array-max-count) packed-type
  by (intro-classes; simp add: td-fafu-idem-intro-simps td-fa-hi-intro-simps)

instance array :: (array-inner-packed, array-max-count) array-outer-packed ..

```

13.6 Theorems about packed types

13.6.1 *td-fa-hi*

```

lemma heap-independence:
   $\llbracket \text{length } h = \text{size-of } \text{TYPE}('a :: \text{packed-type}); \text{length } h' = \text{size-of } \text{TYPE}('a) \rrbracket$ 
   $\implies \text{access-ti } (\text{typ-info-t } \text{TYPE}('a)) v h = \text{access-ti } (\text{typ-info-t } \text{TYPE}('a)) v h'$ 
  by (rule td-fa-hi-heap-independence(1)[OF td-fa-hi], simp-all add: size-of-def)

```

```

theorem packed-heap-update-collapse:
  fixes u::'a::packed-type
  fixes v::'a
  shows heap-update p v (heap-update p u h) = heap-update p v h
  unfolding heap-update-def
  apply(rule ext)
  subgoal for x
    apply(cases x  $\in$  {ptr-val p..+size-of TYPE('a)})
    apply(simp add: heap-update-mem-same-point)
    apply(simp add:to-bytes-def)
    apply(subst heap-independence, simp)
    prefer 2
    apply(rule refl)
    apply(simp)
    apply(simp add: heap-update-nmem-same)
  done
done

```

```

lemma packed-heap-update-collapse-hrs:
  fixes p :: 'a :: packed-type ptr
  shows hrs-mem-update (heap-update p v) (hrs-mem-update (heap-update p v')
  hp) =
    hrs-mem-update (heap-update p v) hp
  unfolding hrs-mem-update-def
  by (simp add: split-def packed-heap-update-collapse)

```

13.6.2 *td-fafu-idem*

```

lemma order-leE:

```

```

fixes x :: 'a :: order
shows  $\llbracket x \leq y; x = y \implies P; x < y \implies P \rrbracket \implies P$ 
by (auto simp: order-le-less)

```

```

lemma of-nat-mono-maybe-le:
  shows  $\llbracket X < 2 \wedge \text{len-of } \text{TYPE}('a); Y \leq X \rrbracket \implies (\text{of-nat } Y :: 'a :: \text{len word}) \leq$ 
of-nat X
  apply (erule order-leE)
  apply simp
  apply (rule order-less-imp-le)
  apply (erule (1) of-nat-mono-maybe)
done

```

```

lemma intvl-le-lower:
  fixes x :: 'a :: len word
  shows  $\llbracket x \in \{y..+n\}; y \leq y + \text{of-nat } (n - 1); n < 2 \wedge \text{len-of } \text{TYPE}('a) \rrbracket \implies$ 
y ≤ x
  apply (drule intvlD)
  apply (elim conjE exE)
  apply (erule ssubst)
  apply (erule word-plus-mono-right2)
  apply (rule of-nat-mono-maybe-le)
  apply simp
  apply simp
done

```

```

lemma intvl-less-upper:
  fixes x :: 'a :: len word
  shows  $\llbracket x \in \{y..+n\}; y \leq y + \text{of-nat } (n - 1); n < 2 \wedge \text{len-of } \text{TYPE}('a) \rrbracket \implies$ 
x ≤ y + of-nat (n - 1)
  apply (drule intvlD)
  apply (elim conjE exE)
  apply (erule ssubst)
  apply (rule word-plus-mono-right; assumption?)
  apply (rule of-nat-mono-maybe-le; simp)
done

```

```

lemma packed-type-access-ti:
  fixes v :: 'a :: packed-type
  assumes lbs: length bs = size-of TYPE('a)
  shows access-ti (typ-info-t TYPE('a)) v bs = access-ti0 (typ-info-t TYPE('a)) v
  unfolding access-ti0-def
  by (rule heap-independence; simp add: lbs size-of-def)

```

```

lemma c-guard-field-lvalue:
  fixes p :: 'a :: mem-type ptr
  assumes cg: c-guard p
  and fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (t, n)
  and eu: export-uinfo t = typ-uinfo-t TYPE('b :: mem-type)

```

```

shows  $c\text{-guard } (Ptr \ \&(p \rightarrow f)) :: 'b :: mem\text{-type } ptr$ 
unfolding  $c\text{-guard-def}$ 
proof (rule  $conjI$ )
  from  $cg \ fl \ eu$  show  $ptr\text{-aligned } (Ptr \ \&(p \rightarrow f)) :: 'b \ ptr$ 
    by (rule  $c\text{-guard-ptr-aligned-fl}$ )
next
  from  $eu$  have  $std: \ size\text{-td } t = \ size\text{-of } TYPE('b)$  using  $fl$ 
    by ( $simp \ add: \ export\text{-size-of}$ )

  from  $cg$  have  $c\text{-null-guard } p$  unfolding  $c\text{-guard-def} \ ..$ 
  thus  $c\text{-null-guard } (Ptr \ \&(p \rightarrow f)) :: 'b \ ptr$  unfolding  $c\text{-null-guard-def}$ 
    apply (rule  $contrapos\text{-nn}$ )
    apply (rule  $subsetD \ [OF \ field\text{-tag-sub}, \ OF \ fl]$ )
    apply ( $simp \ add: \ std$ )
  done
qed

lemma  $word\text{-wrap-of-natD}$ :
  fixes  $x :: 'a :: len \ word$ 
  assumes  $wraps: \neg x \leq x + of\text{-nat } n$ 
  shows  $\exists k. x + of\text{-nat } k = 0 \wedge k \leq n$ 
proof -
  show  $?thesis$ 
  proof (rule  $exI \ [where \ x = \ unat \ (- \ x)], \ intro \ conjI$ )
    show  $x + of\text{-nat } (unat \ (-x)) = 0$ 
      by  $simp$ 
  next
    show  $unat \ (-x) \leq n$ 
      by ( $metis \ add.\ commute \ no\text{-plus-overflow-neg} \ not\text{-less} \ olen\text{-add-eqv} \ word\text{-unat-less-le}$ 
 $wraps$ )
  qed
qed

theorem  $packed\text{-heap-super-field-update}$ :
  fixes  $v :: 'a :: packed\text{-type}$  and  $p :: 'b :: packed\text{-type} \ ptr$ 
  assumes  $fl: \ field\text{-lookup } (typ\text{-info-t } TYPE('b)) \ f \ 0 = \ Some \ (t, \ n)$ 
  and  $cgrd: \ c\text{-guard } p$ 
  and  $eu: \ export\text{-uinfo } t = \ typ\text{-uinfo-t } TYPE('a)$ 
  shows  $heap\text{-update } (Ptr \ \&(p \rightarrow f)) \ v \ hp = \ heap\text{-update } p \ (update\text{-ti } t \ (to\text{-bytes-p}$ 
 $v) \ (h\text{-val } hp \ p)) \ hp$ 
  unfolding  $heap\text{-update-def} \ to\text{-bytes-def}$ 
  apply ( $simp \ add: \ packed\text{-type-access-ti}, \ rule \ ext$ )
proof -
  fix  $x$ 
  let  $?LHS = \ heap\text{-update-list } \&(p \rightarrow f) \ (to\text{-bytes-p } v) \ hp \ x$ 
  let  $?RHS = \ heap\text{-update-list } (ptr\text{-val } p) \ (to\text{-bytes-p } (update\text{-ti } t \ (to\text{-bytes-p } v)$ 
 $(h\text{-val } hp \ p))) \ hp \ x$ 

  from  $cgrd$  have  $al: \ ptr\text{-val } p \leq \ ptr\text{-val } p + of\text{-nat } (size\text{-of } TYPE('b) - 1)$  by

```

(rule c-guard-no-wrap)

have *szb*: size-of TYPE('b) < 2 ^ len-of TYPE(addr-bitsize)
 apply (fold card-word)
 apply (fold addr-card-def)
 apply (rule max-size)
done

have *szt*: n + size-td t ≤ size-of TYPE('b)
 unfolding size-of-def
 by (subst add.commute, rule field-lookup-offset-size [OF fl])
moreover **have** *t0*: 0 < size-td t **using** fl wf-size-desc
 by (rule field-lookup-wf-size-desc-gt)
ultimately **have** *szn*: n < size-of TYPE('b) **by** simp
from *szt* **have** *szt1*: n + (size-td t - 1) ≤ size-of TYPE('b)
 by simp

have *b0*: 0 < size-of (TYPE ('b)) **using** wf-size-desc
 unfolding size-of-def
 by (rule wf-size-desc-gt)

have *uofn*: unat (of-nat n :: addr-bitsize word) = n **using** *szn szb*
 by (metis le-unat-uoι nat-less-le unat-of-nat-len)

from *eu* **have** *std*: size-td t = size-of TYPE('a) **using** fl
 by (simp add: export-size-of)

hence ?LHS = (if x ∈ {&(p→f)..+size-td t} then (to-bytes-p v) ! unat (x -
&(p→f)) else hp x)
 by (simp add: heap-update-mem-same-point heap-update-nmem-same)
also **have** ... = ?RHS
 apply (simp, intro impI conjI)
proof -
 assume *xin*: x ∈ {&(p→f)..+size-td t}
 have to-bytes-p v ! unat (x - &(p→f)) = to-bytes-p (update-ti t (to-bytes-p v)
(h-val hp p)) ! unat (x - ptr-val p)
 proof (simp add: to-bytes-p-def to-bytes-def, subst field-access-update-nth-inner(1)[OF
fl, simplified])

have c-guard (Ptr &(p→f) :: 'a ptr) **using** cgrd fl *eu*
 by (rule c-guard-field-lvalue)
hence *pft*: &(p→f) ≤ &(p→f) + of-nat (size-td t - 1)
 apply -
 apply (drule c-guard-no-wrap)
 apply (simp add: std)
done

have *szt'*: size-td t < 2 ^ len-of TYPE(addr-bitsize)
 apply (subst std)

```

apply (fold card-word)
apply (fold addr-card-def)
apply (rule max-size)
done

have ofn: of-nat  $n \leq x - \text{ptr-val } p$ 
proof (rule le-minus')
  from xin show ptr-val  $p + \text{of-nat } n \leq x$  using pft szt'
    unfolding field-lvalue-def field-lookup-offset-eq [OF fl]
    by (rule intvl-le-lower)
next
  from szb szn have of-nat  $n \leq (\text{of-nat } (\text{size-of } \text{TYPE}'b) - 1) :: \text{addr-bitsize}$ 
word)
  apply -
  apply (rule of-nat-mono-maybe-le)
  apply simp-all
  done
  with al show ptr-val  $p \leq \text{ptr-val } p + \text{of-nat } n$ 
    by (rule word-plus-mono-right2)
qed

thus nlt:  $n \leq \text{unat } (x - \text{ptr-val } p)$ 
  by (metis uofn word-less-eq-iff-unsigned)

have  $x \leq \text{ptr-val } p + (\text{of-nat } n + \text{of-nat } (\text{size-td } t - 1))$  using xin pft szt' t0
  unfolding field-lvalue-def field-lookup-offset-eq [OF fl]
  by (metis (no-types) add.assoc intvl-less-upper)
moreover have  $x \in \{\text{ptr-val } p..+\text{size-of } \text{TYPE}'b\}$  using fl xin
  by (rule subsetD [OF field-tag-sub])
ultimately have  $x - \text{ptr-val } p \leq (\text{of-nat } n + \text{of-nat } (\text{size-td } t - 1))$  using
al szb
  by (metis add-diff-cancel-left' intvl-le-lower word-diff-ls(4))
moreover have  $\text{unat } (\text{of-nat } n + \text{of-nat } (\text{size-td } t - 1)) :: \text{addr-bitsize word}$ 
=  $n + \text{size-td } t - 1$ 
  using t0 order-le-less-trans [OF szt1 szb]
  by (metis Nat.add-diff-assoc One-nat-def Suc-leI of-nat-add unat-of-nat-len)
ultimately have  $\text{unat } (x - \text{ptr-val } p) \leq n + \text{size-td } t - 1$ 
  by (simp add: word-le-nat-alt)
thus  $\text{unat } (x - \text{ptr-val } p) < n + \text{size-td } t$  using t0
  by simp

show td-fafu-idem t
  by (rule field-lookup-td-fafu-idem(1)[OF fl td-fafu-idem])

show wf-fd t
  by (rule wf-fd-field-lookupD [OF fl wf-fd])

show length (access-ti (typ-info-t TYPE('a)) v (replicate (size-of TYPE('a))
0)) = size-td t

```

```

using wf-fd [where 'a = 'a]
by (simp add: length-fa-ti size-of-def std)

show length (replicate (size-of TYPE('b)) 0) = size-td (typ-info-t TYPE('b))
by (simp add: size-of-def)

have unat (x - &(p→f)) = unat ((x - ptr-val p) - of-nat n)
by (simp add: field-lvalue-def field-lookup-offset-eq [OF fl])
also have ... = unat (x - ptr-val p) - n
by (metis ofn unat-sub uofn)
finally have unat (x - &(p→f)) = unat (x - ptr-val p) - n .

thus access-ti (typ-info-t TYPE('a)) v (replicate (size-of TYPE('a)) 0) ! unat
(x - &(p→f)) =
  access-ti (typ-info-t TYPE('a)) v (replicate (size-of TYPE('a)) 0) ! (unat
(x - ptr-val p) - n)
by simp
qed

thus to-bytes-p v ! unat (x - &(p→f)) = ?RHS
apply (subst heap-update-mem-same-point, simp-all)
proof -
show x ∈ {ptr-val p..+size-of TYPE('b)} using fl xin
by (rule subsetD [OF field-tag-sub])
qed
next
assume xni: x ∉ {&(p→f)..+size-td t}
have ?RHS = (if x ∈ {ptr-val p..+size-of TYPE('b)}
  then (to-bytes-p (update-ti t (to-bytes-p v) (h-val hp p))) ! unat (x - ptr-val
p) else hp x)
by (simp add: heap-update-mem-same-point heap-update-nmem-same)

also
{
assume xin: x ∈ {ptr-val p..+size-of TYPE('b)}

hence access-ti (typ-info-t TYPE('b))
(update-ti-t t (access-ti (typ-info-t TYPE('a)) v (replicate (size-of TYPE('a))
0)) (h-val hp p))
(replicate (size-of TYPE('b)) 0) ! unat (x - ptr-val p) = hp x
proof (subst field-access-update-nth-disjD [OF fl])
have x - ptr-val p ≤ of-nat (size-of TYPE('b) - 1)
proof (rule word-diff-ls(4)[where xa=x and x=x for x, simplified])
from xin show x ≤ of-nat (size-of TYPE('b) - 1) + ptr-val p using al
szb
by (subst add.commute, rule intvl-less-upper)
show ptr-val p ≤ x using xin al szb
by (rule intvl-le-lower)
qed

```

```

thus unx:  $\text{unat } (x - \text{ptr-val } p) < \text{size-td } (\text{typ-info-t } \text{TYPE}('b))$  using szb
b0
apply (simp)
by (metis nat-le-Suc-less add.right-neutral b0 id-apply
len-of-addr-card max-size neq0-conv of-nat-Suc of-nat-eq-id size-of-def
unat-of-nat-minus-1 word-less-eq-iff-unsigned)

show  $\text{unat } (x - \text{ptr-val } p) < n - 0 \vee n - 0 + \text{size-td } t \leq \text{unat } (x - \text{ptr-val } p)$ 
using xin xni
unfolding field-lvalue-def field-lookup-offset-eq [OF fl]
apply -
apply (erule intvl-cut)
apply simp
apply (rule max-size)
done

show wf-fd (typ-info-t TYPE('b)) by (rule wf-fd)

show  $\text{length } (\text{access-ti } (\text{typ-info-t } \text{TYPE}('a)) v (\text{replicate } (\text{size-of } \text{TYPE}('a)) 0)) = \text{size-td } t$ 
using wf-fd [where 'a = 'a]
by (simp add: length-fa-ti size-of-def std)

show  $\text{length } (\text{replicate } (\text{size-of } \text{TYPE}('b)) 0) = \text{size-td } (\text{typ-info-t } \text{TYPE}('b))$ 
by (simp add: size-of-def)

have heap-list hp (size-td (typ-info-t TYPE('b))) (ptr-val p) !  $\text{unat } (x - \text{ptr-val } p) = \text{hp } x$ 
apply (subst heap-list-nth)
apply (rule unx)
apply simp
done

thus  $\text{access-ti } (\text{typ-info-t } \text{TYPE}('b)) (h\text{-val } \text{hp } p) (\text{replicate } (\text{size-of } \text{TYPE}('b)) 0) ! \text{unat } (x - \text{ptr-val } p) = \text{hp } x$ 
unfolding h-val-def
by (simp add: from-bytes-def update-ti-t-def size-of-def field-access-update-same(1)[OF td-fafu-idem wf-fd])
qed
}
hence ... = hp x
by (simp add: to-bytes-p-def to-bytes-def update-ti-update-ti-t length-fa-ti [OF wf-fd] std size-of-def)
finally show hp x = ?RHS by simp
qed
finally show ?LHS = ?RHS .
qed

```

13.6.3 Proof automation for packed types

definition *td-packed* :: ('a,'b) *typ-info* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *bool*
where *td-packed* *t sz al* \longleftrightarrow
td-fafu-idem *t* \wedge *td-fa-hi* *t* \wedge *aggregate* *t* \wedge *size-td* *t* = *sz* \wedge *align-td* *t* = *al*

lemma *packed-type-class-intro*:
td-packed (*typ-info-t* *TYPE*('a::*mem-type*)) *s a*
 \implies *OFCLASS*('a::*mem-type*, *packed-type-class*)
by *standard* (*simp-all* *add*: *td-packed-def*)

lemma *td-fa-hi-map-align*[*simp*]:*td-fa-hi* (*map-align* *f t*) = *td-fa-hi* *t*
by (*cases* *t*) *auto*

lemma *td-packed-final-pad*:
 \llbracket *td-packed* *t s a*; $2 \wedge (\max \text{algn } a) \text{ dvd } s \rrbracket \implies$ *td-packed* (*final-pad* *algn t*) *s* (*max* *algn a*)
by (*simp* *add*: *padup-dvd* [*symmetric*] *td-packed-def* *final-pad-def*)

lemma *td-packed-final-pad'*:
assumes *packed-t*: *td-packed* *t s a*
assumes *le*: *algn* \leq *a*
assumes *dvd*: $2 \wedge a \text{ dvd } s$
shows *td-packed* (*final-pad* *algn t*) *s a*
proof –
from *le* **have** *max* *algn a* = *a* **by** *simp*
from *td-packed-final-pad*[*OF* *packed-t*, *of* *algn*, *simplified* *this*, *OF* *dvd*]
show *?thesis* .
qed

lemma *td-packed-ti-typ-combine*:
 \llbracket *td-packed* (*td*::'a::*c-type* *xtyp-info*) *s a*;
align-of *TYPE*('b::*packed-type*) *dvd s*; *fg-cons* *xf xfu*; *aggregate* *td* \rrbracket
 \implies *td-packed* (*ti-typ-combine* *TYPE*('b) *xf xfu* *algn nm td*)
(*s* + *size-td* (*typ-info-t* *TYPE*('b)))
(*max* *a* (*max* *algn* (*align-td* (*typ-info-t* *TYPE*('b)))))
unfolding *td-packed-def*
apply *safe*
apply (*rule* *td-fafu-idem-ti-typ-combine*; *assumption*?)
apply (*rule* *td-fafu-idem*)
apply (*rule* *td-fa-hi-ti-typ-combine*; *assumption*?)
apply (*rule* *td-fa-hi*)
apply *simp*
apply (*simp* *only*: *size-td-lt-ti-typ-combine*)
apply *simp*
done

lemma *td-packed-ti-typ-pad-combine*:
 \llbracket *td-packed* (*td*::'a::*c-type* *xtyp-info*) *s a*;
align-of *TYPE*('b::*packed-type*) *dvd s*; *algn* \leq *align-td* (*typ-info-t* *TYPE*('b));

fg-cons *xf xfu*; *aggregate td*]]
 \Rightarrow *td-packed* (*ti-typ-pad-combine* *TYPE('b)* *xf xfu* *algn nm td*)
 $(s + \text{size-td } (\text{typ-info-t } \text{TYPE('b)}))$
 $(\max a (\text{align-td } (\text{typ-info-t } \text{TYPE('b)})))$
apply (*subgoal-tac padup* ($\max (2 \hat{\ } \text{algn}) (\text{align-of } \text{TYPE('b)})$) (*size-td td*) = 0)
apply (*simp add: ti-typ-pad-combine-def* *Let-def td-packed-ti-typ-combine*)
apply (*auto simp add: padup-dvd td-packed-def packed-type-intro-simps size-td-lt-ti-typ-combine*
max-2-exp max-absorb2)
done

lemma *td-packed-ti-typ-combine-array*:
[[*td-packed* (*td::'a::c-type xtyp-info*) *s a*;
align-of *TYPE('b::packed-type)* *dvd s*; $0 < \text{CARD('n)}$; $\text{algn} \leq \text{align-td } (\text{typ-info-t } \text{TYPE('b)})$]; *fg-cons* *xf xfu*]]
 \Rightarrow *td-packed*
 $(\text{ti-typ-combine } \text{TYPE('b } ['n :: \text{finite}])$ *xf xfu* *algn nm td*)
 $(s + \text{size-td } (\text{typ-info-t } \text{TYPE('b)}) * \text{CARD('n)})$
 $(\max a (\text{align-td } (\text{typ-info-t } \text{TYPE('b)})))$
apply (*clarsimp simp: ti-typ-combine-def td-packed-def*
packed-type-intro-simps td-fafu-idem-extend-ti
td-fa-hi-extend-ti td-fa-hi-adjust-ti
size-td-extend-ti size-of-def
td-fafu-idem-adjust-ti
align-td-array-info max-absorb2)
done

lemma *td-packed-ti-typ-pad-combine-array*:
[[*td-packed* (*td::'a::c-type xtyp-info*) *s a*;
align-of *TYPE('b::packed-type)* *dvd s*; $0 < \text{CARD('n)}$; $\text{algn} \leq \text{align-td } (\text{typ-info-t } \text{TYPE('b)})$]; *fg-cons* *xf xfu*]]
 \Rightarrow *td-packed* (*ti-typ-pad-combine* *TYPE('b } ['n :: \text{finite}])* *xf xfu* *algn nm td*)
 $(s + \text{size-td } (\text{typ-info-t } \text{TYPE('b)}) * \text{CARD('n)})$
 $(\max a (\text{align-td } (\text{typ-info-t } \text{TYPE('b)})))$
apply (*subgoal-tac padup* (*align-of* *TYPE('b['n])*) (*size-td td*) = 0)
apply (*clarsimp simp add: ti-typ-pad-combine-def* *Let-def td-packed-ti-typ-combine-array*
align-td-array-info align-of-def max-2-exp max-absorb2)
apply (*simp add: td-packed-ti-typ-combine-array*)
apply (*simp add: align-of-def padup-dvd td-packed-def align-td-array*)
done

lemma *td-packed-empty-typ-info*:
td-packed (*empty-typ-info* 0 *fn*) 0 0
apply (*unfold td-packed-def, safe*)
apply (*rule td-fafu-idem-empty-typ-info*)
apply (*rule td-fa-hi-empty-typ-info*)
apply (*rule aggregate-empty-typ-info*)
apply (*rule size-td-empty-typ-info*)
apply (*rule align-of-empty-typ-info'*)

```

done

lemmas td-packed-intros =
  td-packed-final-pad
  td-packed-empty-typ-info
  td-packed-ti-typ-combine
  td-packed-ti-typ-pad-combine
  td-packed-ti-typ-combine-array
  td-packed-ti-typ-pad-combine-array

end

```

13.7 Prettier Printing for Programs

```

theory PrettyProgs
imports Simpl.Vcg
begin

syntax (output)
  -Assign      :: 'b => 'b => ('a,'p,'f) com  ((2- ::= / -) [30, 30] 23)

  -seq::('s,'p,'f) com => ('s,'p,'f) com => ('s,'p,'f) com (-; //- [20, 21] 20)

  -While-inv  :: 'a bexp => 'a assn => bdy => ('a,'p,'f) com
    ((0WHILE (-)//INV (-)//-) [25, 0, 81] 71)

  -Do :: ('a,'p,'f) com => bdy (DO// (-)//OD [0] 1000)

  -Cond      :: 'a bexp => ('a,'p,'f) com => ('a,'p,'f) com => ('a,'p,'f) com
    ((0IF - THEN// (-)//ELSE// (-)//FI) [0, 0, 0] 71)
  -Cond-no-else:: 'a bexp => ('a,'p,'f) com => ('a,'p,'f) com
    ((0IF - THEN// (-)//FI) [0, 0] 71)

  -Try-Catch:: ('a,'p,'f) com => ('a,'p,'f) com => ('a,'p,'f) com
    ((0TRY// (-)//CATCH -//END) [0,0] 71)

end

```

```

theory StaticFun
imports
  Main
begin

datatype ('a, 'b) Tree = Node 'a 'b ('a, 'b) Tree ('a, 'b) Tree | Leaf

primrec
  lookup-tree :: ('a, 'b) Tree => ('a => 'c :: linorder) => 'a => 'b option
where

```

```

lookup-tree Leaf fn x = None
| lookup-tree (Node y v l r) fn x = (if fn x = fn y then Some v
                                     else if fn x < fn y then lookup-tree l fn x
                                     else lookup-tree r fn x)

```

definition *optional-strict-range* :: ('a :: linorder) option ⇒ 'a option ⇒ 'a set
where

```

optional-strict-range x y = {z. (x = None ∨ the x < z) ∧ (y = None ∨ z < the
y)}

```

lemma *optional-strict-range-split*:

```

z ∈ optional-strict-range x y
⇒ optional-strict-range x (Some z) = ({..< z} ∩ optional-strict-range x y)
  ∧ optional-strict-range (Some z) y = ({z <..} ∩ optional-strict-range x y)
by (auto simp add: optional-strict-range-def)

```

lemma *optional-strict-rangeI*:

```

z ∈ optional-strict-range None None
z < y ⇒ z ∈ optional-strict-range None (Some y)
x < z ⇒ z ∈ optional-strict-range (Some x) None
x < z ⇒ z < y ⇒ z ∈ optional-strict-range (Some x) (Some y)
by (simp-all add: optional-strict-range-def)

```

definition

```

tree-eq-fun-in-range :: ('a, 'b) Tree ⇒ ('a ⇒ 'c :: linorder) ⇒ ('a → 'b) ⇒ 'c set
⇒ bool

```

where

```

tree-eq-fun-in-range T ord f S ≡ ∀x. (ord x ∈ S) → f x = lookup-tree T ord x

```

lemma *tree-eq-fun-in-range-from-def*:

```

[[ f ≡ lookup-tree T ord ]]
⇒ tree-eq-fun-in-range T ord f (optional-strict-range None None)
by (simp add: tree-eq-fun-in-range-def)

```

lemma *tree-eq-fun-in-range-split*:

```

tree-eq-fun-in-range (Node z v l r) ord f (optional-strict-range x y)
⇒ ord z ∈ optional-strict-range x y
⇒ tree-eq-fun-in-range l ord f (optional-strict-range x (Some (ord z)))
  ∧ f z = Some v
  ∧ tree-eq-fun-in-range r ord f (optional-strict-range (Some (ord z)) y)

```

apply (simp add: tree-eq-fun-in-range-def optional-strict-range-split)

apply fastforce

done

ML ‹

```

structure StaticFun = struct

```

```

(* Actually build the tree -- theta (n lg(n)) *)

```

```

fun build-tree' - mk-leaf [] = mk-leaf
| build-tree' mk-node mk-leaf xs = let
  val len = length xs
  val (ys, zs) = chop (len div 2) xs
in case zs of [] => error build-tree': impossible
  | ((a, b) :: zs) => mk-node a b (build-tree' mk-node mk-leaf ys)
  (build-tree' mk-node mk-leaf zs)
end

fun build-tree ord xs = case xs of [] => error build-tree : empty
| (idx, v) :: - => let
  val idxT = fastype-of idx
  val vT = fastype-of v
  val treeT = Type (@{type-name StaticFun.Tree}, [idxT, vT])
  val mk-leaf = Const (@{const-name StaticFun.Leaf}, treeT)
  val node = Const (@{const-name StaticFun.Node},
    idxT ---> vT ---> treeT ---> treeT ---> treeT)
  fun mk-node a b l r = node $ a $ b $ l $ r
  val lookup = Const (@{const-name StaticFun.lookup-tree},
    treeT ---> fastype-of ord ---> idxT
    ---> Type (@{type-name option}, [vT]))
in
  lookup $ (build-tree' mk-node mk-leaf xs) $ ord
end

fun define-partial-map-tree name mappings ord ctxt = let
  val tree = build-tree ord mappings
in Local-Theory.define
  ((name, NoSyn), ((Thm.def-binding name, []), tree)) ctxt
  |> apfst (apsnd snd)
end

fun prove-partial-map-thms thm ctxt = let
  val init = thm RS @{thm tree-eq-fun-in-range-from-def}
  fun rec-tree thm = case Thm.concl-of thm of
  @{term Trueprop} $ (Const (@{const-name tree-eq-fun-in-range}, -)
    $ (Const (@{const-name Node}, -) $ z $ v $ - $ -) $ - $ -) => let
    val t' = thm RS @{thm tree-eq-fun-in-range-split}
    val solve-simp-tac = SUBGOAL (fn (t, i) =>
      (simp-tac ctxt THEN-ALL-NEW SUBGOAL (fn (t', -) =>
        raise TERM (prove-partial-map-thms: unsolved, [t, t']))) i)
    val r = t' |> (resolve-tac ctxt @{thms optional-strict-rangeI}
      THEN-ALL-NEW solve-simp-tac) 1 |> Seq.hd
    val l = r RS @{thm conjunct1}
    val kr = r RS @{thm conjunct2}
    val k = kr RS @{thm conjunct1}
    val r = kr RS @{thm conjunct2}
    in rec-tree l @ [(z, v), k] @ rec-tree r end
  | - => []

```

```

in rec-tree init end

fun define-tree-and-save-thms name names mappings ord exsimps ctxt = let
  val ((tree, def-thm), ctxt) = define-partial-map-tree name mappings ord ctxt
  val thms = prove-partial-map-thms def-thm (ctxt addsimps exsimps)
  val (idents, thms) = map-split I thms
  val - = map (fn ((x, y), (x', y')) => (x aconv x' andalso y aconv y')
    or else raise TERM (define-tree-and-thms: different, [x, y, x', y']))
    (mappings ~~ idents)
  val (-, ctxt) = Local-Theory.notes
    (map (fn (s, t) => ((Binding.name s, []), [(t, [])]))
      (names ~~ thms)) ctxt
in (tree, ctxt) end

fun define-tree-and-thms-with-defs name names key-defs opt-values ord ctxt = let
  val data = names ~~ (key-defs ~~ opt-values)
  |> map-filter (fn (-, (-, NONE)) => NONE | (nm, (thm, SOME v))
    => SOME (nm, (fst (Logic.dest-equals (Thm.concl-of thm)), v)))
  val (names, mappings) = map-split I data
in define-tree-and-save-thms name names mappings ord key-defs ctxt end

end

›

end

theory IndirectCalls

imports
  PrettyProgs

begin

lemma hoare-indirect-call-known-proc:
  assumes spec:  $\Gamma \vdash P$  (call-exn init q return result-exn c)  $Q, A$ 
  shows  $\Gamma \vdash (\{s. p\ s = q\} \cap P)$  (dynCall-exn f UNIV init p return result-exn c)  $Q, A$ 
  using spec
  apply -
  apply (rule hoare-complete, drule hoare-sound)
  apply (clarsimp simp: cvalid-def HoarePartialDef.valid-def dynCall-exn-def)
  apply (auto elim: exec-Normal-elim-cases)
  done

lemma hoare-indirect-call-guard:

```

```

assumes conseq:  $P \subseteq g \cap R$ 
assumes spec:  $\Gamma \vdash R$  (dynCall-exn f UNIV init p return result-exn c) Q,A
shows  $\Gamma \vdash P$  (dynCall-exn f g init p return result-exn c) Q,A
  using spec conseq
  apply –
  apply (rule hoare-complete, drule hoare-sound)
  apply (clarsimp simp: cvalid-def HoarePartialDef.valid-def dynCall-exn-def maybe-guard-def)
  apply (cases g = UNIV)
  subgoal
    by force
  subgoal
    apply (clarsimp elim: exec-Normal-elim-cases )
    by (meson exec-Normal-elim-cases(5) imageI subsetD)
  done

```

end

```

theory ModifiesProofs
imports CLanguage
begin

```

definition

```

  modifies-inv-refl :: ('a  $\Rightarrow$  'a set)  $\Rightarrow$  bool
where
  modifies-inv-refl P  $\equiv \forall x. x \in P$ 

```

definition

```

  modifies-inv-incl :: ('a  $\Rightarrow$  'a set)  $\Rightarrow$  bool
where
  modifies-inv-incl P  $\equiv \forall x y. y \in P \ x \longrightarrow P$   $y \subseteq P$ 

```

definition

```

  modifies-inv-prop :: ('a  $\Rightarrow$  'a set)  $\Rightarrow$  bool
where
  modifies-inv-prop P  $\equiv$  modifies-inv-refl P  $\wedge$  modifies-inv-incl P

```

lemma *modifies-inv-prop*:

```

  modifies-inv-refl P  $\Longrightarrow$  modifies-inv-incl P  $\Longrightarrow$  modifies-inv-prop P
by (simp add: modifies-inv-prop-def)

```

named-theorems *modifies-inv-intros*

locale *modifies-assertion* =

```

  fixes P :: 's  $\Rightarrow$  's set
  assumes p: modifies-inv-prop P
begin

```

lemmas *modifies-inv-prop'* =
p[unfolded modifies-inv-prop-def modifies-inv-refl-def modifies-inv-incl-def]

lemma *modifies-inv-prop-lift*:
assumes $c: \bigwedge \sigma. \Gamma, \Theta \vdash_{/F} \{\sigma\} c (P \sigma), (P \sigma)$
shows $\Gamma, \Theta \vdash_{/F} (P \sigma) c (P \sigma), (P \sigma)$
using *modifies-inv-prop'* **by** (*fastforce intro: c hoarep.Conseq*)

lemma *modifies-inv-prop-lower*:
assumes $c: \bigwedge \sigma. \Gamma, \Theta \vdash_{/F} (P \sigma) c (P \sigma), (P \sigma)$
shows $\Gamma, \Theta \vdash_{/F} \{\sigma\} c (P \sigma), (P \sigma)$
using *modifies-inv-prop'* **by** (*fastforce intro: c hoarep.Conseq*)

Note that the C-Parser associates sequential composition to the right. So the first statement is typically already an 'atomic' statement (or at least no further sequential composition) that can be solved. We place it as the second precondition because the *modifies*-tactic follows the canonical order of tactical reasoning and solves the subgoals from the back. So *c1* is already solved before further decomposing *c2*. This keeps the number of subgoals (and thus the overall goal-state) small.

lemma *modifies-inv-Seq* [*modifies-inv-intros*]:
assumes $c: \bigwedge \sigma. \Gamma, \Theta \vdash_{/F} \{\sigma\} c2 (P \sigma), (P \sigma) \bigwedge \sigma. \Gamma, \Theta \vdash_{/F} \{\sigma\} c1 (P \sigma), (P \sigma)$
shows $\Gamma, \Theta \vdash_{/F} \{\sigma\} c1 ;; c2 (P \sigma), (P \sigma)$
by (*intro modifies-inv-prop-lower HoarePartial.SeqSwap[OF c[THEN modifies-inv-prop-lift]]*)

lemma *modifies-inv-Cond* [*modifies-inv-intros*]:
assumes $c: \bigwedge \sigma. \Gamma, \Theta \vdash_{/F} \{\sigma\} c1 (P \sigma), (P \sigma) \bigwedge \sigma. \Gamma, \Theta \vdash_{/F} \{\sigma\} c2 (P \sigma), (P \sigma)$
shows $\Gamma, \Theta \vdash_{/F} \{\sigma\} \text{Cond } b c1 c2 (P \sigma), (P \sigma)$
by (*auto intro: HoarePartial.Cond c*)

lemma *modifies-inv-Guard-strip* [*modifies-inv-intros*]:
assumes $c: \bigwedge \sigma. \Gamma, \Theta \vdash_{/UNIV} \{\sigma\} c (P \sigma), (P \sigma)$
shows $\Gamma, \Theta \vdash_{/UNIV} \{\sigma\} \text{Guard } f b c (P \sigma), (P \sigma)$
by (*rule HoarePartial.GuardStrip[OF subset-refl c UNIV-I]*)

lemma *modifies-inv-Skip* [*modifies-inv-intros*]:
shows $\Gamma, \Theta \vdash_{/F} \{\sigma\} \text{SKIP} (P \sigma), (P \sigma)$
using *modifies-inv-prop'* **by** (*auto intro: modifies-inv-prop-lift HoarePartial.Skip*)

lemma *modifies-inv-Skip'* [*modifies-inv-intros*]:
shows $\Gamma, \Theta \vdash_{/F} \{\sigma\} \text{SKIP} (P \sigma)$
using *modifies-inv-prop'* **by** (*auto intro: modifies-inv-prop-lift HoarePartial.Skip*)

lemma *modifies-inv-whileAnno* [*modifies-inv-intros*]:
assumes $c: \bigwedge \sigma. \Gamma, \Theta \vdash_{/F} \{\sigma\} c (P \sigma), (P \sigma)$

shows $\Gamma, \Theta \vdash_F \{\sigma\}$ *whileAnno* $b \ I \ V \ c \ (P \ \sigma), (P \ \sigma)$
apply (*rule* *HoarePartial.reannotateWhileNoGuard*[**where** $I=P \ \sigma$])
by (*intro* *HoarePartial.While* *hoarep.Conseq*;
fastforce simp: modifies-inv-prop' *intro: modifies-inv-prop-lift* c)

lemma *modifies-inv-While* [*modifies-inv-intros*]:
assumes $c: \bigwedge \sigma. \Gamma, \Theta \vdash_F \{\sigma\} \ c \ (P \ \sigma), (P \ \sigma)$
shows $\Gamma, \Theta \vdash_F \{\sigma\}$ *While* $b \ c \ (P \ \sigma), (P \ \sigma)$
by (*intro* *modifies-inv-whileAnno*[*unfolded whileAnno-def*] c)

lemma *modifies-inv-Throw* [*modifies-inv-intros*]:
shows $\Gamma, \Theta \vdash_F \{\sigma\}$ *THROW* $(P \ \sigma), (P \ \sigma)$
using *modifies-inv-prop'* **by** (*auto intro: modifies-inv-prop-lift* *HoarePartial.Throw*)

lemma *modifies-inv-Catch* [*modifies-inv-intros*]:
assumes $c: \bigwedge \sigma. \Gamma, \Theta \vdash_F \{\sigma\} \ c1 \ (P \ \sigma), (P \ \sigma)$
 $\bigwedge \sigma. \Gamma, \Theta \vdash_F \{\sigma\} \ c2 \ (P \ \sigma), (P \ \sigma)$
shows $\Gamma, \Theta \vdash_F \{\sigma\}$ *TRY* $c1 \ CATCH \ c2 \ END \ (P \ \sigma), (P \ \sigma)$
by (*intro* *modifies-inv-prop-lower* *HoarePartial.Catch*[*OF* c [*THEN* *modifies-inv-prop-lift*]])

lemma *modifies-inv-Catch-all* [*modifies-inv-intros*]:
assumes $1: \bigwedge \sigma. \Gamma, \Theta \vdash_F \{\sigma\} \ c1 \ (P \ \sigma), (P \ \sigma)$
assumes $2: \bigwedge \sigma. \Gamma, \Theta \vdash_F \{\sigma\} \ c2 \ (P \ \sigma)$
shows $\Gamma, \Theta \vdash_F \{\sigma\}$ *TRY* $c1 \ CATCH \ c2 \ END \ (P \ \sigma)$
apply (*intro* *HoarePartial.Catch*[*OF* 1] *hoarep.Conseq*, *clarsimp*)
apply (*metis* *modifies-inv-prop'* 2 *singletonI*)
done

lemma *modifies-inv-switch-Nil* [*modifies-inv-intros*]:
shows $\Gamma, \Theta \vdash_F \{\sigma\}$ *switch* $v \ [] \ (P \ \sigma), (P \ \sigma)$
by (*auto intro: modifies-inv-Skip*)

lemma *modifies-inv-switch-Cons* [*modifies-inv-intros*]:
assumes $c: \bigwedge \sigma. \Gamma, \Theta \vdash_F \{\sigma\} \ c \ (P \ \sigma), (P \ \sigma)$
 $\bigwedge \sigma. \Gamma, \Theta \vdash_F \{\sigma\} \ switch \ p \ vcs \ (P \ \sigma), (P \ \sigma)$
shows $\Gamma, \Theta \vdash_F \{\sigma\}$ *switch* $p \ ((v, c) \# \ vcs) \ (P \ \sigma), (P \ \sigma)$
by (*auto intro: c* *modifies-inv-Cond*)

end

locale *modifies-state-assertion* = *modifies-assertion* P **for**
 $P :: ('g, 'l, 'e, 'x) \ state\ scheme \Rightarrow ('g, 'l, 'e, 'x) \ state\ scheme \ set +$
assumes $p: \text{modifies-inv-prop } P$
begin

lemma *modifies-inv-creturn* [*modifies-inv-intros*]:
assumes $c: \bigwedge \sigma. \Gamma, \Theta \vdash_F \{\sigma\} \ Basic \ (\lambda s. \ xfu \ (\lambda -. \ v \ s) \ s) \ (P \ \sigma), (P \ \sigma)$

$\wedge \sigma. \Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ Basic } (rtu (\lambda-. \text{Return})) (P \sigma), (P \sigma)$
shows $\Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ creturn } rtu \text{ xfu } v (P \sigma), (P \sigma)$
unfolding *creturn-def* **by** (*intro p c modifies-inv-intros*)

lemma *modifies-inv-creturn-void* [*modifies-inv-intros*]:
assumes $c: \wedge \sigma. \Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ Basic } (rtu (\lambda-. \text{Return})) (P \sigma), (P \sigma)$
shows $\Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ creturn-void } rtu (P \sigma), (P \sigma)$
unfolding *creturn-void-def* **by** (*intro p c modifies-inv-intros*)

lemma *modifies-inv-cbreak* [*modifies-inv-intros*]:
assumes $c: \wedge \sigma. \Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ Basic } (rtu (\lambda-. \text{Break})) (P \sigma), (P \sigma)$
shows $\Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ cbreak } rtu (P \sigma), (P \sigma)$
unfolding *cbreak-def* **by** (*intro p c modifies-inv-intros*)

lemma *modifies-inv-ccatchbrk* [*modifies-inv-intros*]:
shows $\Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ ccatchbrk } rt (P \sigma), (P \sigma)$
unfolding *ccatchbrk-def* **by** (*intro p modifies-inv-intros*)

lemma *modifies-inv-cgoto* [*modifies-inv-intros*]:
assumes $c: \wedge \sigma. \Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ Basic } (rtu (\lambda-. \text{Goto } l)) (P \sigma), (P \sigma)$
shows $\Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ cgoto } l \text{ rtu } (P \sigma), (P \sigma)$
unfolding *cgoto-def* **by** (*intro p c modifies-inv-intros*)

lemma *modifies-inv-ccatchgoto* [*modifies-inv-intros*]:
shows $\Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ ccatchgoto } l \text{ rt } (P \sigma), (P \sigma)$
unfolding *ccatchgoto-def* **by** (*intro p modifies-inv-intros*)

lemma *modifies-inv-ccatchreturn* [*modifies-inv-intros*]:
shows $\Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ ccatchreturn } rt (P \sigma), (P \sigma)$
unfolding *ccatchreturn-def* **by** (*intro p modifies-inv-intros*)

end

lemma *On-Exit-wp*:
assumes *cleanup*: $\Gamma, \Theta \vdash_{/F} R \text{ cleanup } Q, A$
assumes *cleanup-catch*: $\Gamma, \Theta \vdash_{/F} B \text{ cleanup } A, A$
assumes $c: \Gamma, \Theta \vdash_{/F} P \text{ c } R, B$
shows $\Gamma, \Theta \vdash_{/F} P \text{ On-Exit } c \text{ cleanup } Q, A$
unfolding *On-Exit-def*
apply (*rule HoarePartial.SeqSwap* [**where** $R=R$])
apply (*rule HoarePartial.conseq-no-aux*)
apply (*rule cleanup*)
apply *simp*
apply (*rule HoarePartial.Catch*)
apply (*rule c*)
apply (*rule HoarePartial.SeqSwap*)
apply (*rule HoarePartial.Throw*)

apply (*rule subset-refl*)
apply (*rule cleanup-catch*)
done

lemma *DynCom-fix-pre*: $\forall s \in P. \Gamma, \Theta \vdash /_F \{s\} (c\ s) Q, A$

\implies
 $\Gamma, \Theta \vdash /_F P (DynCom\ c) Q, A$

by (*smt (verit, best) HoarePartial.conseq-exploit-pre Int-insert-left-if1 hoarep.DynCom inf-bot-left singletonD*)

lemma $\Gamma, \Theta \vdash /_F \{s. (\forall t. (s, t) \in r \longrightarrow t \in Q) \wedge (\exists t. (s, t) \in r)\} Spec\ r\ Q, A$

by (*rule hoarep.Spec*)

lemma *hoarep-Spec-fixed*: $(\exists t. (s, t) \in r) \implies \Gamma, \Theta \vdash /_F \{s\} Spec\ r\ \{t. (s, t) \in r\}, A$

by (*simp add: HoarePartial.Spec*)

context *stack-heap-state*

begin

lemma *With-Fresh-Stack-Ptr-tight*:

assumes *c*: $\bigwedge s\ p\ d\ vs\ bs. s \in P \implies vs \in init\ s \implies length\ vs = n \implies length\ bs = n * size-of\ TYPE('a) \implies$

$(p, d) \in stack-allocs\ n\ \mathcal{S}\ TYPE('a::mem-type) (htd\ s) \implies$

$\Gamma, \Theta \vdash /_F \{hmem-upd (fold (\lambda i. heap-update-padding (p +_p\ int\ i) (vs!i) (take (size-of\ TYPE('a)) (drop (i * size-of\ TYPE('a))\ bs))) [0..<n]) (htd-upd (\lambda -. d) s)\}$

$(c\ p)$

$\{t. \forall bs. length\ bs = n * size-of\ TYPE('a) \longrightarrow hmem-upd (heap-update-list (ptr-val\ p) bs) (htd-upd (stack-releases\ n\ p) t) \in Q\},$

$\{t. \forall bs. length\ bs = n * size-of\ TYPE('a) \longrightarrow hmem-upd (heap-update-list (ptr-val\ p) bs) (htd-upd (stack-releases\ n\ p) t) \in A\}$

assumes *no-overflow*: $StackOverflow \in F$

shows $\Gamma, \Theta \vdash /_F P\ With-Fresh-Stack-Ptr\ n\ init\ c\ Q, A$

unfolding *With-Fresh-Stack-Ptr-def*

apply (*rule hoarep.Guarantee [OF no-overflow]*)

apply (*rule DynCom-fix-pre*)

apply (*rule ballI*)

apply *clarsimp*

apply (*rule HoarePartial.Seq*)

apply (*rule hoarep-Spec-fixed*)

apply *clarsimp*

subgoal using *Ex-list-of-length*

by (*metis equals0I old.prod.exhaust*)

apply *clarsimp*

apply (*rule DynCom-fix-pre*)

apply (*rule ballI*)

```

apply clarsimp
apply (rule On-Exit-wp)

    apply (rule HoarePartial.Spec)
    apply (rule subset-refl)
    apply (rule HoarePartial.Spec)
    apply (rule subset-refl)
    subgoal premises prems for s vs p d' vsa bs
    proof –
      have eq1: length vs = n using prems by simp
      have eq2:(allocated-ptrs n S TYPE('a) (htd s) d') = p using prems by (simp
add: stack-allocs-allocated-ptrs)
      show ?thesis
      apply (simp add: eq1 eq2 Ex-list-of-length)
      apply (rule HoarePartialDef.conseqPost [OF c [where s1=s and vs1=vsa
and d1=d' and p1=p]])
      using prems
      apply auto
    done
  qed
done

```

lemma *With-Fresh-Stack-Ptr-tight-wp:*

```

assumes conseq:  $\bigwedge s p d vs bs. s \in P \implies vs \in \text{init } s \implies \text{length } vs = n \implies$ 
length bs = n * size-of TYPE('a)  $\implies$ 
  (p, d)  $\in$  stack-allocs n S TYPE('a::mem-type) (htd s)  $\implies$ 
  (hmem-upd (fold ( $\lambda i. \text{heap-update-padding } (p +_p \text{ int } i) (vs!i) (take (size-of$ 
TYPE('a)) (drop (i * size-of TYPE('a)) bs))) [0..<n]) (htd-upd ( $\lambda-. d) s) \in R s$ 
p d vs)
assumes c:  $\bigwedge s p d vs bs. s \in P \implies vs \in \text{init } s \implies \text{length } vs = n \implies \text{length}$ 
bs = n * size-of TYPE('a)  $\implies$ 
  (p, d)  $\in$  stack-allocs n S TYPE('a::mem-type) (htd s)  $\implies$ 
  (hmem-upd (fold ( $\lambda i. \text{heap-update-padding } (p +_p \text{ int } i) (vs!i) (take (size-of$ 
TYPE('a)) (drop (i * size-of TYPE('a)) bs))) [0..<n]) (htd-upd ( $\lambda-. d) s) \in R s$ 
p d vs  $\implies$ 
   $\Gamma, \Theta \vdash_{/F} (R s p d vs) (c p)$ 
  {t.  $\forall bs. \text{length } bs = n * \text{size-of TYPE('a)} \longrightarrow \text{hmem-upd (heap-update-list$ 
(ptr-val p) bs) (htd-upd (stack-releases n p) t) \in Q},
  {t.  $\forall bs. \text{length } bs = n * \text{size-of TYPE('a)} \longrightarrow \text{hmem-upd (heap-update-list$ 
(ptr-val p) bs) (htd-upd (stack-releases n p) t) \in A}
assumes no-overflow: StackOverflow  $\in F$ 
shows  $\Gamma, \Theta \vdash_{/F} P$  With-Fresh-Stack-Ptr n init c Q, A
apply (rule With-Fresh-Stack-Ptr-tight [OF - no-overflow])
subgoal for s p d vs bs
  apply (rule HoarePartial.conseq)
  apply (rule allI)

```

```

apply (rule c[where  $s1=s$  and  $vs1=vs$  and  $d1=d$ ])
  apply assumption
  apply assumption
  apply assumption
  apply assumption
  apply assumption
apply (rule conseq)
  apply assumption
  apply assumption
  apply assumption
  apply assumption
apply clarsimp
subgoal premises prems
proof -
  from prems have  $eq: \text{length } vs = n$  by simp
  show ?thesis
    apply (simp add: eq)
    apply (rule conseq)
    using prems by auto
qed
done
done

```

Caution: this WP-setup was developed to solve the modified clauses. It might not be the best fit for WP-style reasoning in general. Also note that it is currently not invoked by the automatic modifies-proofs triggered by the C-parser as *With-Fresh-Stack-Ptr* is first decomposed by the (see the rule below).

```

declaration <fn phi =>
  let
    val rule = Morphism.thm phi @{thm With-Fresh-Stack-Ptr-tight}
    fun tac cont-tac ctxt mode i =
      resolve-tac ctxt [rule] i THEN
      SOLVED-verbose no-overflow ctxt (simp-tac ctxt) (i + 1) THEN
      cont-tac true i
  in
    Hoare.add-hoare-tacs [(With-Fresh-Stack-Ptr-tac, tac)]
  end
  >
end

```

```

locale modifies-assertion-stack-heap-state =
  modifies-assertion P + stack-heap-state htd htd-upd hmem hmem-upd S
for  $P::'s \Rightarrow 's$  set and
  htd::  $'s \Rightarrow \text{heap-tyt-desc}$  and
  htd-upd:: ( $\text{heap-tyt-desc} \Rightarrow \text{heap-tyt-desc}$ )  $\Rightarrow 's \Rightarrow 's$  and
  hmem::  $'s \Rightarrow \text{heap-mem}$  and
  hmem-upd:: ( $\text{heap-mem} \Rightarrow \text{heap-mem}$ )  $\Rightarrow 's \Rightarrow 's$  and

```

$\mathcal{S}::\text{addr set} +$
assumes *hmem-upd-inv*: $\bigwedge \sigma m. \text{hmem-upd } m \sigma \in (P \sigma)$
assumes *htd-upd-inv*: $\bigwedge \sigma d. \text{htd-upd } d \sigma \in (P \sigma)$
begin

lemma *modifies-With-Fresh-Stack-Ptr* [*modifies-inv-intros*]:
assumes *no-overflow*: $\text{StackOverflow} \in F$
assumes *c*: $\bigwedge \sigma p. \Gamma, \Theta \vdash_{/F} \{\sigma\} (c (p::'a::\text{mem-type ptr})) (P \sigma), (P \sigma)$
shows $\Gamma, \Theta \vdash_{/F} \{\sigma\} \text{With-Fresh-Stack-Ptr } n \text{ init } c (P \sigma), (P \sigma)$
apply (*rule With-Fresh-Stack-Ptr-tight-wp* [**where** $R = \lambda s p d v. P s$])
subgoal for $s p d vs bs$
proof –
have $(\text{htd-upd } (\lambda-. d) s) \in P s$ **by** (*rule htd-upd-inv*)
moreover have $\text{hmem-upd } (\text{fold } (\lambda i. \text{heap-update-padding } (p +_p \text{int } i) (vs!i)) (\text{take } (\text{size-of TYPE('a)}) (\text{drop } (i * \text{size-of TYPE('a)}) bs))) [0..<n]) (\text{htd-upd } (\lambda-. d) s) \in P (\text{htd-upd } (\lambda-. d) s)$
by (*rule hmem-upd-inv*)
ultimately
show $\text{hmem-upd } (\text{fold } (\lambda i. \text{heap-update-padding } (p +_p \text{int } i) (vs!i)) (\text{take } (\text{size-of TYPE('a)}) (\text{drop } (i * \text{size-of TYPE('a)}) bs))) [0..<n]) (\text{htd-upd } (\lambda-. d) s) \in P s$
using *modifies-inv-prop' p* **by** *blast*
qed
subgoal for $s p d v$
apply (*rule HoarePartial.conseq* [**where** $P'=P$ and $Q'=P$ and $A'=P$])
apply (*rule allI*)
subgoal for Z
apply (*rule modifies-inv-prop-lift*)
apply (*rule c*)
done
using *modifies-inv-prop' hmem-upd-inv htd-upd-inv*
by (*smt (verit, best) mem-Collect-eq singletonD subset-eq*)
apply (*rule no-overflow*)
done

end

locale *modifies-assertion-stack-heap-raw-state* =
modifies-assertion $P + \text{stack-heap-raw-state } t\text{-hrs } t\text{-hrs-update } \mathcal{S}$
for $P::'s \Rightarrow 's$ **set and**
 $t\text{-hrs}:: 's \Rightarrow \text{heap-raw-state}$ **and**
 $t\text{-hrs-update}:: (\text{heap-raw-state} \Rightarrow \text{heap-raw-state}) \Rightarrow 's \Rightarrow 's$ **and**
 $\mathcal{S}::\text{addr set} +$
assumes *hrs-upd-inv*: $\bigwedge \sigma m. t\text{-hrs-update } m \sigma \in (P \sigma)$
begin

sublocale *modifies-assertion-stack-heap-state*
 P

```

λs. hrs-htd (t-hrs s) λupd. t-hrs-update (hrs-htd-update upd)
λs. hrs-mem (t-hrs s) λupd. t-hrs-update (hrs-mem-update upd)
S
using hrs-upd-inv by unfold-locales auto

end

end

```

13.8 Modelling Local Variables

```

theory CLocals
  imports UMM
    HOL-Library.Code-Binary-Nat
    ML-Record-Antiquotation
begin

ML ‹
structure Code-Simproc =
struct

val meta-eq-to-prop = @{lemma ‹(PROP P ≡ Trueprop True) ⇒ PROP P› for
P by simp}

fun solved-eq eq =
  case Thm.prop-of eq of
    @{term-pat - ≡ Trueprop True} => true
  | - => false

fun code-prove ctxt prop =
  let
    val code-eq = SOME (Code-Runtime.dynamic-holds-conv ctxt prop) handle ER-
ROR x => (warning x; NONE)
    val - = Utils.verbose-msg 4 ctxt (fn - => code-prove: ^@{make-string} code-eq)
  in
    code-eq |> Option.mapPartial (fn eq =>
      if solved-eq eq then
        SOME (meta-eq-to-prop OF [Code-Simp.dynamic-conv ctxt prop])
      else NONE)
  end

fun fill-default default xs =
  let
    fun fill ys - [] = rev ys
      | fill ys n (zs as ((i, z)::zs')) =
        if n = i then
          fill (z::ys) (n + 1) zs'

```

```

      else
        if n < i then fill (default::ys) (n + 1) zs
        else error (fill-default: expecting indexes in ascending order ^@{make-string}
(map fst xs))
    in
      fill [] 0 xs
    end

fun code-simp-prems cond-eq-rule0 ps ctxt ct =
  let
    val cond-eq-rule = Thm.transfer' ctxt cond-eq-rule0
    val - = Utils.verbose-msg 4 ctxt (fn - => code-simp-prems: ^@{make-string}
(ct, cond-eq-rule))
    val ps = sort int-ord ps
    val lhs = cond-eq-rule |> Thm.cconcl-of |> Thm.dest-equals-lhs
    val eq = Thm.instantiate (Thm.match (lhs, ct)) cond-eq-rule
    val prems = Thm.cprems-of eq
    fun solve i = code-prove ctxt (nth prems i)
    val solved = map-filter solve ps
  in
    if length solved = length ps then
      SOME (eq OF (fill-default Drule.asm-rl (ps ~~~ solved)))
    else
      NONE
  end

fun is-eq (@{term-pat Trueprop (- = -)}) = true
  | is-eq (@{term-pat - ≡ -}) = true
  | is-eq - = false

fun positions-of len interval =
  case interval of
    Facts.FromTo (i, j) => (@{assert} (j <= len) ; (i - 1) upto (j - 1))
  | Facts.From i => (@{assert} (i <= len) ; (i - 1) upto (len - 1))
  | Facts.Single i => (@{assert} (i <= len) ; [i - 1])

fun positions-of-intervals len =
  map (positions-of len) #> flat #> sort-distinct int-ord

fun lhs-pattern ctxt eq =
  let
    val ((-, [eq]), -) = Variable.import false [eq] ctxt
  in
    eq |> Thm.concl-of |> Logic.dest-equals |> fst
  end

structure Data = Generic-Data (
  type T = simproc list Symtab.table
  val empty = Symtab.empty

```

```

    val merge = Syntab.merge (K false))

fun get-simprocs ctxt named-thms =
  let
    val simprocs = Data.get (Context.Proof ctxt)
  in
    Syntab.lookup-list simprocs named-thms
  end

(* piggy back name-space of named theorems *)
fun check context named-thms =
  let
    val ctxt = Context.proof-of context
    val facts = (Proof-Context.facts-of ctxt);
  in
    Facts.check context facts named-thms
  end

fun add-simproc pos named-thms (intervals: Facts.interval list) thm context =
  let
    val ctxt = Context.proof-of context
    val named-thms = if fst (named-thms) = then else (check context named-thms
)
    val - = is-eq (Thm.concl-of thm) orelse error (Code-Simproc.add-simproc: con-
clusion is not an equality)
    val - = Thm.has-name-hint thm orelse error (Code-Simproc.add-simproc: theo-
rem has no name)
    val name-hint = Thm.get-name-hint thm
    val base-name = Long-Name.base-name name-hint
    val thm1 = safe-mk-meta-eq thm
    val nprems = Thm.nprems-of thm1
    val - = nprems > 0 orelse error (Code-Simproc.add-simprocs: theorem has no
premisses)
    val positions = if null intervals then 0 upto nprems - 1 else positions-of-intervals
nprems intervals
    val patterns = thm1 |> lhs-pattern ctxt |> single
    val thm2 = Thm.trim-context thm1
  in
    if Local-Theory.level ctxt = 0 then context
    else
      context
      |> Context.map-proof-result (Simplifier.define-simproc {name = Binding.make
(base-name, pos), passive=false, identifier=[],
      lhs = patterns, proc = fn - => code-simp-prems thm2 positions})
      |-> (fn simproc => Context.map-proof (
        Local-Theory.declaration {pervasive=false, syntax=false, pos = here }
(fn - =>
          (Data.map (Syntab.map-default (named-thms, [simproc]) (cons sim-
proc))))))

```



```

    end
  end
>

```

```

setup <
  let
    fun position scan = (Scan.ahead (Scan.one (K true)) >> Token.pos-of) -- scan
  >> Library.swap;
  in
    Attrib.setup binding <code-simproc>
      (Scan.lift (position
        (Scan.optional Parse.name-position (, Position.none) -- Scan.optional Parse.thm-sel
          [])) >>
        (fn ((named-thms, intervals), pos) =>
          Thm.declaration-attribute (fn thm => Code-Simproc.add-simproc pos named-thms
            intervals thm))))
      solve preconditions with code-simp #>
    ML-Antiquotation.value-embedded binding <code-simprocs>
      (Args.context -- Scan.lift Parse.embedded-position >>
        (fn (ctxt, name) => Code-Simproc.get-simprocs ML-context ^
          ML-Syntax.print-string (Code-Simproc.check (Context.Proof ctxt) name)))
  end
>

```

```

ML <
  String.str
>

```

```

type-synonym locals = nat => byte list

```

```

definition clookup :: nat => locals => 'a::mem-type where
  clookup n l = from-bytes (l n)

```

```

definition cupdate :: nat => ('a::mem-type => 'a) => locals => locals where
  cupdate n f l = l(n := to-bytes (f (from-bytes (l (n)))) (replicate (size-of TYPE('a))
    0))

```

```

lemma clookup-cupdate-same[simp, state-simp]: clookup n (cupdate n f l) = f
(clookup n l)
  by (simp add: clookup-def cupdate-def)

```

```

lemma clookup-cupdate-same-cond[code-simproc state-simp]:
  n = 0 ==> clookup n (cupdate 0 f l) = f (clookup n l)
  n = 0 ==> clookup 0 (cupdate n f l) = f (clookup n l)
  n = 1 ==> clookup n (cupdate 1 f l) = f (clookup n l)
  n = Suc 0 ==> clookup n (cupdate (Suc 0) f l) = f (clookup n l)
  n = 1 ==> clookup 1 (cupdate n f l) = f (clookup n l)
  n = Suc 0 ==> clookup (Suc 0) (cupdate n f l) = f (clookup n l)

```

$n = \text{numeral } m \implies \text{clookup } n (\text{cupdate } (\text{numeral } m) f l) = f (\text{clookup } n l)$
 $n = \text{numeral } m \implies \text{clookup } (\text{numeral } m) (\text{cupdate } n f l) = f (\text{clookup } n l)$
by (*auto simp add: clookup-def cupdate-def*)

lemma *clookup-refl-cond*[*code-simproc state-simp*]:
 $n = m \implies \text{clookup } n l = \text{clookup } m l \longleftrightarrow \text{True}$
by (*simp-all add: clookup-def*)

lemma *clookup-cupdate-other*[*code-simproc state-simp*]: $n \neq m \implies \text{clookup } n (\text{cupdate } m f l) = (\text{clookup } n l)$
by (*auto simp add: clookup-def cupdate-def*)

lemma *cupdate-compose*[*simp, state-simp*]: $\text{cupdate } n f (\text{cupdate } n g l) = \text{cupdate } n (f o g) l$
by (*simp add: cupdate-def*)

lemma *cupdate-compose-cond*[*code-simproc state-simp*]:
 $n = 0 \implies \text{cupdate } n f (\text{cupdate } 0 g l) = \text{cupdate } n (f o g) l$
 $n = 0 \implies \text{cupdate } 0 f (\text{cupdate } n g l) = \text{cupdate } n (f o g) l$
 $n = 1 \implies \text{cupdate } n f (\text{cupdate } 1 g l) = \text{cupdate } n (f o g) l$
 $n = \text{Suc } 0 \implies \text{cupdate } n f (\text{cupdate } (\text{Suc } 0) g l) = \text{cupdate } n (f o g) l$
 $n = \text{Suc } 0 \implies \text{cupdate } (\text{Suc } 0) f (\text{cupdate } n g l) = \text{cupdate } n (f o g) l$
 $n = \text{numeral } m \implies \text{cupdate } n f (\text{cupdate } (\text{numeral } m) g l) = \text{cupdate } n (f o g) l$
 $n = \text{numeral } m \implies \text{cupdate } (\text{numeral } m) f (\text{cupdate } n g l) = \text{cupdate } n (f o g) l$
by (*auto simp add: cupdate-def*)

lemma *clookup-cupdate-other-numeral*[*simplified, simp, state-simp*]:
 $\text{clookup } 0 (\text{cupdate } 1 f l) = (\text{clookup } 0 l)$
 $\text{clookup } 0 (\text{cupdate } (\text{numeral } m) f l) = (\text{clookup } 0 l)$
 $\text{clookup } 1 (\text{cupdate } 0 f l) = (\text{clookup } 1 l)$
 $\text{numeral } m \neq (1::\text{nat}) \implies$
 $\text{clookup } 1 (\text{cupdate } (\text{numeral } m) f l) = (\text{clookup } 1 l)$

$\text{clookup } (\text{numeral } n) (\text{cupdate } 0 f l) = (\text{clookup } (\text{numeral } n) l)$
 $\text{numeral } n \neq (1::\text{nat}) \implies$
 $\text{clookup } (\text{numeral } n) (\text{cupdate } 1 f l) = (\text{clookup } (\text{numeral } n) l)$

$n \neq m \implies$
 $\text{clookup } (\text{numeral } n) (\text{cupdate } (\text{numeral } m) f l) = (\text{clookup } (\text{numeral } n) l)$
by (*auto simp add: clookup-def cupdate-def*)

lemma *cupdate-commute*: $n \neq m \implies \text{cupdate } n f (\text{cupdate } m g l) = \text{cupdate } m g (\text{cupdate } n f l)$
by (*auto simp add: cupdate-def fun-upd-def*)

lemma *cupdate-commute-ordered*[*code-simproc state-simp*]: $n < m \implies \text{cupdate } n f (\text{cupdate } m g l) = \text{cupdate } m g (\text{cupdate } n f l)$
by (*auto simp add: cupdate-def fun-upd-def*)

lemma *cupdate-commute-numeral-simp*[*simplified, simp, state-simp*]:
 $\text{cupdate } 0 \ f \ (\text{cupdate } 1 \ g \ l) = \text{cupdate } 1 \ g \ (\text{cupdate } 0 \ f \ l)$
 $\text{cupdate } 0 \ f \ (\text{cupdate } (\text{numeral } m) \ g \ l) = \text{cupdate } (\text{numeral } m) \ g \ (\text{cupdate } 0 \ f \ l)$

$\text{numeral } m \neq (1::\text{nat}) \implies$
 $\text{cupdate } 1 \ f \ (\text{cupdate } (\text{numeral } m) \ g \ l) = \text{cupdate } (\text{numeral } m) \ g \ (\text{cupdate } 1 \ f \ l)$

$n < m \implies \text{cupdate } (\text{numeral } n) \ f \ (\text{cupdate } (\text{numeral } m) \ g \ l) = \text{cupdate } (\text{numeral } m) \ g \ (\text{cupdate } (\text{numeral } n) \ f \ l)$
by (*auto simp add: cupdate-def fun-upd-def*)

lemma *const-compose* [*simp, state-simp*]:
 $\text{cupdate } n \ ((\lambda-. x) \circ f) = \text{cupdate } n \ (\lambda-. x)$
 $\text{cupdate } n \ (f \circ (\lambda-. x)) = \text{cupdate } n \ (\lambda-. f \ x)$
by (*auto simp add: comp-def*)

lemma *K-eq-cong*: $((\lambda-. x) = (\lambda-. y)) \longleftrightarrow x = y$
by (*simp add: fun-eq-iff*)

ML-file $\langle \text{positional-symbol-table.ML} \rangle$

named-theorems *locals*

consts *clocals-string-embedding* :: *string* \Rightarrow *nat*
consts *exit-'::nat*

definition *global-exn-var-clocal* = *clocals-string-embedding* "global-exn-var"

bundle *clocals-string-embedding*
begin
notation *clocals-string-embedding* (*S*)
end

ML \langle
 $\text{structure } CLocals =$
 struct
 $\text{structure } Locals = \text{Positional-Symbol-Table}(\text{type } \text{key} = \text{string } \text{val } \text{ord} = \text{fast-string-ord});$

$\text{@}\{\text{record } \langle \text{datatype } \text{entry} = \text{Entry of } \{\text{typ} : \text{typ}, \text{term} : \text{term}, \text{def} : \text{thm}, \text{kind} : \text{NameGeneration.var-kind}\}\}$

$\text{fun } \text{morph-entry } \text{phi } \{\text{typ}, \text{term}, \text{def}, \text{kind}\} = \text{make-entry}$
 $\{\text{typ} = \text{Morphism.typ } \text{phi } \text{typ}, \text{term} = \text{Morphism.term } \text{phi } \text{term}, \text{def} = \text{Morphism.thm } \text{phi } \text{def}, \text{kind} = \text{kind}\};$

$\text{fun } \text{entry-ord } (\text{Entry } \{\text{kind} = \text{kind1}, \text{typ} = \text{T1}, \text{term} = \text{t1}, \text{def} = \text{thm1}\}, \text{Entry } \{\text{kind} = \text{kind2}, \text{typ} = \text{T2}, \text{term} = \text{t2}, \text{def} = \text{thm2}\})$

```

= prod-ord NameGeneration.var-kind-ord (prod-ord Term-Ord.typ-ord (prod-ord
Term-Ord.fast-term-ord Thm.thm-ord))
  ((kind1, (T1, (t1, thm1))), (kind2, ((T2, (t2, thm2))))))

val entry-eq = is-equal o entry-ord

type locals-scope = {locals: entry Locals.symbol-table, scope: Locals.qualifier}

structure Data = Generic-Data (struct
  type T = locals-scope
  val empty = {locals = Locals.empty, scope = []}
  fun merge ({locals=locals1, ...}, {locals=locals2, ...}) =
    {locals = Locals.merge entry-eq (locals1, locals2),
     scope = []}
end)

fun map-locals f ({locals, scope}:locals-scope) =
  ({locals = f locals, scope = scope}:locals-scope)

fun map-scope f ({locals, scope}:locals-scope) = ({locals = locals, scope = f scope}:locals-scope)

fun locals-map f = Data.map (map-locals f)
fun scope-map f = Data.map (map-scope f)
fun enter-scope n = map-scope (fn xs => xs @ [n])
val exit-scope = map-scope (fst o split-last)

fun switch-scope fname =
  Context.proof-map (Data.map (exit-scope #> enter-scope fname))

fun program-name ctxt =
  case #scope (Data.get (Context.Proof ctxt)) of
    (prog-name::-) => prog-name
  | - =>

fun function-name ctxt =
  case #scope (Data.get (Context.Proof ctxt)) of
    [-, fun-name] => fun-name
  | - =>

fun read-function-pointer ctxt s =
  let
    val s = if Long-Name.is-qualified s then s else Long-Name.qualify (program-name
ctxt) s
  in
    Proof-Context.read-const {proper=true, strict=true} ctxt s
  end

```

```

fun mk-lookup-positional T i =
  Const <lookup T> $ HLogic.mk-number @{typ nat} i

fun lookup-by-short-name ctxt x =
  let
    val {scope, locals} = Data.get (Context.Proof ctxt)
    val x = NameGeneration.un-varname x
    val (qualifier, base) = Long-Name.explode x |> split-last
    val common-scope = rev scope |> drop (length qualifier) |> rev
    val full-scope = (if null common-scope then scope else common-scope) @ qualifier
    fun proj (name, (pos, Entry {typ, term,...})) = (name, (typ, term))
    val hits =
      Locals.lookup locals full-scope base |> Option.map (proj o pair x)
  in
    case hits of
      SOME v => v
    | NONE => error (lookup-by-short-name: ^ quote base ^ not defined in scope
    ^ @ {make-string} full-scope)
  end

fun info-from-term ctxt t =
  let
    val {scope, locals} = Data.get (Context.Proof ctxt)
  in
    case t of
      Const (n, -) =>
        let
          val name = NameGeneration.un-varname (Long-Name.base-name n)
        in Locals.lookup locals scope name |> Option.map (fn (p, x) => (name, (p,
x))) end
      | - => try Utils.dest-nat-or-number t
        |> Option.mapPartial (fn p =>
          Locals.lookup-positional locals scope p
          |> Option.map (fn (name, x) => (name, (p, x))))
    end

fun kind-from-term ctxt t = info-from-term ctxt t
  |> Option.map (fn (-, (-, Entry {kind, ...})) => kind)

fun is-defined ctxt x =
  is-some (try (lookup-by-short-name ctxt) x)

fun gen-mk-lookup ctxt (NameGeneration.Named x) =
  let val (-, (typ, term)) = lookup-by-short-name ctxt x
  in (Const <lookup typ> $ term, typ) end
  | gen-mk-lookup - (NameGeneration.Positional (i, T)) =
    (Const <lookup T> $ HLogic.mk-number @{typ nat} i, T)

fun get-position ctxt n =

```

```

let
  val {scope, locals} = Data.get (Context.Proof ctxt)
  in Locals.lookup locals scope n |> Option.map fst end

fun get-default-position ctxt n = the-default (~1) (get-position ctxt n)

fun positional-ord ctxt = int-ord o apply2 (get-default-position ctxt)
fun get-locals ctxt =
  let
    val {scope, locals} = Data.get (Context.Proof ctxt)
    val locals = Locals.get-scope (locals) scope
  in
    locals
    |> Locals.Keytab.dest
    |> map (fn (n, (-, Entry {typ, term, ...})) =>
      (Long-Name.base-name n, (typ, (Const <lookup typ> $ term))))
  end

fun mk-lookup ctxt x = fst (gen-mk-lookup ctxt (NameGeneration.Named x))

fun mk-update ctxt x =
  let val (-, (typ, term)) = lookup-by-short-name ctxt x
  in Const <update typ> $ term end

fun mk-update-positional T i =
  Const <update T> $ HLogic.mk-number @{typ nat} i

fun get-name ctxt n = try <
  let val (n, (typ, -)) = lookup-by-short-name ctxt n
  in (n, typ) end
  catch - => raise Match>

(* Name hints for bound variable names. Special treatment to unprime return variable name *)
fun return-name-hint ret = ret'
  | return-name-hint x = x

fun dest-return-name-hint ret' = ret
  | dest-return-name-hint x = x

fun embedded-name-hint n = @{const clocals-string-embedding} $ Utils.encode-isa-string n

fun name-hint ctxt n =
  if n = NameGeneration.global-exn-var-name then
    @{const global-exn-var-clocal}
  else
    (case try (lookup-by-short-name ctxt) (return-name-hint n) of

```

```

    SOME (-, (-, term)) => term
  | NONE =>
    let
      val - = warning (name-tag: no match for  $\hat{\text{quote}} n \hat{\text{}}$ . Using string
embedding.)
    in
      embedded-name-hint n
    end)

fun name-hints ctxt = map (name-hint ctxt) #> Utils.encode-isa-list @ {typ nat}

fun dest-name-hint (Const- <locals-string-embedding for s>) = HOLogic.dest-string
s
| dest-name-hint (const <global-exn-var-clocal>) = NameGeneration.global-exn-var-name
| dest-name-hint (Const (n, @ {typ nat})) = dest-return-name-hint (NameGeneration.un-varname
(Long-Name.base-name n))
| dest-name-hint - = raise Match

val dest-name-hints = Utils.decode-isa-list #> map dest-name-hint

fun split-scope n =
  case rev (Long-Name.explode n) of
    (base:: fname :: prog-name:: -) => ([prog-name, fname], base)
  | - => raise Match

fun strip-common [] [] = []
| strip-common (x::xs) (y::ys) = if x = y then strip-common xs ys else (y::ys)
| strip-common [] ys = ys
| strip-common - [] = []

val short-names = Attrib.setup-config-bool binding <locals-short-names> (K false);

fun print-tr ctxt (c as (Const (n, T))) =
  let
    val {scope, locals} = Data.get (Context.Proof ctxt)
    val (name-scope, base-name) = split-scope n
    val base-name = NameGeneration.un-varname base-name
    val min-scope = strip-common scope name-scope
    val short-name = base-name
    |> not (Config.get ctxt short-names) ? fold Long-Name.qualify min-scope
  in
    case Locals.lookup locals name-scope base-name of
      SOME (-, Entry {typ, ...}) =>
        Syntax.const syntax-const <-constrain> $ Const (short-name, T) $
Syntax-Phases.term-of-tyt ctxt typ
    | - => raise Match
  end
| print-tr - t = raise Match

```

```

fun is-lookup ctxt (Const (@{const-name clookup}, -) $ -) = true
  | is-lookup ctxt (Const (@{const-syntax clookup}, -) $ -) = true
  | is-lookup ctxt - = raise Match

fun lookup-tr' ctxt (Const (@{const-name clookup}, -) $ (c as Const -)) = print-tr
  ctxt c
  | lookup-tr' ctxt (Const (@{const-syntax clookup}, -) $ (c as Const -)) = print-tr
  ctxt c
  | lookup-tr' ctxt t = raise Match

fun dest-update-tr' ctxt ((c as (Const (@{const-syntax cupdate}, -) $ Const -))
  $ v $ s) =
  (c, v, SOME s)
  | dest-update-tr' ctxt ((c as (Const (@{const-name cupdate}, -) $ Const -))
  $ v $ s) =
  (c, v, SOME s)
  | dest-update-tr' ctxt ((c as (Const (@{const-syntax cupdate}, -) $ Const -))
  $ v) =
  (c, v, NONE)
  | dest-update-tr' ctxt ((c as (Const (@{const-name cupdate}, -) $ Const -))
  $ v) =
  (c, v, NONE)
  | dest-update-tr' ctxt t = raise Match

fun update-tr' ctxt (Const (@{const-name cupdate}, -) $ (c as Const -)) = print-tr
  ctxt c
  | update-tr' ctxt (Const (@{const-syntax cupdate}, -) $ (c as Const -)) = print-tr
  ctxt c
  | update-tr' ctxt t = t

fun locals-insert qualifier pos name typ kind def tab =
  let
    val tab' = tab |> Locals.update-new qualifier (name,
      Entry {typ=typ, term = Thm.concl-of def |> Utils.lhs-of-eq, def = Thm.trim-context
        def, kind = kind})
    val - = @assert ((Locals.lookup tab' qualifier name |> Option.map fst) =
      SOME pos)
  in
    tab'
  end

fun add-entry qualifier pos name typ kind def =
  locals-map (locals-insert qualifier pos name typ kind def)

fun add-entry-attr qualifier pos name typ kind =
  Thm.declaration-attribute (add-entry qualifier pos name typ kind)

```



```

fun define-locals qualifier decls thy =
  let
    val - = @{assert} (not (null qualifier))
    val fname = split-last qualifier |> snd
    val lthy = Named-Target.theory-init thy
    fun define (i, (name, typ, kind)) lthy =
      let
        val vname = NameGeneration.ensure-varname name
        val b = Binding.make (vname, here) |> fold-rev (Binding.qualify true)
      in
        qualifier
          val attrib = Attrib.internal here (fn - => add-entry-attr qualifier i name
            typ kind)
          val rhs = HOLogic.mk-number @{typ nat} i
          val ((t, (s, thm)), lthy) = lthy
            |> Local-Theory.define ((b, Mixfix.NoSyn), ((Binding.suffix-name -def b,
              [attrib] @ @{attributes [locals]} ), rhs))
          in
            lthy |> Code.declare-default-eqns [(thm, true)]
          (*|>
            Local-Theory.declaration {pervasive=true, syntax=false} (fn - =>
              Context.mapping
                (Code.declare-default-eqns-global [(thm, true)])
                I*)
          end
          val lthy = lthy
            |> fold define (tag-list 0 decls)
            |> Bundle.bundle (Binding.make (suffix -scope fname, here),
              Attrib.internal-declaration here (Morphism.entity (fn - => scope-map (K
                (qualifier)))))) []
          in
            lthy |> Local-Theory.exit-global
          end
      end
  end

fun symmetric ctxt thm = fst (Thm.apply-attribute Calculation.symmetric thm
  (Context.Proof ctxt))

fun gen-unfolded {sym} qualifier ctxt thm =
  let
    val {scope, locals} = Data.get (Context.Proof ctxt)
    val qualifier = if null qualifier then scope else qualifier
    val defs0 = Locals.get-scope locals qualifier |> Locals.Keytab.dest
      |> map (fn (-, (-, Entry {def,...})) => Thm.transfer' ctxt def)
    val defs = defs0 |> sym ? map (symmetric ctxt)
  in
    Simplifier.simplify (Simplifier.put-simpset HOL-basic-ss ctxt addsimps defs) thm
  end

val unfolded-with = gen-unfolded {sym = false}
val folded-with = gen-unfolded {sym = true}

```

```

val unfolded = unfolded-with []
val folded = folded-with []

fun check-scope ctxt (qualifier, pos) =
  let
    val {locals, scope} = Data.get (Context.Proof ctxt)
    val exploded = Long-Name.explode qualifier
    val extended = if length exploded = 1 then
      take 1 scope @ exploded
    else
      exploded
  in
    if null extended orelse Locals.defined-scope locals extended then
      extended
    else
      error (undefined scope: ^ quote (Long-Name.implode extended) ^ Position.here
pos)
  end
end

>

setup <
Attrib.setup binding <fold-locals>
  (Args.context -- Scan.lift (Scan.optional Parse.name-position (, Position.none))
>>
  (fn (ctxt, (qualifier, pos)) =>
    let
      val exploded = CLocals.check-scope ctxt (qualifier, pos)
    in
      Thm.rule-attribute [] (CLocals.folded-with exploded o Context.proof-of)
    end))
  fold local variables within optional scope qualifier #>

Attrib.setup binding <unfold-locals>
  (Args.context -- Scan.lift (Scan.optional Parse.name-position (, Position.none))
>>
  (fn (ctxt, (qualifier, pos)) =>
    let
      val exploded = CLocals.check-scope ctxt (qualifier, pos)
    in
      Thm.rule-attribute [] (CLocals.unfolded-with exploded o Context.proof-of)
    end))
  unfold local variables within optional scope qualifier
>

```

nonterminal *localsupbinds* and *localsupbind*

syntax

```
-localsupbind :: 'a ⇒ 'a ⇒ localsupbind          ((λ- := λ / -))  
              :: localsupbind ⇒ localsupbinds      (-)  
-localsupbinds:: localsupbind ⇒ localsupbinds ⇒ localsupbinds (-, / -)
```

syntax

```
-statespace-lookup :: locals ⇒ 'name ⇒ 'c (- · - [60, 60] 60)  
-statespace-locals-lookup :: ('g, locals, 'e, 'x) state-scheme ⇒ 'name ⇒ 'c  
  (- · λ - [60, 60] 60)  
  
-statespace-update :: locals ⇒ 'name ⇒ ('c ⇒ 'c) ⇒ locals  
-statespace-updates :: locals ⇒ updbinds ⇒ locals (-(-) [900, 0] 900)  
  
-statespace-locals-update :: ('g, locals, 'e, 'x) state-scheme ⇒ 'name ⇒ ('c ⇒ 'c)  
⇒ ('g, locals, 'e, 'x) state-scheme  
-statespace-locals-updates :: locals ⇒ localsupbinds ⇒ locals (-(-) [900, 0] 900)  
  
-statespace-locals-map ::  
  'name ⇒ ('c ⇒ 'c) ⇒ ('g, locals, 'e, 'x) state-scheme ⇒ ('g, locals, 'e, 'x)  
state-scheme  
((λ- := λ / -) [1000, 1000] 1000)
```

translations

```
-statespace-updates f (-updbinds b bs) ==  
  -statespace-updates (-statespace-updates f b) bs  
s⟨x := y⟩ == -statespace-update s x y  
  
-statespace-locals-updates f (-localsupbinds b bs) ==  
  -statespace-locals-updates (-statespace-locals-updates f b) bs  
s⟨x := λ y⟩ == -statespace-locals-update s x y
```

parse-translation

```
<  
let  
fun lookup-tr ctxt [s, x] =  
  let  
  in  
    (case Term-Position.strip-positions x of  
      Free (n, -) => CLocals.mk-lookup ctxt n $ s  
    | - => raise Match)  
  end
```

```
fun locals-lookup-tr ctxt [s, x] =  
  (case Term-Position.strip-positions x of
```

```

Free (n,-) =>
  CLocals.mk-lookup ctxt n $ (Syntax.const const-name <locals> $ s)
| - => raise Match);

fun update-tr ctxt [s, x, v] =
  (case Term-Position.strip-positions x of
    Free (n, -) =>
      let
        val upd = CLocals.mk-update ctxt n
      in upd $ v $ s end
    | - => raise Match);

fun locals-update-tr ctxt [s, x, v] =
  let
    val upd $ n $ v' $ - = update-tr ctxt [s, x, v]
  in
    Syntax.const const-name <locals-update> $ (upd $ n $ v') $ s
  end

fun locals-map-tr ctxt [x, v] =
  let
    val upd $ s = locals-update-tr ctxt [Bound 0, x, v]
  in
    upd
  end

in

[(syntax-const <-statespace-lookup>, lookup-tr),
 (syntax-const <-statespace-locals-lookup>, locals-lookup-tr),
 (syntax-const <-statespace-update>, update-tr),
 (syntax-const <-statespace-locals-update>, locals-update-tr),
 (syntax-const <-statespace-locals-map>, locals-map-tr)]
end
>

print-translation <
let

fun dest-number (Const (const-syntax <Groups.zero>, -)) = 0
  | dest-number (Const (const-syntax <Groups.one>, -)) = 1
  | dest-number (Const (const-syntax <numeral>, -) $ n) = Numeral.dest-num-syntax
n

fun lookup-tr' ctxt (args as [n, s]) =
  let
    val n' = CLocals.print-tr ctxt n

```

```

in
  case s of
    Const (const-syntax ⟨locals⟩, -) $ s'' =>
      Syntax.const syntax-const ⟨-statespace-locals-lookup⟩ $ s'' $ n'
    | Const (syntax-const ⟨-antiquoteCur⟩, -) $ Const (get-actuals, -) =>
      Syntax.const syntax-const ⟨-antiquoteCur⟩ $ n'
    | - => Syntax.const syntax-const ⟨-statespace-lookup⟩ $ s $ n'
  end

fun update-tr' ctxt (args as [n, v, s]) =
  let
    val n' = CLocals.print-tr ctxt n
  in
    Syntax.const syntax-const ⟨-statespace-update⟩ $ s $ n' $ v
  end

fun locals-update-tr' ctxt (args as [upd, s]) =
  let
    val - $ s' $ n' $ v = update-tr' ctxt ((snd (strip-comb upd)) @ [s])
  in
    Syntax.const syntax-const ⟨-statespace-locals-update⟩ $ s' $ n' $ v
  end
| locals-update-tr' ctxt (args as [upd]) =
  let
    val - $ - $ n' $ v = update-tr' ctxt ((snd (strip-comb upd)) @ [Bound 0])
  in
    Syntax.const syntax-const ⟨-statespace-locals-map⟩ $ n' $ v
  end

fun assign-tr' ctxt (args as [(aq as Const (-antiquoteCur, -)) $ (c as Const (locals, -)), upd]) =
  let
    fun dest-K (Abs (-, -, v)) = v
      | dest-K t = t

    val s = Bound 0
    val - $ - $ n $ v = locals-update-tr' ctxt [upd, s]
  in
    Syntax.const syntax-const ⟨-Assign⟩ $ (aq $ n) $ (dest-K v)
  end
| assign-tr' ctxt t = raise Match

in
  [(const-syntax ⟨lookup⟩, lookup-tr'),
   (const-syntax ⟨update⟩, update-tr'),
   (const-syntax ⟨locals-update⟩, locals-update-tr'),
   (syntax-const ⟨-Assign⟩, assign-tr')]

```

end
>

end

Chapter 14

Setup Lex / Yacc and Translation from C to Simpl

```
theory CTranslationSetup
imports
  UMM
  PackedTypes
  PrettyProgs
  StaticFun
  IndirectCalls
  ModifiesProofs
  HOL-Eisbach.Eisbach
  ML-Record-Antiquotation
  Option-Scanner
  Misc-Antiquotation
  MkTermAntiquote
  TermPatternAntiquote
  CLocals
keywords
  cond-sorry-modifies-proofs :: thy-decl
and
  mlex
  mlyacc:: thy-load
begin

ML ‹
structure Coerce-Syntax =
struct

  val show-types = Ptr-Syntax.show-ptr-types

  fun coerce-tr' cnst ctxt typ ts = if Config.get ctxt show-types then
    case Term.strip-type typ of
      ([S], T) =>
```

```

      list-comb
      (Syntax.const cnst $ Syntax-Phases.term-of-typ ctxt S
       $ Syntax-Phases.term-of-typ ctxt T
       , ts)
    | - => raise Match
  else raise Match
end
›

```

definition *coerce*::'a::mem-type ⇒ 'b::mem-type **where**
coerce v = from-bytes (to-bytes-p v)

syntax

-coerce :: type ⇒ type ⇒ logic ((1COERCE/(1'(- → -))))

translations

COERCE('a → 'b) => CONST coerce :: ('a ⇒ 'b)

typed-print-translation

⟨ [(@{const-syntax coerce}, Coerce-Syntax.coerce-tr' @ {syntax-const -coerce})] ⟩

lemma *coerce-id*[simp]:

shows *coerce* v = v

by (simp add: *coerce-def*)

lemma *coerce-cancel-packed*[simp]:

fixes v::'a::packed-type

assumes sz-eq: size-of (TYPE('a)) = size-of (TYPE('b))

shows *coerce* ((*coerce* v)::'b::packed-type) = v

using *assms*

apply (simp add: *coerce-def*)

by (metis (mono-tags, lifting) *access-ti₀-def access-ti₀-to-bytes*
field-access-update-same(1) inv-p length-replicate length-to-bytes-p
packed-type-intro-simps(1) wf-fd)

definition *coerce-map*:: ('a::mem-type ⇒ 'a) ⇒ ('b::mem-type ⇒ 'b) **where**

coerce-map f v = *coerce* (f (*coerce* v))

lemma *coerce-map-id*[simp]: *coerce-map* f (*coerce* v) = f v

by (simp add: *coerce-map-def*)

lemma *coerce-coerce-map-cancel-packed*[simp]:

fixes f::'a::packed-type ⇒ 'a

fixes v::'b::packed-type

assumes sz-eq[simp]: size-of (TYPE('a)) = size-of (TYPE('b))

shows ((*coerce* (*coerce-map* f v))::'a) = f (*coerce* v)

by (simp add: *coerce-map-def*)

named-theorems *global-const-defs* **and**
global-const-array-selectors **and**
global-const-non-array-selectors **and**
global-const-selectors

named-theorems *fun-ptr-simps*
named-theorems *fun-ptr-intros*
named-theorems *fun-ptr-distinct*
named-theorems *fun-ptr-subtree*

We integrate `mlex` and `mlyacc` directly into Isabelle:

- We compile the SML files according to the description in `tools/mlyacc/src/FILES`
- We export the necessary signatures and structures to the Isabelle/ML environment.
- As `mlex` / `mlyacc` operate directly on files we invoke them on temporary files and redirect `stdout` / `stderr` to display the messages within PIDE. We wrap this in Isabelle commands `mlex`, `mlyacc`.

SML-file *<tools/mlex/mlex.ML>*

SML-export *<structure LexGen = LexGen>*

SML-file *<tools/mlyacc/mlyacclib/MY-base-sig.ML>*

SML-file *<tools/mlyacc/mlyacclib/MY-stream.ML>*

SML-file *<tools/mlyacc/mlyacclib/MY-lrtable.ML>*

SML-file *<tools/mlyacc/mlyacclib/MY-join.ML>*

SML-file *<tools/mlyacc/mlyacclib/MY-parser2.ML>*

SML-file *<tools/mlyacc/src/utills.ML>*

SML-file *<tools/mlyacc/src/sigs.ML>*

SML-file *<tools/mlyacc/src/hdr.ML>*

SML-file *<tools/mlyacc/src/yacc-grm-sig.sml>*

SML-file *<tools/mlyacc/src/yacc-grm.sml>*

SML-file *<tools/mlyacc/src/yacc.lex.sml>*

SML-file *<tools/mlyacc/src/parse.ML>*

SML-file *<tools/mlyacc/src/grammar.ML>*

SML-file *<tools/mlyacc/src/core.ML>*

SML-file *<tools/mlyacc/src/coreutills.ML>*

SML-file *<tools/mlyacc/src/graph.ML>*

SML-file *<tools/mlyacc/src/look.ML>*

SML-file *<tools/mlyacc/src/lalr.ML>*

SML-file *<tools/mlyacc/src/mklrtable.ML>*

SML-file *<tools/mlyacc/src/mkprstruct.ML>*

SML-file *<tools/mlyacc/src/shrink.ML>*

SML-file *<tools/mlyacc/src/verbose.ML>*

SML-file *<tools/mlyacc/src/absyn-sig.ML>*

SML-file *<tools/mlyacc/src/absyn.ML>*

```

SML-file <tools/mlyacc/src/yacc.ML>
SML-file <tools/mlyacc/src/link.ML>

SML-export <signature LR-TABLE = LR-TABLE>
SML-export <structure LrTable = LrTable>
SML-export <signature LR-PARSER = LR-PARSER>
SML-export <structure LrParser = LrParser>
SML-export <signature TOKEN = TOKEN>
SML-export <signature PARSER-DATA = PARSER-DATA>
SML-export <signature PARSE-GEN = PARSE-GEN>
SML-export <signature LEXER = LEXER>
SML-export <signature PARSER-DATA = PARSER-DATA>
SML-export <signature PARSER = PARSER>
SML-export <signature ARG-PARSER = ARG-PARSER>
SML-export <structure ParseGen = ParseGen>

SML-export <functor Join(structure Lex : LEXER
  structure ParserData: PARSER-DATA
  structure LrParser : LR-PARSER
  sharing ParserData.LrTable = LrParser.LrTable
  sharing ParserData.Token = LrParser.Token
  sharing type Lex.UserDeclarations.svalue = ParserData.svalue
  sharing type Lex.UserDeclarations.pos = ParserData.pos
  sharing type Lex.UserDeclarations.token = ParserData.Token.token):
  PARSER =
    Join(structure Lex = Lex
      structure ParserData = ParserData
      structure LrParser = LrParser)>
SML-export <functor JoinWithArg(structure Lex : ARG-LEXER
  structure ParserData: PARSER-DATA
  structure LrParser : LR-PARSER
  sharing ParserData.LrTable = LrParser.LrTable
  sharing ParserData.Token = LrParser.Token
  sharing type Lex.UserDeclarations.svalue = ParserData.svalue
  sharing type Lex.UserDeclarations.pos = ParserData.pos
  sharing type Lex.UserDeclarations.token = ParserData.Token.token)
  : ARG-PARSER =
    JoinWithArg(structure Lex = Lex
      structure ParserData = ParserData
      structure LrParser = LrParser)
>

ML <
  local

  val tmp-io = Synchronized.var tmp-io ()

  fun system-command cmd =
    if Isabelle-System.bash cmd <> 0 then error (System command failed: ^ cmd)

```

```

else ();

fun copy-file qualify-ref src0 dst0 =
  let
    val src = Path.expand src0;
    val dst = Path.expand dst0;
    val target = if File.is-dir dst then Path.append dst (Path.base src) else dst;
  in
    if File.eq (src, target) then
      ()
    else if qualify-ref then
      ignore (system-command (sed 's/ ref / Unsynchronized.ref /g' ^ File.bash-path
src ^ > ^ File.bash-path target))
    else
      ignore (system-command (cp -f ^ File.bash-path src ^ ^ File.bash-path
target))
    end;
end;

fun file-command qualify-ref tag exts f files thy =
  let
    val (files, thy) = files thy
    val {src-path, lines,...}: Token.file = the-single files
    val filename = Path.file-name src-path
    val full-src-path = Path.append (Resources.master-directory thy) src-path
    val - = Isabelle-System.with-tmp-dir tag (fn tmp-dir => Synchronized.change
tmp-io (fn - =>
  let
    val orig-stdout = TextIO.getOutstream TextIO.stdout
    val orig-stderr = TextIO.getOutstream TextIO.stderr
    val out-file = (Utils.sanitized-path thy tmp-dir (Path.ext out src-path))
    val err-file = (Utils.sanitized-path thy tmp-dir (Path.ext err src-path))
    val dir = Path.dir out-file
    val - = Isabelle-System.make-directory dir
    val stdout = TextIO.openOut (File.standard-path out-file)
    val stderr = TextIO.openOut (File.standard-path err-file)

    val - = TextIO.setOutstream (TextIO.stdout, TextIO.getOutstream stdout)
    val - = TextIO.setOutstream (TextIO.stderr, TextIO.getOutstream stderr)

    val tmp-file = Utils.sanitized-path thy tmp-dir src-path

    val - = File.write tmp-file (cat-lines lines)
    val res = try f (File.standard-path tmp-file)

    val - = TextIO.setOutstream (TextIO.stdout, orig-stdout)
    val - = TextIO.setOutstream (TextIO.stderr, orig-stderr)
    val - = TextIO.closeOut stdout

```

```

val - = TextIO.closeOut stderr
val out = File.read-lines out-file
val err = File.read-lines err-file

val msg = Pretty.string-of (Pretty.strs (out @ err))

in
  case res of
    SOME () =>
      let
        val result-files = map (fn ext => (Path.ext ext tmp-file,
          Path.ext ext full-src-path)) exts
        val - = app (fn (src, dst) => copy-file qualify-ref src dst) result-files
        val - = tracing (quote (tag ^ ^ filename) ^ succeeded:\n ^ msg)
        in () end
      | NONE => (error (quote (tag ^ ^ filename) ^ failed:\n ^ msg); ())
    end))

in
  thy
end

in
val - =
  Outer-Syntax.command
  @{command-keyword mllex} generate lexer
  (Resources.provide-parse-files single >> (Toplevel.theory o (file-command true
mllex [sm] LexGen.lexGen)))
val - =
  Outer-Syntax.command
  @{command-keyword mlyacc} generate parser
  (Resources.provide-parse-files single >> (Toplevel.theory o (file-command true
mlyacc [sig, sm] ParseGen.parseGen)))

end
>

```

```

primrec map-of-default ::
  ('p ⇒ 'a) ⇒ ('p * 'a) list ⇒ 'p ⇒ 'a
where
  map-of-default d [] x = d x
| map-of-default d (x # xs) x' = (if fst x = x' then snd x else map-of-default d xs
x')

```

lemma map-of-default-append: $\langle \text{map-of-default } d \text{ (xs @ ys)} = \text{map-of-default (map-of-default } d \text{ ys) xs} \rangle$
by (induction xs arbitrary: d ys) auto

lemma *map-of-default-map-of-conv*:
 $\langle \text{map-of-default } d \text{ } xs \text{ } p = (\text{case map-of } xs \text{ } p \text{ of } \text{Some } f \Rightarrow f \mid \text{None} \Rightarrow d \text{ } p) \rangle$
by (*induction xs*) (*auto simp add: fun-upd-same fun-upd-other*)

lemma *map-of-default-fallthrough*:
 $p \notin \text{set } (\text{map fst } xs) \Longrightarrow \text{map-of-default } d \text{ } xs \text{ } p = d \text{ } p$
by (*induct xs*) *auto*

lemma *map-of-default-distinct*:
assumes *distinct (map fst xs)*
shows *list-all* ($\lambda(p, f). \text{map-of-default } d \text{ } xs \text{ } p = f$) *xs*
using *assms*
apply (*induction xs*)
apply *simp-all*
by (*smt (verit, ccfv-SIG) image-iff list.pred-mono-strong prod.case-eq-if*)

lemma *map-of-default-default-conv*:
assumes *list-all* ($\lambda(p, f). d \text{ } p = f$) *xs*
shows $\text{map-of-default } d \text{ } xs = d$
using *assms*
by (*induct xs*) *auto*

lemma *map-of-default-monotone-cons*[*partial-function-mono*]:
assumes *f1* [*partial-function-mono*]: *monotone R X f1*
assumes [*partial-function-mono*]: *monotone R X* ($\lambda f. \text{map-of-default } d \text{ } (xs \text{ } f) \text{ } p$)
shows *monotone R X* ($\lambda f. \text{map-of-default } d \text{ } ((p1, f1 \text{ } f)\#xs \text{ } f) \text{ } p$)
apply (*simp only: map-of-default.simps fst-conv snd-conv cong: if-cong*)
apply (*intro partial-function-mono*)
done

hide-const (**open**) *StaticFun.Node*

primrec *tree-of* :: $'a \text{ list} \Rightarrow 'a \text{ tree}$
where
 $\text{tree-of } [] = \text{Tip}$
 $\text{tree-of } (x\#xs) = \text{Node } (\text{Tip}) \text{ } x \text{ } \text{False } (\text{tree-of } xs)$

lemma *set-of-tree-of*: $\text{set-of } (\text{tree-of } xs) = \text{set } xs$
by (*induct xs*) *auto*

lemma *all-distinct-tree-of*:
assumes *all-distinct (tree-of xs)*
shows *distinct xs*
using *assms* **by** (*induct xs*) (*auto simp add: set-of-tree-of*)

lemma *all-distinct-tree-of'*:
 $\text{all-distinct } t \Longrightarrow \text{tree-of } xs \equiv t \Longrightarrow \text{distinct } xs$
by (*simp add: all-distinct-tree-of*)

lemma *map-of-default-fallthrough'*:
 $map\ fst\ xs \equiv ps \implies tree-of\ ps \equiv t \implies p \notin set-of\ t \implies map-of-default\ d\ xs\ p = d\ p$
apply (*rule map-of-default-fallthrough*)
using *set-of-tree-of* [*of map fst xs*]
apply *simp*
done

primrec *list-of* :: 'a tree \Rightarrow 'a list
where
 $list-of\ Tip = []$
 $| list-of\ (Node\ l\ x\ d\ r) = list-of\ l\ @\ (if\ d\ then\ []\ else\ [x])\ @\ list-of\ r$

lemma *list-of-tree-of-conv* [*simp*]: $list-of\ (tree-of\ xs) = xs$
by (*induct xs*) *auto*

lemma *set-list-of-set-of-conv*: $set\ (list-of\ t) = set-of\ t$
by (*induct t*) *auto*

lemma *all-distinct-list-of*:
assumes *all-distinct t*
shows *distinct (list-of t)*
using *assms* **by** (*induct t*) (*auto simp add: set-list-of-set-of-conv*)

lemma *map-of-default-distinct-lookup-list-all*:
 $distinct\ (map\ fst\ xs) \implies list-all\ (\lambda(p, f). map-of-default\ d\ xs\ p = f)\ xs$
by (*induct xs*) (*auto simp add: case-prod-beta' image-iff list.pred-mono-strong*)

lemma *map-of-default-distinct-lookup-list-all'*:
assumes $ps: map\ fst\ xs \equiv ps$
assumes $t: tree-of\ ps \equiv t$
assumes *dist: all-distinct t*
shows $list-all\ (\lambda(p, f). map-of-default\ d\ xs\ p = f)\ xs$
using *all-distinct-tree-of'* [*OF dist t*] *ps*
by (*simp add: map-of-default-distinct-lookup-list-all*)

lemma *map-of-default-distinct-lookup-list-all''*:
assumes $t: list-of\ t \equiv ps$
assumes $ps: map\ fst\ xs \equiv ps$
assumes *dist: all-distinct t*
shows $list-all\ (\lambda(p, f). map-of-default\ d\ xs\ p = f)\ xs$
by (*metis all-distinct-list-of dist map-of-default-distinct-lookup-list-all ps t*)

lemma *map-of-default-other-lookup-list-all*:
 $set\ ps \cap set\ (map\ fst\ xs) = \{ \} \implies list-all\ (\lambda p. map-of-default\ d\ xs\ p = d\ p)\ ps$
using *map-of-default-fallthrough* [**where** $d=d$ **and** $xs=xs$]
apply (*induct xs*)
apply (*clarsimp simp add: list.pred-True*)

by (smt (verit, best) IntI ball-empty list.pred-set)

lemma *delete-Some-subset*: *DistinctTreeProver.delete* $x\ t = \text{Some } t' \implies \text{set-of } t \subseteq \{x\} \cup \text{set-of } t'$

by (induct t arbitrary: t') (auto split: option.splits if-split-asm)

lemma *delete-Some-set-of-union*:

assumes *del*: *DistinctTreeProver.delete* $x\ t = \text{Some } t'$ shows $\text{set-of } t = \{x\} \cup \text{set-of } t'$

proof –

from *delete-Some-set-of* [OF *del*] have $\text{set-of } t' \subseteq \text{set-of } t$.

moreover

from *delete-Some-x-set-of* [OF *del*] obtain $x \in \text{set-of } t \ x \notin \text{set-of } t'$

by *simp*

ultimately have $\{x\} \cup \text{set-of } t' \subseteq \text{set-of } t$

by *blast*

with *delete-Some-subset* [OF *del*]

show *?thesis* by *blast*

qed

primrec *undeleted* :: 'a tree \implies bool

where

undeleted *Tip* = *True*

| *undeleted* (Node *l y d r*) = $(\neg d \wedge \text{undeleted } l \wedge \text{undeleted } r)$

lemma *undeleted-tree-of*[*simp*]: *undeleted* (tree-of *xs*)

by (induct *xs*) auto

lemma *subtract-union-subset*:

subtract $t_1\ t_2 = \text{Some } t \implies \text{undeleted } t_1 \implies \text{set-of } t_2 \subseteq \text{set-of } t_1 \cup \text{set-of } t$

proof (induct t_1 arbitrary: $t_2\ t$)

case *Tip* thus *?case* by *simp*

next

case (Node *l x b r*)

have *sub*: *subtract* (Node *l x b r*) $t_2 = \text{Some } t$ by *fact*

from *Node.prem*s obtain *not-b*: $\neg b$ and *undeleted-l*: *undeleted* *l* and *undeleted-r*: *undeleted* *r*

by *simp*

from *subtract-Some-set-of* [OF *sub*] have *sub-t2*: $\text{set-of } (\text{Node } l\ x\ b\ r) \subseteq \text{set-of } t_2$.

show *?case*

proof (cases *DistinctTreeProver.delete* $x\ t_2$)

case (Some t_2')

note *del-x-Some* = *this*

from *delete-Some-set-of-union* [OF *Some*]

have *t2'-t2*: $\text{set-of } t_2 = \{x\} \cup \text{set-of } t_2'$.

show *?thesis*

```

proof (cases subtract l t₂)
  case (Some t₂')
  note sub-l-Some = this
  from Node.hyps (1) [OF Some undeleted-l]
  have t₂''-t₂': set-of t₂' ⊆ set-of l ∪ set-of t₂'' .
  show ?thesis
  proof (cases subtract r t₂'')
    case (Some t₂''')
    from Node.hyps (2) [OF Some undeleted-r]
    have set-of t₂'' ⊆ set-of r ∪ set-of t₂''' .
    with Some sub-l-Some del-x-Some sub t₂''-t₂' t₂'-t₂ not-b
    show ?thesis
      by auto
  next
  case None
  with del-x-Some sub-l-Some sub
  show ?thesis
    by simp
  qed
next
  case None
  with del-x-Some sub
  show ?thesis
    by simp
  qed
next
  case None
  with sub show ?thesis by simp
  qed
qed

```

```

lemma subtract-union-eq:
  assumes sub: subtract t₁ t₂ = Some t
  assumes und: undeleted t₁
  shows set-of t₂ = set-of t₁ ∪ set-of t
proof
  from sub und
  show set-of t₂ ⊆ set-of t₁ ∪ set-of t
    by (rule subtract-union-subset)
  next
  from subtract-Some-set-of-res [OF sub] have set-of t ⊆ set-of t₂.
  moreover from subtract-Some-set-of [OF sub] have set-of t₁ ⊆ set-of t₂ .
  ultimately
  show set-of t₁ ∪ set-of t ⊆ set-of t₂ by blast
qed

```


lemma *subtract-empty*:
assumes *sub*: *subtract* t_1 $t_2 = \text{Some } t$
assumes *und*: *undeleted* t_1
assumes *empty*: *set-of* $t = \{\}$
shows *set-of* $t_1 = \text{set-of } t_2$
using *subtract-union-eq* [*OF sub und*] *empty* **by** *simp*

lemma *map-of-default-other-lookup-Ball*:
assumes *ps*: *list-of* $t \equiv ps$
assumes *map-fst*: *map fst xs* $\equiv ps$
assumes *sub*: *subtract* t *t-all* = *Some t-sub*
shows $\forall p \in \text{set-of } t\text{-sub. map-of-default } d \text{ } xs \text{ } p = d \text{ } p$
by (*metis disjoint-iff map-fst map-of-default-fallthrough ps set-list-of-set-of-conv sub subtract-Some-dist-res*)

lemma *subtract-set-of-exchange-first*:
assumes *sub1*: *subtract* t_1 $t = \text{Some } t'$
assumes *sub2*: *subtract* t_2 $t = \text{Some } t''$
assumes *und1*: *undeleted* t_1
assumes *und2*: *undeleted* t_2
assumes *seq*: *set-of* $t_1 = \text{set-of } t_2$
shows *set-of* $t' = \text{set-of } t''$
using *subtract-union-eq* [*OF sub1 und1*] *subtract-union-eq* [*OF sub2 und2*] *seq*
subtract-Some-dist-res [*OF sub1*] *subtract-Some-dist-res* [*OF sub2*]
by *blast*

lemma *TWO*: *Suc* (*Suc* 0) = 2 **by** *arith*

definition

fun-addr-of :: *int* \Rightarrow *unit ptr* **where**
fun-addr-of $i \equiv \text{Ptr } (\text{word-of-int } i)$

definition

ptr-range :: '*a*::*c-type ptr* \Rightarrow *addr set* **where**
ptr-range $p \equiv \{ \text{ptr-val } (p::'a \text{ ptr}) \dots < \text{ptr-val } p + \text{word-of-int}(\text{int}(\text{size-of } (\text{TYPE}'a))) \}$
}

lemma *guarded-spec-body-wp* [*vcg-hoare*]:

$P \subseteq \{s. (\forall t. (s,t) \in R \longrightarrow t \in Q) \wedge (Ft \notin F \longrightarrow (\exists t. (s,t) \in R))\}$
 $\Longrightarrow \Gamma, \Theta \vdash_{/F} P$ (*guarded-spec-body* Ft R) Q, A
apply (*simp add: guarded-spec-body-def*)
apply (*cases* $Ft \in F$)
apply (*erule* *HoarePartialDef.Guarantee*)
apply (*rule* *HoarePartialDef.conseqPre*, *rule* *HoarePartialDef.Spec*)

```

apply auto[1]
apply (rule HoarePartialDef.conseqPre, rule HoarePartialDef.Guard[where P=P])
apply (rule HoarePartialDef.conseqPre, rule HoarePartialDef.Spec)
apply auto[1]
apply simp
apply (erule order-trans)
apply (auto simp: image-def Bex-def)
done

ML-file ../lib/ml-helpers/StringExtras.ML
ML-file topo-sort.ML
ML-file isa-termstypes.ML

mlex StrictC.lex
mlyacc StrictC.grm

ML-file StrictC.grm.sig
ML-file StrictC.grm.sml
ML-file StrictC.lex.sml

ML-file StrictCParser.ML
ML-file complit.ML
ML-file hp-termstypes.ML
ML-file termstypes-sig.ML
term word-of-int
ML-file termstypes.ML
ML-file recursive-records/recursive-record-pp.ML
ML-file recursive-records/recursive-record-package.ML
ML-file UMM-termstypes.ML
ML-file expression-typing.ML
ML-file program-analysis.ML

context
begin
ML-file cached-theory-simproc.ML
ML-file UMM-Proofs.ML
end

simproc-setup size-of-bound (⟨size-of TYPE('a::c-type) ≤ n⟩) = ⟨K (fn ctxt =>
fn ct =>
  let
    val - = (case Thm.term-of ct of
      @_{term-pat size-of ?T ≤ - } => if UMM-Proofs.is-ctype T then () else raise
Match
    | - => raise Match)
    val ctxt' = ctxt addsimps (Named-Theorems.get ctxt @_{named-theorems size-simps})
    val eq = Simplifier.asm-full-rewrite ctxt' ct

```

```

    val - = Utils.verbose-msg 5 ctxt (fn - => size-of-bound: ^ Thm.string-of-thm
    ctxt eq)
    val rhs = Thm.rhs-of eq |> Thm.term-of
  in
    if rhs aconv term <True> orelse rhs aconv term <False> then
      SOME eq
    else
      NONE
  end
  handle Match => NONE)
>

```

```

ML-file heapstatetype.ML
ML-file MemoryModelExtras-sig.ML
ML-file MemoryModelExtras.ML
ML-file calculate-state.ML
ML-file syntax-transforms.ML
ML-file expression-translation.ML
ML-file modifies-proofs.ML
ML-file HPIInter.ML
ML-file stmt-translation.ML

```

```

method-setup vcg = <Hoare.vcg (*|> then-shorten-names*)>
  Simpl 'vcg' followed by C parser 'shorten-names'

```

```

method-setup vcg-step = <Hoare.vcg-step (*|> then-shorten-names*)>
  Simpl 'vcg-step' followed by C parser 'shorten-names'

```

```

declare typ-info-word [simp del]
declare typ-info-ptr [simp del]

```

```

lemma valid-call-Spec-eq-subset:

```

```

   $\Gamma' \text{procname} = \text{Some} (\text{Spec } R) \implies$ 
   $\text{HoarePartialDef.valid } \Gamma' \text{NF } P (\text{Call procname}) \ Q \ A = (P \subseteq \text{fst } 'R \wedge (R \subseteq (-$ 
   $P) \times \text{UNIV} \cup \text{UNIV} \times Q))$ 

```

```

apply (safe; simp?)

```

```

subgoal for  $x$ 

```

```

  apply (clarsimp simp: HoarePartialDef.valid-def)

```

```

  apply (rule ccontr)

```

```

  apply (drule-tac  $x = \text{Normal } x$  in spec, elim allE)

```

```

  apply (drule mp, erule exec.Call, rule exec.SpecStuck)

```

```

  apply (auto simp: image-def)[2]

```

```

done

```

```

apply (clarsimp simp: HoarePartialDef.valid-def)

```

```

apply (elim allE, drule mp, erule exec.Call, erule exec.Spec)

```

```

apply auto[1]

```

```

apply (clarsimp simp: HoarePartialDef.valid-def)

```

```

apply (erule exec-Normal-elim-cases, simp-all)

```

apply (*erule exec-Normal-elim-cases, auto*)
done

lemma *creturn-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s. (\text{exnupd } (\lambda-. \text{Return})) (\text{rvupd } (\lambda-. v s) s) \in A\}$
shows $\Gamma, \Theta \vdash_{/F} P \text{creturn exnupd rvupd } v Q, A$
unfolding *creturn-def*
by *vcg*

lemma *creturn-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \{s. (\text{exnupd } (\lambda-. \text{Return})) (\text{rvupd } (\lambda-. v s) s) \in A\}$
shows $\Gamma, \Theta \vdash_{t/F} P \text{creturn exnupd rvupd } v Q, A$
unfolding *creturn-def*
by *vcg*

lemma *creturn-void-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s. (\text{exnupd } (\lambda-. \text{Return})) s \in A\}$
shows $\Gamma, \Theta \vdash_{/F} P \text{creturn-void exnupd } Q, A$
unfolding *creturn-void-def*
by *vcg*

lemma *creturn-void-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \{s. (\text{exnupd } (\lambda-. \text{Return})) s \in A\}$
shows $\Gamma, \Theta \vdash_{t/F} P \text{creturn-void exnupd } Q, A$
unfolding *creturn-void-def*
by *vcg*

lemma *cbreak-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s. (\text{exnupd } (\lambda-. \text{Break})) s \in A\}$
shows $\Gamma, \Theta \vdash_{/F} P \text{cbreak exnupd } Q, A$
unfolding *cbreak-def*
by *vcg*

lemma *cbreak-wp-totoal* [*vcg-hoare*]:
assumes $P \subseteq \{s. (\text{exnupd } (\lambda-. \text{Break})) s \in A\}$
shows $\Gamma, \Theta \vdash_{t/F} P \text{cbreak exnupd } Q, A$
unfolding *cbreak-def*
by *vcg*

lemma *ccatchbrk-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s. (\text{exnupd } s = \text{Break} \longrightarrow s \in Q) \wedge$
 $(\text{exnupd } s \neq \text{Break} \longrightarrow s \in A)\}$
shows $\Gamma, \Theta \vdash_{/F} P \text{ccatchbrk exnupd } Q, A$
unfolding *ccatchbrk-def*
by *vcg*

lemma *ccatchbrk-wp-total* [vcg-hoare]:
assumes $P \subseteq \{s. (\text{exnupd } s = \text{Break} \longrightarrow s \in Q) \wedge$
 $(\text{exnupd } s \neq \text{Break} \longrightarrow s \in A)\}$
shows $\Gamma, \Theta \vdash_{t/F} P \text{ ccatchbrk exnupd } Q, A$
unfolding *ccatchbrk-def*
by *vcg*

lemma *cgoto-wp* [vcg-hoare]:
assumes $P \subseteq \{s. (\text{exnupd } (\lambda-. \text{Goto } l)) s \in A\}$
shows $\Gamma, \Theta \vdash_{t/F} P \text{ cgoto } l \text{ exnupd } Q, A$
unfolding *cgoto-def*
by *vcg*

lemma *cgoto-wp-total* [vcg-hoare]:
assumes $P \subseteq \{s. (\text{exnupd } (\lambda-. \text{Goto } l)) s \in A\}$
shows $\Gamma, \Theta \vdash_{t/F} P \text{ cgoto } l \text{ exnupd } Q, A$
unfolding *cgoto-def*
by *vcg*

lemma *ccatchgoto-wp* [vcg-hoare]:
assumes $P \subseteq \{s. (\text{exnupd } s = \text{Goto } l \longrightarrow s \in Q) \wedge$
 $(\text{exnupd } s \neq \text{Goto } l \longrightarrow s \in A)\}$
shows $\Gamma, \Theta \vdash_{t/F} P \text{ ccatchgoto } l \text{ exnupd } Q, A$
unfolding *ccatchgoto-def*
by *vcg*

lemma *ccatchgoto-wp-total* [vcg-hoare]:
assumes $P \subseteq \{s. (\text{exnupd } s = \text{Goto } l \longrightarrow s \in Q) \wedge$
 $(\text{exnupd } s \neq \text{Goto } l \longrightarrow s \in A)\}$
shows $\Gamma, \Theta \vdash_{t/F} P \text{ ccatchgoto } l \text{ exnupd } Q, A$
unfolding *ccatchgoto-def*
by *vcg*

lemma *ccatchreturn-wp* [vcg-hoare]:
assumes $P \subseteq \{s. (\text{is-local } (\text{exnupd } s) \longrightarrow s \in Q) \wedge$
 $(\neg \text{is-local } (\text{exnupd } s) \longrightarrow s \in A)\}$
shows $\Gamma, \Theta \vdash_{t/F} P \text{ ccatchreturn exnupd } Q, A$
unfolding *ccatchreturn-def*
by *vcg*

lemma *ccatchreturn-wp-total* [vcg-hoare]:
assumes $P \subseteq \{s. (\text{is-local } (\text{exnupd } s) \longrightarrow s \in Q) \wedge$
 $(\neg \text{is-local } (\text{exnupd } s) \longrightarrow s \in A)\}$
shows $\Gamma, \Theta \vdash_{t/F} P \text{ ccatchreturn exnupd } Q, A$
unfolding *ccatchreturn-def*
by *vcg*

lemma *cexit-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s. \text{exnupd } s \in A\}$
shows $\Gamma, \Theta \vdash_{/F} P \text{ cexit exnupd } Q, A$
unfolding *cexit-def*
by *vcg*

lemma *cexit-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \{s. \text{exnupd } s \in A\}$
shows $\Gamma, \Theta \vdash_{t/F} P \text{ cexit exnupd } Q, A$
unfolding *cexit-def*
by *vcg*

lemma *cchaos-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s. \forall x. (v \ x \ s) \in Q\}$
shows $\Gamma, \Theta \vdash_{/F} P \text{ cchaos } v \ Q, A$
unfolding *cchaos-def*
using *assms* **by** (*blast intro: HoarePartial.Spec*)

lemma *cchaos-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \{s. \forall x. (v \ x \ s) \in Q\}$
shows $\Gamma, \Theta \vdash_{t/F} P \text{ cchaos } v \ Q, A$
unfolding *cchaos-def*
using *assms* **by** (*blast intro: HoareTotal.Spec*)

lemma *lvar-nondet-init-wp* [*vcg-hoare*]:
 $P \subseteq \{s. \forall v. (\text{upd } (\lambda \cdot. v)) \ s \in Q\} \implies \Gamma, \Theta \vdash_{/F} P \text{ lvar-nondet-init upd } Q, A$
unfolding *lvar-nondet-init-def*
by (*rule HoarePartialDef.conseqPre, vcg, auto*)

lemma *lvar-nondet-init-wp-total* [*vcg-hoare*]:
 $P \subseteq \{s. \forall v. (\text{upd } (\lambda \cdot. v)) \ s \in Q\} \implies \Gamma, \Theta \vdash_{t/F} P \text{ lvar-nondet-init upd } Q, A$
unfolding *lvar-nondet-init-def*
by (*rule HoareTotalDef.conseqPre, vcg, auto*)

lemma *Seq-propagate-precond*:
 $[[\Gamma, \Theta \vdash_{/F} P \ c_1 \ P, A; \Gamma, \Theta \vdash_{/F} P \ c_2 \ Q, A]] \implies \Gamma, \Theta \vdash_{/F} P \ (\text{Seq } c_1 \ c_2) \ Q, A$
apply (*rule hoarep.Seq*)
apply *assumption*
apply *assumption*
done

lemma *Seq-propagate-precond-total*:
 $[[\Gamma, \Theta \vdash_{t/F} P \ c_1 \ P, A; \Gamma, \Theta \vdash_{t/F} P \ c_2 \ Q, A]] \implies \Gamma, \Theta \vdash_{t/F} P \ (\text{Seq } c_1 \ c_2) \ Q, A$
apply (*rule hoaret.Seq*)
apply *assumption*
apply *assumption*
done

```

lemma mem-safe-lvar-init [simp,intro]:
  assumes upd:  $\bigwedge g v s. \text{globals-update } g (\text{upd } (\lambda-. v) s) = \text{upd } (\lambda-. v) (\text{globals-update } g s)$ 
  assumes acc:  $\bigwedge v s. \text{globals } (\text{upd } (\lambda-. v) s) = \text{globals } s$ 
  shows mem-safe (lvar-nondet-init upd) x
  apply (clarsimp simp: mem-safe-def lvar-nondet-init-def)
  apply (erule exec.cases; clarsimp simp: restrict-safe-def)
  apply (fastforce simp: restrict-safe-OK-def restrict-htd-def upd acc intro: exec.Spec)
  apply (simp add: exec-fatal-def)
  apply (fastforce simp: exec-fatal-def restrict-htd-def upd acc intro!: exec.SpecStuck)
  done

lemma intra-safe-lvar-nondet-init [simp]:
  intra-safe (lvar-nondet-init upd :: (('a::heap-state-type', 'l, 'e, 'x) state-scheme, 'p, 'f) com) =
  ( $\forall \Gamma. \text{mem-safe } (\text{lvar-nondet-init upd} :: (('a::\text{heap-state-type}', 'l, 'e, 'x) \text{state-scheme}, 'p, 'f) \text{com}))$ 
  ( $\Gamma :: (('a, 'l, 'e, 'x) \text{state-scheme}, 'p, 'f) \text{body}$ ))
  by (simp add: lvar-nondet-init-def)

lemma proc-deps-lvar-nondet-init [simp]:
  proc-deps (lvar-nondet-init upd)  $\Gamma = \{\}$ 
  by (simp add: lvar-nondet-init-def)

declare word-neq-0-conv[simp]

declare [[hoare-use-generalise=true]]
end

theory Array-Selectors
  imports
    CTranslationSetup
  keywords array-selectors :: thy-defn
begin

named-theorems array-selectors-simps

lemmas [array-selectors-simps] =— numerals
  Arrays.arr-fupdate-same
  Arrays.arr-fupdate-other
  Numeral-Type.card-num0
  Numeral-Type.card-num1
  Numeral-Type.card-bit0
  Numeral-Type.card-bit1
  Nat.One-nat-def
  Nat.mult-Suc-right

```

Nat.mult-0-right
Nat.Suc-not-Zero
Num.Suc-numeral
Num.eq-numeral-Suc
Num.Suc-eq-numeral
Num.less-Suc-numeral
Num.mult-num-simps
Num.add-num-simps
Num.pred-numeral-simps
Num.numeral-times-numeral
Num.num.distinct
Num.num.inject
Nat.add-0-right
Num.arith-simps
Num.more-arith-simps
Num.rel-simps
Nat.Zero-not-Suc

lemmas [*array-selectors-simps*] =
comp-apply
fcp-beta

ML-file *array-selectors.ML*

experiment
begin

definition *my-array* \equiv *fupdate* 3
(*apfst* (λ -. *0x30*) *o* *apsnd* (λ -. *43*))
(*fupdate* 2 (*apfst* (λ -. *0x20*) *o* *apsnd* (λ -. *42*))
(*fupdate* 1 (*apfst* (λ -. *0x10*) *o* *apsnd* (λ -. *41*))
(*fupdate* 0 (*apfst* (λ -. *0x0*) *o* *apsnd* (λ -. *40*))
(*ARRAY* -. (*0*, *0*))::(*32 word* \times *nat*)[*4*]))

lemmas [*array-selectors-simps*] =
apfst-conv
apsnd-conv

— in applications, the update functions are from the recursive record package, therefore the recursive record package simpset is included by default

array-selectors (*no-recursive-record-simpset*)— does not make a difference here
my-array-sels **is** *my-array-def*

lemma *my-array*.*[0]* \equiv (*0*, *40*)
my-array.*[1]* \equiv (*0x10*, *41*)
my-array.*[Suc 0]* \equiv (*0x10*, *41*)
my-array.*[2]* \equiv (*0x20*, *42*)
my-array.*[3]* \equiv (*0x30*, *43*)
by (*rule my-array-sels*)**+**

end

end

theory *CTranslation*

imports

CTranslationSetup

Array-Selectors

keywords

new-C-include-dir:: thy-decl

and

include-C-file

install-C-types

install-C-file :: thy-load

begin

ML-file *isar-install.ML*

end

Chapter 15

Misc. Lemmas

```
theory TypHeapLib
imports CTranslation
begin
```

15.1 Abbreviations and helpers

```
definition is-an-abbreviation  $\equiv$  True
```

```
abbreviation
  clift  $\equiv$  lift-t c-guard
```

```
lemma clift-def: is-an-abbreviation by (simp add: is-an-abbreviation-def)
```

15.2 Basic operations

15.2.1 clift

```
lemma c-guard-clift:
  clift hp p = Some x  $\implies$  c-guard p
  by (rule lift-t-g)
```

```
lemma clift-heap-update:
  fixes p :: 'a :: mem-type ptr
  shows hrs-htd hp  $\models_t$  p  $\implies$  clift (hrs-mem-update (heap-update p v) hp) = (clift
hp)(p  $\mapsto$  v)
  unfolding hrs-mem-update-def
  apply (cases hp)
  apply (simp add: split-def hrs-htd-def)
  apply (erule lift-t-heap-update)
  done
```

```
lemma clift-heap-update-same:
  fixes p :: 'a :: mem-type ptr
```

shows $\llbracket \text{hrs-htd } hp \models_t p; \text{typ-uinfo-t } TYPE('a) \perp_t \text{typ-uinfo-t } TYPE('b) \rrbracket$
 $\implies \text{clift } (\text{hrs-mem-update } (\text{heap-update } p \ v) \ hp) = (\text{clift } hp :: 'b :: \text{mem-type}$
 $\text{typ-heap})$
unfolding *hrs-mem-update-def*
apply (*cases hp*)
apply (*simp add: split-def hrs-htd-def*)
apply (*erule lift-t-heap-update-same*)
apply *simp*
done

lemmas *clift-heap-update-same-td-name = clift-heap-update-same [OF - tag-disj-via-td-name,*
unfolded pad-typ-name-def]

15.2.2 *h-val*

lemmas *h-val-clift = lift-t-lift [where g = c-guard, unfolded CTypesDefs.lift-def,*
simplified]

lemma *h-val-clift'*:

$\text{clift } hp \ p = \text{Some } v \implies \text{h-val } (\text{hrs-mem } hp) \ p = v$
unfolding *hrs-mem-def* **by** (*cases hp, simp add: h-val-clift*)

15.2.3 *h-t-valid*

lemma *clift-Some-eq-valid*:

$(\exists v. \text{clift } hp \ p = \text{Some } v) = (\text{hrs-htd } hp \models_t p)$
apply (*cases hp*)
apply (*simp add: lift-t-if hrs-htd-def*)
done

lemma *h-t-valid-clift-Some-iff*:

$(\text{hrs-htd } hp \models_t p) = (\exists v. \text{clift } hp \ p = \text{Some } v)$
apply (*cases hp*)
apply (*simp add: lift-t-if hrs-htd-def*)
done

lemma *h-t-valid-clift*:

$\text{clift } hp \ p = \text{Some } v \implies \text{hrs-htd } hp \models_t p$
apply (*cases hp, simp add: hrs-htd-def*)
apply (*erule lift-t-h-t-valid*)
done

lemma *c-guard-h-t-valid*:

$\text{hrs-htd } hp \models_t p \implies \text{c-guard } p$
by (*simp add: h-t-valid-def*)

15.3 *field-lvalue*

15.3.1 *heap-update*

lemma *heap-update-field*:

```
[[field-ti TYPE('a :: packed-type) f = Some t; c-guard p;  
export-uinfo t = export-uinfo (typ-info-t TYPE('b :: packed-type))]]  
⇒ heap-update (Ptr &(p→f) :: 'b ptr) v hp =  
heap-update p (update-ti t (to-bytes-p v) (h-val hp p)) hp  
apply (erule field-ti-field-lookupE)  
apply (erule (2) packed-heap-super-field-update [unfolded typ-uinfo-t-def])  
done
```

lemma *heap-update-field'*:

```
[[field-ti TYPE('a :: packed-type) f = Some t; c-guard p;  
export-uinfo t = export-uinfo (typ-info-t TYPE('b :: packed-type))]]  
⇒ heap-update (Ptr &(p→f) :: 'b ptr) v hp =  
heap-update p (update-ti-t t (to-bytes-p v) (h-val hp p)) hp  
apply (erule field-ti-field-lookupE)  
apply (subst packed-heap-super-field-update [unfolded typ-uinfo-t-def])  
apply assumption+  
apply (drule export-size-of [simplified typ-uinfo-t-def])  
apply (simp add: update-ti-t-def)  
done
```

lemma *heap-update-field-hrs*:

```
fixes p :: 'a :: packed-type ptr and v :: 'b :: packed-type  
shows [[field-ti TYPE('a) f = Some t; c-guard p;  
export-uinfo t = export-uinfo (typ-info-t TYPE('b))]]  
⇒ hrs-mem-update (heap-update (Ptr &(p→f)) v) hp =  
hrs-mem-update (heap-update p (update-ti-t t (to-bytes-p v) (h-val (hrs-mem hp)  
p))) hp  
unfolding hrs-mem-update-def  
apply (simp add: split-def)  
apply (subst heap-update-field)  
apply assumption  
apply assumption  
apply assumption  
apply (frule export-size-of [unfolded typ-uinfo-t-def])  
apply (simp add: update-ti-t-def hrs-mem-def)  
done
```

lemma *heap-update-field-ext*:

```
[[field-ti TYPE('a :: packed-type) f = Some t; c-guard p;  
export-uinfo t = export-uinfo (typ-info-t TYPE('b :: packed-type))]]  
⇒ heap-update (Ptr &(p→f) :: 'b ptr) =  
(λv hp. heap-update p (update-ti t (to-bytes-p v) (h-val hp p)) hp)  
apply (rule ext, rule ext)  
apply (erule (2) heap-update-field)  
done
```

15.3.2 *c-guard*

lemma *c-guard-field*:

```
[[c-guard (p :: 'a :: mem-type ptr); field-ti TYPE('a :: mem-type) f = Some t;
export-uinfo t = export-uinfo (typ-info-t TYPE('b :: mem-type))]]
⇒ c-guard (Ptr &(p→f) :: 'b :: mem-type ptr)
apply (erule field-ti-field-lookupE)
apply (erule (1) c-guard-field-lvalue)
apply (simp add: typ-uinfo-t-def)
done
```

15.3.3 *clift*

lemma *clift-field*:

```
fixes v :: 'a :: mem-type and p :: 'a :: mem-type ptr
assumes lf: clift hp p = Some v
and fl: field-ti TYPE('a) f = Some t
and eu: export-uinfo t = export-uinfo (typ-info-t TYPE('b :: mem-type))
shows clift hp (Ptr &(p→f) :: 'b :: mem-type ptr) = Some (from-bytes (access-ti0
t v))
using lf fl eu
apply (clarsimp elim!: field-ti-field-lookupE)
apply (erule (2) lift-t-mono [unfolded typ-uinfo-t-def])
apply (rule c-guard-mono)
done
```

Updates

lemma *clift-field-update*:

```
fixes val :: 'b :: mem-type and ptr :: 'a :: mem-type ptr
assumes fl: field-ti TYPE('a) f = Some t
and eu: export-uinfo t = export-uinfo (typ-info-t TYPE('b))
and cl: clift hp ptr = Some z
shows (clift (hrs-mem-update (heap-update (Ptr &(ptr→f)) val) hp)) =
(clift hp)(ptr ↦ field-update (field-desc t) (to-bytes-p val) z)
(is ?LHS = ?RHS)
```

proof –

```
have cl': clift (fst hp, snd hp) ptr = Some z using cl by simp

have ?LHS = super-field-update-t (Ptr &(ptr→f)) val (clift (fst hp, snd hp))
unfolding hrs-mem-update-def split-def
proof (rule lift-t-super-field-update [OF h-t-valid-sub, unfolded typ-uinfo-t-def])
from cl' show snd hp ⊨t ptr by (rule lift-t-h-t-valid)
show TYPE('b) ≤τ TYPE('a) using fl eu by (rule field-ti-sub-typ [unfolded
typ-uinfo-t-def])
qed fact+
```

```
also have ... = ?RHS using fl eu cl
apply –
apply (erule field-ti-field-lookupE)
```

```

apply (subst super-field-update-lookup [unfolded typ-uinfo-t-def])
apply assumption
apply simp
apply simp
apply simp
done

```

finally show ?thesis .
qed

lemma *clift-field-update-padding*:

```

fixes val :: 'b :: mem-type and ptr :: 'a :: mem-type ptr
assumes fl: field-ti TYPE('a) f = Some t
and eu: export-uinfo t = export-uinfo (typ-info-t TYPE('b))
and cl: clift hp ptr = Some z
and lbs: length bs = size-of TYPE('b)
shows (clift (hrs-mem-update (heap-update-padding (Ptr &(ptr→f)) val bs) hp))
=
  (clift hp)(ptr ↦ field-update (field-desc t) (to-bytes-p val) z)
  (is ?LHS = ?RHS)

```

proof –

```

have cl': clift (fst hp, snd hp) ptr = Some z using cl by simp

```

```

have ?LHS = super-field-update-t (Ptr &(ptr→f)) val (clift (fst hp, snd hp))
  unfolding hrs-mem-update-def split-def

```

```

proof (rule lift-t-super-field-update-padding [OF h-t-valid-sub - lbs, unfolded typ-uinfo-t-def])

```

```

  from cl' show snd hp  $\models_t$  ptr by (rule lift-t-h-t-valid)

```

```

  show TYPE('b)  $\leq_\tau$  TYPE('a) using fl eu by (rule field-ti-sub-typ [unfolded
typ-uinfo-t-def])

```

```

qed fact+

```

```

also have ... = ?RHS using fl eu cl

```

```

apply –

```

```

apply (erule field-ti-field-lookupE)

```

```

apply (subst super-field-update-lookup [unfolded typ-uinfo-t-def])

```

```

apply assumption

```

```

apply simp

```

```

apply simp

```

```

apply simp

```

```

done

```

finally show ?thesis .
qed

15.3.4 cparent

definition

```

cparent :: ('a :: c-type) ptr  $\Rightarrow$  string list  $\Rightarrow$  ('b :: c-type) ptr
where

```

$cparent\ p\ fs \equiv THE\ p'.\ p = Ptr\ \&(p' \rightarrow fs)$

lemma *cparent-field* [simp]:
 $cparent\ (Ptr\ \&(p \rightarrow fs))\ fs = p$
unfolding *cparent-def*
by (*simp add: field-lvalue-def*)

lemma *cparent-def2*:
fixes $p :: 'b :: c\text{-type}\ ptr$
shows $cparent\ p\ f \equiv (Ptr\ (ptr\text{-val}\ p - of\text{-nat}\ (field\text{-offset}\ TYPE('a :: c\text{-type})\ f))$
 $:: 'a :: c\text{-type}\ ptr)$
(is $cparent\ p\ f \equiv ?p')$
proof –
have $pv: p = Ptr\ \&(p' \rightarrow f)$
by (*simp add: field-lvalue-def field-simps*)
show $cparent\ p\ f \equiv ?p'$
by (*subst pv*) *simp*
qed

lemma *field-cparent* [simp]:
fixes $p :: 'a :: c\text{-type}\ ptr$
shows $(Ptr\ \&(cparent\ p\ f :: 'b :: c\text{-type}\ ptr \rightarrow f)) = p$
by (*simp add: cparent-def2 field-lvalue-def*)

lemma *clift-cparentE*:
fixes $v :: 'a :: mem\text{-type}$ **and** $p :: 'b :: mem\text{-type}\ ptr$
assumes $lf: clift\ hp\ (cparent\ p\ fs :: 'a\ ptr) = Some\ v$
and $ft: field\text{-ti}\ TYPE('a)\ fs = Some\ t$
and $eu: export\text{-uinfo}\ t = export\text{-uinfo}\ (typ\text{-info}\text{-t}\ TYPE('b))$
shows $clift\ hp\ p = Some\ (from\text{-bytes}\ (access\text{-ti}_0\ t\ v))$
unfolding *cparent-def*
proof –
have $pv: p = Ptr\ \&((Ptr\ (ptr\text{-val}\ p - of\text{-nat}\ (field\text{-offset}\ TYPE('a)\ fs)) :: 'a$
 $ptr) \rightarrow fs)$
(is $p = Ptr\ \&(p' \rightarrow fs)$) **by** (*simp add: field-lvalue-def field-simps*)
have $cp: cparent\ p\ fs = ?p'$
by (*subst pv*) *simp*

from lf **have** $lf': clift\ hp\ ?p' = Some\ v$
by (*simp add: cparent-def2*)

have $clift\ hp\ (Ptr\ \&(p' \rightarrow fs) :: 'b\ ptr) = Some\ (from\text{-bytes}\ (access\text{-ti}_0\ t\ v))$ **using**
 $lf'\ ft\ eu$
by (*rule clift-field*)

thus *?thesis*
by (*simp add: pv [symmetric]*)
qed

lemma *heap-update-to-cparent*:
fixes $p :: 'b :: \text{packed-type ptr}$ **and** $fs :: \text{char list list}$
defines $cp \equiv \text{cparent } p$ $fs :: 'a :: \text{packed-type ptr}$
assumes $fl: \text{field-ti TYPE}('a :: \text{packed-type}) fs = \text{Some } t$
and $cg: \text{c-guard } cp$
and $eu: \text{export-uinfo } t = \text{export-uinfo } (\text{typ-info-t TYPE}('b))$
shows $\text{heap-update } p \ v \ hp = \text{heap-update } cp \ (\text{update-ti } t \ (\text{to-bytes-p } v) \ (\text{h-val } hp \ cp)) \ hp$
(is $?LHS = ?RHS$ **)**
proof –
have $?LHS = \text{heap-update } (Ptr \ \&(cp \rightarrow fs)) \ v \ hp$
unfolding $cp\text{-def}$ **by** $simp$

moreover have $\dots = ?RHS$ **using** $fl \ cg \ eu$
by $(\text{fastforce intro: heap-update-field})$

finally show $?thesis$.
qed

lemma *c-guard-cparent*:
 $\llbracket \text{c-guard } ((\text{cparent } p \ f)::'a::\text{mem-type ptr});$
 $\text{field-ti TYPE}('a) \ f = \text{Some } t;$
 $\text{export-uinfo } t = \text{typ-uinfo-t TYPE}('b) \rrbracket \implies$
 $\text{c-guard } (p::'b::\text{mem-type ptr})$
apply $(\text{subst field-cparent}[\text{symmetric}, \text{where } f=f])$
apply $(\text{rule c-guard-field}, (\text{simp add:typ-uinfo-t-def})+)$
done

lemma *parent-update-child*:
fixes $p::'b::\text{packed-type ptr}$
shows
 $\llbracket \text{c-guard } ((\text{cparent } p \ f)::'a::\text{packed-type ptr}); \text{field-ti TYPE}('a) \ f = \text{Some } t;$
 $\text{export-uinfo } t = \text{export-uinfo } (\text{typ-info-t TYPE}('b)) \rrbracket$
 $\implies \text{hrs-mem-update } (\text{heap-update } p \ v) \ hp =$
 hrs-mem-update
 $(\text{heap-update } ((\text{cparent } p \ f)::'a \ \text{ptr})$
 $(\text{update-ti-t } t \ (\text{to-bytes-p } v) \ (\text{h-val } (\text{hrs-mem } hp) \ (\text{cparent } p \ f))))$
 hp
apply $(\text{subst field-cparent}[\text{symmetric}, \text{where } f=f \ \text{and } 'b='a])$
apply $(\text{rule heap-update-field-hrs}, \text{simp}+)$
done

15.3.5 *h-val*

lemma *h-val-field-clift*:
fixes $pa :: 'a :: \text{mem-type ptr}$
assumes $cl: \text{clift } (h, d) \ pa = \text{Some } v$
and $fl: \text{field-ti TYPE}('a) \ f = \text{Some } t$


```

and    eu: export-uinto t = export-uinto (typ-info-t TYPE('b :: mem-type))
shows  h-val h (Ptr &(pa→f) :: 'b :: mem-type ptr) = from-bytes (access-ti0 t
v)
using cl ft eu
apply -
apply (rule h-val-clift)
apply (clarsimp simp: field-ti-def split: option.splits)
apply (erule (2) lift-t-mono [unfolded typ-uinto-t-def])
apply (rule c-guard-mono)
done

```

lemma *h-val-field-clift'*:

```

fixes pa :: 'a :: mem-type ptr
assumes cl: clift hp pa = Some v
and    ft: field-ti TYPE('a) f = Some t
and    eu: export-uinto t = typ-uinto-t TYPE('b :: mem-type)
shows  h-val (hrs-mem hp) (Ptr &(pa→f) :: 'b :: mem-type ptr) = from-bytes
(access-ti0 t v)
using cl ft eu
apply (cases hp)
apply (simp add: h-val-field-clift hrs-mem-def typ-uinto-t-def)
done

```

lemma *clift-subtype*:

```

[[ clift hp ((cparent p f)::'a::mem-type ptr) = Some v;
  field-ti TYPE('a) f = Some t;
  export-uinto t = export-uinto (typ-info-t TYPE('b::mem-type)) ]] ==>
clift hp (p::'b ptr) = Some (from-bytes (access-ti0 t v))
apply(subst field-cparent[symmetric, where f=f])
apply(rule clift-field)
apply(simp)+
done

```

15.3.6 *h-t-valid*

lemma *h-t-valid-field*:

```

fixes p :: 'a :: mem-type ptr
assumes htv: d ⊨t p
and    fti: field-ti TYPE('a :: mem-type) f = Some t
and    eu: export-uinto t = export-uinto (typ-info-t TYPE('b :: mem-type))
shows  d ⊨t (Ptr &(p→f) :: 'b :: mem-type ptr)
using htv fti eu
apply -
apply (clarsimp simp: field-ti-def split: option.splits)
apply (erule (1) h-t-valid-mono [rule-format, unfolded typ-uinto-t-def])
apply (rule c-guard-mono)
apply assumption
done

```

lemma *h-t-valid-field'*:
fixes $p::'a::mem\text{-}type\ ptr$
shows
 $\llbracket field\text{-}ti\ TYPE('a)\ f = Some\ t;$
 $export\text{-}uinfo\ t = typ\text{-}uinfo\text{-}t\ TYPE('b);$
 $d, g \models_t p; g' ((Ptr\ \&(p \rightarrow f))::'b::mem\text{-}type\ ptr) \rrbracket \implies d, g' \models_t Ptr\ \&(p \rightarrow f)$
apply(*simp add:h-t-valid-guard-subst[OF h-t-valid-sub]*)
done

lemma *h-t-valid-c-guard-field*:
fixes $p::'a::mem\text{-}type\ ptr$
shows
 $\llbracket d \models_t p;$
 $field\text{-}ti\ TYPE('a)\ f = Some\ t;$
 $export\text{-}uinfo\ t = typ\text{-}uinfo\text{-}t\ TYPE('b) \rrbracket \implies$
 $d \models_t ((Ptr\ \&(p \rightarrow f))::'b::mem\text{-}type\ ptr)$
apply(*simp add:h-t-valid-field c-guard-field h-t-valid-c-guard typ-uinfo-t-def*)
done

lemma *h-t-valid-cparent*:
 $\llbracket field\text{-}ti\ TYPE('a)\ f = Some\ t;$
 $export\text{-}uinfo\ t = typ\text{-}uinfo\text{-}t\ TYPE('b);$
 $d, g \models_t ((cparent\ p\ f)::'a::mem\text{-}type\ ptr); g' (p::'b::mem\text{-}type\ ptr) \rrbracket \implies$
 $d, g' \models_t p$
apply(*subst field-cparent[symmetric, where f=f]*)
apply(*rule h-t-valid-field'*)
apply(*assumption*)
apply(*simp add:typ-uinfo-t-def*)
apply(*assumption*)
apply(*simp*)
done

lemma *h-t-valid-c-guard-cparent*:
fixes $p::'b::mem\text{-}type\ ptr$
shows
 $\llbracket d \models_t ((cparent\ p\ f)::'a::mem\text{-}type\ ptr);$
 $field\text{-}ti\ TYPE('a)\ f = Some\ t;$
 $export\text{-}uinfo\ t = typ\text{-}uinfo\text{-}t\ TYPE('b) \rrbracket \implies$
 $d \models_t p$
apply(*rule h-t-valid-cparent*)
apply(*assumption*)
apply(*simp add:typ-uinfo-t-def*)
apply(*assumption*)
apply(*rule c-guard-cparent*)
apply(*rule h-t-valid-c-guard, assumption*)
apply(*assumption*)
apply(*simp add:typ-uinfo-t-def*)
done

```

lemma c-guard-array-c-guard:
  c-guard (ptr-coerce p :: ('b :: c-type, 'a :: finite) array ptr)  $\implies$  c-guard (p :: 'b
ptr)
  apply (clarsimp simp: c-guard-def)
  apply (rule conjI)
  apply (clarsimp simp: ptr-aligned-def align-of-def align-td-array)
  apply (simp add: c-null-guard-def)
  apply (erule contra-subsetD [rotated])
  apply (rule intvl-start-le)
  apply simp
  done

lemma c-guard-array-field:
  assumes parent-cguard: c-guard (p :: 'a :: mem-type ptr)
  and subfield: field-ti TYPE('a :: mem-type) f = Some t
  and type-match: export-uinfo t = export-uinfo (typ-info-t TYPE(''b :: array-outer-max-size,
'c :: array-max-count) array))
  shows c-guard (Ptr &(p→f) :: 'b ptr)
  by (metis c-guard-array-c-guard c-guard-field parent-cguard ptr-coerce.simps sub-
field type-match)

instantiation ptr :: (type) enum
begin

  definition enum-ptr  $\equiv$  map Ptr enum-class.enum
  definition enum-all-ptr P  $\equiv$  enum-class.enum-all ( $\lambda v. P (Ptr v)$ )
  definition enum-ex-ptr P  $\equiv$  enum-class.enum-ex ( $\lambda v. P (Ptr v)$ )

instance
  apply (intro-classes)
  apply (clarsimp simp: enum-ptr-def)
  apply (metis ptr.exhaust surj-def)
  apply (clarsimp simp: enum-ptr-def distinct-map)
  apply (metis injI ptr.inject)
  apply (clarsimp simp: enum-all-ptr-def)
  apply (rule iffI)
  apply (rule allI)
  apply (rename-tac x)
  apply (erule-tac x=ptr-val x in allE)
  apply force
  apply force
  apply (clarsimp simp: enum-ex-ptr-def)
  apply (rule iffI)
  apply force
  apply clarsimp
  subgoal for P x
  apply (rule exI[where x=ptr-val x])
  apply clarsimp
  done

```

done
end

15.3.7 Type Combinators and Padding

lemma *ti-typ-pad-combine-empty-ti*:

fixes $tp :: 'b :: c\text{-type}\ \textit{itself}$
shows $ti\text{-typ}\text{-pad}\text{-combine}\ tp\ lu\ upd\ algn\ fld\ (empty\text{-typ}\text{-info}\ algn'\ n) =$
 $TypDesc\ (max\ algn'\ (max\ algn\ (align\text{-td}\ (typ\text{-info}\text{-t}\ TYPE('b))))))$
 $(TypAggregate\ [DTuple\ (adjust\text{-ti}\ (typ\text{-info}\text{-t}\ TYPE('b))\ lu\ upd)\ fld$
 $(\backslash field\text{-access} = xto\text{-bytes} \circ lu,$
 $field\text{-update} = upd \circ xfrom\text{-bytes},$
 $field\text{-sz} = size\text{-of}\ TYPE('b)\backslash)])\ n$
by (*simp add: ti-typ-pad-combine-def ti-typ-combine-def empty-typ-info-def Let-def*)

lemma *ti-typ-combine-empty-ti*:

fixes $tp :: 'b :: c\text{-type}\ \textit{itself}$
shows $ti\text{-typ}\text{-combine}\ tp\ lu\ upd\ algn\ fld\ (empty\text{-typ}\text{-info}\ algn'\ n) =$
 $TypDesc\ (max\ algn'\ (max\ algn\ (align\text{-td}\ (typ\text{-info}\text{-t}\ TYPE('b))))))$
 $(TypAggregate\ [DTuple\ (adjust\text{-ti}\ (typ\text{-info}\text{-t}\ TYPE('b))\ lu\ upd)\ fld$
 $(\backslash field\text{-access} = xto\text{-bytes} \circ lu,$
 $field\text{-update} = upd \circ xfrom\text{-bytes},$
 $field\text{-sz} = size\text{-of}\ TYPE('b)\backslash)])\ n$
by (*simp add: ti-typ-combine-def empty-typ-info-def Let-def*)

lemma *ti-typ-pad-combine-td*:

fixes $tp :: 'b :: c\text{-type}\ \textit{itself}$
shows $padup\ (max\ (2 \wedge algn)\ (align\text{-of}\ TYPE('b)))\ (size\text{-td}\text{-struct}\ st) = 0 \implies$
 $ti\text{-typ}\text{-pad}\text{-combine}\ tp\ lu\ upd\ algn\ fld\ (TypDesc\ algn'\ st\ n) =$
 $TypDesc\ (max\ algn'\ (max\ algn\ (align\text{-td}\ (typ\text{-info}\text{-t}\ TYPE('b))))))$
 $(extend\text{-ti}\text{-struct}\ st\ (adjust\text{-ti}\ (typ\text{-info}\text{-t}\ TYPE('b))\ lu\ upd)\ fld$
 $(\backslash field\text{-access} = xto\text{-bytes} \circ lu,$
 $field\text{-update} = upd \circ xfrom\text{-bytes},$
 $field\text{-sz} = size\text{-of}\ TYPE('b)\backslash)])\ n$
by (*simp add: ti-typ-pad-combine-def ti-typ-combine-def Let-def*)

lemma *ti-typ-combine-td*:

fixes $tp :: 'b :: c\text{-type}\ \textit{itself}$
shows $padup\ (align\text{-of}\ TYPE('b))\ (size\text{-td}\text{-struct}\ st) = 0 \implies$
 $ti\text{-typ}\text{-combine}\ tp\ lu\ upd\ algn\ fld\ (TypDesc\ algn'\ st\ n) =$
 $TypDesc\ (max\ algn'\ (max\ algn\ (align\text{-td}\ (typ\text{-info}\text{-t}\ TYPE('b))))))$
 $(extend\text{-ti}\text{-struct}\ st\ (adjust\text{-ti}\ (typ\text{-info}\text{-t}\ TYPE('b))\ lu\ upd)\ fld$
 $(\backslash field\text{-access} = xto\text{-bytes} \circ lu,$
 $field\text{-update} = upd \circ xfrom\text{-bytes},$
 $field\text{-sz} = size\text{-of}\ TYPE('b)\backslash)])\ n$
by (*simp add: ti-typ-combine-def Let-def*)

lemma *update-ti-t-pad-combine*:

assumes $std: size\text{-td}\ td' \bmod 2 \wedge (max\ algn\ (align\text{-td}\ (typ\text{-info}\text{-t}\ TYPE('a) ::$

```

c-type))) = 0
shows update-ti-t (ti-tyt-pad-combine TYPE('a :: c-type) lu upd algn fld td') bs
v =
  update-ti-t (ti-tyt-combine TYPE('a :: c-type) lu upd algn fld td') bs v
using std
by (simp add: ti-tyt-pad-combine-def size-td-simps Let-def max-2-exp)

```

15.3.8 The orphanage: miscellaneous lemmas pulled up to (roughly) where they belong.

```

lemma uinfo-array-tag-n-m-not-le-tyt-name:
  typ-name (typ-info-t TYPE('b)) @ 'array' @ nat-to-bin-string m
    ∉ td-names (typ-info-t TYPE('a))
    ⇒ ¬ uinfo-array-tag-n-m TYPE('b :: c-type) n m ≤ typ-uinfo-t TYPE('a ::
c-type)
apply (clarsimp simp: tyt-tag-le-def typ-uinfo-t-def)
apply (drule td-set-td-names)
apply (clarsimp simp: uinfo-array-tag-n-m-def typ-uinfo-t-def)
apply (drule arg-cong[where f=λxs. set "r" ⊆ set xs], simp)
apply (simp add: uinfo-array-tag-n-m-def typ-uinfo-t-def)
done

```

```

lemma tag-not-le-via-td-name:
  typ-name (typ-info-t TYPE('a)) ∉ td-names (typ-info-t TYPE('b))
    ⇒ typ-name (typ-info-t TYPE('a)) ≠ pad-tyt-name
    ⇒ ¬ typ-uinfo-t TYPE('a :: c-type) ≤ typ-uinfo-t TYPE ('b :: c-type)
apply (clarsimp simp: tyt-tag-le-def typ-uinfo-t-def)
apply (drule td-set-td-names, simp+)
done

```

```

lemmas typ-heap-simps =
  — c-guard
  c-guard-field
  c-guard-h-t-valid
  — h-t-valid
  h-t-valid-field
  h-t-valid-clift
  — h-val
  h-val-field-clift'
  h-val-clift'
  — clift
  clift-field
  clift-field-update
  heap-update-field-hrs
  heap-update-field'

```

```

    clift-heap-update
    clift-heap-update-same-td-name — Try this last (is expensive)

end

theory LemmaBucket-C
imports
  More-Lib
  WordSetup
  TypHeapLib
  ArrayAssertion
begin

declare word-neq-0-conv [simp del]

lemma Ptr-not-null-pointer-not-zero:  $(Ptr\ p \neq\ NULL) = (p \neq 0)$ 
by simp

lemma hrs-mem-f:  $f\ (hrs\text{-}mem\ s) = hrs\text{-}mem\ (hrs\text{-}mem\text{-}update\ f\ s)$ 
apply (cases s)
apply (clarsimp simp: hrs-mem-def hrs-mem-update-def)
done

lemma hrs-mem-heap-update:
   $heap\text{-}update\ p\ v\ (hrs\text{-}mem\ s) = hrs\text{-}mem\ (hrs\text{-}mem\text{-}update\ (heap\text{-}update\ p\ v)\ s)$ 
apply (rule hrs-mem-f)
done

lemma surj-Ptr [simp]:
  surj Ptr
by (rule surjI [where f = ptr-val], simp)

lemma inj-Ptr [simp]:
  inj Ptr
apply (rule injI)
apply simp
done

lemma bij-Ptr :
  bij Ptr
by (simp add: bijI)

lemma exec-Guard:
   $(G \vdash \langle Guard\ Err\ S\ c,\ Normal\ s \rangle \Rightarrow s')$ 
   $= (if\ s \in S\ then\ G \vdash \langle c,\ Normal\ s \rangle \Rightarrow s'$ 
      $else\ s' = Fault\ Err)$ 
by (auto split: if-split elim!: exec-elim-cases intro: exec.intros)

```

```

lemma byte-ptr-guarded:ptr-val ( $x::8$  word ptr)  $\neq 0 \implies$  c-guard x
  unfolding c-guard-def c-null-guard-def ptr-aligned-def
  by (clarsimp simp: intvl-Suc)

lemma intvl-aligned-bottom-eq:
  fixes  $p :: 'a::len$  word
  assumes al1: is-aligned x n
  and al2: is-aligned p bits
  and nb:  $\neg n < bits$ 
  and off:  $off \leq 2^{\wedge} bits$   $off \neq 0$ 
  shows ( $x \in \{p ..+ off\}$ ) = ( $x = p$ )
proof (rule iffI)
  assume  $x = p$ 
  thus  $x \in \{p ..+ off\}$  using off
    by (simp add: intvl-self)
next
  assume x-in-intvl:  $x \in \{p ..+ off\}$ 

  show  $x = p$ 
proof cases
  assume wb: bits < len-of TYPE('a)

  from x-in-intvl obtain kp where  $xp: x = p + of\text{-}nat\ kp$  and kp: kp < off
    by (clarsimp dest!: intvlD)

  hence is-aligned ( $p + of\text{-}nat\ kp$ ) n using al1 by simp
  hence  $2^{\wedge} n \text{ dvd } unat\ (p + of\text{-}nat\ kp)$  unfolding is-aligned-def .
  hence  $2^{\wedge} n \text{ dvd } unat\ p + kp$  using kp off wb
  apply -
  apply (subst (asm) iffD1 [OF unat-plus-simple])
  apply (rule is-aligned-no-wrap' [OF al2])
  apply (rule of-nat-power)
  apply simp-all[2]
  apply (subst (asm) unat-of-nat)
  apply (subst (asm) mod-less)
  apply (erule order-less-le-trans)
  apply (erule order-trans)
  apply simp
  apply simp
  done

  moreover from al2 obtain q2 where  $pbits: p = 2^{\wedge} bits * of\text{-}nat\ q2$ 
    and  $q2: q2 < 2^{\wedge} (len\text{-of}\ TYPE('a) - bits)$ 
    by (rule is-alignedE)

  moreover from nb obtain kn where  $nbits: n = bits + kn$ 
    by (clarsimp simp: linorder-not-less le-iff-add)

```

```

ultimately have  $2^{\text{bits}} \text{ dvd } 2^{\text{bits}} * q2 + kp$ 
  apply (simp add: power-add)
  apply (simp add: unat-mult-power-lem [OF q2])
  apply (erule dvd-mult-left)
done

hence  $2^{\text{bits}} \text{ dvd } kp$  by (simp add: dvd-reduce-multiple)
with  $kp$  have  $kp = 0$ 
  apply -
  apply (erule contrapos-pp)
  apply (simp add: linorder-not-less)
  apply (drule (1) dvd-imp-le)
  apply (erule order-trans [OF off(1)])
done

thus ?thesis using  $xp$  by simp
next
  assume  $wb: \neg \text{bits} < \text{len-of TYPE}(a)$ 
  with  $assms$ 
  show ?thesis by (simp add: is-aligned-mask mask-def power-overflow)
qed
qed

lemma intvl-mem-weaken:  $x \in \{p..+a - n\} \implies x \in \{p..+a\}$ 
  apply -
  apply (drule intvlD)
  apply clarsimp
  apply (rule intvlI)
  apply simp
done

lemma upto-intvl-eq:
  fixes  $x :: 'a::\text{len word}$ 
  assumes  $al: \text{is-aligned } x \ n$ 
  shows  $\{x..+2^n\} = \{x .. x + 2^n - 1\}$ 
proof cases
  assume  $n < \text{len-of TYPE}(a)$ 
  with  $assms$  show ?thesis
  unfolding intvl-def
  apply simp
  apply standard
  apply clarsimp
  apply (subgoal-tac of-nat  $k < (2 :: 'a \text{ word})^n$ )
  apply (intro conjI)
  apply (erule (1) is-aligned-no-wrap')
  apply (subst p-assoc-help)
  apply (rule word-plus-mono-right)
  apply (simp add: word-less-sub-1)

```



```

    apply (simp add: field-simps is-aligned-no-overflow)
    apply (simp add: of-nat-power)
  apply clarsimp
  subgoal for xa
    apply (rule exI[where x = unat (xa - x)])
    apply clarsimp
    apply (rule unat-less-power, assumption)
    apply (subst word-less-sub-le [symmetric])
    apply assumption
    apply (rule word-diff-ls'(4))
    apply (simp add: field-simps)
    apply assumption
  done
done
next
assume  $\neg n < \text{len-of } \text{TYPE}('a)$ 
with assms show ?thesis
  apply (simp add: is-aligned-mask mask-def power-overflow intvl-def)
  apply (rule set-eqI)
  apply clarsimp
  by (metis (no-types, opaque-lifting) le-less-trans nat-power-less-imp-less not-le
power-eq-0-iff
power-zero-numeral unat-lt2p word-unat.Rep-inverse)
qed

```

```

lemma upto-intvl-eq':
  fixes x :: 'a :: len word
  shows  $\llbracket x \leq x + (\text{of-nat } b - 1); b \neq 0; b \leq 2 \wedge \text{len-of } \text{TYPE}('a) \rrbracket \implies \{x..+b\}$ 
=  $\{x .. x + \text{of-nat } b - 1\}$ 
  unfolding intvl-def
  supply unsigned-of-nat
  apply standard
  apply clarsimp
  apply (subgoal-tac of-nat k  $\leq$  (of-nat (b - 1) :: 'a word))
  apply (intro conjI)
  apply (erule word-random)
  apply simp
  apply (subst field-simps [symmetric], rule word-plus-mono-right)
  apply simp
  apply assumption
  apply (subst More-Word.of-nat-mono-maybe-le [symmetric])
  apply simp
  apply simp
  apply simp
  apply clarsimp
  subgoal for xa
    apply (rule exI[where x = unat (xa - x)])
    apply simp

```

```

apply (simp add: unat-sub)
apply (rule nat-diff-less)
apply (subst (asm) word-le-nat-alt, erule order-le-less-trans)
apply (subst add-diff-eq[symmetric], subst unat-plus-if')
apply (simp add: no-olen-add-nat)
apply (simp add: le-eq-less-or-eq)
apply (erule disjE)
  apply (subst unat-minus-one)
    apply (erule (1) of-nat-neq-0)
      apply (simp add: unat-of-nat)
        apply (erule ssubst, rule unat-lt2p)
          apply (simp add: word-le-nat-alt)
            done
done

lemma intvl-aligned-top:
  fixes x :: 'a::len word
  assumes al1: is-aligned x n
  and al2: is-aligned p bits
  and nb: n ≤ bits
  and offn: off < 2 ^ n
  and wb: bits < len-of TYPE('a)
  shows (x ∈ {p ..+ 2 ^ bits - off}) = (x ∈ {p ..+ 2 ^ bits})
proof (rule iffI)
  assume x ∈ {p..+2 ^ bits - off}
  thus x ∈ {p..+2 ^ bits} by (rule intvl-mem-weaken)
next
  assume asm: x ∈ {p..+2 ^ bits}

  show x ∈ {p..+2 ^ bits - off}
  proof (cases n = 0)
    case True
      with offn asm show ?thesis by simp
    next
      case False

  from asm have x ∈ {p .. p + 2 ^ bits - 1}
    by (simp add: upto-intvl-eq [OF al2])
  then obtain q where xp: x = p + of-nat (q * 2 ^ n) and qb: q < 2 ^ (bits -
n) using False nb
    by (fastforce dest!: is-aligned-diff[OF al1 al2 wb,simplified field-simps])

  have q * 2 ^ n < 2 ^ bits - off
  proof -
    show ?thesis using offn qb nb
      apply (simp add: less-diff-conv)
      apply (erule (1) nat-add-offset-less)
      apply arith
      done

```

```

qed

with xp show ?thesis
  apply -
  apply (erule ssubst)
  apply (erule intvlI)
  done
qed
qed

lemma heap-update-list-update:
  fixes v :: word8
  shows  $x \neq y \implies \text{heap-update-list } s \text{ } xs \text{ } (\text{hp}(y := v)) \text{ } x = \text{heap-update-list } s \text{ } xs \text{ } \text{hp } x$ 
  apply (induct xs rule: rev-induct)
  apply simp
  apply (simp add: heap-update-list-append cong: if-cong)
  done

lemma heap-update-list-append2:
   $\text{length } xs + \text{length } ys < 2 \wedge \text{word-bits} \implies$ 
   $\text{heap-update-list } s \text{ } (xs @ ys) \text{ } \text{hp}$ 
   $= \text{heap-update-list } s \text{ } xs \text{ } (\text{heap-update-list } (s + \text{of-nat } (\text{length } xs)) \text{ } ys \text{ } \text{hp})$ 
proof (induct xs arbitrary: hp s)
  case Nil
  show ?case by simp
next
  case (Cons v' vs')

  have  $(1 :: \text{addr}) + \text{of-nat } (\text{length } vs') = \text{of-nat } (\text{length } (v' \# vs'))$ 
  by simp
  also have  $\dots \neq 0$  using Cons.prems
  apply -
  apply (rule of-nat-neq-0)
  apply simp
  apply (simp add: word-bits-conv)
  done
  finally have neq0:  $(1 :: \text{addr}) + \text{of-nat } (\text{length } vs') \neq 0$  .

  have  $(1 :: \text{addr}) + \text{of-nat } (\text{length } vs') = \text{of-nat } (\text{length } (v' \# vs'))$ 
  by simp
  also have  $\text{unat } \dots + \text{length } ys < 2 \wedge \text{word-bits}$  using Cons.prems
  apply (subst unat-of-nat)
  apply (simp add: word-bits-conv)
  done
  finally have lt:  $\text{unat } ((1 :: \text{addr}) + \text{of-nat } (\text{length } vs')) + \text{length } ys < 2 \wedge$ 

```

word-bits .

```
from Cons.prems have  $\text{length } vs' + \text{length } ys < 2 \wedge \text{word-bits}$  by simp
thus ?case
  apply simp
  apply (subst Cons.hyps, assumption)
  apply (rule arg-cong [where  $f = \text{heap-update-list } (s + 1) \text{ vs}$ ])
  apply (rule ext)
  subgoal for  $x$ 
    apply (cases  $x = s$ )
    apply simp
    apply (subst heap-update-nmem-same)
    apply (subst add.assoc)
    apply (rule intvl-nowrap[OF neq0 order-less-imp-le
      [OF lt[unfolded word-bits-def]]])
    apply simp
    apply (clarsimp simp: heap-update-list-update field-simps)
  done
done
qed
```

```
lemma heap-update-word8:
  heap-update  $p$  ( $v :: \text{word8}$ )  $hp = hp(\text{ptr-val } p := v)$ 
  unfolding heap-update-def by (simp add: to-bytes-word8)
```

```
lemma index-foldr-update2:
   $\llbracket n \leq i; i < \text{CARD}('b::\text{finite}) \rrbracket \implies \text{index } (\text{foldr } (\lambda n \text{ arr. } \text{Arrays.update arr } n$ 
 $m) [0..<n] (x :: ('a,'b) \text{array})) i = \text{index } x i$ 
  apply (induct  $n$  arbitrary: x)
  apply simp
  apply simp
  done
```

```
lemma index-foldr-update:
   $\llbracket i < n; n \leq \text{CARD}('b::\text{finite}) \rrbracket \implies \text{index } (\text{foldr } (\lambda n \text{ arr. } \text{Arrays.update arr } n$ 
 $m) [0..<n] (x :: ('a,'b) \text{array})) i = m$ 
  apply (induct  $n$  arbitrary: x)
  apply simp
  apply simp
  apply (erule less-SucE)
  apply simp
  apply simp
  apply (subst index-foldr-update2)
  apply simp
  apply simp
  apply simp
  done
```

```
lemma intvl-disjoint1:
```

```

fixes  $a :: 'a :: \text{len word}$ 
assumes  $abc: a + \text{of-nat } b \leq c$ 
and  $alb: a \leq a + \text{of-nat } b$ 
and  $cld: c \leq c + \text{of-nat } d$ 
and  $blt: b < 2 \wedge \text{len-of TYPE}('a)$ 
and  $dlt: d < 2 \wedge \text{len-of TYPE}('a)$ 
shows  $\{a..+b\} \cap \{c..+d\} = \{\}$ 
proof (rule disjointI, rule notI)
  fix  $x y$ 
  assume  $x: x \in \{a..+b\}$  and  $y: y \in \{c..+d\}$  and  $xy: x = y$ 

  from  $x$  obtain  $kx$  where  $x = a + \text{of-nat } kx$  and  $kx: kx < b$ 
    by (clarsimp dest!: intvlD)

  moreover from  $y$  obtain  $ky$  where  $y = c + \text{of-nat } ky$  and  $ky: ky < d$ 
    by (clarsimp dest!: intvlD)

  ultimately have  $ac: a + \text{of-nat } kx = c + \text{of-nat } ky$  using  $xy$  by simp

  have  $\text{of-nat } kx < (\text{of-nat } b :: 'a \text{ word})$  using  $blt$   $kx$ 
    by (rule of-nat-mono-maybe)
  hence  $a + \text{of-nat } kx < a + \text{of-nat } b$  using  $alb$ 
    by (rule word-plus-strict-mono-right)

  also have  $\dots \leq c$  by (rule abc)
  also have  $\dots \leq c + \text{of-nat } ky$  using  $cld$   $dlt$   $ky$ 
    by - (rule word-random [OF - iffD1 [OF More-Word.of-nat-mono-maybe-le]],
simp+)
  finally show False using  $ac$  by simp
qed

```

```

lemma intvl-disjoint2:
  fixes  $a :: 'a :: \text{len word}$ 
  assumes  $abc: a + \text{of-nat } b \leq c$ 
  and  $alb: a \leq a + \text{of-nat } b$ 
  and  $cld: c \leq c + \text{of-nat } d$ 
  and  $blt: b < 2 \wedge \text{len-of TYPE}('a)$ 
  and  $dlt: d < 2 \wedge \text{len-of TYPE}('a)$ 
  shows  $\{c..+d\} \cap \{a..+b\} = \{\}$ 
  using  $abc$   $alb$   $cld$   $blt$   $dlt$ 
  by (subst Int-commute, rule intvl-disjoint1)

```

```

lemma typ-slice-t-self:
   $td \in \text{fst } \text{' set } (\text{typ-slice-t } td \ m)$ 
  apply (cases td)
  apply (simp split: if-split)
  done

```

```

lemma index-fold-update:

```

```

[[ distinct xs; set xs ⊆ {..< card (UNIV :: 'b set)}; n < card (UNIV :: 'b set) ]]
⇒
  index (foldr (λn (arr :: 'a['b :: finite]). Arrays.update arr n (f n (index arr n)))
xs v) n
  = (if n ∈ set xs then f n (index v n) else index v n)
apply (induct xs)
apply simp
subgoal for x xs by (cases x = n, auto)
done

```

lemma *hrs-mem-update-cong*:

```

[[ ∧x. f x = f' x ]] ⇒ hrs-mem-update f = hrs-mem-update f'
by (simp add: hrs-mem-update-def)

```

lemma *Guard-no-cong*:

```

[[ A=A'; c=c' ]] ⇒ Guard A P c = Guard A' P c'
by simp

```

lemma *coerce-heap-update-to-heap-updates*:

```

assumes n: n = chunk * m and len: length xs = n
shows heap-update-list x xs
  = (λs. foldl (λs n. heap-update-list (x + (of-nat n * of-nat chunk))
    (take chunk (drop (n * chunk) xs)) s)
    s [0 ..< m])
using len[simplified n]
apply (induct m arbitrary: x xs)
apply (rule ext, simp)
apply (rule ext)
subgoal for m x xs s
  apply (simp only: upt-conv-Cons map-Suc-upt[symmetric])
  apply (subgoal-tac ∃ ys zs. length ys = chunk ∧ xs = ys @ zs)
  apply (clarsimp simp: heap-update-list-concat-unfold foldl-map
    field-simps)
  apply (rule exI[where x=take chunk xs])
  apply (rule exI[where x=drop chunk xs])
  apply simp
done
done

```

lemma *update-ti-list-array'*:

```

[[ update-ti-list-t (map f [0 ..< n]) xs v = y;
  ∀ n. size-td-tuple (f n) = v∃; length xs = v∃ * n;
  ∀ m xs v'. length xs = v∃ ∧ m < n →
  update-ti-tuple-t (f m) xs v' = Arrays.update v' m (update-ti-t (g m) xs (index
v' m)) ]]
⇒ y = foldr (λn arr. Arrays.update arr n (update-ti-t (g n) (take v∃ (drop

```

```

(v3 * n) xs)) (index arr n))) [0 ..< n] v
  apply (subgoal-tac  $\forall$  ys. size-td-list (map f ys) = v3 * length ys)
  prefer 2
  apply (rule allI)
  subgoal for ys by (induct ys) auto
  apply (induct n arbitrary: xs y v)
  apply simp
  apply (simp add: access-ti-append)
  apply (elim meta-allE, drule(1) meta-mp)
  apply simp
  apply (rule foldr-cong, (rule refl)+)
  apply (simp add: take-drop)
  apply (subst min.absorb1)
  apply (fold mult-Suc-right, rule mult-le-mono2)
  apply simp
  apply simp
done

```

lemma *update-ti-list-array*:

```

[[ update-ti-list-t (map f [0 ..< n]) xs v = (y :: 'a['b :: finite]);
   $\forall$  n. size-td-tuple (f n) = v3; length xs = v3 * n;
   $\forall$  m xs v'. length xs = v3  $\wedge$  m < n  $\longrightarrow$ 
  update-ti-tuple-t (f m) xs v' = Arrays.update v' m (update-ti-t (g m) xs (index
v' m));
  n  $\leq$  card (UNIV :: 'b set) ]]
 $\implies \forall$  m < n. update-ti-t (g m) (take v3 (drop (v3 * m) xs)) (index v m) =
index y m
  apply (subst update-ti-list-array'[where y=y], assumption+)
  apply clarsimp
  apply (subst index-fold-update)
  apply clarsimp+
done

```

lemma *access-in-array*:

```

fixes y :: ('a :: c-type)['b :: finite]
assumes assms: h-val hp x = y
           n < card (UNIV :: 'b set)
  and subst:  $\forall$  xs v. length xs = size-of TYPE('a)
              $\longrightarrow$  update-ti-t (typ-info-t TYPE('a)) xs v = f xs
shows h-val hp
      (Ptr (ptr-val x + of-nat (n * size-of TYPE('a)))) = index y n
using assms
  apply (simp add: h-val-def drop-heap-list-le2 del: of-nat-mult)
  apply (subst take-heap-list-le[symmetric, where n=card (UNIV :: 'b set) * size-of
TYPE ('a)])
  apply (fold mult-Suc, rule mult-le-mono1)
  apply simp
  apply (simp add: from-bytes-def typ-info-array')
  apply (drule update-ti-list-array, simp+)

```

```

  apply (simp add: size-of-def)
  apply (clarsimp simp: update-ti-s-adjust-ti)
  apply (rule refl)
  apply simp
  apply (drule spec, drule(1) mp)
  apply (simp add: size-of-def ac-simps drop-take)
  apply (subgoal-tac length v = size-of TYPE('a) for v)
  apply (subst subst, assumption)
  apply (subst(asm) subst, assumption)
  apply simp
  apply (simp add: size-of-def)
  apply (subst le-diff-conv2)
  apply simp
  apply (fold mult-Suc, rule mult-le-mono1)
  apply simp
done

```

lemma *access-ti-list-array*:

```

[[  $\forall n. \text{size-td-tuple } (f\ n) = v3; \text{length } xs = v3 * n;$ 
   $\forall m. m < n \wedge v3 \leq \text{length } (\text{drop } (v3 * m) xs)$ 
   $\longrightarrow \text{access-ti-tuple } (f\ m) (FCP\ g) (\text{take } v3 (\text{drop } (v3 * m) xs)) = (h\ m)$ 
  ]]  $\implies$ 
  access-ti-list (map f [0 ..< n]) (FCP g) xs
  = foldl (@) [] (map h [0 ..< n])
  apply (subgoal-tac  $\forall ys. \text{size-td-list } (\text{map } f\ ys) = v3 * \text{length } ys$ )
  prefer 2
  apply (rule allI)
  subgoal for ys by (induct ys, simp+)
  apply (induct n arbitrary: xs)
  apply simp
  apply (simp add: access-ti-append)
  apply (erule-tac x=take (v3 * n) xs in meta-allE)
  apply simp
  apply (frule spec, drule mp, rule conjI, rule lessI)
  apply simp
  apply simp
  apply (erule meta-mp)
  apply (auto simp add: drop-take)
done

```

lemma *take-drop-foldl-concat*:

```

[[  $\bigwedge y. y < m \implies \text{length } (f\ y) = n; x < m$  ]]
 $\implies \text{take } n (\text{drop } (x * n) (\text{foldl } (@) [] (\text{map } f [0 ..< m]))) = f\ x$ 
  apply (subst split-upt-on-n, assumption)
  apply (simp only: foldl-concat-concat map-append)
  apply (subst drop-append-miracle)
  apply (induct x, simp-all)[1]
  apply simp
done

```



```

lemma heap-update-Array:
  heap-update (p ::('a::packed-type['b::finite]) ptr) arr
    = (λs. foldl (λs n. heap-update (array-ptr-index p False n)
      (Arrays.index arr n) s) s [0 ..< card (UNIV :: 'b set)])
  apply (rule ext, simp add: heap-update-def)
  apply (subst coerce-heap-update-to-heap-updates
    [OF - refl, where chunk=size-of TYPE('a) and m=card (UNIV :: 'b set)])
  apply (simp)
  apply (rule foldl-cong[OF refl refl])
  apply (simp add: array-ptr-index-def CTypesDefs.ptr-add-def)
  subgoal for s a x
    apply (rule arg-cong[where f=λxs. heap-update-list p xs s for p s])
    apply (simp add: to-bytes-def size-of-def
      packed-type-access-ti)
    apply (simp add: typ-info-array')
    apply (subst fcp-eta[symmetric], subst access-ti-list-array)
    apply simp
    apply simp
    apply (simp add: packed-type-access-ti size-of-def)
    apply fastforce
    apply (rule take-drop-foldl-concat)
    apply (simp add: size-of-def)
    apply simp
  done

```

```

lemma from-bytes-Array-element:
  fixes p :: ('a::mem-type['b::finite]) ptr
  assumes less: of-nat n < card (UNIV :: 'b set)
  assumes len: length bs = size-of TYPE('a) * CARD('b)
  shows
    index (from-bytes bs :: 'a['b]) n
      = from-bytes (take (size-of TYPE('a)) (drop (n * size-of TYPE('a)) bs))
  using less
  apply (simp add: from-bytes-def size-of-def typ-info-array')
  apply (subst update-ti-list-array'[OF refl])
  apply simp
  apply (simp add: len size-of-def)
  apply (clarsimp simp: update-ti-s-adjust-ti)
  apply (rule refl)
  apply (simp add: split-upt-on-n[OF less])
  apply (rule trans, rule foldr-does-nothing-to-xf[where xf=λs. index s n])
  apply simp+
  apply (subst foldr-does-nothing-to-xf[where xf=λs. index s n])
  apply simp
  apply (simp add: mult commute)
  apply (frule Suc-leI)

```

```

apply (drule-tac k=size-of TYPE('a) in mult-le-mono2)
apply (rule upd-rf)
apply (simp add: size-of-def len mult commute)
done

```

```

lemma heap-access-Array-element':
fixes p :: ('a::mem-type['b::finite]) ptr
assumes less: of-nat n < card (UNIV :: 'b set)
shows
  index (h-val hp p) n
    = h-val hp (array-ptr-index p False n)
using less
apply (simp add: array-ptr-index-def CTypesDefs.ptr-add-def h-val-def)
apply (simp add: from-bytes-Array-element)
apply (simp add: drop-heap-list-le take-heap-list-le)
apply (subst take-heap-list-le)
apply (simp add: le-diff-conv2)
apply (drule Suc-leI)
apply (drule-tac k=size-of TYPE('a) in mult-le-mono2)
apply (simp add: mult commute)
apply simp
done

```

```

lemmas heap-access-Array-element
  = heap-access-Array-element'[simplified array-ptr-index-simps]

```

```

lemma heap-update-id:
  h-val hp ptr = (v :: 'a :: packed-type)
     $\implies$  heap-update ptr v hp = hp
apply (simp add: h-val-def heap-update-def)
apply (rule heap-update-list-id'[where n=size-of TYPE('a)])
apply clarsimp
apply (simp add: from-bytes-def to-bytes-def update-ti-t-def
  size-of-def field-access-update-same
  td-fafu-idem)
done

```

```

lemma heap-update-Array-update:
assumes n: n < CARD('b :: finite)
assumes size: CARD('b) * size-of TYPE('a :: packed-type) < 2 ^ addr-bitsize
shows heap-update p (Arrays.update (arr :: 'a['b]) n v) hp
  = heap-update (array-ptr-index p False n) v (heap-update p arr hp)
proof -

```

```

  have P:  $\bigwedge x k. \llbracket x < \text{CARD}('b); k < \text{size-of TYPE}('a) \rrbracket$ 
     $\implies$  unat (of-nat x * of-nat (size-of TYPE('a)) + (of-nat k :: addr))
      = x * size-of TYPE('a) + k

```

```

using size
supply unsigned-of-nat
apply (cases size-of TYPE('a), simp-all)
apply (cases CARD('b), simp-all)
apply (subst unat-add-lem[THEN iffD1])
apply (simp add: unat-word-ariths unat-of-nat less-Suc-eq-le)
apply (subgoal-tac Suc x * size-of TYPE('a) < 2 ^ addr-bitsize, simp-all)
apply (erule order-le-less-trans[rotated], simp add: add-mono)
apply (subst unat-mult-lem[THEN iffD1])
apply (simp add: unat-of-nat unat-add-lem[THEN iffD1])
apply (rule order-less-le-trans, erule order-le-less-trans[rotated],
        rule add-mono, simp+)
apply (simp add: less-Suc-eq-le trans-le-add2)
apply simp
apply (simp add: unat-of-nat unat-add-lem[THEN iffD1])
done

```

```

let ?key-upd = heap-update (array-ptr-index p False n) v
note commute = fold-commute-apply[where h=?key-upd
    and xs=[Suc n ..< CARD('b)], where g=f' and f=f' for f']

```

```

show ?thesis using n
apply (simp add: heap-update-Array split-upt-on-n[OF n]
        foldl-conv-fold)
apply (subst commute)
apply (simp-all add: packed-heap-update-collapse
        cong: fold-cong')
apply (rule ext, simp)
subgoal for x
apply (rule heap-update-commute, simp-all add: ptr-add-def)
apply (simp add: array-ptr-index-def CTypesDefs.ptr-add-def intvl-def Suc-le-eq)
apply (rule set-eqI, clarsimp)
apply (drule word-unat.Rep-inject[THEN iffD2])
apply (clarsimp simp: P nat-eq-add-iff1)
apply (cases x, simp-all add: less-Suc-eq-le Suc-diff-le)
done
done

```

qed

```

lemma heap-update-id-Array:
fixes arr :: ('a :: packed-type)['b :: finite]
shows arr = h-val hp p
     $\implies$  heap-update p arr hp = hp
apply (simp add: heap-update-Array)
apply (rule foldl-does-nothing[where s=hp])
apply (simp add: heap-access-Array-element' heap-update-id)
done

```

```

lemma heap-update-Array-element'':

```

```

fixes p' :: (('a :: packed-type)['b::finite]) ptr
fixes p :: ('a :: packed-type) ptr
fixes hp w
assumes p: p = array-ptr-index p' False n
assumes n: n < CARD('b)
assumes size: CARD('b) * size-of TYPE('a) < 2 ^ addr-bitsize
shows heap-update p' (Arrays.update (h-val hp p') n w) hp
  = heap-update p w hp
apply (subst heap-update-Array-update[OF n size])
apply (simp add: heap-update-id-Array p)
done

lemmas heap-update-Array-element'
  = heap-update-Array-element'[simplified array-ptr-index-simps]

lemma array-count-size:
  CARD('b :: array-max-count) * size-of TYPE('a :: array-outer-max-size) < 2 ^
  addr-bitsize
  using array-outer-max-size-ax[where 'a='a] array-max-count-ax[where 'a='b]
  apply (clarsimp dest!: nat-le-Suc-less-imp)
  apply (drule(1) mult-mono, simp+)
  done

lemmas heap-update-Array-element
  = heap-update-Array-element'[OF refl - array-count-size]

lemma typ-slice-list-cut:
  [ (∀ x ∈ set xs. size-td (dt-fst x) = m); m ≠ 0; n < (length xs * m) ]
  ⇒ typ-slice-list xs n =
    typ-slice-tuple (xs ! (n div m)) (n mod m)
  apply (induct xs arbitrary: n, simp-all)
  subgoal for x1 xs n
    apply (intro conjI impI)
    apply simp
    apply (subgoal-tac ∃ n'. n = n' + m)
    apply clarsimp
    apply (metis (no-types, opaque-lifting) add commute add.left-neutral cancel-comm-monoid-add-class.diff-cancel
      div-add1-eq div-mult-self-is-m mod-div-trivial modulo-nat-def mult.left-neutral
      nth-Cons-Suc plus-1-eq-Suc)
    apply (rule exI[where x=n - m])
    apply simp
  done
done

lemma typ-slice-t-array:
  [ n < CARD('b); y < size-of TYPE('a) ]
  ⇒ typ-slice-t (export-uinfo (typ-info-t TYPE('a))) y ≤
  typ-slice-t (export-uinfo (array-tag TYPE('a['b :: finite])))

```

```

      (y + size-of TYPE('a :: mem-type) * n)
apply (simp add: array-tag-def array-tag-n-eq
        split del: if-split)
apply (rule disjI2)
apply (subgoal-tac y + (size-of TYPE('a) * n) < CARD('b) * size-of TYPE('a))
  apply (simp add: typ-slice-list-cut[where m=size-of TYPE('a)]
        map-td-list-map o-def size-of-def
        sz-nzero[unfolded size-of-def])
  apply (simp flip: export-uinfo-def)
apply (rule order-less-le-trans[where y=Suc n * size-of TYPE('a)])
  apply (simp add: size-of-def)
apply (simp only: size-of-def mult-le-mono1)
done

```

lemma *h-t-valid-Array-element'*:

```

[[ htd ⊨t (p :: (('a :: mem-type)['b :: finite]) ptr); coerce ∨ n < CARD('b) ]
  ⇒ htd ⊨t array-ptr-index p coerce n for coerce
apply (clarsimp simp only: h-t-valid-def valid-footprint-def Let-def
  c-guard-def c-null-guard-def)
apply (subgoal-tac ∃ offs. array-ptr-index p coerce n = CTypesDefs.ptr-add (ptr-coerce
p) (of-nat offs)
  ∧ offs < CARD ('b))
apply (clarsimp simp: size-td-array size-of-def typ-uinfo-t-def
  typ-info-array array-tag-def)
subgoal for offs
apply (intro conjI)
  apply (clarsimp simp: CTypesDefs.ptr-add-def
    field-simps)
  apply (rename-tac y)
  apply (drule-tac x=offs * size-of TYPE('a) + y in spec)
  apply (drule mp)
  apply (rule order-less-le-trans[where y=Suc offs * size-of TYPE('a)])
  apply (simp add: size-of-def)
  apply (simp only: size-of-def mult-le-mono1)
  apply (clarsimp simp: field-simps)
  apply (erule map-le-trans[rotated])
  apply (rule list-map-mono)
  apply (subst mult.commute, rule typ-slice-t-array[unfolded array-tag-def])
  apply assumption
  apply (simp add: size-of-def)
apply (simp add: ptr-aligned-def align-of-def align-td-array
  array-ptr-index-def
  CTypesDefs.ptr-add-def unat-word-ariths unat-of-nat)
using align-size-of[where 'a='a] align[where 'a='a]
  apply (simp add: align-of-def size-of-def addr-card-def card-word)
  apply (simp add: dvd-mod)
apply (thin-tac ∀ x. P x for P)
apply (clarsimp simp: intvl-def)
apply (rename-tac k)

```

```

apply (drule-tac x=offs * size-of TYPE('a) + k in spec)
apply (drule mp)
apply (simp add: array-ptr-index-def CTypesDefs.ptr-add-def field-simps)
apply (erule notE)
apply (rule order-less-le-trans[where y=Suc offs * size-of TYPE('a)])
apply (simp add: size-of-def)
apply (simp only: size-of-def mult-le-mono1)
done
subgoal by (auto simp add: array-ptr-index-def intro: exI[where x=0])
done

lemma h-t-valid-Array-element:
  [[ htd  $\models_t$  (p :: (('a :: mem-type)['b :: finite]) ptr); 0 ≤ n; n < int CARD('b) ]
    ⇒ htd  $\models_t$  ((ptr-coerce p :: 'a ptr) +p n)
apply (drule-tac n=nat n and coerce=False in h-t-valid-Array-element')
apply simp
apply (simp add: array-ptr-index-def)
done

lemma ptr-safe-Array-element:
  [[ ptr-safe (p :: (('a :: mem-type)['b :: finite]) ptr) htd; coerce ∨ n < CARD('b) ]
    ⇒ ptr-safe (array-ptr-index p coerce n) htd for coerce
apply (simp add: ptr-safe-def)
apply (erule order-trans[rotated])
apply (subgoal-tac ∃ offs. array-ptr-index p coerce n = CTypesDefs.ptr-add (ptr-coerce
p) (of-nat offs)
  ∧ offs < CARD ('b))
prefer 2
subgoal by (auto simp: array-ptr-index-def intro: exI[where x=0])[1]
apply (clarsimp simp: s-footprint-def s-footprint-untyped-def
  CTypesDefs.ptr-add-def
  size-td-array size-of-def)
subgoal for offs x k
apply (rule exI[where x=offs * size-of TYPE('a) + x])
apply (simp add: size-of-def)
apply (rule conjI)
apply (rule order-less-le-trans[where y=Suc offs * size-of TYPE('a)])
  apply (simp add: size-of-def)
apply (simp only: size-of-def)
apply (rule mult-le-mono1)
apply simp
apply (thin-tac coerce ∨ P for P)
apply (elim disjE exE conjE, simp-all add: typ-uinfo-t-def)
apply (erule order-less-le-trans)
apply (rule prefix-length-le)
apply (rule order-trans, erule typ-slice-t-array)
  apply (simp add: size-of-def)
apply (simp add: size-of-def field-simps typ-info-array)
done

```

done

lemma *from-bytes-eq*:

from-bytes [x] = x

apply (*clarsimp simp:from-bytes-def update-ti-t-def typ-info-word*)

apply (*simp add:word-rcat-def*)

done

lemma *bytes-disjoint*: $(x::('a::c-type) ptr) \neq y \implies \{ptr-val\ x + a ..+ 1\} \cap \{ptr-val\ y + a ..+ 1\} = \{\}$

by (*clarsimp simp:intvl-def*)

lemma *byte-ptrs-disjoint*: $(x::('a::c-type) ptr) \neq y \implies \forall i < of-nat\ (size-of\ TYPE('a)).\ ptr-val\ x + i \neq ptr-val\ y + i$

by *force*

lemma *le-step*: $\llbracket (x::('a::len) word) < y + 1; x \neq y \rrbracket \implies x < y$

by (*metis less-x-plus-1 max-word-max order-less-le*)

lemma *ptr-add-disjoint*:

$\llbracket ptr-val\ y \notin \{ptr-val\ x ..+ size-of\ TYPE('a)\};$

$ptr-val\ (x::('a::c-type) ptr) < ptr-val\ (y::('b::c-type) ptr);$

$a < of-nat\ (size-of\ TYPE('a)) \rrbracket \implies$

$ptr-val\ x + a < ptr-val\ y$

apply (*erule swap*)

apply (*rule intvl-inter-le [where k=0 and ka=unat (ptr-val y - ptr-val x)]*)

apply *clarsimp*

apply (*metis (opaque-lifting, mono-tags) add-diff-cancel2 add-diff-inverse diff-add-cancel*

trans-less-add1 unat-less-helper word-le-less-eq word-less-add-right

word-of-nat-less word-unat.Rep-inverse)

apply *simp*

done

lemma *ptr-add-disjoint2*:

$\llbracket ptr-val\ x \notin \{ptr-val\ y ..+ size-of\ TYPE('a)\};$

$ptr-val\ (y::('b::c-type) ptr) < ptr-val\ (x::('a::c-type) ptr);$

$a < of-nat\ (size-of\ TYPE('a)) \rrbracket \implies$

$ptr-val\ y + a < ptr-val\ x$

apply (*erule swap*)

apply (*rule intvl-inter-le [where k=0 and ka=unat (ptr-val x - ptr-val y)]*)

apply *clarsimp*

apply (*metis (no-types, opaque-lifting) add commute less-imp-le less-le-trans not-le unat-less-helper*

word-diff-ls'(4))

apply *simp*

done

lemma *ptr-aligned-is-aligned*: $\llbracket ptr-aligned\ (x::('a::c-type) ptr); align-of\ TYPE('a) = 2 \wedge n \rrbracket \implies is-aligned\ (ptr-val\ x)\ n$

by (clarsimp simp: ptr-aligned-def is-aligned-def)

lemma *intvl-no-overflow*:

assumes *no-overflow*: $\text{unat } a + b < 2 \wedge \text{len-of } \text{TYPE}('a::\text{len})$

shows $(x \in \{(a :: 'a \text{ word}) ..+ b\}) = (a \leq x \wedge x < (a + \text{of-nat } b))$

proof –

obtain $sk :: 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow \text{nat} \Rightarrow \text{nat}$

where $f1: \bigwedge x y z. x \notin \{y..+z\} \vee x = y + \text{of-nat } (sk \ x \ y \ z) \wedge sk \ x \ y \ z < z$

using $[[\text{metis-new-skolem}]]$ **by** (metis *intvlD*)

have $f2: \bigwedge x. a + x < a + \text{of-nat } b \vee \neg x < \text{of-nat } b$

using *no-overflow*

by (metis *PackedTypes.of-nat-mono-maybe-le add-lessD1 le-add1 add commute olen-add-eqv unat-of-nat-eq word-arith-nat-add word-plus-strict-mono-right*)

have $f3: \forall x y. y \notin \{x..+b\} \vee \text{of-nat } (sk \ y \ x \ b) < (\text{of-nat } b :: 'a \text{ word})$

using *no-overflow f1*

by (metis *add-lessD1 add commute of-nat-mono-maybe*)

have $x < a + \text{of-nat } b \vee \neg \text{of-nat } (sk \ x \ a \ b) < (\text{of-nat } b :: 'a \text{ word}) \vee ?thesis$

using *f1 f2* **by** metis

hence $x < a + \text{of-nat } b \vee ?thesis$

using *f3* **by** metis

thus *?thesis*

apply (rule *disjE*)

apply (rule *iffI*)

apply (clarsimp simp: *intvl-def*)

apply (rename-tac *k*)

apply (clarsimp simp: *unat-sub-if-size word-le-nat-alt word-less-nat-alt*)

apply (cut-tac *no-overflow*)

apply (subgoal-tac $k + (b + \text{unat } a) < 2 \wedge \text{len-of } (\text{TYPE}('a)) + b$)

apply (subgoal-tac $k + \text{unat } a < 2 \wedge \text{len-of } (\text{TYPE}('a))$)

apply (metis *add-lessD1 le-def less-not-refl2 add commute unat-eq-of-nat word-arith-nat-add*)

apply *clarsimp*

apply *clarsimp*

apply (clarsimp simp: *intvl-def*)

apply (rule *exI* [where $x = \text{unat } (x - a)$])

apply (clarsimp simp: *unat-sub-if-size word-le-nat-alt word-less-nat-alt*)

apply (cut-tac *no-overflow*)

apply (metis *diff-le-self le-add-diff-inverse le-diff-conv le-eq-less-or-eq le-unat-woi add commute nat-neq-iff unat-of-nat-eq word-arith-nat-add*)

apply *simp*

done

qed

lemma *FCP-arg-cong*: $f = g \implies \text{FCP } f = \text{FCP } g$
by *simp*

lemma *h-val-id*:
 $h\text{-val } (hrs\text{-mem } (hrs\text{-mem-update } (heap\text{-update } x \ y) \ s)) \ x = (y::'a::mem\text{-type})$
apply (*subst hrs-mem-update*)
apply (*rule h-val-heap-update*)
done

lemma *h-val-id-padding*:
 $length \ bs = size\text{-of } \text{TYPE}('a) \implies h\text{-val } (hrs\text{-mem } (hrs\text{-mem-update } (heap\text{-update-padding } x \ y \ bs) \ s)) \ x = (y::'a::mem\text{-type})$
apply (*subst hrs-mem-update*)
apply (*simp add: h-val-heap-update-padding*)
done

lemma *heap-update-id2*:
 $hrs\text{-mem-update } (heap\text{-update } p \ ((h\text{-val } (hrs\text{-mem } s) \ p)::'a::packed\text{-type})) \ s = s$
apply (*clarsimp simp:hrs-mem-update-def case-prod-beta*)
apply (*subst heap-update-id*)
apply (*simp add:hrs-mem-def*)
done

lemma *intvlI-unat*: $unat \ b < unat \ c \implies a + b \in \{a \ .. + unat \ c\}$
by (*metis intvlI word-unat.Rep-inverse*)

lemma *neg-imp-bytes-disjoint*:
 $\llbracket c\text{-guard } (x::'a::c\text{-type } ptr); c\text{-guard } y; unat \ j < align\text{-of } \text{TYPE}('a); unat \ i < align\text{-of } \text{TYPE}('a); x \neq y; 2 \wedge n = align\text{-of } \text{TYPE}('a); n < 32 \rrbracket$
 \implies
 $ptr\text{-val } x + j \neq ptr\text{-val } y + i$
apply (*rule ccontr*)
apply (*subgoal-tac is-aligned (ptr-val x) n*)
apply (*subgoal-tac is-aligned (ptr-val y) n*)
apply (*subgoal-tac (ptr-val x + j && ~ mask n) = (ptr-val y + i && ~ mask n)*)
apply (*subst (asm) neg-mask-add-aligned, simp, simp add: word-less-nat-alt*)
apply (*subst (asm) neg-mask-add-aligned, simp, simp add: word-less-nat-alt*)
apply *clarsimp*
apply *simp*
apply (*clarsimp simp: c-guard-def ptr-aligned-def is-aligned-def*)
apply (*clarsimp simp: c-guard-def ptr-aligned-def is-aligned-def*)
done

lemma *heap-update-list-base'*: $heap\text{-update-list } p \ [] = id$

by (rule ext, simp)

lemma hrs-mem-update-id3: hrs-mem-update id = id
unfolding hrs-mem-update-def by simp

lemma h-t-valid-ptr-retyp-inside-eq:
fixes $p :: 'a :: \text{mem-type ptr}$ and $p' :: 'a :: \text{mem-type ptr}$
assumes inside: $\text{ptr-val } p' \in \{\text{ptr-val } p \text{ ..+ size-of TYPE('a)}\}$
and ht: $\text{ptr-retyp } p \text{ td, } g \models_t p'$
shows $p = p'$
using ptr-retyp-same-cleared-region[OF ht] inside mem-type-self[where $p=p'$]
by blast

lemma typ-slice-t-self-nth:
 $\exists n < \text{length } (\text{typ-slice-t } \text{td } m). \exists b. \text{typ-slice-t } \text{td } m ! n = (\text{td}, b)$
using typ-slice-t-self [where $\text{td} = \text{td}$ and $m = m$]
by (fastforce simp add: in-set-conv-nth)

lemma ptr-retyp-other-cleared-region:
fixes $p :: 'a :: \text{mem-type ptr}$ and $p' :: 'b :: \text{mem-type ptr}$
assumes ht: $\text{ptr-retyp } p \text{ td, } g \models_t p'$
and tdisj: $\text{typ-uinfo-t TYPE('a)} \perp_t \text{typ-uinfo-t TYPE('b)} :: \text{mem-type}$
and clear: $\forall x \in \{\text{ptr-val } p \text{ ..+ size-of TYPE('a)}\}. \forall n b. \text{snd } (\text{td } x) n \neq \text{Some } (\text{typ-uinfo-t TYPE('b)}, b)$
shows $\{\text{ptr-val } p' \text{ ..+ size-of TYPE('b)}\} \cap \{\text{ptr-val } p \text{ ..+ size-of TYPE('a)}\} = \{\}$
proof (rule classical)
assume asm: $\{\text{ptr-val } p' \text{ ..+ size-of TYPE('b)}\} \cap \{\text{ptr-val } p \text{ ..+ size-of TYPE('a)}\} \neq \{\}$
then obtain mv where $mvp: mv \in \{\text{ptr-val } p \text{ ..+ size-of TYPE('a)}\}$
and $mvp': mv \in \{\text{ptr-val } p' \text{ ..+ size-of TYPE('b)}\}$
by blast

then obtain k' where $mv: mv = \text{ptr-val } p' + \text{of-nat } k'$ and $klt: k' < \text{size-td } (\text{typ-uinfo-t TYPE('b)})$
by (clarsimp dest!: intvlD simp: size-of-def)

let $?mv = \text{ptr-val } p' + \text{of-nat } k'$

obtain $n b$ where $nl: n < \text{length } (\text{typ-slice-t } (\text{typ-uinfo-t TYPE('b)}) k')$
and $tseq: \text{typ-slice-t } (\text{typ-uinfo-t TYPE('b)}) k' ! n = (\text{typ-uinfo-t TYPE('b)}, b)$
using typ-slice-t-self-nth [where $\text{td} = \text{typ-uinfo-t TYPE('b)}$ and $m = k'$]
byclarsimp

with ht have $\text{snd } (\text{ptr-retyp } p \text{ td } ?mv) n = \text{Some } (\text{typ-uinfo-t TYPE('b)}, b)$
unfolding h-t-valid-def
apply –
apply (clarsimp simp: valid-footprint-def Let-def)
apply (drule spec, drule mp [OF - klt])
apply (clarsimp simp: map-le-def)

```

apply (drule bspec)
apply simp
apply simp
done

moreover {
  assume snd (ptr-retyp p empty-htd ?mv) n = Some (typ-uinfo-t TYPE('b), b)
  hence (typ-uinfo-t TYPE('b)) ∈ fst ‘ set (typ-slice-t (typ-uinfo-t TYPE('a))
    (unat (ptr-val p' + of-nat k' - ptr-val p)))

    using asm mv mvp
    apply –
    apply (rule image-eqI[where x = (typ-uinfo-t TYPE('b), b)])
    apply simp
    apply (fastforce simp add: ptr-retyp-footprint list-map-eq in-set-conv-nth split:
if-split-asm)
    done

    with typ-slice-set have (typ-uinfo-t TYPE('b)) ∈ fst ‘ td-set (typ-uinfo-t
TYPE('a)) 0
    by (rule subsetD)

    hence False using tdisj by (clarsimp simp: tag-disj-def typ-tag-le-def)
} ultimately show ?thesis using mvp mvp' mv unfolding h-t-valid-def valid-footprint-def
apply –
apply (subst (asm) ptr-retyp-d-eq-snd)
apply (auto simp add: map-add-Some-iff clear)
done
qed

end

```

theory *Cong-Tactic*

imports

Main

HOL-Eisbach.Eisbach-Tools

begin

Simple congruence prover

Replaces a goal of the shape:

$$f\ x\ (g\ y)\ (\lambda x. x + 1) = f\ x'\ (i\ y')\ (\lambda z. z + Suc\ 0)$$

by

$$x = x'\ g\ y = i\ y'\ 1 = Suc\ 0$$

The tactic essentially applies $\llbracket ?f = ?g; ?x = ?y \rrbracket \implies ?f\ ?x = ?g\ ?y$, but using first-order matching.

ML ‹

fun fo-cong-tac ctxt = CSUBGOAL (fn (cgoal, i) =>

```

let
  val goal = Thm.term-of cgoal;
in
  (case Logic.strip-assums-concl goal of
   - $ (- $ t $ s) =>
     let
       val (f, xs) = strip-comb t
       val (g, ys) = strip-comb s
     in
       if f aconv g andalso length xs = length ys
       then REPEAT-DETERM-N (length xs) (cong-tac ctxt i)
       THEN resolve-tac ctxt [@{thm refl}] i
       else no-tac
     end
   | - => no-tac)
end);

```

```

fun abs-cong-tac ctxt = CSUBGOAL (fn (cgoal, i) =>
  let
    val goal = Thm.term-of cgoal;
  in
    (case Logic.strip-assums-concl goal of
     - $ (- $ Abs - $ Abs -) => resolve-tac ctxt [@{thm ext}] i
     | - => no-tac)
  end);

```

>

```

method-setup app-cong =
  ‹Scan.succeed (fn ctxt => Method.SIMPLE-METHOD' (fo-cong-tac ctxt))›
  congruence method

```

```

method-setup abs-cong =
  ‹Scan.succeed (fn ctxt => Method.SIMPLE-METHOD' (abs-cong-tac ctxt))›
  congruence method

```

```

method cong-step = (app-cong | abs-cong | rule refl | rule eq-reflection)
method cong = (cong-step ; cong?)

```

Preserve context information from [cong] rules

```

ML ‹
fun const-cong ctxt = CSUBGOAL (fn (cgoal, i) =>
  let
    val goal = Thm.term-of cgoal;
  in
    (case Logic.strip-assums-concl goal of
     - $ (- $ t $ s) => (case (head-of t, head-of s) of
      (Const (c, -), Const (c', -)) =>

```

```

    if c = c'
    then
      case AList.lookup (op =) (#congs (dest-ss (simpset-of ctxt))) (true, c)
of
  SOME thm =>
    resolve-tac ctxt [Thm.transfer' ctxt thm RS meta-eq-to-obj-eq] i
  | NONE => no-tac
  else no-tac
  | - => no-tac
)
| - => no-tac)
end);
>

```

```

method-setup const-cong =
  ⟨Scan.succeed (fn ctxt => Method.SIMPLE-METHOD' (const-cong ctxt))⟩
  congruence method (constant with [cong])

```

```

method cong-context-step = (const-cong | cong-step | rule simp-impliesI)
method cong-context = (cong-context-step ; cong-context?)

```

```

end

```

Part III
AutoCorres

Chapter 16

Spec-Monad

```
theory Spec-Monad
imports
  Basic-Runs-To-VCG
  HOL-Library.Complete-Partial-Order2
  HOL-Library.Monad-Syntax
  AutoCorres-Utills
begin
```

16.1 *rel-map* and *rel-project*

```
definition rel-map :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  bool where
  rel-map f r x = (x = f r)
```

```
lemma rel-map-direct[simp]: rel-map f a (f a)
by (simp add: rel-map-def)
```

```
abbreviation rel-project  $\equiv$  rel-map
```

```
lemmas rel-project-def = rel-map-def
```

```
lemma rel-project-id: rel-project id = (=)
  rel-project ( $\lambda v. v$ ) = (=)
by (auto simp add: rel-project-def fun-eq-iff)
```

```
lemma rel-project-unit: rel-project ( $\lambda-. ()$ ) x y = True
by (simp add: rel-project-def)
```

```
lemma rel-projectI: y = prj x  $\implies$  rel-project prj x y
by (simp add: rel-project-def)
```

```
lemma rel-project-conv: rel-project prj x y = (y = prj x)
by (simp add: rel-project-def)
```

16.2 Misc Theorems

declare *case-unit-Unity* [*simp*] — without this rule simplifier seems loops in unexpected ways

lemma *abs-const-unit*: $(\lambda(v::unit). f) = (\lambda(). f)$
by *auto*

lemma *SUP-mono''*: $(\bigwedge x. x \in A \implies f x \leq g x) \implies (\bigsqcup x \in A. f x) \leq (\bigsqcup x \in A. g x ::$
 $-\text{::complete-lattice})$
by (*rule SUP-mono*) *auto*

lemma *wf-nat-bound*: *wf* $\{(a, b). b < a \wedge b \leq (n::nat)\}$
apply (*rule wf-subset*)
apply (*rule wf-measure*[**where** $f = \lambda a. \text{Suc } n - a$])
apply *auto*
done

lemma (**in** *complete-lattice*) *admissible-le*:
ccpo.admissible Inf $(\leq) (\lambda x. (y \leq x))$
by (*simp add: ccpo.admissibleI local.Inf-greatest*)

lemma *mono-const*: *mono* $(\lambda x. c)$
by (*simp add: mono-def*)

lemma *mono-lam*: $(\bigwedge a. \text{mono } (\lambda x. F x a)) \implies \text{mono } F$
by (*simp add: mono-def le-fun-def*)

lemma *mono-app*: *mono* $(\lambda x. x a)$
by (*simp add: mono-def le-fun-def*)

lemma *all-cong-map*:
assumes $f \cdot f': \bigwedge y. f (f' y) = y$ **and** $P \cdot Q: \bigwedge x. P x \longleftrightarrow Q (f x)$
shows $(\forall x. P x) \longleftrightarrow (\forall y. Q y)$
using *assms* **by** *metis*

lemma *rel-set-refl*: $(\bigwedge x. x \in A \implies R x x) \implies \text{rel-set } R A A$
by (*auto simp: rel-set-def*)

lemma *rel-set-converse-iff*: $\text{rel-set } R X Y \longleftrightarrow \text{rel-set } R^{-1-1} Y X$
by (*auto simp add: rel-set-def*)

lemma *rel-set-weaken*:
 $(\bigwedge x y. x \in A \implies y \in B \implies P x y \implies Q x y) \implies \text{rel-set } P A B \implies \text{rel-set } Q A B$
by (*force simp: rel-set-def*)

lemma *sim-set-refl*: $(\bigwedge x. x \in X \implies R x x) \implies \text{sim-set } R X X$
by (*auto simp: sim-set-def*)

16.3 Galois Connections

lemma *mono-of-Sup-cont*:

fixes $f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice$
assumes $cont: \bigwedge X. f (Sup X) = (SUP x \in X. f x)$
assumes $xy: x \leq y$
shows $f x \leq f y$

proof –

have $f x \leq sup (f x) (f y)$ **by** (*rule sup-ge1*)
also have $\dots = (SUP x \in \{x, y\}. f x)$ **by** *simp*
also have $\dots = f (Sup \{x, y\})$ **by** (*rule cont[symmetric]*)
finally show *?thesis* **using** xy **by** (*simp add: sup-absorb2*)

qed

lemma *gc-of-Sup-cont*:

fixes $f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice$
assumes $cont: \bigwedge X. f (Sup X) = (SUP x \in X. f x)$
shows $f x \leq y \longleftrightarrow x \leq Sup \{x. f x \leq y\}$

proof *safe*

assume $f x \leq y$ **then show** $x \leq Sup \{x. f x \leq y\}$
by (*intro Sup-upper*) *simp*

next

assume $x \leq Sup \{x. f x \leq y\}$
then have $f x \leq f (Sup \{x. f x \leq y\})$
by (*rule mono-of-Sup-cont[OF cont]*)
also have $\dots = (SUP x \in \{x. f x \leq y\}. f x)$ **by** (*rule cont*)
also have $\dots \leq y$ **by** (*rule Sup-least*) *auto*
finally show $f x \leq y$.

qed

lemma *mono-of-Inf-cont*:

fixes $f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice$
assumes $cont: \bigwedge X. f (Inf X) = (INF x \in X. f x)$
assumes $xy: x \leq y$
shows $f x \leq f y$

proof –

have $f x = f (Inf \{x, y\})$ **using** xy **by** (*simp add: inf-absorb1*)
also have $\dots = (INF x \in \{x, y\}. f x)$ **by** (*rule cont*)
also have $\dots = inf (f x) (f y)$ **by** *simp*
also have $\dots \leq f y$ **by** (*rule inf-le2*)
finally show *?thesis* .

qed

lemma *gc-of-Inf-cont*:

fixes $f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice$
assumes $cont: \bigwedge X. f (Inf X) = (INF x \in X. f x)$
shows $Inf \{y. x \leq f y\} \leq y \longleftrightarrow x \leq f y$

proof *safe*

assume $x \leq f y$ **then show** $Inf \{y. x \leq f y\} \leq y$

```

    by (intro Inf-lower) simp
next
  assume *: Inf {y. x ≤ f y} ≤ y
  have x ≤ (INF y∈{y. x ≤ f y}. f y) by (rule Inf-greatest) auto
  also have ... = f (Inf {y. x ≤ f y}) by (rule cont[symmetric])
  also have ... ≤ f y
    using * by (rule mono-of-Inf-cont[OF cont])
  finally show x ≤ f y .
qed

```

```

lemma gfp-fusion:
  assumes f-g:  $\bigwedge x y. g x \leq y \longleftrightarrow x \leq f y$ 
  assumes a: mono a
  assumes b: mono b
  assumes *:  $\bigwedge x. f (a x) = b (f x)$ 
  shows f (gfp a) = gfp b
  apply (intro antisym gfp-upperbound)
  subgoal
    apply (subst *[symmetric])
    apply (subst gfp-fixpoint[OF a])
    ..
  apply (rule f-g[THEN iffD1])
  apply (rule gfp-upperbound)
  apply (rule f-g[THEN iffD2])
  apply (subst *)
  apply (subst gfp-fixpoint[OF b, symmetric])
  apply (rule monoD[OF b])
  apply (rule f-g[THEN iffD1])
  apply (rule order-refl)
  done

```

```

lemma lfp-fusion:
  assumes f-g:  $\bigwedge x y. g x \leq y \longleftrightarrow x \leq f y$ 
  assumes a: mono a
  assumes b: mono b
  assumes *:  $\bigwedge x. g (a x) = b (g x)$ 
  shows g (lfp a) = lfp b
  apply (intro antisym lfp-lowerbound)
  subgoal
    apply (rule f-g[THEN iffD2])
    apply (rule lfp-lowerbound)
    apply (rule f-g[THEN iffD1])
    apply (subst *)
    apply (subst (2) lfp-fixpoint[OF b, symmetric])
    apply (rule monoD[OF b])
    apply (rule f-g[THEN iffD2])
    apply (rule order-refl)
  done
subgoal

```

```

apply (subst *[symmetric])
apply (subst lfp-fixpoint[OF a])
..
done

```

16.4 *post-state* type

datatype *'r post-state* = *Failure* | *Success 'r set*

Failure is supposed to model things like undefined behaviour in C. We usually have to show the absence of *Failure* for all possible executions of the program. Moreover, it is used to model the 'result' of a non terminating computation.

lemma *split-post-state*:

```

x = (case x of Success X  $\Rightarrow$  Success X | Failure  $\Rightarrow$  Failure)
for x::'a post-state
by (cases x) auto

```

instantiation *post-state* :: (type) order
begin

inductive *less-eq-post-state* :: '*a post-state* \Rightarrow '*a post-state* \Rightarrow bool **where**
Failure-le[simp, intro]: less-eq-post-state p Failure
| *Success-le-Success[intro]: r \subseteq q \Longrightarrow less-eq-post-state (Success r) (Success q)*

definition *less-post-state* :: '*a post-state* \Rightarrow '*a post-state* \Rightarrow bool **where**
less-post-state p q \longleftrightarrow p \leq q \wedge \neg q \leq p

instance

proof

```

fix p q r :: 'a post-state
show p  $\leq$  p by (cases p) auto
show p  $\leq$  q  $\Longrightarrow$  q  $\leq$  r  $\Longrightarrow$  p  $\leq$  r by (blast elim: less-eq-post-state.cases)
show p  $\leq$  q  $\Longrightarrow$  q  $\leq$  p  $\Longrightarrow$  p = q by (blast elim: less-eq-post-state.cases)
qed (fact less-post-state-def)
end

```

lemma *Success-le-Success-iff[simp]: Success r \leq Success q \longleftrightarrow r \subseteq q*
by (auto elim: less-eq-post-state.cases)

lemma *Failure-le-iff[simp]: Failure \leq q \longleftrightarrow q = Failure*
by (auto elim: less-eq-post-state.cases)

instantiation *post-state* :: (type) complete-lattice
begin

definition *top-post-state* :: '*a post-state* **where**
top-post-state = Failure

definition *bot-post-state* :: 'a post-state **where**

bot-post-state = Success {}

definition *inf-post-state* :: 'a post-state \Rightarrow 'a post-state \Rightarrow 'a post-state **where**

inf-post-state =
(λ Failure \Rightarrow id | Success res1 \Rightarrow
(λ Failure \Rightarrow Success res1 | Success res2 \Rightarrow Success (res1 \cap res2)))

definition *sup-post-state* :: 'a post-state \Rightarrow 'a post-state \Rightarrow 'a post-state **where**

sup-post-state =
(λ Failure \Rightarrow (λ -. Failure) | Success res1 \Rightarrow
(λ Failure \Rightarrow Failure | Success res2 \Rightarrow Success (res1 \cup res2)))

definition *Inf-post-state* :: 'a post-state set \Rightarrow 'a post-state **where**

Inf-post-state s = (if Success - ' s = {} then Failure else Success (\cap (Success - ' s)))

definition *Sup-post-state* :: 'a post-state set \Rightarrow 'a post-state **where**

Sup-post-state s = (if Failure \in s then Failure else Success (\cup (Success - ' s)))

instance

proof

fix x y z :: 'a post-state **and** A :: 'a post-state set

show *inf* x y \leq y **by** (simp add: *inf-post-state-def* split: post-state.split)

show *inf* x y \leq x **by** (simp add: *inf-post-state-def* split: post-state.split)

show x \leq y \Rightarrow x \leq z \Rightarrow x \leq *inf* y z

by (auto simp add: *inf-post-state-def* elim!: less-eq-post-state.cases)

show x \leq *sup* x y **by** (simp add: *sup-post-state-def* split: post-state.split)

show y \leq *sup* x y **by** (simp add: *sup-post-state-def* split: post-state.split)

show x \leq y \Rightarrow z \leq y \Rightarrow *sup* x z \leq y

by (auto simp add: *sup-post-state-def* elim!: less-eq-post-state.cases)

show x \in A \Rightarrow *Inf* A \leq x **by** (cases x) (auto simp add: *Inf-post-state-def*)

show (\bigwedge x. x \in A \Rightarrow z \leq x) \Rightarrow z \leq *Inf* A **by** (cases z) (force simp add:

Inf-post-state-def)+

show x \in A \Rightarrow x \leq *Sup* A **by** (cases x) (auto simp add: *Sup-post-state-def*)

show (\bigwedge x. x \in A \Rightarrow x \leq z) \Rightarrow *Sup* A \leq z **by** (cases z) (force simp add:

Sup-post-state-def)+

qed (simp-all add: *top-post-state-def* *Inf-post-state-def* *bot-post-state-def* *Sup-post-state-def*)

end

primrec *holds-post-state* :: ('a \Rightarrow bool) \Rightarrow 'a post-state \Rightarrow bool **where**

holds-post-state P Failure \longleftrightarrow False

| *holds-post-state* P (Success X) \longleftrightarrow (\forall x \in X. P x)

primrec *holds-partial-post-state* :: ('a \Rightarrow bool) \Rightarrow 'a post-state \Rightarrow bool **where**

holds-partial-post-state P Failure \longleftrightarrow True

| *holds-partial-post-state* P (Success X) \longleftrightarrow (\forall x \in X. P x)

inductive

sim-post-state :: ('a ⇒ 'b ⇒ bool) ⇒ 'a post-state ⇒ 'b post-state ⇒ bool

for *R*

where

[*simp*]: $\bigwedge p. \text{sim-post-state } R \ p \ \text{Failure}$

| $\bigwedge A \ B. \text{sim-set } R \ A \ B \implies \text{sim-post-state } R \ (\text{Success } A) \ (\text{Success } B)$

inductive

rel-post-state :: ('a ⇒ 'b ⇒ bool) ⇒ 'a post-state ⇒ 'b post-state ⇒ bool

for *R*

where

[*simp*]: *rel-post-state* *R* *Failure* *Failure*

| $\bigwedge A \ B. \text{rel-set } R \ A \ B \implies \text{rel-post-state } R \ (\text{Success } A) \ (\text{Success } B)$

primrec *lift-post-state* :: ('a ⇒ 'b ⇒ bool) ⇒ 'b post-state ⇒ 'a post-state **where**

lift-post-state *R* *Failure* = *Failure*

| *lift-post-state* *R* (*Success* *Y*) = *Success* {*x*. $\exists y \in Y. R \ x \ y$ }

primrec *unlift-post-state* :: ('a ⇒ 'b ⇒ bool) ⇒ 'a post-state ⇒ 'b post-state **where**

unlift-post-state *R* *Failure* = *Failure*

| *unlift-post-state* *R* (*Success* *X*) = *Success* {*y*. $\forall x. R \ x \ y \longrightarrow x \in X$ }

primrec *map-post-state* :: ('a ⇒ 'b) ⇒ 'a post-state ⇒ 'b post-state **where**

map-post-state *f* *Failure* = *Failure*

| *map-post-state* *f* (*Success* *r*) = *Success* (*f* ' *r*)

primrec *vmap-post-state* :: ('a ⇒ 'b) ⇒ 'b post-state ⇒ 'a post-state **where**

vmap-post-state *f* *Failure* = *Failure*

| *vmap-post-state* *f* (*Success* *r*) = *Success* (*f* -' *r*)

definition *pure-post-state* :: 'a ⇒ 'a post-state **where**

pure-post-state *v* = *Success* {*v*}

primrec *bind-post-state* :: 'a post-state ⇒ ('a ⇒ 'b post-state) ⇒ 'b post-state

where

bind-post-state *Failure* *p* = *Failure*

| *bind-post-state* (*Success* *res*) *p* = \bigsqcup (*p* ' *res*)

16.4.1 Order Properties

lemma *top-ne-bot*[*simp*]: ($\perp :: - \text{ post-state}$) $\neq \top$

and *bot-ne-top*[*simp*]: ($\top :: - \text{ post-state}$) $\neq \perp$

by (*auto simp: top-post-state-def bot-post-state-def*)

lemma *Success-ne-top*[*simp*]:

Success *X* $\neq \top$ $\top \neq \text{Success } X$

by (*auto simp: top-post-state-def*)

lemma *Success-eq-bot-iff*[*simp*]:

$Success\ X = \perp \longleftrightarrow X = \{\}$ $\perp = Success\ X \longleftrightarrow X = \{\}$
by (*auto simp: bot-post-state-def*)

lemma *Sup-Success*: $(\bigsqcup x \in X. Success\ (f\ x)) = Success\ (\bigcup x \in X. f\ x)$
by (*auto simp: Sup-post-state-def*)

lemma *Sup-Success-pair*: $(\bigsqcup (x, y) \in X. Success\ (f\ x\ y)) = Success\ (\bigcup (x, y) \in X. f\ x\ y)$
by (*simp add: split-beta' Sup-Success*)

16.4.2 holds-post-state

lemma *holds-post-state-iff*:
 $holds\text{-post}\text{-state}\ P\ p \longleftrightarrow (\exists X. p = Success\ X \wedge (\forall x \in X. P\ x))$
by (*cases p auto*)

lemma *holds-post-state-weaken*:
 $holds\text{-post}\text{-state}\ P\ p \Longrightarrow (\bigwedge x. P\ x \Longrightarrow Q\ x) \Longrightarrow holds\text{-post}\text{-state}\ Q\ p$
by (*cases p; auto*)

lemma *holds-post-state-combine*:
 $holds\text{-post}\text{-state}\ P\ p \Longrightarrow holds\text{-post}\text{-state}\ Q\ p \Longrightarrow$
 $(\bigwedge x. P\ x \Longrightarrow Q\ x \Longrightarrow R\ x) \Longrightarrow holds\text{-post}\text{-state}\ R\ p$
by (*cases p; auto*)

lemma *holds-post-state-Ball*:
 $Y \neq \{\} \vee p \neq Failure \Longrightarrow$
 $holds\text{-post}\text{-state}\ (\lambda x. \forall y \in Y. P\ x\ y)\ p \longleftrightarrow (\forall y \in Y. holds\text{-post}\text{-state}\ (\lambda x. P\ x\ y)\ p)$
by (*cases p; auto*)

lemma *holds-post-state-All*:
 $holds\text{-post}\text{-state}\ (\lambda x. \forall y. P\ x\ y)\ p \longleftrightarrow (\forall y. holds\text{-post}\text{-state}\ (\lambda x. P\ x\ y)\ p)$
by (*cases p; auto*)

lemma *holds-post-state-BexI*:
 $y \in Y \Longrightarrow holds\text{-post}\text{-state}\ (\lambda x. P\ x\ y)\ p \Longrightarrow holds\text{-post}\text{-state}\ (\lambda x. \exists y \in Y. P\ x\ y)\ p$
by (*cases p; auto*)

lemma *holds-post-state-conj*:
 $holds\text{-post}\text{-state}\ (\lambda x. P\ x \wedge Q\ x)\ p \longleftrightarrow holds\text{-post}\text{-state}\ P\ p \wedge holds\text{-post}\text{-state}\ Q\ p$
by (*cases p; auto*)

lemma *sim-post-state-iff*:
 $sim\text{-post}\text{-state}\ R\ p\ q \longleftrightarrow$
 $(\forall P. holds\text{-post}\text{-state}\ (\lambda y. \forall x. R\ x\ y \longrightarrow P\ x)\ q \longrightarrow holds\text{-post}\text{-state}\ P\ p)$
apply (*cases p; cases q; simp add: sim-set-def sim-post-state.simps*)

apply *safe*
apply *force*
apply *force*
subgoal **premises** *prems* **for** *A B*
using *prems*(\mathcal{I})[*THEN spec, of* $\lambda a. \exists b \in B. R a b$] *prems*(\mathcal{I})
by *auto*
done

lemma *post-state-le-iff*:
 $p \leq q \iff (\forall P. \text{holds-post-state } P q \longrightarrow \text{holds-post-state } P p)$
by (*cases p; cases q; force simp add: subset-eq Ball-def*)

lemma *post-state-eq-iff*:
 $p = q \iff (\forall P. \text{holds-post-state } P p \iff \text{holds-post-state } P q)$
by (*simp add: order-eq-iff post-state-le-iff*)

lemma *holds-top-post-state[simp]*: $\neg \text{holds-post-state } P \top$
by (*simp add: top-post-state-def*)

lemma *holds-bot-post-state[simp]*: $\text{holds-post-state } P \perp$
by (*simp add: bot-post-state-def*)

lemma *holds-Sup-post-state[simp]*: $\text{holds-post-state } P (\text{Sup } F) \iff (\forall f \in F. \text{holds-post-state } P f)$
by (*subst (2) split-post-state*)
(auto simp: Sup-post-state-def split: post-state.splits)

lemma *holds-post-state-gfp*:
 $\text{holds-post-state } P (\text{gfp } f) \iff (\forall p. p \leq f p \longrightarrow \text{holds-post-state } P p)$
by (*simp add: gfp-def*)

lemma *holds-post-state-gfp-apply*:
 $\text{holds-post-state } P (\text{gfp } f x) \iff (\forall p. p \leq f p \longrightarrow \text{holds-post-state } P (p x))$
by (*simp add: gfp-def*)

lemma *holds-lift-post-state[simp]*:
 $\text{holds-post-state } P (\text{lift-post-state } R x) \iff \text{holds-post-state } (\lambda y. \forall x. R x y \longrightarrow P x) x$
by (*cases x auto*)

lemma *holds-map-post-state[simp]*:
 $\text{holds-post-state } P (\text{map-post-state } f x) \iff \text{holds-post-state } (\lambda x. P (f x)) x$
by (*cases x auto*)

lemma *holds-vmap-post-state[simp]*:
 $\text{holds-post-state } P (\text{vmap-post-state } f x) \iff \text{holds-post-state } (\lambda x. \forall y. f y = x \longrightarrow P y) x$

by (*cases x*) *auto*

lemma *holds-pure-post-state[simp]*: *holds-post-state P (pure-post-state x) \longleftrightarrow P x*
by (*simp add: pure-post-state-def*)

lemma *holds-bind-post-state[simp]*:
holds-post-state P (bind-post-state f g) \longleftrightarrow holds-post-state ($\lambda x.$ holds-post-state P (g x)) f
by (*cases f*) *auto*

lemma *holds-post-state-False*: *holds-post-state ($\lambda x.$ False) f \longleftrightarrow f = \perp*
by (*cases f*) (*auto simp: bot-post-state-def*)

16.4.3 *holds-post-state-partial*

lemma *holds-partial-post-state-of-holds*:
holds-post-state P p \implies holds-partial-post-state P p
by (*cases p; simp*)

lemma *holds-partial-post-state-iff*:
holds-partial-post-state P p \longleftrightarrow ($\forall X. p = \text{Success } X \longrightarrow (\forall x \in X. P x)$)
by (*cases p*) *auto*

lemma *holds-partial-post-state-True[simp]*: *holds-partial-post-state ($\lambda x.$ True) p*
by (*cases p; simp*)

lemma *holds-partial-post-state-weaken*:
holds-partial-post-state P p \implies ($\bigwedge x. P x \implies Q x$) \implies holds-partial-post-state Q p
by (*cases p; auto*)

lemma *holds-partial-post-state-Ball*:
holds-partial-post-state ($\lambda x. \forall y \in Y. P x y$) p \longleftrightarrow ($\forall y \in Y. \text{holds-partial-post-state } (\lambda x. P x y) p$)
by (*cases p; auto*)

lemma *holds-partial-post-state-All*:
holds-partial-post-state ($\lambda x. \forall y. P x y$) p \longleftrightarrow ($\forall y. \text{holds-partial-post-state } (\lambda x. P x y) p$)
by (*cases p; auto*)

lemma *holds-partial-post-state-conj*:
holds-partial-post-state ($\lambda x. P x \wedge Q x$) p \longleftrightarrow
holds-partial-post-state P p \wedge holds-partial-post-state Q p
by (*cases p; auto*)

lemma *holds-partial-top-post-state[simp]*: *holds-partial-post-state P \top*
by (*simp add: top-post-state-def*)

lemma *holds-partial-bot-post-state*[simp]: *holds-partial-post-state* $P \perp$
by (*simp add: bot-post-state-def*)

lemma *holds-partial-pure-post-state*[simp]: *holds-partial-post-state* P (*pure-post-state* x) $\longleftrightarrow P x$
by (*simp add: pure-post-state-def*)

lemma *holds-partial-Sup-post-stateI*:
 $(\bigwedge x. x \in X \implies \text{holds-partial-post-state } P x) \implies \text{holds-partial-post-state } P (\text{Sup } X)$
by (*force simp: Sup-post-state-def*)

lemma *holds-partial-bind-post-stateI*:
 $\text{holds-partial-post-state } (\lambda x. \text{holds-partial-post-state } P (g x)) f \implies$
 $\text{holds-partial-post-state } P (\text{bind-post-state } f g)$
by (*cases f*) (*auto intro: holds-partial-Sup-post-stateI*)

lemma *holds-partial-map-post-state*[simp]:
 $\text{holds-partial-post-state } P (\text{map-post-state } f x) \longleftrightarrow \text{holds-partial-post-state } (\lambda x. P (f x)) x$
by (*cases x*) *auto*

lemma *holds-partial-vmap-post-state*[simp]:
 $\text{holds-partial-post-state } P (\text{vmap-post-state } f x) \longleftrightarrow$
 $\text{holds-partial-post-state } (\lambda x. \forall y. f y = x \longrightarrow P y) x$
by (*cases x*) *auto*

lemma *holds-partial-bind-post-state*:
 $\text{holds-partial-post-state } (\lambda x. \text{holds-partial-post-state } P (g x)) f \implies$
 $\text{holds-partial-post-state } P (\text{bind-post-state } f g)$
by (*cases f*) (*auto intro: holds-partial-Sup-post-stateI*)

16.4.4 *sim-post-state*

lemma *sim-post-state-eq-iff-le*: *sim-post-state* $(=)$ $p q \longleftrightarrow p \leq q$
by (*simp add: post-state-le-iff sim-post-state-iff*)

lemma *sim-post-state-Success-Success-iff*[simp]:
 $\text{sim-post-state } R (\text{Success } r) (\text{Success } q) \longleftrightarrow (\forall a \in r. \exists b \in q. R a b)$
by (*auto simp add: sim-set-def elim: sim-post-state.cases intro: sim-post-state.intros*)

lemma *sim-post-state-Success2*:
 $\text{sim-post-state } R f (\text{Success } q) \longleftrightarrow \text{holds-post-state } (\lambda a. \exists b \in q. R a b) f$
by (*cases f*; *simp add: sim-post-state.simps sim-set-def*)

lemma *sim-post-state-Failure1*[simp]: *sim-post-state* R *Failure* $q \longleftrightarrow q = \text{Failure}$
by (*auto elim: sim-post-state.cases intro: sim-post-state.intros*)

lemma *sim-post-state-top2*[simp, *intro*]: *sim-post-state* $R p \top$

by (*simp add: top-post-state-def*)

lemma *sim-post-state-top1* [*simp*]: *sim-post-state* $R \top q \longleftrightarrow q = \top$
 using *sim-post-state-Failure1* by (*metis top-post-state-def*)

lemma *sim-post-state-bot2* [*simp, intro*]: *sim-post-state* $R p \perp \longleftrightarrow p = \perp$
 by (*cases p; simp add: bot-post-state-def*)

lemma *sim-post-state-bot1* [*simp, intro*]: *sim-post-state* $R \perp q$
 by (*cases q; simp add: bot-post-state-def*)

lemma *sim-post-state-le1*: *sim-post-state* $R f' g \implies f \leq f' \implies \text{sim-post-state } R f g$
 by (*simp add: post-state-le-iff sim-post-state-iff*)

lemma *sim-post-state-le2*: *sim-post-state* $R f g \implies g \leq g' \implies \text{sim-post-state } R f g'$
 by (*simp add: post-state-le-iff sim-post-state-iff*)

lemma *sim-post-state-Sup1*:
sim-post-state $R (\text{Sup } A) f \longleftrightarrow (\forall a \in A. \text{sim-post-state } R a f)$
 by (*auto simp add: sim-post-state-iff*)

lemma *sim-post-state-Sup2*:
 $a \in A \implies \text{sim-post-state } R f a \implies \text{sim-post-state } R f (\text{Sup } A)$
 by (*auto simp add: sim-post-state-iff*)

lemma *sim-post-state-Sup*:
 $\forall a \in A. \exists b \in B. \text{sim-post-state } R a b \implies \text{sim-post-state } R (\text{Sup } A) (\text{Sup } B)$
 by (*auto simp: sim-post-state-Sup1 intro: sim-post-state-Sup2*)

lemma *sim-post-state-weaken*:
sim-post-state $R f g \implies (\bigwedge x y. R x y \implies Q x y) \implies \text{sim-post-state } Q f g$
 by (*cases f; cases g; force*)

lemma *sim-post-state-trans*:
sim-post-state $R f g \implies \text{sim-post-state } Q g h \implies (\bigwedge x y z. R x y \implies Q y z \implies S x z) \implies$
sim-post-state $S f h$
 by (*fastforce elim!: sim-post-state.cases intro: sim-post-state.intros simp: sim-set-def*)

lemma *sim-post-state-refl'*: *holds-partial-post-state* $(\lambda x. R x x) f \implies \text{sim-post-state } R f f$
 by (*cases f; auto simp: rel-set-def*)

lemma *sim-post-state-refl*: $(\bigwedge x. R x x) \implies \text{sim-post-state } R f f$
 by (*simp add: sim-post-state-refl'*)

lemma *sim-post-state-pure-post-state2*:

sim-post-state $F f$ (*pure-post-state* x) \longleftrightarrow *holds-post-state* $(\lambda y. F y x) f$
by (*cases* f ; *simp* *add*: *pure-post-state-def*)

16.4.5 *rel-post-state*

lemma *rel-post-state-top*[*simp*, *intro!*]: *rel-post-state* $R \top \top$
by (*auto* *simp*: *top-post-state-def*)

lemma *rel-post-state-top-iff*[*simp*]:
rel-post-state $R \top p \longleftrightarrow p = \top$
rel-post-state $R p \top \longleftrightarrow p = \top$
by (*auto* *simp*: *top-post-state-def* *rel-post-state.simps*)

lemma *rel-post-state-Success-iff*[*simp*]:
rel-post-state R (*Success* A) (*Success* B) \longleftrightarrow *rel-set* $R A B$
by (*auto* *elim*: *rel-post-state.cases* *intro*: *rel-post-state.intros*)

lemma *rel-post-state-bot*[*simp*, *intro!*]: *rel-post-state* $R \perp \perp$
by (*auto* *simp*: *bot-post-state-def* *Lifting-Set.empty-transfer*)

lemma *rel-post-state-eq-sim-post-state*:
rel-post-state $R p q \longleftrightarrow$ *sim-post-state* $R p q \wedge$ *sim-post-state* $R^{-1-1} q p$
by (*auto* *simp*: *rel-set-def* *sim-set-def* *elim!*: *sim-post-state.cases* *rel-post-state.cases*)

lemma *rel-post-state-weaken*:
rel-post-state $R f g \implies (\bigwedge x y. R x y \implies Q x y) \implies$ *rel-post-state* $Q f g$
by (*auto* *intro*: *sim-post-state-weaken* *simp*: *rel-post-state-eq-sim-post-state*)

lemma *rel-post-state-eq[relator-eq]*: *rel-post-state* $(=) = (=)$
by (*simp* *add*: *rel-post-state-eq-sim-post-state* *fun-eq-iff* *sim-post-state-eq-iff-le-order-eq-iff*)

lemma *rel-post-state-mono[relator-mono]*:
 $A \leq B \implies$ *rel-post-state* $A \leq$ *rel-post-state* B
using *rel-set-mono[of A B]*
by (*auto* *simp* *add*: *le-fun-def* *intro!*: *rel-post-state.intros* *elim!*: *rel-post-state.cases*)

lemma *rel-post-state-refl'*: *holds-partial-post-state* $(\lambda x. R x x) f \implies$ *rel-post-state* $R f f$
by (*cases* f ; *auto* *simp*: *rel-set-def*)

lemma *rel-post-state-refl*: $(\bigwedge x. R x x) \implies$ *rel-post-state* $R f f$
by (*simp* *add*: *rel-post-state-refl'*)

16.4.6 *lift-post-state*

lemma *lift-post-state-top*: *lift-post-state* $R \top = \top$
by (*auto* *simp*: *top-post-state-def*)

lemma *lift-post-state-unlift-post-state*: $\text{lift-post-state } R \ p \leq q \longleftrightarrow p \leq \text{unlift-post-state } R \ q$

by (*cases p*; *cases q*; *auto simp add: subset-eq*)

lemma *lift-post-state-eq-Sup*: $\text{lift-post-state } R \ p = \text{Sup } \{q. \text{sim-post-state } R \ q \ p\}$

unfolding *post-state-eq-iff holds-Sup-post-state*

using *holds-lift-post-state sim-post-state-iff*

by *blast*

lemma *le-lift-post-state-iff*: $q \leq \text{lift-post-state } R \ p \longleftrightarrow \text{sim-post-state } R \ q \ p$

by (*simp add: post-state-le-iff sim-post-state-iff*)

lemma *lift-post-state-eq[simp]*: $\text{lift-post-state } (=) \ p = p$

by (*simp add: post-state-eq-iff*)

lemma *lift-post-state-comp*:

$\text{lift-post-state } R \ (\text{lift-post-state } Q \ p) = \text{lift-post-state } (R \ OO \ Q) \ p$

by (*cases p*) *auto*

lemma *sim-post-state-lift*:

$\text{sim-post-state } Q \ q \ (\text{lift-post-state } R \ p) \longleftrightarrow \text{sim-post-state } (Q \ OO \ R) \ q \ p$

unfolding *le-lift-post-state-iff[symmetric] lift-post-state-comp ..*

lemma *lift-post-state-Sup*: $\text{lift-post-state } R \ (\text{Sup } F) = (\text{SUP } f \in F. \text{lift-post-state } R \ f)$

by (*simp add: post-state-eq-iff*)

lemma *lift-post-state-mono*: $p \leq q \implies \text{lift-post-state } R \ p \leq \text{lift-post-state } R \ q$

by (*simp add: post-state-le-iff*)

16.4.7 *map-post-state*

lemma *map-post-state-top*: $\text{map-post-state } f \ \top = \top$

by (*auto simp: top-post-state-def*)

lemma *mono-map-post-state*: $s1 \leq s2 \implies \text{map-post-state } f \ s1 \leq \text{map-post-state } f \ s2$

by (*simp add: post-state-le-iff*)

lemma *map-post-state-eq-lift-post-state*: $\text{map-post-state } f \ p = \text{lift-post-state } (\lambda a \ b. a = f \ b) \ p$

by (*cases p*) *auto*

lemma *map-post-state-Sup*: $\text{map-post-state } f \ (\text{Sup } X) = (\text{SUP } x \in X. \text{map-post-state } f \ x)$

by (*simp add: post-state-eq-iff*)

lemma *map-post-state-comp*: $\text{map-post-state } f \ (\text{map-post-state } g \ p) = \text{map-post-state } (f \ o \ g) \ p$

by (*simp add: post-state-eq-iff*)

lemma *map-post-state-id*[*simp*]: *map-post-state id p = p*
by (*simp add: post-state-eq-iff*)

lemma *map-post-state-pure*[*simp*]: *map-post-state f (pure-post-state x) = pure-post-state (f x)*
by (*simp add: pure-post-state-def*)

lemma *sim-post-state-map-post-state1*:
sim-post-state R (map-post-state f p) q \longleftrightarrow sim-post-state ($\lambda x y. R (f x) y$) p q
by (*cases p; cases q; simp*)

lemma *sim-post-state-map-post-state2*:
sim-post-state R p (map-post-state f q) \longleftrightarrow sim-post-state ($\lambda x y. R x (f y)$) p q
unfolding *map-post-state-eq-lift-post-state sim-post-state-lift* **by** (*simp add: OO-def[abs-def]*)

lemma *map-post-state-eq-top*[*simp*]: *map-post-state f p = \top \longleftrightarrow p = \top*
by (*cases p; simp add: top-post-state-def*)

lemma *map-post-state-eq-bot*[*simp*]: *map-post-state f p = \perp \longleftrightarrow p = \perp*
by (*cases p; simp add: bot-post-state-def*)

16.4.8 *vmap-post-state*

lemma *vmap-post-state-top*: *vmap-post-state f \top = \top*
by (*auto simp: top-post-state-def*)

lemma *vmap-post-state-Sup*:
vmap-post-state f (Sup X) = (SUP $x \in X. vmap-post-state f x$)
by (*simp add: post-state-eq-iff*)

lemma *vmap-post-state-le-iff*: *(vmap-post-state f p \leq q) = (p \leq \sqcup {p. vmap-post-state f p \leq q})*
using *vmap-post-state-Sup* **by** (*rule gc-of-Sup-cont*)

lemma *vmap-post-state-eq-lift-post-state*: *vmap-post-state f p = lift-post-state ($\lambda a b. f a = b$) p*
by (*cases p*) *auto*

lemma *vmap-post-state-comp*: *vmap-post-state f (vmap-post-state g p) = vmap-post-state (g \circ f) p*
apply (*simp add: post-state-eq-iff*)
apply (*intro allI arg-cong[where f= $\lambda P. holds-post-state P p$]*)
apply *auto*
done

lemma *vmap-post-state-id*: *vmap-post-state id p = p*
by (*simp add: post-state-eq-iff*)

lemma *sim-post-state-vmap-post-state2*:
 $sim\text{-}post\text{-}state\ R\ p\ (vmap\text{-}post\text{-}state\ f\ q) \longleftrightarrow$
 $sim\text{-}post\text{-}state\ (\lambda x\ y.\ \exists y'.\ f\ y' = y \wedge R\ x\ y')\ p\ q$
by (*cases p; cases q; auto*) *blast+*

lemma *vmap-post-state-le-iff-le-map-post-state*:
 $map\text{-}post\text{-}state\ f\ p \leq q \longleftrightarrow p \leq vmap\text{-}post\text{-}state\ f\ q$
by (*simp flip: sim-post-state-eq-iff-le*
add: sim-post-state-map-post-state1 sim-post-state-vmap-post-state2)

16.4.9 *pure-post-state*

lemma *pure-post-state-Failure[simp]*: $pure\text{-}post\text{-}state\ v \neq Failure$
by (*simp add: pure-post-state-def*)

lemma *pure-post-state-top[simp]*: $pure\text{-}post\text{-}state\ v \neq \top$
by (*simp add: pure-post-state-def top-post-state-def*)

lemma *pure-post-state-bot[simp]*: $pure\text{-}post\text{-}state\ v \neq \perp$
by (*simp add: pure-post-state-def bot-post-state-def*)

lemma *pure-post-state-inj[simp]*: $pure\text{-}post\text{-}state\ v = pure\text{-}post\text{-}state\ w \longleftrightarrow v = w$
by (*simp add: pure-post-state-def*)

lemma *sim-pure-post-state-iff[simp]*:
 $sim\text{-}post\text{-}state\ R\ (pure\text{-}post\text{-}state\ a)\ (pure\text{-}post\text{-}state\ b) \longleftrightarrow R\ a\ b$
by (*simp add: pure-post-state-def*)

lemma *rel-pure-post-state-iff[simp]*:
 $rel\text{-}post\text{-}state\ R\ (pure\text{-}post\text{-}state\ a)\ (pure\text{-}post\text{-}state\ b) \longleftrightarrow R\ a\ b$
by (*simp add: rel-post-state-eq-sim-post-state*)

lemma *pure-post-state-le[simp]*: $pure\text{-}post\text{-}state\ v \leq pure\text{-}post\text{-}state\ w \longleftrightarrow v = w$
by (*simp add: pure-post-state-def*)

lemma *Success-eq-pure-post-state[simp]*: $Success\ X = pure\text{-}post\text{-}state\ x \longleftrightarrow X = \{x\}$
by (*auto simp add: pure-post-state-def*)

lemma *pure-post-state-eq-Success[simp]*: $pure\text{-}post\text{-}state\ x = Success\ X \longleftrightarrow X = \{x\}$
by (*auto simp add: pure-post-state-def*)

lemma *pure-post-state-le-Success[simp]*: $pure\text{-}post\text{-}state\ x \leq Success\ X \longleftrightarrow x \in X$
by (*auto simp add: pure-post-state-def*)

lemma *Success-le-pure-post-state[simp]*: $Success\ X \leq pure\text{-}post\text{-}state\ x \longleftrightarrow X \subseteq \{x\}$

by (auto simp add: pure-post-state-def)

16.4.10 bind-post-state

lemma *bind-post-state-top*: $\text{bind-post-state } \top g = \top$
by (auto simp: top-post-state-def)

lemma *bind-post-state-Sup1*[simp]:
 $\text{bind-post-state } (\text{Sup } F) g = (\text{SUP } f \in F. \text{bind-post-state } f g)$
by (simp add: post-state-eq-iff)

lemma *bind-post-state-Sup2*[simp]:
 $G \neq \{\} \vee f \neq \text{Failure} \implies \text{bind-post-state } f (\text{Sup } G) = (\text{SUP } g \in G. \text{bind-post-state } f g)$
by (simp add: post-state-eq-iff holds-post-state-Ball)

lemma *bind-post-state-sup1*[simp]:
 $\text{bind-post-state } f (\text{sup } f1 f2) g = \text{sup } (\text{bind-post-state } f1 g) (\text{bind-post-state } f2 g)$
using *bind-post-state-Sup1*[of {f1, f2} g] by simp

lemma *bind-post-state-sup2*[simp]:
 $\text{bind-post-state } f (\text{sup } g1 g2) = \text{sup } (\text{bind-post-state } f g1) (\text{bind-post-state } f g2)$
using *bind-post-state-Sup2*[of {g1, g2} f] by simp

lemma *bind-post-state-top1*[simp]: $\text{bind-post-state } \top f = \top$
by (simp add: post-state-eq-iff)

lemma *bind-post-state-bot1*[simp]: $\text{bind-post-state } \perp f = \perp$
by (simp add: post-state-eq-iff)

lemma *bind-post-state-eq-top*:
 $\text{bind-post-state } f g = \top \iff \neg \text{holds-post-state } (\lambda x. g x \neq \top) f$
by (cases f) (force simp add: Sup-post-state-def simp flip: top-post-state-def)+

lemma *bind-post-state-eq-bot*:
 $\text{bind-post-state } f g = \perp \iff \text{holds-post-state } (\lambda x. g x = \perp) f$
by (cases f) (auto simp flip: top-post-state-def)

lemma *lift-post-state-bind-post-state*:
 $\text{lift-post-state } R (\text{bind-post-state } x g) = \text{bind-post-state } x (\lambda v. \text{lift-post-state } R (g v))$
by (simp add: post-state-eq-iff)

lemma *vmap-post-state-bind-post-state*:
 $\text{vmap-post-state } f (\text{bind-post-state } p g) = \text{bind-post-state } p (\lambda v. \text{vmap-post-state } f (g v))$
by (simp add: post-state-eq-iff)

lemma *map-post-state-bind-post-state*:

$\text{map-post-state } f \text{ (bind-post-state } x \text{ } g) = \text{bind-post-state } x \text{ (}\lambda v. \text{map-post-state } f \text{ (} g \text{ } v))$

by (*simp add: post-state-eq-iff*)

lemma *bind-post-state-pure-post-state1*[*simp*]:

$\text{bind-post-state (pure-post-state } v) f = f v$

by (*simp add: post-state-eq-iff*)

lemma *bind-post-state-pure-post-state2*:

$\text{bind-post-state } p \text{ (}\lambda x. \text{pure-post-state (} f \text{ } x)) = \text{map-post-state } f \text{ } p$

by (*simp add: post-state-eq-iff*)

lemma *bind-post-state-map-post-state*:

$\text{bind-post-state (map-post-state } f \text{ } p) g = \text{bind-post-state } p \text{ (}\lambda x. g \text{ (} f \text{ } x))$

by (*simp add: post-state-eq-iff*)

lemma *bind-post-state-assoc*[*simp*]:

$\text{bind-post-state (bind-post-state } f \text{ } g) h =$

$\text{bind-post-state } f \text{ (}\lambda v. \text{bind-post-state (} g \text{ } v) \text{ } h)$

by (*simp add: post-state-eq-iff*)

lemma *sim-bind-post-state'*:

$\text{sim-post-state (}\lambda x y. \text{sim-post-state } R \text{ (} g \text{ } x) \text{ (} k \text{ } y)) f h \implies$

$\text{sim-post-state } R \text{ (bind-post-state } f \text{ } g) \text{ (bind-post-state } h \text{ } k)$

by (*cases f; cases h; simp add: sim-post-state-Sup*)

lemma *sim-bind-post-state-left-iff*:

$\text{sim-post-state } R \text{ (bind-post-state } f \text{ } g) h \longleftrightarrow$

$h = \text{Failure} \vee \text{holds-post-state (}\lambda x. \text{sim-post-state } R \text{ (} g \text{ } x) \text{ } h) f$

by (*cases f; cases h (simp-all add: sim-post-state-Sup1)*)

lemma *sim-bind-post-state-left*:

$\text{holds-post-state (}\lambda x. \text{sim-post-state } R \text{ (} g \text{ } x) \text{ } h) f \implies$

$\text{sim-post-state } R \text{ (bind-post-state } f \text{ } g) h$

by (*simp add: sim-bind-post-state-left-iff*)

lemma *sim-bind-post-state-right*:

$g \neq \perp \implies \text{holds-partial-post-state (}\lambda x. \text{sim-post-state } R \text{ } f \text{ (} h \text{ } x)) g \implies$

$\text{sim-post-state } R \text{ } f \text{ (bind-post-state } g \text{ } h)$

by (*cases g (auto intro: sim-post-state-Sup2)*)

lemma *sim-bind-post-state*:

$\text{sim-post-state } Q \text{ } f \text{ } h \implies (\bigwedge x y. Q \text{ } x \text{ } y \implies \text{sim-post-state } R \text{ (} g \text{ } x) \text{ (} k \text{ } y)) \implies$

$\text{sim-post-state } R \text{ (bind-post-state } f \text{ } g) \text{ (bind-post-state } h \text{ } k)$

by (*rule sim-bind-post-state'[OF sim-post-state-weaken, of Q]*)

lemma *rel-bind-post-state'*:

$\text{rel-post-state (}\lambda a b. \text{rel-post-state } R \text{ (} g1 \text{ } a) \text{ (} g2 \text{ } b)) f1 \text{ } f2 \implies$

$\text{rel-post-state } R \text{ (bind-post-state } f1 \text{ } g1) \text{ (bind-post-state } f2 \text{ } g2)$

by (*auto simp: rel-post-state-eq-sim-post-state*
intro: sim-bind-post-state' sim-post-state-weaken)

lemma *rel-bind-post-state*:
 $rel\text{-}post\text{-}state\ Q\ f1\ f2 \implies (\bigwedge a\ b.\ Q\ a\ b \implies rel\text{-}post\text{-}state\ R\ (g1\ a)\ (g2\ b)) \implies$
 $rel\text{-}post\text{-}state\ R\ (bind\text{-}post\text{-}state\ f1\ g1)\ (bind\text{-}post\text{-}state\ f2\ g2)$
by (*rule rel-bind-post-state'[OF rel-post-state-weaken, of Q]*)

lemma *mono-bind-post-state*: $f1 \leq f2 \implies g1 \leq g2 \implies bind\text{-}post\text{-}state\ f1\ g1 \leq$
 $bind\text{-}post\text{-}state\ f2\ g2$
by (*simp flip: sim-post-state-eq-iff-le add: le-fun-def sim-bind-post-state*)

lemma *Sup-eq-Failure[simp]*: $Sup\ X = Failure \longleftrightarrow Failure \in X$
by (*simp add: Sup-post-state-def*)

lemma *Failure-inf-iff*: $(Failure = x \sqcap y) \longleftrightarrow (x = Failure \wedge y = Failure)$
by (*metis Failure-le-iff le-inf-iff*)

lemma *Success-vimage-singleton-cancel*: $Success\ \text{-}'\ \{Success\ X\} = \{X\}$
by *auto*

lemma *Success-vimage-image-cancel*: $Success\ \text{-}'\ (\lambda x.\ Success\ (f\ x))\ \text{'}\ X = f\ \text{'}\ X$
by *auto*

lemma *Success-image-comp*: $(\lambda x.\ Success\ (g\ x))\ \text{'}\ X = Success\ \text{'}\ (g\ \text{'}\ X)$
by *auto*

16.5 *exception-or-result* type

instantiation *option* :: (*type*) *default*
begin

definition *default-option* = *None*

instance ..
end

lemma *Some-ne-default[simp]*:
 $Some\ x \neq default\ default \neq Some\ x$
 $default = None \longleftrightarrow True\ None = default \longleftrightarrow True$
by (*auto simp: default-option-def*)

typedef (**overloaded**) (*'a::default, 'b*) *exception-or-result* =
 $Inl\ \text{'}\ (UNIV - \{default\}) \cup Inr\ \text{'}\ UNIV :: ('a + 'b)\ set$
by *auto*

setup-lifting *type-definition-exception-or-result*

context *assumes* *SORT-CONSTRAINT('e::default)*
begin

lift-definition *Exception* :: 'e \Rightarrow ('e, 'v) *exception-or-result* **is**
 $\lambda e.$ if $e = \text{default}$ then *Inr undefined* else *Inl e*
by *auto*

lift-definition *Result* :: 'v \Rightarrow ('e, 'v) *exception-or-result* **is**
 $\lambda v.$ *Inr v*
by *auto*

end

lift-definition
case-exception-or-result :: ('e::default \Rightarrow 'a) \Rightarrow ('v \Rightarrow 'a) \Rightarrow ('e, 'v) *exception-or-result* \Rightarrow 'a
is *case-sum* .

declare [[*case-translation case-exception-or-result Exception Result*]]

lemma *Result-eq-Result[simp]*: *Result a = Result b* \longleftrightarrow $a = b$
by *transfer simp*

lemma *Exception-eq-Exception[simp]*: *Exception a = Exception b* \longleftrightarrow $a = b$
by *transfer simp*

lemma *Result-eq-Exception[simp]*: *Result a = Exception e* \longleftrightarrow ($e = \text{default} \wedge a = \text{undefined}$)
by *transfer simp*

lemma *Exception-eq-Result[simp]*: *Exception e = Result a* \longleftrightarrow ($e = \text{default} \wedge a = \text{undefined}$)
by (*metis Result-eq-Exception*)

lemma *exception-or-result-cases[case-names Exception Result, cases type: exception-or-result]*:
 $(\bigwedge e. e \neq \text{default} \Longrightarrow x = \text{Exception } e \Longrightarrow P) \Longrightarrow (\bigwedge v. x = \text{Result } v \Longrightarrow P) \Longrightarrow P$
by (*transfer fixing: P*) *auto*

lemma *case-exception-or-result-Result[simp]*:
(case Result v of Exception e \Rightarrow f e | Result w \Rightarrow g w) = g v
by *transfer simp*

lemma *case-exception-or-result-Exception[simp]*:
(case Exception e of Exception e \Rightarrow f e | Result w \Rightarrow g w) = (if $e = \text{default}$ then g undefined else f e)
by *transfer simp*

Caution: for split rules don't use syntax *case r of Exception e \Rightarrow f e |*

Result $w \Rightarrow g w$, as this introduces non eta-contracted f and g , which don't work with the splitter.

lemma *exception-or-result-split*:

P (*case-exception-or-result* $f g r$) \longleftrightarrow
 $(\forall e. e \neq \text{default} \longrightarrow r = \text{Exception } e \longrightarrow P (f e)) \wedge$
 $(\forall v. r = \text{Result } v \longrightarrow P (g v))$
by (*cases* r) *simp-all*

lemma *exception-or-result-split-asm*:

P (*case-exception-or-result* $f g r$) \longleftrightarrow
 $\neg ((\exists e. r = \text{Exception } e \wedge e \neq \text{default} \wedge \neg P (f e)) \vee$
 $(\exists v. r = \text{Result } v \wedge \neg P (g v)))$
by (*cases* r) *simp-all*

lemmas *exception-or-result-splits* = *exception-or-result-split exception-or-result-split-asm*

lemma *split-exception-or-result*:

$r = (\text{case } r \text{ of } \text{Exception } e \Rightarrow \text{Exception } e \mid \text{Result } v \Rightarrow \text{Result } v)$
by (*cases* r) *auto*

lemma *exception-or-result-nchotomy*:

$\neg ((\forall e. e \neq \text{default} \longrightarrow x \neq \text{Exception } e) \wedge (\forall v. x \neq \text{Result } v))$ **by** (*cases* x)
auto

lemma *val-split*:

fixes $r::(\text{unit}, 'a)$ *exception-or-result*
shows
 P (*case-exception-or-result* $f g r$) \longleftrightarrow
 $(\forall v. r = \text{Result } v \longrightarrow P (g v))$
by (*cases* r) *simp-all*

lemma *val-split-asm*:

fixes $r::(\text{unit}, 'a)$ *exception-or-result*
shows P (*case-exception-or-result* $f g r$) \longleftrightarrow
 $\neg (\exists v. r = \text{Result } v \wedge \neg P (g v))$
by (*cases* r) *simp-all*

lemmas *val-splits*[*split*] = *val-split val-split-asm*

instantiation *exception-or-result* :: ($\{\text{equal}, \text{default}\}, \text{equal}$) *equal* **begin**

definition *equal-exception-or-result* $a b =$

(*case* a *of*
 $\text{Exception } e \Rightarrow (\text{case } b \text{ of } \text{Exception } e' \Rightarrow \text{HOL.equal } e e' \mid \text{Result } s \Rightarrow \text{False})$
 $\mid \text{Result } r \Rightarrow (\text{case } b \text{ of } \text{Exception } e \Rightarrow \text{False} \mid \text{Result } s \Rightarrow \text{HOL.equal } r s)$
 $)$

instance **proof** **qed** (*simp add: equal-exception-or-result-def equal-eq split: exception-or-result-splits*)

end

definition *is-Exception*:: ('e::default, 'a) exception-or-result \Rightarrow bool **where**
is-Exception x \equiv (case x of Exception e \Rightarrow e \neq default | Result - \Rightarrow False)

definition *is-Result*:: ('e::default, 'a) exception-or-result \Rightarrow bool **where**
is-Result x \equiv (case x of Exception e \Rightarrow e = default | Result - \Rightarrow True)

definition *the-Exception*:: ('e::default, 'a) exception-or-result \Rightarrow 'e **where**
the-Exception x \equiv (case x of Exception e \Rightarrow e | Result - \Rightarrow default)

definition *the-Result*:: ('e::default, 'a) exception-or-result \Rightarrow 'a **where**
the-Result x \equiv (case x of Exception e \Rightarrow undefined | Result v \Rightarrow v)

lemma *is-Exception-simps*[simp]:
is-Exception (Exception e) = (e \neq default)
is-Exception (Result v) = False
by (auto simp add: *is-Exception-def*)

lemma *is-Result-simps*[simp]:
is-Result (Exception e) = (e = default)
is-Result (Result v) = True
by (auto simp add: *is-Result-def*)

lemma *the-Exception-simp*[simp]:
the-Exception (Exception e) = e
by (auto simp add: *the-Exception-def*)

lemma *the-Exception-Result*:
the-Exception (Result v) = default
by (simp add: *the-Exception-def*)

syntax -Res :: ptrn \Rightarrow ptrn (Res -)

definition *undefined-unit*::unit \Rightarrow 'b **where** *undefined-unit* x \equiv undefined

translations λ Res x. b \Leftrightarrow CONST case-exception-or-result (CONST *undefined-unit*)
(λ x. b)

term λ Res x. f x

term λ Res (x, y, z). f x y z

term λ Res (x, y, z) s. P x y s z

lifting-update *exception-or-result.lifting*

lifting-forget *exception-or-result.lifting*

inductive *rel-exception-or-result*:: ('e::default \Rightarrow 'f::default \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b
 \Rightarrow bool) \Rightarrow

$(\text{'e, 'a}) \text{ exception-or-result} \Rightarrow (\text{'f, 'b}) \text{ exception-or-result} \Rightarrow \text{bool}$
where

Exception:

$E e f \Rightarrow e \neq \text{default} \Rightarrow f \neq \text{default} \Rightarrow$
 $\text{rel-exception-or-result } E R (\text{Exception } e) (\text{Exception } f) \mid$

Result:

$R a b \Rightarrow \text{rel-exception-or-result } E R (\text{Result } a) (\text{Result } b)$

lemma *All-exception-or-result-cases:*

$(\forall x. P (x::(-, -) \text{ exception-or-result})) \longleftrightarrow$
 $(\forall \text{err}. \text{err} \neq \text{default} \longrightarrow P (\text{Exception } \text{err})) \wedge (\forall v. P (\text{Result } v))$
by (*metis exception-or-result-cases*)

lemma *Ball-exception-or-result-cases:*

$(\forall x \in s. P (x::(-, -) \text{ exception-or-result})) \longleftrightarrow$
 $(\forall \text{err}. \text{err} \neq \text{default} \longrightarrow \text{Exception } \text{err} \in s \longrightarrow P (\text{Exception } \text{err})) \wedge$
 $(\forall v. \text{Result } v \in s \longrightarrow P (\text{Result } v))$
by (*metis exception-or-result-cases*)

lemma *Bex-exception-or-result-cases:*

$(\exists x \in s. P (x::(-, -) \text{ exception-or-result})) \longleftrightarrow$
 $(\exists \text{err}. \text{err} \neq \text{default} \wedge \text{Exception } \text{err} \in s \wedge P (\text{Exception } \text{err})) \vee$
 $(\exists v. \text{Result } v \in s \wedge P (\text{Result } v))$
by (*metis exception-or-result-cases*)

lemma *Ex-exception-or-result-cases:*

$(\exists x. P (x::(-, -) \text{ exception-or-result})) \longleftrightarrow$
 $(\exists \text{err}. \text{err} \neq \text{default} \wedge P (\text{Exception } \text{err})) \vee$
 $(\exists v. P (\text{Result } v))$
by (*metis exception-or-result-cases*)

lemma *case-distrib-exception-or-result:*

$f (\text{case } x \text{ of } \text{Exception } e \Rightarrow E e \mid \text{Result } v \Rightarrow R v) = (\text{case } x \text{ of } \text{Exception } e \Rightarrow f$
 $(E e) \mid \text{Result } v \Rightarrow f (R v))$
by (*auto split: exception-or-result-split*)

lemma *rel-exception-or-result-Results[simp]:*

$\text{rel-exception-or-result } E R (\text{Result } a) (\text{Result } b) \longleftrightarrow R a b$
by (*simp add: rel-exception-or-result.simps*)

lemma *rel-exception-or-result-Exception[simp]:*

$e \neq \text{default} \Rightarrow f \neq \text{default} \Rightarrow$
 $\text{rel-exception-or-result } E R (\text{Exception } e) (\text{Exception } f) \longleftrightarrow E e f$
by (*simp add: rel-exception-or-result.simps*)

lemma *rel-exception-or-result-Result-Exception[simp]:*

$e \neq \text{default} \Rightarrow \neg \text{rel-exception-or-result } E R (\text{Exception } e) (\text{Result } b)$
 $f \neq \text{default} \Rightarrow \neg \text{rel-exception-or-result } E R (\text{Result } a) (\text{Exception } f)$
by (*auto simp add: rel-exception-or-result.simps*)

lemma *is-Exception-iff*: $is-Exception\ x \longleftrightarrow (\exists e. e \neq default \wedge x = Exception\ e)$
apply (*cases x*)
apply (*auto*)
done

lemma *rel-exception-or-result-converse*:
 $(rel-exception-or-result\ E\ R)^{-1-1} = rel-exception-or-result\ E^{-1-1}\ R^{-1-1}$
apply (*rule ext*)
apply (*auto simp add: rel-exception-or-result.simps*)
done

lemma *rel-exception-or-result-Result[simp]*:
 $rel-exception-or-result\ E\ R\ (Result\ x)\ (Result\ y) = R\ x\ y$
by (*auto simp add: rel-exception-or-result.simps*)

lemma *rel-exception-or-result-eq-conv*: $rel-exception-or-result\ (=)\ (=)\ (=)$
apply (*rule ext*)
by (*auto simp add: rel-exception-or-result.simps*)
(*meson exception-or-result-cases*)

lemma *rel-exception-or-result-sum-eq*: $rel-exception-or-result\ (=)\ (=)\ (=)$
apply (*rule ext*)
apply (*subst (1) split-exception-or-result*)
apply (*auto simp add: rel-exception-or-result.simps split: exception-or-result-splits*)
done

definition

map-exception-or-result::
 $(e::default \Rightarrow 'f::default) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('e, 'a)\ exception-or-result \Rightarrow ('f, 'b)\ exception-or-result$

where

$map-exception-or-result\ E\ F\ x =$
 $(case\ x\ of\ Exception\ e \Rightarrow Exception\ (E\ e) \mid Result\ v \Rightarrow Result\ (F\ v))$

lemma *map-exception-or-result-Exception[simp]*:
 $e \neq default \Longrightarrow map-exception-or-result\ E\ F\ (Exception\ e) = Exception\ (E\ e)$
by (*simp add: map-exception-or-result-def*)

lemma *map-exception-or-result-Result[simp]*:
 $map-exception-or-result\ E\ F\ (Result\ v) = Result\ (F\ v)$
by (*simp add: map-exception-or-result-def*)

lemma *map-exception-or-result-id*: $map-exception-or-result\ id\ id\ x = x$
by (*cases x; simp*)

lemma *map-exception-or-result-comp*:
assumes $E2: \bigwedge x. x \neq default \Longrightarrow E2\ x \neq default$
shows $map-exception-or-result\ E1\ F1\ (map-exception-or-result\ E2\ F2\ x) =$

map-exception-or-result ($E1 \circ E2$) ($F1 \circ F2$) x
by (*cases* x ; *auto dest*: $E2$)

lemma *le-bind-post-state-exception-or-result-cases*[*case-names* $Exception$ $Result$]:
assumes
holds-partial-post-state ($\lambda(x, t). \forall e. e \neq \text{default} \longrightarrow x = \text{Exception } e \longrightarrow X e$
 $t \leq X' e t$) x
assumes *holds-partial-post-state* ($\lambda(x, t). \forall v. x = \text{Result } v \longrightarrow V v t \leq V' v t$)
 x
shows *bind-post-state* x ($\lambda(r, t). \text{case } r \text{ of } \text{Exception } e \Rightarrow X e t \mid \text{Result } v \Rightarrow V$
 $v t$) \leq
bind-post-state x ($\lambda(r, t). \text{case } r \text{ of } \text{Exception } e \Rightarrow X' e t \mid \text{Result } v \Rightarrow V' v t$)
using *assms*
by (*cases* x ; *force intro!*: *SUP-mono'' split*: *exception-or-result-splits prod.splits*)

16.6 *spec-monad* type

typedef (**overloaded**) ($'e::\text{default}, 'a, 's$) *spec-monad* =
 $UNIV :: ('s \Rightarrow (('e::\text{default}, 'a) \text{exception-or-result} \times 's) \text{post-state}) \text{set}$
morphisms *run Spec*
by (*fact UNIV-witness*)

lemma *run-case-prod-distrib*[*simp*]:
 $\text{run } (\text{case } x \text{ of } (r, s) \Rightarrow f r s) t = (\text{case } x \text{ of } (r, s) \Rightarrow \text{run } (f r s) t)$
by (*rule prod.case-distrib*)

lemma *image-case-prod-distrib*[*simp*]:
 $(\lambda(r, t). f r t) ' R = (\lambda v. (g v, h v)) ' R = (\lambda v. (f (g v) (h v))) ' R$
by *auto*

lemma *run-Spec*[*simp*]: $\text{run } (\text{Spec } p) s = p s$
by (*simp add: Spec-inverse*)

setup-lifting *type-definition-spec-monad*

lemma *spec-monad-ext*: $(\bigwedge s. \text{run } f s = \text{run } g s) \Longrightarrow f = g$
apply *transfer*
apply *auto*
done

lemma *spec-monad-ext-iff*: $f = g \longleftrightarrow (\forall s. \text{run } f s = \text{run } g s)$
using *spec-monad-ext* **by** *auto*

instantiation *spec-monad* :: (*default, type, type*) *complete-lattice*
begin

lift-definition

less-eq-spec-monad :: ($'e::\text{default}, 'r, 's$) *spec-monad* \Rightarrow ($'e, 'r, 's$) *spec-monad* \Rightarrow
bool

is (\leq) .

lift-definition

less-spec-monad :: ('e::default, 'r, 's) spec-monad \Rightarrow ('e, 'r, 's) spec-monad \Rightarrow bool

is ($<$) .

lift-definition *bot-spec-monad* :: ('e::default, 'r, 's) spec-monad **is** bot .

lift-definition *top-spec-monad* :: ('e::default, 'r, 's) spec-monad **is** top .

lift-definition *Inf-spec-monad* :: ('e::default, 'r, 's) spec-monad set \Rightarrow ('e, 'r, 's) spec-monad

is *Inf* .

lift-definition *Sup-spec-monad* :: ('e::default, 'r, 's) spec-monad set \Rightarrow ('e, 'r, 's) spec-monad

is *Sup* .

lift-definition

sup-spec-monad ::

('e::default, 'r, 's) spec-monad \Rightarrow ('e, 'r, 's) spec-monad \Rightarrow ('e, 'r, 's) spec-monad

is *sup* .

lift-definition

inf-spec-monad ::

('e::default, 'r, 's) spec-monad \Rightarrow ('e, 'r, 's) spec-monad \Rightarrow ('e, 'r, 's) spec-monad

is *inf* .

instance

apply (*standard*; *transfer*)

subgoal by (*rule less-le-not-le*)

subgoal by (*rule order-refl*)

subgoal by (*rule order-trans*)

subgoal by (*rule antisym*)

subgoal by (*rule inf-le1*)

subgoal by (*rule inf-le2*)

subgoal by (*rule inf-greatest*)

subgoal by (*rule sup-ge1*)

subgoal by (*rule sup-ge2*)

subgoal by (*rule sup-least*)

subgoal by (*rule Inf-lower*)

subgoal by (*rule Inf-greatest*)

subgoal by (*rule Sup-upper*)

subgoal by (*rule Sup-least*)

subgoal by (*rule Inf-empty*)

subgoal by (*rule Sup-empty*)

done

end

lift-definition *fail* :: ('e::default, 'a, 's) spec-monad
is λs . *Failure* .

lift-definition

bind-exception-or-result ::
('e::default, 'a, 's) spec-monad \Rightarrow
(('e, 'a) exception-or-result \Rightarrow ('f::default, 'b, 's) spec-monad) \Rightarrow ('f, 'b, 's)
spec-monad
is
 $\lambda f h s$. *bind-post-state* (f s) ($\lambda(v, t)$. h v t) .

lift-definition *bind-handle* ::

('e::default, 'a, 's) spec-monad \Rightarrow
('a \Rightarrow ('f, 'b, 's) spec-monad) \Rightarrow ('e \Rightarrow ('f, 'b, 's) spec-monad) \Rightarrow
('f::default, 'b, 's) spec-monad
is $\lambda f g h s$. *bind-post-state* (f s) ($\lambda(r, t)$. case r of *Exception* e \Rightarrow h e t | *Result* v
 \Rightarrow g v t) .

lift-definition *yield* :: ('e, 'a) exception-or-result \Rightarrow ('e::default, 'a, 's) spec-monad
is $\lambda r s$. *pure-post-state* (r, s) .

abbreviation *return* r \equiv *yield* (*Result* r)

abbreviation *skip* \equiv *return* ()

abbreviation *throw-exception-or-result* e \equiv *yield* (*Exception* e)

lift-definition *get-state* :: ('e::default, 's, 's) spec-monad
is λs . *pure-post-state* (*Result* s, s) .

lift-definition *set-state* :: 's \Rightarrow ('e::default, unit, 's) spec-monad
is $\lambda t s$. *pure-post-state* (*Result* (), t) .

lift-definition *map-value* ::

(('e::default, 'a) exception-or-result \Rightarrow ('f::default, 'b) exception-or-result) \Rightarrow
('e, 'a, 's) spec-monad \Rightarrow ('f, 'b, 's) spec-monad
is $\lambda f g s$. *map-post-state* (apfst f) (g s) .

lift-definition *vmap-value* ::

(('e::default, 'a) exception-or-result \Rightarrow ('f::default, 'b) exception-or-result) \Rightarrow
('f, 'b, 's) spec-monad \Rightarrow ('e, 'a, 's) spec-monad
is $\lambda f g s$. *vmap-post-state* (apfst f) (g s) .

definition *bind* ::

('e::default, 'a, 's) spec-monad \Rightarrow ('a \Rightarrow ('e, 'b, 's) spec-monad) \Rightarrow ('e, 'b, 's)
spec-monad

where

bind f g = *bind-handle* f g *throw-exception-or-result*

adhoc-overloading*Monad-Syntax.bind bind***lift-definition** *lift-state* ::
$$('s \Rightarrow 't \Rightarrow \text{bool}) \Rightarrow ('e::\text{default}, 'a, 't) \text{ spec-monad} \Rightarrow ('e, 'a, 's) \text{ spec-monad}$$

is $\lambda R p s. \text{lift-post-state } (\text{rel-prod } (=) R) (SUP\ t \in \{t. R\ s\ t\}. p\ t) .$

definition *exec-concrete* ::
$$('s \Rightarrow 't) \Rightarrow ('e::\text{default}, 'a, 's) \text{ spec-monad} \Rightarrow ('e, 'a, 't) \text{ spec-monad}$$
where

$$\text{exec-concrete } st = \text{lift-state } (\lambda t\ s. t = st\ s)$$
definition *exec-abstract* ::
$$('s \Rightarrow 't) \Rightarrow ('e::\text{default}, 'a, 't) \text{ spec-monad} \Rightarrow ('e, 'a, 's) \text{ spec-monad}$$
where

$$\text{exec-abstract } st = \text{lift-state } (\lambda s\ t. t = st\ s)$$
definition *select-value* :: $('e, 'a)$ *exception-or-result set* $\Rightarrow ('e::\text{default}, 'a, 's)$ *spec-monad***where**

$$\text{select-value } R = (SUP\ r \in R. \text{yield } r)$$
definition *select* :: $'a$ *set* $\Rightarrow ('e::\text{default}, 'a, 's)$ *spec-monad* **where**

$$\text{select } S = (SUP\ a \in S. \text{return } a)$$
definition *unknown* :: $('e::\text{default}, 'a, 's)$ *spec-monad* **where**

$$\text{unknown} \equiv \text{select } UNIV$$
definition *gets* :: $('s \Rightarrow 'a) \Rightarrow ('e::\text{default}, 'a, 's)$ *spec-monad* **where**

$$\text{gets } f = \text{bind } \text{get-state } (\lambda s. \text{return } (f\ s))$$
definition *assert-opt* :: $'a$ *option* $\Rightarrow ('e::\text{default}, 'a, 's)$ *spec-monad* **where**

$$\text{assert-opt } v = (\text{case } v \text{ of } None \Rightarrow \text{fail} \mid Some\ v \Rightarrow \text{return } v)$$
definition *gets-the* :: $('s \Rightarrow 'a \text{ option}) \Rightarrow ('e::\text{default}, 'a, 's)$ *spec-monad* **where**

$$\text{gets-the } f = \text{bind } (\text{gets } f) \text{ assert-opt}$$
definition *modify* :: $('s \Rightarrow 's) \Rightarrow ('e::\text{default}, \text{unit}, 's)$ *spec-monad* **where**

$$\text{modify } f = \text{bind } \text{get-state } (\lambda s. \text{set-state } (f\ s))$$
definition *assume* :: *bool* $\Rightarrow ('e::\text{default}, \text{unit}, 's)$ *spec-monad* **where**

$$\text{assume } P = (\text{if } P \text{ then return } () \text{ else bot})$$
definition *assert* :: *bool* $\Rightarrow ('e::\text{default}, \text{unit}, 's)$ *spec-monad* **where**

$$\text{assert } P = (\text{if } P \text{ then return } () \text{ else top})$$
definition *assuming* :: $('s \Rightarrow \text{bool}) \Rightarrow ('e::\text{default}, \text{unit}, 's)$ *spec-monad* **where**

$$\text{assuming } P = \text{do } \{s \leftarrow \text{get-state}; \text{assume } (P\ s)\}$$

definition *guard*:: ('s ⇒ bool) ⇒ ('e::default, unit, 's) spec-monad **where**
guard P = do {s ← get-state; assert (P s)}

definition *assume-result-and-state* :: ('s ⇒ ('a × 's) set) ⇒ ('e::default, 'a, 's) spec-monad **where**
assume-result-and-state f = do {s ← get-state; (v, t) ← select (f s); set-state t; return v}

lift-definition *assume-outcome* ::
('s ⇒ (('e, 'a) exception-or-result × 's) set) ⇒ ('e::default, 'a, 's) spec-monad
is λf s. Success (f s) .

definition *assert-result-and-state* ::
('s ⇒ ('a × 's) set) ⇒ ('e::default, 'a, 's) spec-monad
where
assert-result-and-state f =
do {s ← get-state; assert (f s ≠ {}); (v, t) ← select (f s); set-state t; return v}

abbreviation *state-select* :: ('s × 's) set ⇒ ('e::default, unit, 's) spec-monad
where
state-select r ≡ *assert-result-and-state* (λs. {((), s'). (s, s') ∈ r})

definition *condition* ::
('s ⇒ bool) ⇒ ('e::default, 'a, 's) spec-monad ⇒ ('e, 'a, 's) spec-monad ⇒
('e, 'a, 's) spec-monad
where
condition C T F =
bind get-state (λs. if C s then T else F)

notation (output)
condition ((condition (-)// (-)// (-)) [1000,1000,1000] 999)

definition *when* :: bool ⇒ ('e::default, unit, 's) spec-monad ⇒ ('e, unit, 's) spec-monad
where *when* c f ≡ *condition* (λ-. c) f skip

abbreviation *unless* :: bool ⇒ ('e::default, unit, 's) spec-monad ⇒ ('e, unit, 's) spec-monad
where *unless* c ≡ *when* (¬c)

definition *on-exit'* ::
('e::default, 'a, 's) spec-monad ⇒ ('e, unit, 's) spec-monad ⇒ ('e, 'a, 's) spec-monad
where
on-exit' f c ≡
bind-exception-or-result f (λr. do { c; yield r })

definition *on-exit* ::
('e::default, 'a, 's) spec-monad ⇒ ('s × 's) set ⇒ ('e, 'a, 's) spec-monad

where

$on\text{-}exit\ f\ cleanup \equiv on\text{-}exit'\ f\ (state\text{-}select\ cleanup)$

abbreviation $guard\text{-}on\text{-}exit ::$

$('e::default, 'a, 's)\ spec\text{-}monad \Rightarrow ('s \Rightarrow bool) \Rightarrow ('s \times 's)\ set \Rightarrow ('e, 'a, 's)\ spec\text{-}monad$

where

$guard\text{-}on\text{-}exit\ f\ grd\ cleanup \equiv on\text{-}exit'\ f\ (bind\ (guard\ grd)\ (\lambda\cdot. state\text{-}select\ cleanup))$

abbreviation $assume\text{-}on\text{-}exit ::$

$('e::default, 'a, 's)\ spec\text{-}monad \Rightarrow ('s \Rightarrow bool) \Rightarrow ('s \times 's)\ set \Rightarrow ('e, 'a, 's)\ spec\text{-}monad$

where

$assume\text{-}on\text{-}exit\ f\ grd\ cleanup \equiv on\text{-}exit'\ f\ (bind\ (assuming\ grd)\ (\lambda\cdot. state\text{-}select\ cleanup))$

lift-definition $run\text{-}bind ::$

$('e::default, 'a, 't)\ spec\text{-}monad \Rightarrow 't \Rightarrow ((('e, 'a)\ exception\text{-}or\text{-}result \Rightarrow 't \Rightarrow ('f::default, 'b, 's)\ spec\text{-}monad) \Rightarrow ('f::default, 'b, 's)\ spec\text{-}monad)$

is $\lambda f t g s. bind\text{-}post\text{-}state\ (f\ t)\ (\lambda(r, t). g\ r\ t\ s)$.

— $run\text{-}bind$ might be a more canonical building block compared to $lift\text{-}state$.

See subsection on $run\text{-}bind$ below.

type-synonym $('e, 'a, 's)\ predicate = ('e, 'a)\ exception\text{-}or\text{-}result \Rightarrow 's \Rightarrow bool$

lift-definition $runs\text{-}to ::$

$('e::default, 'a, 's)\ spec\text{-}monad \Rightarrow 's \Rightarrow ('e, 'a, 's)\ predicate \Rightarrow bool$
(- / · - ¶ - ¶ [61, 1000, 0] 30) — syntax $\text{-}do\text{-}block$ has 62

is $\lambda f s Q. holds\text{-}post\text{-}state\ (\lambda(r, t). Q\ r\ t)\ (f\ s)$.

lift-definition $runs\text{-}to\text{-}partial ::$

$('e::default, 'a, 's)\ spec\text{-}monad \Rightarrow 's \Rightarrow ('e, 'a, 's)\ predicate \Rightarrow bool$
(- / · - ?¶ - ¶ [61, 1000, 0] 30)

is $\lambda f s Q. holds\text{-}partial\text{-}post\text{-}state\ (\lambda(r, t). Q\ r\ t)\ (f\ s)$.

lift-definition $refines ::$

$('e::default, 'a, 's)\ spec\text{-}monad \Rightarrow ('f::default, 'b, 't)\ spec\text{-}monad \Rightarrow 's \Rightarrow 't \Rightarrow (((('e, 'a)\ exception\text{-}or\text{-}result \times 's) \Rightarrow (('f, 'b)\ exception\text{-}or\text{-}result \times 't) \Rightarrow bool) \Rightarrow bool)$

is $\lambda f g s t R. sim\text{-}post\text{-}state\ R\ (f\ s)\ (g\ t)$.

lift-definition $rel\text{-}spec ::$

$('e::default, 'a, 's)\ spec\text{-}monad \Rightarrow ('f::default, 'b, 't)\ spec\text{-}monad \Rightarrow 's \Rightarrow 't \Rightarrow (((('e, 'a)\ exception\text{-}or\text{-}result \times 's) \Rightarrow (('f, 'b)\ exception\text{-}or\text{-}result \times 't) \Rightarrow bool) \Rightarrow bool)$

is

$\lambda f g s t R. rel\text{-}post\text{-}state\ R\ (f\ s)\ (g\ t)$.

definition *rel-spec-monad* ::

$(\text{'s} \Rightarrow \text{'t} \Rightarrow \text{bool}) \Rightarrow ((\text{'e}, \text{'a}) \text{ exception-or-result} \Rightarrow (\text{'f}, \text{'b}) \text{ exception-or-result} \Rightarrow \text{bool}) \Rightarrow$

$(\text{'e}::\text{default}, \text{'a}, \text{'s}) \text{ spec-monad} \Rightarrow (\text{'f}::\text{default}, \text{'b}, \text{'t}) \text{ spec-monad} \Rightarrow \text{bool}$

where

$\text{rel-spec-monad } R \ Q \ f \ g =$

$(\forall s \ t. R \ s \ t \longrightarrow \text{rel-post-state } (\text{rel-prod } Q \ R) (\text{run } f \ s) (\text{run } g \ t))$

lift-definition *always-progress* :: $(\text{'e}::\text{default}, \text{'a}, \text{'s}) \text{ spec-monad} \Rightarrow \text{bool}$ **is**

$\lambda p. \forall s. p \ s \neq \text{bot} .$

named-theorems *run-spec-monad* *Simplification rules to run a Spec monad*

named-theorems *runs-to-iff* *Equivalence theorems for runs to predicate*

named-theorems *always-progress-intros* *intro rules for always-progress predicate*

lemma *runs-to-partialI*: $(\bigwedge r \ x. P \ r \ x) \Longrightarrow f \cdot s \ ?\{ P \}$

by $(\text{cases run } f \ s) (\text{auto simp: runs-to-partial.rep-eq})$

lemma *runs-to-partial-True*: $f \cdot s \ ?\{ \lambda r \ s. \text{True} \}$

by $(\text{simp add: runs-to-partialI})$

lemma *runs-to-partial-conj*:

$f \cdot s \ ?\{ P \} \Longrightarrow f \cdot s \ ?\{ Q \} \Longrightarrow f \cdot s \ ?\{ \lambda r \ s. P \ r \ s \wedge Q \ r \ s \}$

by $(\text{cases run } f \ s) (\text{auto simp: runs-to-partial.rep-eq})$

lemma *refines-iff'*:

$\text{refines } f \ g \ s \ t \ R \longleftrightarrow (\forall P. g \cdot t \ \{ \lambda r \ s. \forall p \ t. R \ (p, t) \ (r, s) \longrightarrow P \ p \ t \} \longrightarrow f \cdot s \ \{ P \})$

apply *transfer*

apply $(\text{simp add: sim-post-state-iff le-fun-def split-beta' all-comm[where 'a='a]})$

apply $(\text{rule all-cong-map[where f'=\lambda P (x, y). P x y and f=\lambda P x y. P (x, y)])}$

apply *auto*

done

lemma *refines-weaken*:

$\text{refines } f \ g \ s \ t \ R \Longrightarrow (\bigwedge r \ s \ q \ t. R \ (r, s) \ (q, t) \Longrightarrow Q \ (r, s) \ (q, t)) \Longrightarrow \text{refines } f \ g \ s \ t \ Q$

by *transfer* $(\text{auto intro: sim-post-state-weaken})$

lemma *refines-mono*:

$(\bigwedge r \ s \ q \ t. R \ (r, s) \ (q, t) \Longrightarrow Q \ (r, s) \ (q, t)) \Longrightarrow \text{refines } f \ g \ s \ t \ R \Longrightarrow \text{refines } f \ g \ s \ t \ Q$

by $(\text{rule refines-weaken})$

lemma *refines-refl*: $(\bigwedge r \ t. R \ (r, t) \ (r, t)) \Longrightarrow \text{refines } f \ f \ s \ s \ R$

by *transfer* $(\text{auto intro: sim-post-state-refl})$

lemma *refines-trans*:

$\text{refines } f \ g \ s \ t \ R \Longrightarrow \text{refines } g \ h \ t \ u \ Q \Longrightarrow$

$(\bigwedge r s p t q u. R (r, s) (p, t) \implies Q (p, t) (q, u) \implies S (r, s) (q, u)) \implies$
 $\text{refines } f h s u S$
by *transfer (auto intro: sim-post-state-trans)*

lemma *refines-trans'*: $\text{refines } f g s t1 R \implies \text{refines } g h t1 t2 Q \implies \text{refines } f h s t2$
 $(R \text{ OO } Q)$
by *(rule refines-trans; assumption?) auto*

lemma *refines-strengthen*:
 $\text{refines } f g s t R \implies f \cdot s \text{ ?}\{ F \} \implies g \cdot t \text{ ?}\{ G \} \implies$
 $(\bigwedge x s y t. R (x, s) (y, t) \implies F x s \implies G y t \implies Q (x, s) (y, t)) \implies$
 $\text{refines } f g s t Q$
by *transfer (fastforce elim!: sim-post-state.cases simp: sim-set-def split-beta')*

lemma *refines-strengthen1*:
 $\text{refines } f g s t R \implies f \cdot s \text{ ?}\{ F \} \implies$
 $(\bigwedge x s y t. R (x, s) (y, t) \implies F x s \implies Q (x, s) (y, t)) \implies$
 $\text{refines } f g s t Q$
by *(rule refines-strengthen[OF - - runs-to-partial-True]; assumption?)*

lemma *refines-strengthen2*:
 $\text{refines } f g s t R \implies g \cdot t \text{ ?}\{ G \} \implies$
 $(\bigwedge x s y t. R (x, s) (y, t) \implies G y t \implies Q (x, s) (y, t)) \implies$
 $\text{refines } f g s t Q$
by *(rule refines-strengthen[OF - runs-to-partial-True]; assumption?)*

lemma *refines-cong-cases*:
assumes $\bigwedge e s e' s'. e \neq \text{default} \implies e' \neq \text{default} \implies$
 $R (\text{Exception } e, s) (\text{Exception } e', s') \longleftrightarrow Q (\text{Exception } e, s) (\text{Exception } e', s')$
assumes $\bigwedge e s x' s'. e \neq \text{default} \implies$
 $R (\text{Exception } e, s) (\text{Result } x', s') \longleftrightarrow Q (\text{Exception } e, s) (\text{Result } x', s')$
assumes $\bigwedge x s e' s'. e' \neq \text{default} \implies$
 $R (\text{Result } x, s) (\text{Exception } e', s') \longleftrightarrow Q (\text{Result } x, s) (\text{Exception } e', s')$
assumes $\bigwedge x s x' s'. R (\text{Result } x, s) (\text{Result } x', s') \longleftrightarrow Q (\text{Result } x, s) (\text{Result } x', s')$
shows $\text{refines } f f' s s' R \longleftrightarrow \text{refines } f f' s s' Q$
apply *(clarsimp intro!: arg-cong[where f=refines f f' s s'] simp: fun-eq-iff del: iffI)*
subgoal for $v s v' s'$ **by** *(cases v; cases v'; simp add: assms)*
done

lemma *runs-to-partial-weaken[runs-to-vcg-weaken]*:
 $f \cdot s \text{ ?}\{ Q \} \implies (\bigwedge r t. Q r t \implies Q' r t) \implies f \cdot s \text{ ?}\{ Q' \}$
by *transfer (auto intro: holds-partial-post-state-weaken)*

lemma *runs-to-weaken[runs-to-vcg-weaken]*: $f \cdot s \{ Q \} \implies (\bigwedge r t. Q r t \implies Q' r t) \implies f \cdot s \{ Q' \}$
by *transfer (auto intro: holds-post-state-weaken)*

lemma *runs-to-partial-imp-runs-to-partial*[*mono*]:
 $(\bigwedge a s. P a s \longrightarrow Q a s) \Longrightarrow f \cdot s \text{ ?}\{ P \} \longrightarrow f \cdot s \text{ ?}\{ Q \}$
using *runs-to-partial-weaken*[*of f s P Q*] **by** *auto*

lemma *runs-to-imp-runs-to*[*mono*]:
 $(\bigwedge a s. P a s \longrightarrow Q a s) \Longrightarrow f \cdot s \{ P \} \longrightarrow f \cdot s \{ Q \}$
using *runs-to-weaken*[*of f s P Q*] **by** *auto*

lemma *refines-imp-refines*[*mono*]:
 $(\bigwedge a s. P a s \longrightarrow Q a s) \Longrightarrow \text{refines } f g s t P \longrightarrow \text{refines } f g s t Q$
using *refines-weaken*[*of f g s t P Q*] **by** *auto*

lemma *runs-to-cong-cases*:
assumes $\bigwedge e s. e \neq \text{default} \Longrightarrow P (\text{Exception } e) s \longleftrightarrow Q (\text{Exception } e) s$
assumes $\bigwedge x s. P (\text{Result } x) s \longleftrightarrow Q (\text{Result } x) s$
shows $f \cdot s \{ P \} \longleftrightarrow f \cdot s \{ Q \}$
apply (*clarsimp intro!*: *arg-cong*[**where** *f=runs-to -*] *simp: fun-eq-iff del: iffI*)
subgoal for *v s* **by** (*cases v*; *simp add: assms*)
done

lemma *le-spec-monad-le-refines-iff*: $f \leq g \longleftrightarrow (\forall s. \text{refines } f g s s (=))$
by *transfer* (*simp add: le-fun-def sim-post-state-eq-iff-le*)

lemma *spec-monad-le-iff*: $f \leq g \longleftrightarrow (\forall P (s::'a). g \cdot s \{ P \} \longrightarrow f \cdot s \{ P \})$
by (*simp add: le-spec-monad-le-refines-iff refines-iff' all-comm*[**where** *'a='a*])

lemma *spec-monad-eq-iff*: $f = g \longleftrightarrow (\forall P s. f \cdot s \{ P \} \longleftrightarrow g \cdot s \{ P \})$
by (*simp add: order-eq-iff spec-monad-le-iff*)

lemma *spec-monad-eqI*: $(\bigwedge P s. f \cdot s \{ P \} \longleftrightarrow g \cdot s \{ P \}) \Longrightarrow f = g$
by (*simp add: spec-monad-eq-iff*)

lemma *runs-to-Sup-iff*: $(\text{Sup } X) \cdot s \{ P \} \longleftrightarrow (\forall x \in X. x \cdot s \{ P \})$
by *transfer simp*

lemma *runs-to-lfp*:
assumes *f: mono f*
assumes *x-s: P x s*
assumes *: $\bigwedge p. (\bigwedge x s. P x s \Longrightarrow p x \cdot s \{ R \}) \Longrightarrow (\bigwedge x s. P x s \Longrightarrow f p x \cdot s \{ R \})$
shows *lfp f x \cdot s \{ R \}*
proof –
have $\forall x s. P x s \longrightarrow \text{lfp } f x \cdot s \{ R \}$
apply (*rule lfp-ordinal-induct*[*OF f*])
subgoal using * **by** *blast*
subgoal by (*simp add: runs-to-Sup-iff*)
done
with *x-s* **show** *?thesis* **by** *auto*
qed

lemma *runs-to-partial-alt*:
 $f \cdot s \text{ ?}\{ Q \} \longleftrightarrow \text{run } f \text{ } s = \top \vee f \cdot s \{ Q \}$
by (*cases run f s; simp add: runs-to.rep-eq runs-to-partial.rep-eq top-post-state-def*)

lemma *runs-to-of-runs-to-partial-runs-to'*:
 $f \cdot s \{ P \} \Longrightarrow f \cdot s \text{ ?}\{ Q \} \Longrightarrow f \cdot s \{ Q \}$
by (*auto simp: runs-to-partial-alt runs-to.rep-eq*)

lemma *runs-to-partial-of-runs-to*:
 $f \cdot s \{ Q \} \Longrightarrow f \cdot s \text{ ?}\{ Q \}$
by (*auto simp: runs-to-partial-alt*)

lemma *runs-to-partial-tivial[simp]*: $f \cdot s \text{ ?}\{ \lambda -. \text{True} \}$
by (*cases run f s; simp add: runs-to-partial.rep-eq*)

lemma *le-spec-monad-le-run-iff*: $f \leq g \longleftrightarrow (\forall s. \text{run } f \text{ } s \leq \text{run } g \text{ } s)$
by (*simp add: le-fun-def less-eq-spec-monad.rep-eq*)

lemma *refines-le-run-trans*: $\text{refines } f \text{ } g \text{ } s \text{ } s1 \text{ } R \Longrightarrow \text{run } g \text{ } s1 \leq \text{run } h \text{ } s2 \Longrightarrow \text{refines } f \text{ } h \text{ } s \text{ } s2 \text{ } R$
by *transfer (rule sim-post-state-le2)*

lemma *le-run-refines-trans*: $\text{run } f \text{ } s \leq \text{run } g \text{ } s1 \Longrightarrow \text{refines } g \text{ } h \text{ } s1 \text{ } s2 \text{ } R \Longrightarrow \text{refines } f \text{ } h \text{ } s \text{ } s2 \text{ } R$
by *transfer (rule sim-post-state-le1)*

lemma *refines-le-trans*: $\text{refines } f \text{ } g \text{ } s \text{ } t \text{ } R \Longrightarrow g \leq h \Longrightarrow \text{refines } f \text{ } h \text{ } s \text{ } t \text{ } R$
by (*auto simp: le-spec-monad-le-run-iff refines-le-run-trans*)

lemma *le-refines-trans*: $f \leq g \Longrightarrow \text{refines } g \text{ } h \text{ } s \text{ } t \text{ } R \Longrightarrow \text{refines } f \text{ } h \text{ } s \text{ } t \text{ } R$
by (*auto simp: le-spec-monad-le-run-iff intro: le-run-refines-trans*)

lemma *le-run-refines-iff*: $\text{run } f \text{ } s \leq \text{run } g \text{ } t \longleftrightarrow \text{refines } f \text{ } g \text{ } s \text{ } t (=)$
by *transfer (simp add: sim-post-state-eq-iff-le)*

lemma *refines-top[simp]*: $\text{refines } f \text{ } \top \text{ } s \text{ } t \text{ } R$
unfolding *top-spec-monad.rep-eq refines.rep-eq* **by** *simp*

lemma *refines-Sup1*: $\text{refines } (\text{Sup } F) \text{ } g \text{ } s \text{ } s' \text{ } R \longleftrightarrow (\forall f \in F. \text{refines } f \text{ } g \text{ } s \text{ } s' \text{ } R)$
by (*simp add: refines.rep-eq Sup-spec-monad.rep-eq image-image sim-post-state-Sup1*)

lemma *monotone-le-iff-refines*:
 $\text{monotone } R (\leq) F \longleftrightarrow (\forall x \ y. R \ x \ y \longrightarrow (\forall s. \text{refines } (F \ x) \ (F \ y) \ s \ s (=)))$
by (*auto simp: monotone-def le-fun-def le-run-refines-iff[symmetric] le-spec-monad-le-run-iff*)

lemma *refines-iff-runs-to*:
 $\text{refines } f \text{ } g \text{ } s \text{ } t \text{ } R \longleftrightarrow (\forall P. g \cdot t \{ \lambda r \ t'. \forall p \ s'. R \ (p, s') \ (r, t') \longrightarrow P \ p \ s' \} \longrightarrow f \cdot s \{ P \})$

by *transfer* (*auto simp: sim-post-state-iff split-beta'*)

lemma *runs-to-refines-weaken'*:

refines $f g s t R \implies g \cdot t \{ \lambda r t'. \forall p s'. R (p, s') (r, t') \longrightarrow P p s' \} \implies f \cdot s \{ P \}$

by (*simp add: refines-iff-runs-to*)

lemma *runs-to-refines-weaken*: *refines* $f f' s t (=) \implies f' \cdot t \{ P \} \implies f \cdot s \{ P \}$

by (*auto simp: runs-to-refines-weaken'*)

lemma *refinesD-runs-to*:

assumes $f \cdot g$: *refines* $f g s t R$ **and** g : $g \cdot t \{ P \}$

shows $f \cdot s \{ \lambda r t. \exists r' t'. R (r, t) (r', t') \wedge P r' t' \}$

by (*rule runs-to-refines-weaken'[OF f-g]*) (*auto intro: runs-to-weaken[OF g]*)

lemma *rel-spec-iff-refines*:

rel-spec $f g s t R \longleftrightarrow \text{refines } f g s t R \wedge \text{refines } g f t s (R^{-1-1})$

unfolding *rel-spec-def* **by** *transfer* (*simp add: rel-post-state-eq-sim-post-state*)

lemma *rel-spec-symm*: *rel-spec* $f g s t R \longleftrightarrow \text{rel-spec } g f t s R^{-1-1}$

by (*simp add: rel-spec-iff-refines ac-simps*)

lemma *rel-specD-refines1*: *rel-spec* $f g s t R \implies \text{refines } f g s t R$

by (*simp add: rel-spec-iff-refines*)

lemma *rel-spec-mono*: $P \leq Q \implies \text{rel-spec } f g s t P \implies \text{rel-spec } f g s t Q$

using *rel-post-state-mono* **by** (*auto simp: rel-spec-def*)

lemma *rel-spec-eq*: *rel-spec* $f g s t (=) \longleftrightarrow \text{run } f s = \text{run } g t$

by (*simp add: rel-spec-def rel-post-state-eq*)

lemma *rel-spec-eq-conv*: $(\forall s. \text{rel-spec } f g s s (=)) \longleftrightarrow f = g$

using *rel-spec-eq spec-monad-ext* **by** *metis*

lemma *rel-spec-eqD*: $(\bigwedge s. \text{rel-spec } f g s s (=)) \implies f = g$

using *rel-spec-eq-conv* **by** *metis*

lemma *rel-spec-refl'*: $f \cdot s \{ \lambda r s. R (r, s) (r, s) \} \implies \text{rel-spec } f f s s R$

by *transfer* (*simp add: rel-post-state-refl' split-beta'*)

lemma *rel-spec-refl*: $(\bigwedge r s. R (r, s) (r, s)) \implies \text{rel-spec } f f s s R$

by (*rule rel-spec-mono[OF - rel-spec-eq[THEN iffD2]]*) *auto*

lemma *refines-same-runs-to-partialI*:

$f \cdot s \{ \lambda r s'. R (r, s') (r, s') \} \implies \text{refines } f f s s R$

by *transfer*

(*auto intro!: sim-post-state-refl' simp: split-beta'*)

lemma *refines-same-runs-toI*:

$f \cdot s \{\!\{ \lambda r s'. R (r, s') (r, s') \}\!\} \Longrightarrow \text{refines } f f s s R$
by (*rule refines-same-runs-to-partialI* [*OF runs-to-partial-of-runs-to*])

lemma *always-progress-case-prod* [*always-progress-intros*]:

$\text{always-progress } (f (fst p) (snd p)) \Longrightarrow \text{always-progress } (\text{case-prod } f p)$
by (*simp add: split-beta*)

lemma *refines-runs-to-partial-fuse*:

$\text{refines } f f' s s' Q \Longrightarrow f \cdot s \{P\} \Longrightarrow$
 $\text{refines } f f' s s' (\lambda(r,t) (r',t'). Q (r,t) (r',t') \wedge P r t)$
by *transfer (auto elim!: sim-post-state.cases simp: sim-set-def split-beta')*

lemma *runs-to-refines*:

$f' \cdot s' \{P\} \Longrightarrow \text{refines } f f' s s' Q \Longrightarrow (\bigwedge x s y t. P x s \Longrightarrow Q (y, t) (x, s) \Longrightarrow$
 $R y t) \Longrightarrow$
 $f \cdot s \{R\}$
by *transfer (force elim!: sim-post-state.cases simp: sim-set-def split-beta')*

lemma *runs-to-partial-subst-Res*: $f \cdot s \{ \lambda r. P (\text{the-Result } r) \} \longleftrightarrow f \cdot s \{ \lambda Res r. P r \}$

by (*intro arg-cong* [**where** $f = \lambda P. f \cdot s \{P\}$]) (*auto simp: fun-eq-iff the-Result-def*)

lemma *runs-to-subst-Res*: $f \cdot s \{ \lambda r. P (\text{the-Result } r) \} \longleftrightarrow f \cdot s \{ \lambda Res r. P r \}$

by (*intro arg-cong* [**where** $f = \lambda P. f \cdot s \{P\}$]) (*auto simp: fun-eq-iff the-Result-def*)

lemma *runs-to-Res* [*simp*]: $f \cdot s \{ \lambda r t. \forall v. r = \text{Result } v \longrightarrow P v t \} \longleftrightarrow f \cdot s \{ \lambda Res v t. P v t \}$

by (*intro arg-cong* [**where** $f = \lambda P. f \cdot s \{P\}$]) (*auto simp: fun-eq-iff the-Result-def*)

lemma *runs-to-partial-Res* [*simp*]:

$f \cdot s \{ \lambda r t. \forall v. r = \text{Result } v \longrightarrow P v t \} \longleftrightarrow f \cdot s \{ \lambda Res v t. P v t \}$
by (*intro arg-cong* [**where** $f = \lambda P. f \cdot s \{P\}$]) (*auto simp: fun-eq-iff the-Result-def*)

lemma *rel-spec-runs-to*:

assumes $f: f \cdot s \{P\}$ *always-progress* f
and $g: g \cdot t \{Q\}$ *always-progress* g
and $P: (\bigwedge r s' p t'. P r s' \Longrightarrow Q p t' \Longrightarrow R (r, s') (p, t'))$
shows *rel-spec* $f g s t R$
using *assms unfolding rel-spec-def always-progress.rep-eq runs-to.rep-eq*
by (*cases run f s; cases run g t; simp add: rel-set-def split-beta'*)
(metis all-not-in-conv bot-post-state-def prod.exhaust-sel)

lemma *runs-to-res-independent-res*: $f \cdot s \{ \lambda -. P \} \longleftrightarrow f \cdot s \{ \lambda Res -. P \}$

by (*rule runs-to-subst-Res*)

lemma *lift-state-spec-monad-eq* [*simp*]: *lift-state* $(=) p = p$

by *transfer (simp add: prod.rel-eq fun-eq-iff rel-post-state-eq)*

lemma *rel-post-state-Sup*:
 $rel\text{-}set (\lambda x y. rel\text{-}post\text{-}state Q (f x) (g y)) X Y \implies rel\text{-}post\text{-}state Q (\bigsqcup x \in X. f x)$
 $(\bigsqcup x \in Y. g x)$
by (*force simp: rel-post-state-eq-sim-post-state rel-set-def intro!: sim-post-state-Sup*)

lemma *rel-set-Result-image-iff*:
 $rel\text{-}set (rel\text{-}prod (\lambda Res v1 Res v2. (P v1 v2)) R)$
 $((\lambda x. case x of (v, s) \implies (Result v, s)) ' Vals1)$
 $((\lambda x. case x of (v, s) \implies (Result v, s)) ' Vals2)$
 \longleftrightarrow
 $rel\text{-}set (rel\text{-}prod P R) Vals1 Vals2$
by (*force simp add: rel-set-def*)

lemma *rel-post-state-converse-iff*:
 $rel\text{-}post\text{-}state R X Y \longleftrightarrow rel\text{-}post\text{-}state R^{-1-1} Y X$
by (*metis conversesep-conversesep rel-post-state-eq-sim-post-state*)

lemma *runs-to-le-post-state-iff*: $runs\text{-}to f s Q \longleftrightarrow run f s \leq Success \{(r, s). Q r s\}$
by (*cases run f s (auto simp add: runs-to.rep-eq)*)

lemma *runs-to-partial-runs-to-iff*: $runs\text{-}to\text{-}partial f s Q \longleftrightarrow (run f s = Failure \vee runs\text{-}to f s Q)$
by (*cases run f s; simp add: runs-to-partial.rep-eq runs-to.rep-eq*)

lemma *run-runs-to-extensionality*:
 $run f s = run g s \longleftrightarrow (\forall P. f \cdot s \{P\} \longleftrightarrow g \cdot s \{P\})$
apply *transfer*
apply (*simp add: post-state-eq-iff*)
apply (*rule all-cong-map[where f'= $\lambda P (x, y). P x y$ and $f = \lambda P x y. P (x, y)$]*)
apply *auto*
done

lemma *le-spec-monad-runI*: $(\bigwedge s. run f s \leq run g s) \implies f \leq g$
by *transfer (simp add: le-fun-def)*

16.6.1 *rel-spec-monad*

lemma *rel-spec-monad-iff-rel-spec*:
 $rel\text{-}spec\text{-}monad R Q f g \longleftrightarrow (\forall s t. R s t \longrightarrow rel\text{-}spec f g s t (rel\text{-}prod Q R))$
unfolding *rel-spec-monad-def rel-fun-def rel-spec.rep-eq ..*

lemma *rel-spec-monadI*:
 $(\bigwedge s t. R s t \implies rel\text{-}spec f g s t (rel\text{-}prod Q R)) \implies rel\text{-}spec\text{-}monad R Q f g$
by (*auto simp: rel-spec-monad-iff-rel-spec*)

lemma *rel-spec-monadD*:
 $rel\text{-}spec\text{-}monad R Q f g \implies R s t \implies rel\text{-}spec f g s t (rel\text{-}prod Q R)$
by (*auto simp: rel-spec-monad-iff-rel-spec*)

lemma *rel-spec-monad-eq-conv*: *rel-spec-monad* (=) (=) = (=)
unfolding *rel-spec-monad-def* **by** *transfer* (*simp add: fun-eq-iff prod.rel-eq rel-post-state-eq*)

lemma *rel-spec-monad-converse-iff*:
rel-spec-monad R Q f g \longleftrightarrow *rel-spec-monad* R^{-1-1} Q^{-1-1} g f
using *rel-post-state-converse-iff*
by (*metis conversesep.cases conversesep.intros prod.rel-conversep rel-spec-monad-def*)

lemma *rel-spec-monad-iff-refines*:
rel-spec-monad S R f g \longleftrightarrow
 $(\forall s t. S s t \longrightarrow (\text{refines } f g s t (\text{rel-prod } R S) \wedge \text{refines } g f t s (\text{rel-prod } R^{-1-1} S^{-1-1})))$
using *rel-spec-iff-refines rel-spec-monad-iff-rel-spec*
by (*metis prod.rel-conversep*)

lemma *rel-spec-monad-rel-exception-or-resultI*:
rel-spec-monad R (*rel-exception-or-result* (=) (=)) f g \implies *rel-spec-monad* R (=)
 f g
by (*simp add: rel-exception-or-result-sum-eq*)

lemma *runs-to-partial-runs-to-fuse*:
assumes *part*: $f \cdot s \ ?\{Q\}$
assumes *tot*: $f \cdot s \ \{P\}$
shows $f \cdot s \ \{\lambda r t. Q r t \wedge P r t\}$
using *assms*
apply (*cases run f s*)
apply (*auto simp add: runs-to-def runs-to-partial-def*)
done

16.7 VCG basic setup

lemma *runs-to-cong-pred-only*:
 $P = Q \implies (p \cdot s \ \{P\}) \longleftrightarrow (p \cdot s \ \{Q\})$
by *simp*

lemma *runs-to-cong-state-only[runs-to-vcg-cong-state-only]*:
 $s = t \implies (p \cdot s \ \{Q\}) \longleftrightarrow (p \cdot t \ \{Q\})$
by *simp*

lemma *runs-to-partial-cong-state-only[runs-to-vcg-cong-state-only]*:
 $s = t \implies (p \cdot s \ ?\{Q\}) \longleftrightarrow (p \cdot t \ ?\{Q\})$
by *simp*

lemma *runs-to-cong-program-only[runs-to-vcg-cong-program-only]*:
 $p = q \implies (p \cdot s \ \{Q\}) \longleftrightarrow (q \cdot s \ \{Q\})$
by *simp*

lemma *runs-to-partial-cong-program-only[runs-to-vcg-cong-program-only]*:

$p = q \implies (p \cdot s \text{ ?}\{\!\! \} Q \{\!\! \}) \longleftrightarrow (q \cdot s \text{ ?}\{\!\! \} Q \{\!\! \})$
by *simp*

lemma *runs-to-case-conv*[*simp*]:
 $((\text{case } (a, b) \text{ of } (x, y) \Rightarrow f x y) \cdot s \{\!\! \} Q \{\!\! \}) \longleftrightarrow ((f a b) \cdot s \{\!\! \} Q \{\!\! \})$
by *simp*

lemma *runs-to-partial-case-conv*[*simp*]:
 $((\text{case } (a, b) \text{ of } (x, y) \Rightarrow f x y) \cdot s \text{ ?}\{\!\! \} Q \{\!\! \}) \longleftrightarrow ((f a b) \cdot s \text{ ?}\{\!\! \} Q \{\!\! \})$
by *simp*

lemma *always-progress-prod-case*[*always-progress-intros*]:
 $\text{always-progress } (f \text{ (fst } p) \text{ (snd } p)) \implies \text{always-progress } (\text{case } p \text{ of } (x, y) \Rightarrow f x y)$
by (*auto simp add: always-progress-def split: prod.splits*)

lemma *runs-to-conj*:
 $(f \cdot s \{\!\! \} \lambda r s. P r s \wedge Q r s \{\!\! \}) \longleftrightarrow (f \cdot s \{\!\! \} \lambda r s. P r s \{\!\! \}) \wedge (f \cdot s \{\!\! \} \lambda r s. Q r s \{\!\! \})$
by (*cases run f s*) (*auto simp: runs-to.rep-eq*)

lemma *runs-to-all*:
 $(f \cdot s \{\!\! \} \lambda r s. \forall x. P x r s \{\!\! \}) \longleftrightarrow (\forall x. f \cdot s \{\!\! \} \lambda r s. P x r s \{\!\! \})$
by (*cases run f s*) (*auto simp: runs-to.rep-eq*)

lemma *runs-to-imp-const*:
 $(f \cdot s \{\!\! \} \lambda r s. P r s \longrightarrow Q \{\!\! \}) \longleftrightarrow (Q \wedge (f \cdot s \{\!\! \} \lambda -. \text{True } \{\!\! \})) \vee (\neg Q \wedge (f \cdot s \{\!\! \} \lambda r s. \neg P r s \{\!\! \}))$
by (*cases run f s*) (*auto simp: runs-to.rep-eq*)

16.7.1 *res-monad* and *exn-monad* Types

type-synonym *'a val* = (*unit*, *'a*) *exception-or-result*
type-synonym (*'e*, *'a*) *xval* = (*'e option*, *'a*) *exception-or-result*
type-synonym (*'a*, *'s*) *res-monad* = (*unit*, *'a*, *'s*) *spec-monad*
type-synonym (*'e*, *'a*, *'s*) *exn-monad* = (*'e option*, *'a*, *'s*) *spec-monad*

definition *Exn* :: *'e* \Rightarrow (*'e*, *'a*) *xval* **where**
 $\text{Exn } e = \text{Exception } (\text{Some } e)$

definition
 $\text{case-xval} :: ('e \Rightarrow 'a) \Rightarrow ('v \Rightarrow 'a) \Rightarrow ('e, 'v) \text{ xval} \Rightarrow 'a$ **where**
 $\text{case-xval } f g x =$
 (*case* *x* of
 Exception *v* \Rightarrow (*case* *v* of *Some* *e* \Rightarrow *f e* | *None* \Rightarrow *undefined*)
 | *Result* *v* \Rightarrow *g v*)

declare [[*case-translation case-xval Exn Result*]]

inductive *rel-xval*:: (*'e* \Rightarrow *'f* \Rightarrow *bool*) \Rightarrow (*'a* \Rightarrow *'b* \Rightarrow *bool*) \Rightarrow
 (*'e*, *'a*) *xval* \Rightarrow (*'f*, *'b*) *xval* \Rightarrow *bool*

where
Exn: $E\ e\ f \implies \text{rel-xval } E\ R\ (\text{Exn } e)\ (\text{Exn } f) \mid$
Result: $R\ a\ b \implies \text{rel-xval } E\ R\ (\text{Result } a)\ (\text{Result } b)$

definition *map-xval* :: $('e \Rightarrow 'f) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('e, 'a)\ \text{xval} \Rightarrow ('f, 'b)\ \text{xval}$ **where**
map-xval $f\ g\ x \equiv \text{case } x\ \text{of } \text{Exn } e \Rightarrow \text{Exn } (f\ e) \mid \text{Result } v \Rightarrow \text{Result } (g\ v)$

lemma *rel-xval-eq*: $\text{rel-xval } (=)\ (=) = (=)$
apply (*rule ext*)
subgoal for $x\ y$
apply (*cases x*)
apply (*auto simp add: rel-xval.simps default-option-def Exn-def*)
done
done

lemma *rel-xval-rel-exception-or-result-conv*:
 $\text{rel-xval } E\ R = \text{rel-exception-or-result } (\text{rel-map the } OO\ E\ OO\ \text{rel-map Some})\ R$
apply (*rule ext*)
subgoal for $x\ y$
apply (*cases x*)
subgoal
by (*auto simp add: rel-exception-or-result.simps rel-xval.simps Exn-def default-option-def rel-map-def*)
(*metis option.sel rel-map-direct relcomppI*)
subgoal
by (*auto simp add: rel-exception-or-result.simps rel-xval.simps Exn-def default-option-def rel-map-def*)
done
done

lemmas *rel-xval-Exn* = *rel-xval.intros(1)*
lemmas *rel-xval-Result* = *rel-xval.intros(2)*

lemma *case-xval-simps[simp]*:
 $\text{case-xval } f\ g\ (\text{Exn } v) = f\ v$
 $\text{case-xval } f\ g\ (\text{Result } e) = g\ e$
by (*auto simp add: case-xval-def Exn-def*)

lemma *case-exception-or-result-Exn[simp]*:
 $\text{case-exception-or-result } f\ g\ (\text{Exn } x) = f\ (\text{Some } x)$
by (*simp add: Exn-def*)

lemma *xval-split*:
 $P\ (\text{case-xval } f\ g\ r) \iff$
 $(\forall e. r = \text{Exn } e \longrightarrow P\ (f\ e)) \wedge$
 $(\forall v. r = \text{Result } v \longrightarrow P\ (g\ v))$
by (*auto simp add: case-xval-def Exn-def split: exception-or-result-splits option.splits*)

lemma *xval-split-asm*:

$P (case-xval f g r) \longleftrightarrow$
 $\neg ((\exists e. r = Exn e \wedge \neg P (f e)) \vee$
 $(\exists v. r = Result v \wedge \neg P (g v)))$
by (*auto simp add: case-xval-def Exn-def split: exception-or-result-splits option.splits*)

lemmas *xval-splits = xval-split xval-split-asm*

lemma *Exn-eq-Exn[simp]*: $Exn x = Exn y \longleftrightarrow x = y$
by (*simp add: Exn-def*)

lemma *Exn-neq-Result[simp]*: $Exn x = Result e \longleftrightarrow False$
by (*simp add: Exn-def*)

lemma *Result-neq-Exn[simp]*: $Result e = Exn x \longleftrightarrow False$
by (*simp add: Exn-def*)

lemma *Exn-eq-Exception[simp]*:
 $Exn x = Exception a \longleftrightarrow a = Some x$
 $Exception a = Exn x \longleftrightarrow a = Some x$
by (*auto simp: Exn-def*)

lemma *map-exception-or-result-Exn[simp]*:
 $(\bigwedge x. f (Some x) \neq None) \implies map-exception-or-result f g (Exn x) = Exn (the (f (Some x)))$
by (*simp add: Exn-def*)

lemma *case-xval-Exception-Some-simp[simp]*: $(case Exception (Some y) of Exn e \Rightarrow f e \mid Result v \Rightarrow g v) = f y$
by (*metis Exn-def case-xval-simps(1)*)

lemma *rel-xval-simps[simp]*:
 $rel-xval E R (Exn e) (Exn f) = E e f$
 $rel-xval E R (Result v) (Result w) = R v w$
 $rel-xval E R (Exn e) (Result w) = False$
 $rel-xval E R (Result v) (Exn f) = False$
by (*auto simp add: rel-xval.simps*)

lemma *map-xval-simps[simp]*:
 $map-xval f g (Exn e) = Exn (f e)$
 $map-xval f g (Result v) = Result (g v)$
by (*auto simp add: map-xval-def*)

lemma *map-xval-Exn*: $map-xval f g x = Exn y \longleftrightarrow (\exists e. x = Exn e \wedge y = f e)$
by (*auto simp add: map-xval-def split: xval-splits*)

lemma *map-xval-Result*: $map-xval f g x = Result y \longleftrightarrow (\exists v. x = Result v \wedge y = g v)$
by (*auto simp add: map-xval-def split: xval-splits*)

lemma *Result-unit-eq*: $(x:: \text{unit val}) = \text{Result } ()$
by *(cases x) auto*

simproc-setup *Result-unit-eq* $(x::(\text{unit val})) = \langle$
let
fun is-Result-unit **Const** $\langle \text{Result } \langle @\{\text{typ unit}\} \rangle \langle @\{\text{typ unit}\} \rangle$ **for** **Const** $\langle \text{Product-Type.Unity} \rangle$
 $= \text{true}$
| *is-Result-unit* $= \text{false}$
in
K $(K \text{ (fn ct =>$
if *is-Result-unit* (Thm.term-of ct) *then* *NONE*
else *SOME* $(\text{mk-meta-eq } @\{\text{thm Result-unit-eq}\}))$
 end
 \rangle

lemma *ex-val-Result1*:
 $\exists v1. (x::'v1 \text{ val}) = \text{Result } v1$
by *(cases x) auto*

lemma *ex-val-Result2*:
 $\exists v1 v2. (x::('v1 * 'v2) \text{ val}) = \text{Result } (v1, v2)$
by *(cases x) auto*

lemma *ex-val-Result3*:
 $\exists v1 v2 v3. (x::('v1 * 'v2 * 'v3) \text{ val}) = \text{Result } (v1, v2, v3)$
by *(cases x) auto*

lemma *ex-val-Result4*:
 $\exists v1 v2 v3 v4. (x::('v1 * 'v2 * 'v3 * 'v4) \text{ val}) = \text{Result } (v1, v2, v3, v4)$
by *(cases x) auto*

lemma *ex-val-Result5*:
 $\exists v1 v2 v3 v4 v5. (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5) \text{ val}) = \text{Result } (v1, v2, v3, v4, v5)$
by *(cases x) auto*

lemma *ex-val-Result6*:
 $\exists v1 v2 v3 v4 v5 v6. (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5 * 'v6) \text{ val}) = \text{Result } (v1, v2, v3, v4, v5, v6)$
by *(cases x) auto*

lemma *ex-val-Result7*:
 $\exists v1 v2 v3 v4 v5 v6 v7. (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5 * 'v6 * 'v7) \text{ val}) = \text{Result } (v1, v2, v3, v4, v5, v6, v7)$
by *(cases x) auto*

lemma *ex-val-Result8*:
 $\exists v1 v2 v3 v4 v5 v6 v7 v8. (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5 * 'v6 * 'v7 * 'v8) \text{ val}) = \text{Result } (v1, v2, v3, v4, v5, v6, v7, v8)$

by (*cases x*) *auto*

lemma *ex-val-Result9*:

$\exists v1\ v2\ v3\ v4\ v5\ v6\ v7\ v8\ v9. (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5 * 'v6 * 'v7 * 'v8 * 'v9)\ val) = \text{Result } (v1, v2, v3, v4, v5, v6, v7, v8, v9)$

by (*cases x*) *auto*

lemma *ex-val-Result10*:

$\exists v1\ v2\ v3\ v4\ v5\ v6\ v7\ v8\ v9\ v10. (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5 * 'v6 * 'v7 * 'v8 * 'v9 * 'v10)\ val) = \text{Result } (v1, v2, v3, v4, v5, v6, v7, v8, v9, v10)$

by (*cases x*) *auto*

lemma *ex-val-Result11*:

$\exists v1\ v2\ v3\ v4\ v5\ v6\ v7\ v8\ v9\ v10\ v11. (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5 * 'v6 * 'v7 * 'v8 * 'v9 * 'v10 * 'v11)\ val) = \text{Result } (v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11)$

by (*cases x*) *auto*

lemmas *ex-val-Result[simp]* =

ex-val-Result1

ex-val-Result2

ex-val-Result3

ex-val-Result4

ex-val-Result5

ex-val-Result6

ex-val-Result7

ex-val-Result8

ex-val-Result9

ex-val-Result10

ex-val-Result11

lemma *all-val-imp-iff*: $(\forall v. (r::'a\ val) = \text{Result } v \longrightarrow P) \longleftrightarrow P$

by (*cases r*) *auto*

definition *map-exn* :: $('e \Rightarrow 'f) \Rightarrow ('e, 'a)\ xval \Rightarrow ('f, 'a)\ xval$ **where**

map-exn f x =

(*case x of*

Exn e \Rightarrow *Exn (f e)*

| *Result v* \Rightarrow *Result v*)

lemma *map-exn-simps[simp]*:

map-exn f (Exn e) = *Exn (f e)*

map-exn f (Result v) = *Result v*

map-exn f x = *Result v* \longleftrightarrow *x* = *Result v*

by (*auto simp add: map-exn-def split: xval-splits*)

lemma *map-exn-id[simp]*: *map-exn* $(\lambda x. x)$ = $(\lambda x. x)$

by (*auto simp add: map-exn-def split: xval-splits*)

definition *unnest-exn* :: ('e + 'a, 'a) xval ⇒ ('e, 'a) xval **where**
unnest-exn x =
 (case x of
 Exn e ⇒ (case e of *Inl* l ⇒ *Exn* l | *Inr* r ⇒ *Result* r)
 | *Result* v ⇒ *Result* v)

lemma *unnest-exn-simps*[*simp*]:
unnest-exn (*Exn* (*Inl* l)) = *Exn* l
unnest-exn (*Exn* (*Inr* r)) = *Result* r
unnest-exn (*Result* v) = *Result* v
by (*auto simp add: unnest-exn-def split: xval-splits*)

lemma *unnest-exn-eq-simps*[*simp*]:
unnest-exn (*Result* r) = *Result* r' ⟷ r = r'
unnest-exn (*Result* e) ≠ *Exn* e
unnest-exn (*Exn* e) = *Result* r ⟷ e = *Inr* r
unnest-exn (*Exn* e) = *Exn* e' ⟷ e = *Inl* e'
by (*auto simp: unnest-exn-def split: xval-splits sum.splits*)

definition *the-Exn* :: ('e, 'a) xval ⇒ 'e **where**
the-Exn x ≡ (case x of *Exn* e ⇒ e | *Result* - ⇒ *undefined*)

abbreviation *the-Res* :: 'a val ⇒ 'a **where** *the-Res* ≡ *the-Result*

lemma *is-Exception-val*[*simp*]: *is-Exception* (x::'a val) = *False*
by (*auto simp add: is-Exception-def*)

lemma *is-Exception-Exn*[*simp*]: *is-Exception* (*Exn* x)
by (*simp add: Exn-def*)

lemma *is-Result-val*[*simp*]: *is-Result* (v::'a val)
by (*auto simp add: is-Result-def*)

lemma *the-Exception-Exn*[*simp*]: *the-Exception* (*Exn* e) = *Some* e
by (*auto simp add: the-Exception-def*)

lemma *the-Exn-Exn*[*simp*]:
the-Exn (*Exn* e) = e
the-Exn (*Exception* (*Some* e)) = e
by (*auto simp add: the-Exn-def case-xval-def*)

lemma *the-Result-simp*[*simp*]:
the-Result (*Result* v) = v
by (*auto simp add: the-Result-def*)

lemma *Result-the-Result-val*[*simp*]: *Result* (*the-Result* (x::'a val)) = x
by (*auto simp add: the-Result-def*)

lemma *rel-exception-or-result-val-apply*:
fixes $x :: 'a \text{ val}$
fixes $y :: 'b \text{ val}$
shows $\text{rel-exception-or-result } E \ R \ x \ y \longleftrightarrow \text{rel-exception-or-result } E' \ R \ x \ y$
by (*auto simp add: rel-exception-or-result.simps*)

lemma *rel-exception-or-result-val*:
shows $((\text{rel-exception-or-result } E \ R) :: 'a \text{ val} \Rightarrow 'b \text{ val} \Rightarrow \text{bool}) = \text{rel-exception-or-result } E' \ R$
apply (*rule ext*)
apply (*auto simp add: rel-exception-or-result.simps*)
done

lemma *rel-exception-or-result-Res-val*:
 $(\lambda \text{Res } x \ \text{Res } y. \ R \ x \ y) = \text{rel-exception-or-result } (\lambda - . \ \text{False}) \ R$
apply (*rule ext*)
apply *auto*
done

definition *unite*: $('a, 'a) \ xval \Rightarrow 'a \text{ val}$ **where**
 $\text{unite } x = (\text{case } x \text{ of } \text{Exn } v \Rightarrow \text{Result } v \mid \text{Result } v \Rightarrow \text{Result } v)$

lemma *unite-simps*[*simp*]:
 $\text{unite } (\text{Exn } v) = \text{Result } v$
 $\text{unite } (\text{Result } v) = \text{Result } v$
by (*auto simp add: unite-def*)

lemma *val-exhaust*: $(\bigwedge v. (x :: 'a \text{ val}) = \text{Result } v \Longrightarrow P) \Longrightarrow P$
by (*cases x auto*)

lemma *val-iff*: $P (x :: 'a \text{ val}) \longleftrightarrow (\exists v. x = \text{Result } v \wedge P (\text{Result } v))$
by (*cases x auto*)

lemma *val-nchotomy*: $\forall (x :: 'a \text{ val}). \exists v. x = \text{Result } v$
by (*meson val-exhaust*)

lemma $(\bigwedge x :: 'a \text{ val}. \text{PROP } P \ x) \Longrightarrow \text{PROP } P (\text{Result } (v :: 'a))$
by (*rule meta-spec*)

lemma *val-Result-the-Result-conv*: $(x :: 'a \text{ val}) = \text{Result } (\text{the-Result } x)$
by (*cases x (auto simp add: the-Result-def)*)

lemma *split-val-all*[*simp*]: $(\bigwedge x :: 'a \text{ val}. \text{PROP } P \ x) \equiv (\bigwedge v :: 'a. \text{PROP } P (\text{Result } v))$
proof
fix $v :: 'a$
assume $\bigwedge x :: 'a \text{ val}. \text{PROP } P \ x$
then show $\text{PROP } P (\text{Result } v)$.

```

next
  fix  $x::'a$  val
  assume  $\bigwedge v::'a. \text{PROP } P (\text{Result } v)$ 
  hence  $\text{PROP } P (\text{Result } (\text{the-Result } x))$  .
  thus  $\text{PROP } P x$ 
    apply (subst val-Result-the-Result-conv)
    apply (assumption)
  done
qed

lemma split-val-Ex[simp]:  $(\exists (x::'a \text{ val}). P x) \longleftrightarrow (\exists (v::'a). P (\text{Result } v))$ 
  by (auto)

lemma split-val-All[simp]:  $(\forall (x::'a \text{ val}). P x) \longleftrightarrow (\forall (v::'a). P (\text{Result } v))$ 
  by (auto)

definition to-xval ::  $('e + 'a) \Rightarrow ('e, 'a) \text{ xval}$  where
  to-xval  $x = (\text{case } x \text{ of } \text{Inl } e \Rightarrow \text{Exn } e \mid \text{Inr } v \Rightarrow \text{Result } v)$ 

lemma to-xval-simps[simp]:
  to-xval (Inl  $e$ ) = Exn  $e$ 
  to-xval (Inr  $v$ ) = Result  $v$ 
  by (auto simp add: to-xval-def)

lemma to-xval-Result-iff[simp]: to-xval  $x = \text{Result } v \longleftrightarrow x = \text{Inr } v$ 
  apply (cases  $x$ )
  apply (auto simp add: to-xval-def default-option-def)
  done

lemma to-xval-Exn-iff[simp]:
  to-xval  $x = \text{Exn } v \longleftrightarrow$ 
     $(x = \text{Inl } v)$ 
  apply (cases  $x$ )
  apply (auto simp add: to-xval-def default-option-def Exn-def)
  done

lemma to-xval-Exception-iff[simp]:
  to-xval  $x = \text{Exception } v \longleftrightarrow$ 
     $((v = \text{None} \wedge x = \text{Inr } \text{undefined}) \vee (\exists e. v = \text{Some } e \wedge x = \text{Inl } e))$ 
  apply (cases  $x$ )
  apply (auto simp add: to-xval-def default-option-def Exn-def)
  done

definition from-xval ::  $('e, 'a) \text{ xval} \Rightarrow ('e + 'a)$  where
  from-xval  $x = (\text{case } x \text{ of } \text{Exn } e \Rightarrow \text{Inl } e \mid \text{Result } v \Rightarrow \text{Inr } v)$ 

lemma from-xval-simps[simp]:
  from-xval (Exn  $e$ ) = Inl  $e$ 

```

$from-xval (Result v) = Inr v$
by (*auto simp add: from-xval-def*)

lemma *from-xval-Inr-iff[simp]*: $from-xval x = Inr v \longleftrightarrow x = Result v$
apply (*cases x*)
apply (*auto simp add: from-xval-def split: xval-splits*)
done

lemma *from-xval-Inl-iff[simp]*: $from-xval x = Inl e \longleftrightarrow x = Exn e$
apply (*cases x*)
apply (*auto simp add: from-xval-def split: xval-splits*)
done

lemma *to-xval-from-xval[simp]*: $to-xval (from-xval x) = x$
apply (*cases x*)
apply (*auto simp add: Exn-def default-option-def*)
done

lemma *from-xval-to-xval[simp]*: $from-xval (to-xval x) = x$
apply (*cases x*)
apply (*auto simp add: Exn-def default-option-def*)
done

lemma *rel-map-to-xval-Exn-iff[simp]*: $rel-map to-xval y (Exn e) \longleftrightarrow y = Inl e$
by (*auto simp add: rel-map-def to-xval-def split: sum.splits*)

lemma *rel-map-to-xval-Inr-iff[simp]*: $rel-map to-xval (Inr r) x \longleftrightarrow x = Result r$
by (*auto simp add: rel-map-def*)

lemma *rel-map-to-xval-Inl-iff[simp]*: $rel-map to-xval (Inl l) x \longleftrightarrow x = Exn l$
by (*auto simp add: rel-map-def*)

lemma *rel-map-to-xval-Result-iff[simp]*: $rel-map to-xval y (Result r) \longleftrightarrow y = Inr r$
by (*auto simp add: rel-map-def to-xval-def split: sum.splits*)

lemma *rel-map-from-xval-Exn-iff[simp]*: $rel-map from-xval (Exn l) x \longleftrightarrow x = Inl l$
by (*auto simp add: rel-map-def*)

lemma *rel-map-from-xval-Inl-iff[simp]*: $rel-map from-xval y (Inl e) \longleftrightarrow y = Exn e$
by (*auto simp add: rel-map-def from-xval-def split: xval-splits*)

lemma *rel-map-from-xval-Result-iff[simp]*: $rel-map from-xval (Result r) x \longleftrightarrow x = Inr r$
by (*auto simp add: rel-map-def*)

lemma *rel-map-from-xval-Inr-iff[simp]*: $rel-map from-xval y (Inr r) \longleftrightarrow y = Re-$

sult r
by (*auto simp add: rel-map-def from-xval-def split: xval-splits*)

16.8 *res-monad* and *exn-monad* functions

abbreviation *throw e* \equiv *yield (Exn e)*

definition *try* :: (*'e + 'a, 'a, 's*) *exn-monad* \Rightarrow (*'e, 'a, 's*) *exn-monad* **where**
try = *map-value unnest-exn*

definition *finally* :: (*'a, 'a, 's*) *exn-monad* \Rightarrow (*'a, 's*) *res-monad* **where**
finally = *map-value unite*

definition
catch :: (*'e, 'a, 's*) *exn-monad* \Rightarrow
(*'e* \Rightarrow (*'f::default, 'a, 's*) *spec-monad*) \Rightarrow
(*'f::default, 'a, 's*) *spec-monad* (**infix** *<catch>* 10)

where
f <catch> handler \equiv *bind-handle f return (handler o the)*

abbreviation *bind-finally* ::
(*'e, 'a, 's*) *exn-monad* \Rightarrow (*'e, 'a*) *xval* \Rightarrow (*'b, 's*) *res-monad* \Rightarrow (*'b, 's*) *res-monad*
where
bind-finally \equiv *bind-exception-or-result*

definition *ignoreE* :: (*'e, 'a, 's*) *exn-monad* \Rightarrow (*'a, 's*) *res-monad* **where**
ignoreE f = *catch f (_ . bot)*

definition *liftE* :: (*'a, 's*) *res-monad* \Rightarrow (*'e, 'a, 's*) *exn-monad* **where**
liftE = *map-value (map-exception-or-result (\lambda x. undefined) id)*

definition *check*:: *'e* \Rightarrow (*'s* \Rightarrow *'a* \Rightarrow *bool*) \Rightarrow (*'e, 'a, 's*) *exn-monad* **where**
check e p =
condition ($\lambda s. \exists x. p s x$)
(*do* { *s* \leftarrow *get-state*; *select* {*x. p s x*} })
(*throw e*)

abbreviation *check'* :: *'e* \Rightarrow (*'s* \Rightarrow *bool*) \Rightarrow (*'e, unit, 's*) *exn-monad* **where**
check' e p \equiv *check e (\lambda s -. p s)*

16.9 Monad operations

16.9.1 \top

declare *top-spec-monad.rep-eq*[*run-spec-monad, simp*]

lemma *always-progress-top*[*always-progress-intros*]: *always-progress* \top
by *transfer simp*

lemma *runs-to-top*[simp]: $\top \cdot s \{ Q \} \longleftrightarrow False$
by *transfer simp*

lemma *runs-to-partial-top*[simp]: $\top \cdot s ?\{ Q \} \longleftrightarrow True$
by *transfer simp*

16.9.2 \perp

declare *bot-spec-monad.rep-eq*[*run-spec-monad*, *simp*]

lemma *always-progress-bot-iff*[*iff*]: *always-progress* $\perp \longleftrightarrow False$
by *transfer simp*

lemma *runs-to-bot*[simp]: $\perp \cdot s \{ Q \} \longleftrightarrow True$
by *transfer simp*

lemma *runs-to-partial-bot*[simp]: $\perp \cdot s ?\{ Q \} \longleftrightarrow True$
by *transfer simp*

16.9.3 *fail*

lemma *always-progress-fail*[*always-progress-intros*]: *always-progress* *fail*
by *transfer (simp add: bot-post-state-def)*

lemma *run-fail*[*run-spec-monad*, *simp*]: *run fail* $s = \top$
by *transfer (simp add: top-post-state-def)*

lemma *runs-to-fail*[simp]: *fail* $\cdot s \{ R \} \longleftrightarrow False$
by *transfer simp*

lemma *runs-to-partial-fail*[simp]: *fail* $\cdot s ?\{ R \} \longleftrightarrow True$
by *transfer simp*

lemma *refines-fail*[simp]: *refines* *f fail* $s t R$
by (*simp add: refines.rep-eq*)

lemma *rel-spec-fail*[simp]: *rel-spec* *fail fail* $s t R$
by (*simp add: rel-spec-def*)

16.9.4 *yield*

lemma *run-yield*[*run-spec-monad*, *simp*]: *run (yield* r) $s = \text{pure-post-state}$ (r , s)
by *transfer simp*

lemma *always-progress-yield*[*always-progress-intros*]: *always-progress* (*yield* r)
by (*simp add: always-progress-def*)

lemma *yield-inj*[simp]: *yield* $x = \text{yield } y \longleftrightarrow x = y$
by *transfer (auto simp: fun-eq-iff)*

lemma *runs-to- yield -iff*[simp]: $((\text{yield } r) \cdot s \{ Q \}) \longleftrightarrow Q \ r \ s$
by *transfer simp*

lemma *runs-to- yield* [runs-to-vcg]: $Q \ r \ s \Longrightarrow \text{yield } r \cdot s \{ Q \}$
by *simp*

lemma *runs-to-partial- yield -iff*[simp]: $((\text{yield } r) \cdot s \ ?\{ Q \}) \longleftrightarrow Q \ r \ s$
by *transfer simp*

lemma *runs-to-partial- yield* [runs-to-vcg]: $Q \ r \ s \Longrightarrow \text{yield } r \cdot s \ ?\{ Q \}$
by *simp*

lemma *refines- yield -iff*[simp]: *refines* $(\text{yield } r) (\text{yield } r') \ s \ s' \ R \longleftrightarrow R \ (r, s) (r', s')$
by *transfer simp*

lemma *refines- yield* : $R \ (a, s) (b, t) \Longrightarrow \text{refines } (\text{yield } a) (\text{yield } b) \ s \ t \ R$
by *simp*

lemma *refines- yield -right-iff*:
refines $f \ (\text{yield } e) \ s \ t \ R \longleftrightarrow (f \cdot s \ \{ \lambda r \ s'. R \ (r, s') \ (e, t) \})$
by *(auto simp: refines.rep-eq runs-to.rep-eq sim-post-state-pure-post-state2 split-beta')*

lemma *rel-spec- yield* : $R \ (a, s) (b, t) \Longrightarrow \text{rel-spec } (\text{yield } a) (\text{yield } b) \ s \ t \ R$
by *(simp add: rel-spec-def)*

lemma *rel-spec- yield -iff*[simp]: *rel-spec* $(\text{yield } x) (\text{yield } y) \ s \ t \ Q \longleftrightarrow Q \ (x, s) (y, t)$
by *(simp add: rel-spec.rep-eq)*

lemma *rel-spec-monad- yield* : $Q \ x \ y \Longrightarrow \text{rel-spec-monad } R \ Q \ (\text{yield } x) (\text{yield } y)$
by *(auto simp add: rel-spec-monad-iff-rel-spec)*

16.9.5 *throw-exception-or-result*

lemma *throw-exception-or-result-bind*[simp]:
 $e \neq \text{default} \Longrightarrow \text{throw-exception-or-result } e \ >>= f = \text{throw-exception-or-result } e$
unfolding *bind-def* **by** *transfer auto*

16.9.6 *throw*

lemma *throw-bind*[simp]: $\text{throw } e \ >>= f = \text{throw } e$
by *(simp add: Exn-def)*

16.9.7 *get-state*

lemma *always-progress-get-state*[always-progress-intros]: *always-progress* *get-state*
by *transfer simp*

lemma *run-get-state*[*run-spec-monad, simp*]: *run get-state s = pure-post-state (Result s, s)*

by *transfer simp*

lemma *runs-to-get-state*[*runs-to-vcg*]: *get-state · s {λr t. r = Result s ∧ t = s}*

by *transfer simp*

lemma *runs-to-get-state-iff*[*runs-to-iff*]: *get-state · s {Q} ↔ (Q (Result s) s)*

by *transfer simp*

lemma *runs-to-partial-get-state*[*runs-to-vcg*]: *get-state · s ?{λr t. r = Result s ∧ t = s}*

by *transfer simp*

lemma *refines-get-state*: *R (Result s, s) (Result t, t) ⇒ refines get-state get-state s t R*

by *transfer simp*

lemma *rel-spec-get-state*: *R (Result s, s) (Result t, t) ⇒ rel-spec get-state get-state s t R*

by (*auto simp: rel-spec-iff-refines intro: refines-get-state*)

16.9.8 set-state

lemma *always-progress-set-state*[*always-progress-intros*]: *always-progress (set-state t)*

by *transfer simp*

lemma *set-state-inj*[*simp*]: *set-state x = set-state y ↔ x = y*

by *transfer (simp add: fun-eq-iff)*

lemma *run-set-state*[*run-spec-monad, simp*]: *run (set-state t) s = pure-post-state (Result (), t)*

by *transfer simp*

lemma *runs-to-set-state*[*runs-to-vcg*]: *set-state t · s {λr t'. t' = t}*

by *transfer simp*

lemma *runs-to-set-state-iff*[*runs-to-iff*]: *set-state t · s {Q} ↔ Q (Result ()) t*

by *transfer simp*

lemma *runs-to-partial-set-state*[*runs-to-vcg*]: *set-state t · s ?{λr t'. t' = t}*

by *transfer simp*

lemma *refines-set-state*:

R (Result (), s') (Result (), t') ⇒ refines (set-state s') (set-state t') s t R

by *transfer simp*

lemma *rel-spec-set-state*:

$R (Result (), s') (Result (), t') \implies rel-spec (set-state s') (set-state t') s t R$
by (*auto simp add: rel-spec-iff-refines intro: refines-set-state*)

16.9.9 *select*

lemma *always-progress-select*[*always-progress-intros*]: $S \neq \{\}$ \implies *always-progress* (*select S*)

unfolding *select-def* **by** *transfer simp*

lemma *runs-to-select*[*runs-to-vcg*]: $(\bigwedge x. x \in S \implies Q (Result x) s) \implies select S \cdot s \{\!| Q |\!\}$

unfolding *select-def* **by** *transfer simp*

lemma *runs-to-select-iff*[*runs-to-iff*]: $select S \cdot s \{\!| Q |\!\} \longleftrightarrow (\forall x \in S. Q (Result x) s)$

unfolding *select-def* **by** *transfer auto*

lemma *runs-to-partial-select*[*runs-to-vcg*]: $(\bigwedge x. x \in S \implies Q (Result x) s) \implies select S \cdot s \{\!| Q |\!\}$

using *runs-to-select* **by** (*rule runs-to-partial-of-runs-to*)

lemma *run-select*[*run-spec-monad, simp*]: $run (select S) s = Success ((\lambda v. (Result v, s)) ' S)$

unfolding *select-def* **by** *transfer (auto simp add: image-image pure-post-state-def Sup-Success)*

lemma *refines-select*:

$(\bigwedge x. x \in P \implies \exists xa \in Q. R (Result x, s) (Result xa, t)) \implies refines (select P) (select Q) s t R$

unfolding *select-def* **by** *transfer (auto intro!: sim-post-state-Sup)*

lemma *rel-spec-select*:

$rel-set (\lambda a b. R (Result a, s) (Result b, t)) P Q \implies rel-spec (select P) (select Q) s t R$

by (*auto simp add: rel-spec-def rel-set-def*)

16.9.10 *unknown*

lemma *runs-to-unknown*[*runs-to-vcg*]: $(\bigwedge x. Q (Result x) s) \implies unknown \cdot s \{\!| Q |\!\}$

by (*simp add: unknown-def runs-to-select-iff*)

lemma *runs-to-unknown-iff*[*runs-to-iff*]: $unknown \cdot s \{\!| Q |\!\} \longleftrightarrow (\forall x. Q (Result x) s)$

by (*simp add: unknown-def runs-to-select-iff*)

lemma *runs-to-partial-unknown*[*runs-to-vcg*]: $(\bigwedge x. Q (Result x) s) \implies unknown \cdot s \{\!| Q |\!\}$

using *runs-to-unknown* **by** (*rule runs-to-partial-of-runs-to*)

lemma *run-unknown*[*run-spec-monad, simp*]: *run unknown s = Success ((λv. (Result v, s)) ‘UNIV)*

unfolding *unknown-def by simp*

lemma *always-progress-unknown*[*always-progress-intros*]: *always-progress unknown*

unfolding *unknown-def by (simp add: always-progress-intros)*

16.9.11 lift-state

lemma *run-lift-state*[*run-spec-monad*]:

run (lift-state R f) s = lift-post-state (rel-prod (=) R) (SUP t ∈ {t. R s t}. run f t)

by *transfer standard*

lemma *runs-to-lift-state-iff*[*runs-to-iff*]:

(lift-state R f) · s ?{ Q } ↔ (∀ s'. R s s' → f · s' ?{λr t'. ∀ t. R t t' → Q r t})

by (*simp add: runs-to.rep-eq lift-state.rep-eq rel-prod-sel split-beta'*)

lemma *runs-to-lift-state*[*runs-to-vcg*]:

(∧ s'. R s s' ⇒ f · s' ?{λr t'. ∀ t. R t t' → Q r t}) ⇒ lift-state R f · s ?{ Q }

by (*simp add: runs-to-lift-state-iff*)

lemma *runs-to-partial-lift-state*[*runs-to-vcg*]:

(∧ s'. R s s' ⇒ f · s' ?{λr t'. ∀ t. R t t' → Q r t}) ⇒ lift-state R f · s ?{ Q }

by (*cases ∀ s'. R s s' → run f s' ≠ Failure*)

(*auto simp: runs-to-partial-alt run-lift-state lift-post-state-Sup image-image image-iff runs-to-lift-state-iff top-post-state-def*)

lemma *mono-lift-state*: *f ≤ f' ⇒ lift-state R f ≤ lift-state R f'*

by *transfer (auto simp: le-fun-def intro!: lift-post-state-mono SUP-mono)*

16.9.12 const exec-concrete

lemma *run-exec-concrete*[*run-spec-monad*]:

run (exec-concrete st f) s = map-post-state (apsnd st) (⊔ (run f ‘ st - ‘ {s}))

by (*auto simp add: exec-concrete-def lift-state-def map-post-state-eq-lift-post-state fun-eq-iff*

intro!: arg-cong2[where f=lift-post-state] SUP-cong)

lemma *runs-to-exec-concrete-iff*[*runs-to-iff*]:

exec-concrete st f · s ?{ Q } ↔ (∀ t. s = st t → f · t ?{λr t. Q r (st t)})

by (*auto simp: runs-to.rep-eq run-exec-concrete split-beta'*)

lemma *runs-to-exec-concrete*[*runs-to-vcg*]:

(∧ t. s = st t ⇒ f · t ?{λr t. Q r (st t)}) ⇒ exec-concrete st f · s ?{ Q }

by (*simp add: runs-to-exec-concrete-iff*)

lemma *runs-to-partial-exec-concrete*[*runs-to-vcg*]:

(∧ t. s = st t ⇒ f · t ?{λr t. Q r (st t)}) ⇒ exec-concrete st f · s ?{ Q }

by (simp add: exec-concrete-def runs-to-partial-lift-state)

lemma *mono-exec-concrete*: $f \leq f' \implies \text{exec-concrete } st \ f \leq \text{exec-concrete } st \ f'$
unfolding *exec-concrete-def* **by** (rule *mono-lift-state*)

lemma *monotone-exec-concrete-le*[*partial-function-mono*]:
monotone $Q (\leq) f \implies \text{monotone } Q (\leq) (\lambda f'. \text{exec-concrete } st \ (f f'))$
using *mono-exec-concrete*
by (*fastforce simp add: monotone-def le-fun-def*)

lemma *monotone-exec-concrete-ge*[*partial-function-mono*]:
monotone $Q (\geq) f \implies \text{monotone } Q (\geq) (\lambda f'. \text{exec-concrete } st \ (f f'))$
using *mono-exec-concrete*
by (*fastforce simp add: monotone-def le-fun-def*)

16.9.13 const *exec-abstract*

lemma *run-exec-abstract*[*run-spec-monad*]:
run (*exec-abstract* $st \ f$) $s = \text{vmap-post-state } (\text{apsnd } st) \ (\text{run } f \ (st \ s))$
by (*auto simp add: exec-abstract-def vmap-post-state-eq-lift-post-state lift-state-def fun-eq-iff*
intro!: arg-cong2[where f=lift-post-state])

lemma *runs-to-exec-abstract-iff*[*runs-to-iff*]:
exec-abstract $st \ f \cdot s \ \{\!\!| \ Q \ \!\!\} \longleftrightarrow f \cdot (st \ s) \ \{\!\!| \lambda r \ t. \forall s'. t = st \ s' \longrightarrow Q \ r \ s' \ \!\!\}$
by (*simp add: runs-to.rep-eq run-exec-abstract split-beta' prod-eq-iff eq-commute*)

lemma *runs-to-exec-abstract*[*runs-to-vcg*]:
 $f \cdot (st \ s) \ \{\!\!| \lambda r \ t. \forall s'. t = st \ s' \longrightarrow Q \ r \ s' \ \!\!\} \implies \text{exec-abstract } st \ f \cdot s \ \{\!\!| \ Q \ \!\!\}$
by (*simp add: runs-to-exec-abstract-iff*)

lemma *runs-to-partial-exec-abstract*[*runs-to-vcg*]:
 $f \cdot (st \ s) \ \{\!\!| \lambda r \ t. \forall s'. t = st \ s' \longrightarrow Q \ r \ s' \ \!\!\} \implies \text{exec-abstract } st \ f \cdot s \ \{\!\!| \ Q \ \!\!\}$
by (*simp add: runs-to-partial.rep-eq run-exec-abstract split-beta' prod-eq-iff eq-commute*)

lemma *mono-exec-abstract*: $f \leq f' \implies \text{exec-abstract } st \ f \leq \text{exec-abstract } st \ f'$
unfolding *exec-abstract-def* **by** (rule *mono-lift-state*)

lemma *monotone-exec-abstract-le*[*partial-function-mono*]:
monotone $Q (\leq) f \implies \text{monotone } Q (\leq) (\lambda f'. \text{exec-abstract } st \ (f f'))$
by (*simp add: monotone-def mono-exec-abstract*)

lemma *monotone-exec-abstract-ge*[*partial-function-mono*]:
monotone $Q (\geq) f \implies \text{monotone } Q (\geq) (\lambda f'. \text{exec-abstract } st \ (f f'))$
by (*simp add: monotone-def mono-exec-abstract*)

16.9.14 *bind-exception-or-result*

lemma *runs-to-bind-exception-or-result-iff*[*runs-to-iff*]:
bind-exception-or-result $f \ g \cdot s \ \{\!\!| \ Q \ \!\!\} \longleftrightarrow f \cdot s \ \{\!\!| \lambda r \ t. g \ r \cdot t \ \{\!\!| \ Q \ \!\!\} \ \!\!\}$

by transfer (simp add: split-beta')

lemma runs-to-bind-exception-or-result[runs-to-vcg]:

$f \cdot s \{ \lambda r t. g r \cdot t \{ Q \} \} \implies \text{bind-exception-or-result } f g \cdot s \{ Q \}$
by (auto simp: runs-to-bind-exception-or-result-iff)

lemma runs-to-partial-bind-exception-or-result[runs-to-vcg]:

$f \cdot s \{ ? \lambda r t. g r \cdot t \{ ? Q \} \} \implies \text{bind-exception-or-result } f g \cdot s \{ ? Q \}$
by transfer (simp add: split-beta' holds-partial-bind-post-state)

lemma refines-bind-exception-or-result:

$\text{refines } f f' s s' (\lambda(r, t) (r', t')). \text{refines } (g r) (g' r') t t' R \implies$
 $\text{refines } (\text{bind-exception-or-result } f g) (\text{bind-exception-or-result } f' g') s s' R$
by transfer (auto intro: sim-bind-post-state)

lemma refines-bind-exception-or-result':

assumes $f: \text{refines } f f' s s' Q$
assumes $g: \bigwedge r t r' t'. Q (r, t) (r', t') \implies \text{refines } (g r) (g' r') t t' R$
shows $\text{refines } (\text{bind-exception-or-result } f g) (\text{bind-exception-or-result } f' g') s s' R$
by (auto intro: refines-bind-exception-or-result refines-mono[OF -] g)

lemma mono-bind-exception-or-result:

$f \leq f' \implies g \leq g' \implies \text{bind-exception-or-result } f g \leq \text{bind-exception-or-result } f' g'$
unfolding le-fun-def
by transfer (auto simp add: le-fun-def intro!: mono-bind-post-state)

lemma monotone-bind-exception-or-result-le[partial-function-mono]:

$\text{monotone } R (\leq) (\lambda f'. f f') \implies (\bigwedge v. \text{monotone } R (\leq) (\lambda f'. g f' v)) \implies$
 $\text{monotone } R (\leq) (\lambda f'. \text{bind-exception-or-result } (f f') (g f'))$
by (simp add: monotone-def) (metis le-fun-def mono-bind-exception-or-result)

lemma monotone-bind-exception-or-result-ge[partial-function-mono]:

$\text{monotone } R (\geq) (\lambda f'. f f') \implies (\bigwedge v. \text{monotone } R (\geq) (\lambda f'. g f' v)) \implies$
 $\text{monotone } R (\geq) (\lambda f'. \text{bind-exception-or-result } (f f') (g f'))$
by (simp add: monotone-def) (metis le-fun-def mono-bind-exception-or-result)

lemma run-bind-exception-or-result-cong:

assumes *: $\text{run } f s = \text{run } f' s$
assumes **: $f \cdot s \{ \lambda x s'. \text{run } (g x) s' = \text{run } (g' x) s' \}$
shows $\text{run } (\text{bind-exception-or-result } f g) s = \text{run } (\text{bind-exception-or-result } f' g')$
s

using assms

by (cases run f' s)

(auto simp: bind-exception-or-result-def runs-to-partial.rep-eq
intro!: SUP-cong split: exception-or-result-splits)

16.9.15 *bind-handle*

lemma *bind-handle-eq*:

bind-handle $f g h =$

bind-exception-or-result $f (\lambda r. \text{case } r \text{ of } \text{Exception } e \Rightarrow h e \mid \text{Result } v \Rightarrow g v)$

unfolding *spec-monad-ext-iff* *bind-handle.rep-eq* *bind-exception-or-result.rep-eq*

by (*intro allI* *arg-cong*[**where** $f = \text{bind-post-state } _$] *ext*)

(*auto split*: *exception-or-result-split*)

lemma *runs-to-bind-handle-iff*[*runs-to-iff*]:

bind-handle $f g h \cdot s \{ Q \} \longleftrightarrow f \cdot s \{ \lambda r t.$

$(\forall v. r = \text{Result } v \longrightarrow g v \cdot t \{ Q \}) \wedge$

$(\forall e. r = \text{Exception } e \longrightarrow e \neq \text{default} \longrightarrow h e \cdot t \{ Q \}) \}$

by *transfer*

(*auto intro!*: *arg-cong*[**where** $f = \lambda p. \text{holds-post-state } p _$] *simp*: *fun-eq-iff*

split: *prod.splits exception-or-result-splits*)

lemma *runs-to-bind-handle*[*runs-to-vcg*]:

$f \cdot s \{ \lambda r t. (\forall v. r = \text{Result } v \longrightarrow g v \cdot t \{ Q \}) \wedge$

$(\forall e. r = \text{Exception } e \longrightarrow e \neq \text{default} \longrightarrow h e \cdot t \{ Q \}) \} \Longrightarrow$

bind-handle $f g h \cdot s \{ Q \}$

by (*auto simp*: *runs-to-bind-handle-iff*)

lemma *runs-to-bind-handle-exception-monad*[*runs-to-vcg*]:

fixes $f :: ('e, 'a, 's) \text{exn-monad}$

assumes f :

$f \cdot s \{ \lambda r t. (\forall v. r = \text{Result } v \longrightarrow (g v \cdot t \{ Q \})) \wedge$

$(\forall e. r = \text{Exception } (\text{Some } e) \longrightarrow (h (\text{Some } e) \cdot t \{ Q \})) \}$

shows *bind-handle* $f g h \cdot s \{ Q \}$

by (*rule runs-to-bind-handle*[*OF runs-to-weaken*[*OF f*]]) (*auto simp*: *default-option-def*)

lemma *runs-to-bind-handle-res-monad*[*runs-to-vcg*]:

fixes $f :: ('a, 's) \text{res-monad}$

assumes f : $f \cdot s \{ \lambda r t. (\forall v. r = \text{Result } v \longrightarrow (g v \cdot t \{ Q \})) \}$

shows *bind-handle* $f g h \cdot s \{ Q \}$

by (*rule runs-to-bind-handle*[*OF runs-to-weaken*[*OF f*]]) (*auto simp*: *default-option-def*)

lemma *runs-to-partial-bind-handle*[*runs-to-vcg*]:

$f \cdot s \{ \lambda r t. (\forall v. r = \text{Result } v \longrightarrow g v \cdot t \{ Q \}) \wedge$

$(\forall e. r = \text{Exception } e \longrightarrow e \neq \text{default} \longrightarrow h e \cdot t \{ Q \}) \} \Longrightarrow$

bind-handle $f g h \cdot s \{ Q \}$

by *transfer*

(*auto intro!*: *holds-partial-bind-post-stateI* *intro*: *holds-partial-post-state-weaken*

split: *exception-or-result-splits prod.splits*)

lemma *runs-to-partial-bind-handle-exception-monad*[*runs-to-vcg*]:

fixes $f :: ('e, 'a, 's) \text{exn-monad}$

assumes f : $f \cdot s \{ \lambda r t. (\forall v. r = \text{Result } v \longrightarrow (g v \cdot t \{ Q \})) \wedge$

$(\forall e. r = \text{Exception } (\text{Some } e) \longrightarrow (h (\text{Some } e) \cdot t \{ Q \})) \}$

shows *bind-handle* $f g h \cdot s \{ Q \}$

by (*rule runs-to-partial-bind-handle*[*OF runs-to-partial-weaken*[*OF f*]])
(auto simp: default-option-def)

lemma *runs-to-partial-bind-handle-res-monad*[*runs-to-vcg*]:

fixes $f :: ('a, 's) \text{res-monad}$

assumes $f: f \cdot s \text{ ?}\{\!\! \} \lambda r t. (\forall v. r = \text{Result } v \longrightarrow (g v \cdot t \text{ ?}\{\!\! \} Q \!\!\})\}$

shows *bind-handle* $f g h \cdot s \text{ ?}\{\!\! \} Q \!\!\}$

by (*rule runs-to-partial-bind-handle*[*OF runs-to-partial-weaken*[*OF f*]])
(auto simp: default-option-def)

lemma *mono-bind-handle*:

$f \leq f' \Longrightarrow g \leq g' \Longrightarrow h \leq h' \Longrightarrow \text{bind-handle } f g h \leq \text{bind-handle } f' g' h'$

unfolding *le-fun-def*

by *transfer*

(auto simp add: le-fun-def intro!: mono-bind-post-state split: exception-or-result-splits)

lemma *monotone-bind-handle-le*[*partial-function-mono*]:

monotone $R (\leq) (\lambda f'. f f') \Longrightarrow (\bigwedge v. \text{monotone } R (\leq) (\lambda f'. g f' v)) \Longrightarrow$

$(\bigwedge e. \text{monotone } R (\leq) (\lambda f'. h f' e)) \Longrightarrow$

monotone $R (\leq) (\lambda f'. \text{bind-handle } (f f') (g f') (h f'))$

by (*simp add: monotone-def*) (*metis le-fun-def mono-bind-handle*)

lemma *monotone-bind-handle-ge*[*partial-function-mono*]:

monotone $R (\geq) (\lambda f'. f f') \Longrightarrow (\bigwedge v. \text{monotone } R (\geq) (\lambda f'. g f' v)) \Longrightarrow$

$(\bigwedge e. \text{monotone } R (\geq) (\lambda f'. h f' e)) \Longrightarrow$

monotone $R (\geq) (\lambda f'. \text{bind-handle } (f f') (g f') (h f'))$

by (*simp add: monotone-def*) (*metis le-fun-def mono-bind-handle*)

lemma *run-bind-handle*[*run-spec-monad*]:

run (*bind-handle* $f g h$) $s = \text{bind-post-state } (\text{run } f s)$

$(\lambda(r, t). \text{case } r \text{ of } \text{Exception } e \Rightarrow \text{run } (h e) t \mid \text{Result } v \Rightarrow \text{run } (g v) t)$

by *transfer simp*

lemma *always-progress-bind-handle*[*always-progress-intros*]:

always-progress $f \Longrightarrow (\bigwedge v. \text{always-progress } (g v)) \Longrightarrow (\bigwedge e. \text{always-progress } (h e))$

$\Longrightarrow \text{always-progress } (\text{bind-handle } f g h)$

by (*auto simp: always-progress.rep-eq run-bind-handle bind-post-state-eq-bot prod-eq-iff*
exception-or-result-nchotomy holds-post-state-False

split: exception-or-result-splits prod.splits)

lemma *refines-bind-handle'*:

refines $f f' s s' (\lambda(r, t) (r', t'))$.

$(\forall e e'. e \neq \text{default} \longrightarrow e' \neq \text{default} \longrightarrow r = \text{Exception } e \longrightarrow r' = \text{Exception } e' \longrightarrow$

$\text{refines } (h e) (h' e') t t' R) \wedge$

$(\forall e x'. e \neq \text{default} \longrightarrow r = \text{Exception } e \longrightarrow r' = \text{Result } x' \longrightarrow$

$\text{refines } (h e) (g' x') t t' R) \wedge$

$(\forall x e'. e' \neq \text{default} \longrightarrow r = \text{Result } x \longrightarrow r' = \text{Exception } e' \longrightarrow$

$\text{refines } (g x) (h' e') t t' R) \wedge$

$(\forall x x'. r = \text{Result } x \longrightarrow r' = \text{Result } x' \longrightarrow$
 $\text{refines } (g x) (g' x') t t' R) \implies$
 $\text{refines } (\text{bind-handle } f g h) (\text{bind-handle } f' g' h') s s' R$
apply *transfer*
apply (*auto intro! sim-bind-post-state'*)[1]
apply (*rule sim-post-state-weaken, assumption*)
apply (*auto split: exception-or-result-splits prod.splits*)
done

lemma *refines-bind-handle-bind-handle:*

assumes *f: refines f f' s s' Q*
assumes *ll: $\bigwedge e e' t t'. Q (\text{Exception } e, t) (\text{Exception } e', t') \implies$*
 $e \neq \text{default} \implies e' \neq \text{default} \implies$
 $\text{refines } (h e) (h' e') t t' R$
assumes *lr: $\bigwedge e v' t t'. Q (\text{Exception } e, t) (\text{Result } v', t') \implies e \neq \text{default} \implies$*
 $\text{refines } (h e) (g' v') t t' R$
assumes *rl: $\bigwedge v e' t t'. Q (\text{Result } v, t) (\text{Exception } e', t') \implies e' \neq \text{default} \implies$*
 $\text{refines } (g v) (h' e') t t' R$
assumes *rr: $\bigwedge v v' t t'. Q (\text{Result } v, t) (\text{Result } v', t') \implies$*
 $\text{refines } (g v) (g' v') t t' R$
shows $\text{refines } (\text{bind-handle } f g h) (\text{bind-handle } f' g' h') s s' R$
apply (*rule refines-bind-handle'*)
apply (*rule refines-weaken[OF f]*)
apply (*auto split: exception-or-result-splits prod.splits intro: ll lr rl rr*)
done

lemma *refines-bind-handle-bind-handle-exn:*

assumes *f: refines f f' s s' Q*
assumes *ll: $\bigwedge e e' t t'. Q (\text{Exn } e, t) (\text{Exn } e', t') \implies$*
 $\text{refines } (h (\text{Some } e)) (h' (\text{Some } e')) t t' R$
assumes *lr: $\bigwedge e v' t t'. Q (\text{Exn } e, t) (\text{Result } v', t') \implies$*
 $\text{refines } (h (\text{Some } e)) (g' v') t t' R$
assumes *rl: $\bigwedge v e' t t'. Q (\text{Result } v, t) (\text{Exn } e', t') \implies$*
 $\text{refines } (g v) (h' (\text{Some } e')) t t' R$
assumes *rr: $\bigwedge v v' t t'. Q (\text{Result } v, t) (\text{Result } v', t') \implies$*
 $\text{refines } (g v) (g' v') t t' R$
shows $\text{refines } (\text{bind-handle } f g h) (\text{bind-handle } f' g' h') s s' R$
apply (*rule refines-bind-handle'*)
apply (*rule refines-weaken[OF f]*)
using *ll lr rl rr*
apply (*auto split: exception-or-result-splits prod.splits simp: Exn-def default-option-def*)
done

lemma *bind-handle-return-spec-monad[simp]:* $\text{bind-handle } (\text{return } v) g h = g v$
by *transfer simp*

lemma *bind-handle-throw-spec-monad[simp]:*

$v \neq \text{default} \implies \text{bind-handle } (\text{throw-exception-or-result } v) g h = h v$
by *transfer simp*

lemma *bind-handle-bind-handle-spec-monad*:

bind-handle (*bind-handle* *f* *g1* *h1*) *g2* *h2* =
bind-handle *f*
 $(\lambda v. \text{bind-handle } (g1 \ v) \ g2 \ h2)$
 $(\lambda e. \text{bind-handle } (h1 \ e) \ g2 \ h2)$

apply *transfer*

apply (*auto simp*: *fun-eq-iff intro!*: *arg-cong*[**where** *f=bind-post-state* -])[1]

subgoal for *g1* *h1* *g2* *h2* *r* *v* **by** (*cases* *r*) *simp-all*

done

lemma *mono-bind-handle-spec-monad*:

mono *f* $\implies (\bigwedge v. \text{mono } (\lambda x. g \ x \ v)) \implies (\bigwedge e. \text{mono } (\lambda x. h \ x \ e)) \implies$
 $\text{mono } (\lambda x. \text{bind-handle } (f \ x) \ (g \ x) \ (h \ x))$

unfolding *mono-def*

apply *transfer*

apply (*auto simp*: *le-fun-def intro!*: *mono-bind-post-state*)[1]

subgoal for *f* *g* *h* *x* *y* *r* *q* **by** (*cases* *r*) *simp-all*

done

lemma *rel-spec-bind-handle*:

rel-spec *f* *f'* *s* *s'* $(\lambda(r, t) \ (r', t')).$

$(\forall e \ e'. \ e \neq \text{default} \longrightarrow e' \neq \text{default} \longrightarrow r = \text{Exception } e \longrightarrow r' = \text{Exception } e' \longrightarrow$

$\text{rel-spec } (h \ e) \ (h' \ e') \ t \ t' \ R) \wedge$

$(\forall e \ x'. \ e \neq \text{default} \longrightarrow r = \text{Exception } e \longrightarrow r' = \text{Result } x' \longrightarrow$

$\text{rel-spec } (h \ e) \ (g' \ x') \ t \ t' \ R) \wedge$

$(\forall x \ e'. \ e' \neq \text{default} \longrightarrow r = \text{Result } x \longrightarrow r' = \text{Exception } e' \longrightarrow$

$\text{rel-spec } (g \ x) \ (h' \ e') \ t \ t' \ R) \wedge$

$(\forall x \ x'. \ r = \text{Result } x \longrightarrow r' = \text{Result } x' \longrightarrow$

$\text{rel-spec } (g \ x) \ (g' \ x') \ t \ t' \ R)) \implies$

rel-spec (*bind-handle* *f* *g* *h*) (*bind-handle* *f'* *g'* *h'*) *s* *s'* *R*

by (*auto simp*: *rel-spec-iff-refines intro!*: *refines-bind-handle' intro*: *refines-weaken*)

lemma *bind-finally-bind-handle-conv*: *bind-finally* *f* *g* = *bind-handle* *f* $(\lambda v. \ g \ (\text{Result } v))$ $(\lambda e. \ g \ (\text{Exception } e))$

apply (*rule spec-monad-eqI*)

apply (*auto simp add*: *runs-to-iff Exn-def* [*symmetric*] *default-option-def elim!*: *runs-to-weaken*)[1]

using *exception-or-result-nchotomy* **by** *force*

16.9.16 (\gg)

lemma *run-bind*[*run-spec-monad*]: *run* (*bind* *f* *g*) *s* =

bind-post-state (*run* *f* *s*) $(\lambda(r, t). \ \text{case } r \ \text{of}$

$\text{Exception } e \Rightarrow \text{pure-post-state } (\text{Exception } e, t)$

$| \text{Result } v \Rightarrow \text{run } (g \ v) \ t)$

by (*auto simp add*: *bind-def run-bind-handle*)

lemma *run-bind-eq-top-iff*:

$run (bind f g) s = \top \iff \neg (f \cdot s \{ \lambda x s. \forall a. x = Result a \longrightarrow run (g a) s \neq \top \})$

by (*simp add: run-bind bind-post-state-eq-top runs-to.rep-eq prod-eq-iff split-beta' split: exception-or-result-splits prod.splits*)

lemma *always-progress-bind*[*always-progress-intros*]:

$always_progress f \implies (\bigwedge v. always_progress (g v)) \implies always_progress (bind f g)$

by (*simp add: always-progress-intros bind-def*)

lemma *run-bind-cong*:

assumes *: $run f s = run f' s$

assumes **: $f \cdot s \{ \lambda x s'. \forall v. x = (Result v) \longrightarrow run (g v) s' = run (g' v) s' \}$

shows $run (bind f g) s = run (bind f' g') s$

using *assms*

by (*cases run f' s*)

(*auto simp: run-bind runs-to-partial.rep-eq*

intro!: SUP-cong split: exception-or-result-splits)

lemma *runs-to-bind-iff*[*runs-to-iff*]:

$bind f g \cdot s \{ Q \} \iff f \cdot s \{ \lambda r t.$

$(\forall v. r = Result v \longrightarrow g v \cdot t \{ Q \}) \wedge$

$(\forall e. r = Exception e \longrightarrow e \neq default \longrightarrow Q (Exception e) t) \}$

by (*simp add: bind-def runs-to-bind-handle-iff*)

lemma *runs-to-bind*[*runs-to-vcg*]:

$f \cdot s \{ \lambda r t. (\forall v. r = Result v \longrightarrow g v \cdot t \{ Q \}) \wedge$

$(\forall e. r = Exception e \longrightarrow e \neq default \longrightarrow Q (Exception e) t) \} \implies$

$bind f g \cdot s \{ Q \}$

by (*simp add: runs-to-bind-iff*)

lemma *runs-to-bind-exception*[*runs-to-vcg*]:

fixes $f :: ('e, 'a, 's) \text{exn-monad}$

assumes [*runs-to-vcg*]: $f \cdot s \{ \lambda r t. (\forall v. r = Result v \longrightarrow g v \cdot t \{ Q \}) \wedge$

$(\forall e. r = Exn e \longrightarrow Q (Exn e) t) \}$

shows $bind f g \cdot s \{ Q \}$

supply *runs-to-bind*[*runs-to-vcg*]

by *runs-to-vcg (auto simp: Exn-def default-option-def)*

lemma *runs-to-bind-res*[*runs-to-vcg*]:

fixes $f :: ('a, 's) \text{res-monad}$

assumes [*runs-to-vcg*]: $f \cdot s \{ \lambda Res v t. g v \cdot t \{ Q \} \}$

shows $bind f g \cdot s \{ Q \}$

supply *runs-to-bind*[*runs-to-vcg*]

by *runs-to-vcg*

lemma *runs-to-partial-bind*[*runs-to-vcg*]:

$f \cdot s \{ \lambda r t. (\forall v. r = Result v \longrightarrow g v \cdot t \{ Q \}) \wedge$

$(\forall e. r = Exception e \longrightarrow e \neq default \longrightarrow Q (Exception e) t) \} \implies$

```

    bind f g · s ?{ Q }
  apply (simp add: bind-def)
  apply (rule runs-to-partial-bind-handle)
  apply simp
  done

```

```

lemma runs-to-partial-bind-exception-monad[runs-to-vcg]:
  fixes f :: ('e, 'a, 's) exn-monad
  assumes [runs-to-vcg]: f · s ?{ λr t. (∀ v. r = Result v → g v · t ?{ Q }) } ∧
    (∀ e. r = Exn e → Q (Exn e) t)
  shows bind f g · s ?{ Q }
  by runs-to-vcg (auto simp add: default-option-def Exn-def)

```

```

lemma runs-to-partial-bind-res-monad[runs-to-vcg]:
  fixes f :: ('a, 's) res-monad
  assumes [runs-to-vcg]: f · s ?{ λRes v t. g v · t ?{ Q } }
  shows bind f g · s ?{ Q }
  by runs-to-vcg

```

```

lemma bind-return[simp]: (bind m return) = m
  apply (clarsimp simp: spec-monad-eq-iff runs-to-bind-iff fun-eq-iff
    intro!: runs-to-cong-pred-only)
  subgoal for P r t
    by (cases r) auto
  done

```

```

lemma bind-skip[simp]: bind m (λx. skip) = m
  using bind-return[of m] by simp

```

```

lemma return-bind[simp]: bind (return x) f = f x
  unfolding bind-def by transfer simp

```

```

lemma bind-assoc: bind (bind f g) h = bind f (λx. bind (g x) h)
  apply (rule spec-monad-ext)
  apply (auto simp: split-beta' run-bind intro!: arg-cong[where f=bind-post-state
-]
    split: exception-or-result-splits)
  done

```

```

lemma mono-bind: f ≤ f' ⇒ g ≤ g' ⇒ bind f g ≤ bind f' g'
  unfolding bind-def
  by (auto intro: mono-bind-handle)

```

```

lemma mono-bind-spec-monad:
  mono f ⇒ (Λv. mono (λx. g x v)) ⇒ mono (λx. bind (f x) (g x))
  unfolding bind-def
  by (intro mono-bind-handle-spec-monad mono-const) auto

```

```

lemma monotone-bind-le[partial-function-mono]:

```

$monotone\ R\ (\leq)\ (\lambda f'.\ f\ f') \implies (\bigwedge v.\ monotone\ R\ (\leq)\ (\lambda f'.\ g\ f'\ v))$
 $\implies monotone\ R\ (\leq)\ (\lambda f'.\ bind\ (f\ f')\ (g\ f'))$
unfolding *bind-def*
by (*auto intro: monotone-bind-handle-le*)

lemma *monotone-bind-ge[partial-function-mono]*:
 $monotone\ R\ (\geq)\ (\lambda f'.\ f\ f') \implies (\bigwedge v.\ monotone\ R\ (\geq)\ (\lambda f'.\ g\ f'\ v))$
 $\implies monotone\ R\ (\geq)\ (\lambda f'.\ bind\ (f\ f')\ (g\ f'))$
unfolding *bind-def*
by (*auto intro: monotone-bind-handle-ge*)

lemma *refines-bind'*:
assumes *f: refines f f' s s' ($\lambda(x, t)\ (x', t')$)*.
 $(\forall e.\ e \neq default \longrightarrow x = Exception\ e \longrightarrow$
 $(\forall e'.\ e' \neq default \longrightarrow x' = Exception\ e' \longrightarrow R\ (Exception\ e,\ t)\ (Exception$
 $e',\ t')) \wedge$
 $(\forall v'.\ x' = Result\ v' \longrightarrow refines\ (throw-exception-or-result\ e)\ (g'\ v')\ t\ t'\ R)))$
 \wedge
 $(\forall v.\ x = Result\ v \longrightarrow$
 $(\forall e'.\ e' \neq default \longrightarrow x' = Exception\ e' \longrightarrow$
 $refines\ (g\ v)\ (throw-exception-or-result\ e')\ t\ t'\ R) \wedge$
 $(\forall v'.\ x' = Result\ v' \longrightarrow refines\ (g\ v)\ (g'\ v')\ t\ t'\ R))))$
shows *refines (bind f g) (bind f' g') s s' R*
unfolding *bind-def*
by (*rule refines-bind-handle'[OF refines-weaken, OF f]*) *auto*

lemma *refines-bind*:
assumes *f: refines f f' s s' Q*
assumes *ll: $\bigwedge e\ e'\ t\ t'.\ Q\ (Exception\ e,\ t)\ (Exception\ e',\ t') \implies e \neq default \implies e' \neq default \implies$*
 $R\ (Exception\ e,\ t)\ (Exception\ e',\ t')$
assumes *lr: $\bigwedge e\ v'\ t\ t'.\ Q\ (Exception\ e,\ t)\ (Result\ v',\ t') \implies e \neq default \implies$*
 $refines\ (yield\ (Exception\ e))\ (g'\ v')\ t\ t'\ R$
assumes *rl: $\bigwedge v\ e'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exception\ e',\ t') \implies e' \neq default \implies$*
 $refines\ (g\ v)\ (yield\ (Exception\ e'))\ t\ t'\ R$
assumes *rr: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \implies$*
 $refines\ (g\ v)\ (g'\ v')\ t\ t'\ R$
shows *refines (bind f g) (bind f' g') s s' R*
by (*rule refines-bind'[OF refines-weaken[OF f]]*)
(auto simp: ll lr rl rr)

lemma *refines-bind-bind-exn*:
assumes *f: refines f f' s s' Q*
assumes *ll: $\bigwedge e\ e'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Exn\ e',\ t') \implies R\ (Exn\ e,\ t)\ (Exn\ e',\ t')$*
assumes *lr: $\bigwedge e\ v'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Result\ v',\ t') \implies refines\ (throw\ e)\ (g'\ v')$*
 $t\ t'\ R$
assumes *rl: $\bigwedge v\ e'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exn\ e',\ t') \implies refines\ (g\ v)\ (throw\ e')\ t$*
 $t'\ R$
assumes *rr: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \implies refines\ (g\ v)\ (g'\ v')\ t$*

$t' R$
shows *refines* $(f \ggg g) (f' \ggg g') s s' R$
using *ll lr rl rr*
by (*intro refines-bind*[*OF f*])
(auto simp: Exn-def default-option-def)

lemma *refines-bind-res*:
assumes f : *refines* $f f' s s' (\lambda(\text{Res } r, t) (\text{Res } r', t')). \text{refines } (g r) (g' r') t t' R$
shows *refines* $((\text{bind } f g)::('a, 's) \text{res-monad}) ((\text{bind } f' g')::('b, 't) \text{res-monad}) s s' R$
by (*rule refines-bind*[*OF f*]) *auto*

lemma *refines-bind-res'*:
assumes f : *refines* $f f' s s' Q$
assumes g : $\bigwedge r t r' t'. Q (\text{Result } r, t) (\text{Result } r', t') \implies \text{refines } (g r) (g' r') t t' R$
shows *refines* $((\text{bind } f g)::('a, 's) \text{res-monad}) ((\text{bind } f' g')::('b, 't) \text{res-monad}) s s' R$
by (*auto intro!*: *refines-bind-res refines-weaken*[*OF f*] g)

lemma *refines-bind-bind-exn-wp*:
assumes f : *refines* $f f' s s' (\lambda(r, t) (r', t'))$.
(case r of
 $\text{Exn } e \Rightarrow (\text{case } r' \text{ of } \text{Exn } e' \Rightarrow R (\text{Exn } e, t) (\text{Exn } e', t') \mid \text{Result } v' \Rightarrow \text{refines } (\text{throw } e) (g' v') t t' R$
 $\mid \text{Result } v \Rightarrow (\text{case } r' \text{ of } \text{Exn } e' \Rightarrow \text{refines } (g v) (\text{throw } e') t t' R \mid \text{Result } v' \Rightarrow \text{refines } (g v) (g' v') t t' R))$
shows *refines* $(f \ggg g) (f' \ggg g') s s' R$
apply (*rule refines-bind'*)
apply (*rule refines-weaken* [*OF f*])
apply (*auto simp add: Exn-def default-option-def split: xval-splits*)
done

lemma *rel-spec-bind-res*:
 $\text{rel-spec } f f' s s' (\lambda(\text{Res } r, t) (\text{Res } r', t')). \text{rel-spec } (g r) (g' r') t t' R \implies$
 $\text{rel-spec } ((\text{bind } f g)::('a, 's) \text{res-monad}) ((\text{bind } f' g')::('b, 't) \text{res-monad}) s s' R$
unfolding *rel-spec-iff-refines*
by (*safe intro!*: *refines-bind-res*; (*rule refines-weaken, assumption, simp*))

lemma *rel-spec-bind-res'*:
assumes f : *rel-spec* $f f' s s' Q$
assumes g : $\bigwedge r t r' t'. Q (\text{Result } r, t) (\text{Result } r', t') \implies \text{rel-spec } (g r) (g' r') t t' R$
shows *rel-spec* $((\text{bind } f g)::('a, 's) \text{res-monad}) ((\text{bind } f' g')::('b, 't) \text{res-monad}) s s' R$
by (*auto intro!*: *rel-spec-bind-res rel-spec-mono*[*OF - f*] g)

lemma *rel-spec-bind-bind*:
assumes f : *rel-spec* $f f' s s' Q$

assumes ll: $\bigwedge e e' t t'. Q (\text{Exception } e, t) (\text{Exception } e', t') \implies e \neq \text{default} \implies e' \neq \text{default} \implies R (\text{Exception } e, t) (\text{Exception } e', t')$
 $Q (\text{Exception } e, t) (\text{Exception } e', t') \implies e \neq \text{default} \implies e' \neq \text{default} \implies R (\text{Exception } e, t) (\text{Exception } e', t')$
assumes lr: $\bigwedge e v' t t'. Q (\text{Exception } e, t) (\text{Result } v', t') \implies e \neq \text{default} \implies \text{rel-spec } (\text{yield } (\text{Exception } e)) (g' v') t t' R$
assumes rl: $\bigwedge v e' t t'. Q (\text{Result } v, t) (\text{Exception } e', t') \implies e' \neq \text{default} \implies \text{rel-spec } (g v) (\text{yield } (\text{Exception } e')) t t' R$
assumes rr: $\bigwedge v v' t t'. Q (\text{Result } v, t) (\text{Result } v', t') \implies \text{rel-spec } (g v) (g' v') t t' R$
shows $\text{rel-spec } (\text{bind } f g) (\text{bind } f' g') s s' R$
using *assms unfolding rel-spec-iff-refines*
apply (*intro conjI*)
subgoal by (*rule refines-bind[where Q=Q]*) *auto*
subgoal by (*rule refines-bind[where Q=Q⁻¹⁻¹]*) *auto*
done

lemma *rel-spec-bind-exn:*

assumes f: $\text{rel-spec } f f' s s' Q$
assumes ll: $\bigwedge e e' t t'. Q (\text{Exn } e, t) (\text{Exn } e', t') \implies R (\text{Exn } e, t) (\text{Exn } e', t')$
assumes lr: $\bigwedge e v' t t'. Q (\text{Exn } e, t) (\text{Result } v', t') \implies \text{rel-spec } (\text{throw } e) (g' v') t t' R$
assumes rl: $\bigwedge v e' t t'. Q (\text{Result } v, t) (\text{Exn } e', t') \implies \text{rel-spec } (g v) (\text{throw } e') t t' R$
assumes rr: $\bigwedge v v' t t'. Q (\text{Result } v, t) (\text{Result } v', t') \implies \text{rel-spec } (g v) (g' v') t t' R$
shows $\text{rel-spec } (\text{bind } f g) (\text{bind } f' g') s s' R$
using *ll lr rl rr*
by (*intro rel-spec-bind-bind[OF f]*)
(auto simp: Exn-def default-option-def)

lemma *refines-bind-right:*

assumes f: $\text{refines } f f' s s' Q$
assumes ll: $\bigwedge e e' t t'. Q (\text{Exception } e, t) (\text{Exception } e', t') \implies e \neq \text{default} \implies e' \neq \text{default} \implies R (\text{Exception } e, t) (\text{Exception } e', t')$
assumes lr: $\bigwedge e v' t t'. Q (\text{Exception } e, t) (\text{Result } v', t') \implies e \neq \text{default} \implies \text{refines } (\text{yield } (\text{Exception } e)) (g' v') t t' R$
assumes rl: $\bigwedge v e' t t'. Q (\text{Result } v, t) (\text{Exception } e', t') \implies e' \neq \text{default} \implies R (\text{Result } v, t) (\text{Exception } e', t')$
assumes rr: $\bigwedge v v' t t'. Q (\text{Result } v, t) (\text{Result } v', t') \implies \text{refines } (\text{return } v) (g' v') t t' R$
shows $\text{refines } f (\text{bind } f' g') s s' R$
proof –
have $\text{refines } (\text{bind } f \text{return}) (\text{bind } f' g') s s' R$
using *ll lr rl rr* **by** (*intro refines-bind[OF f]*) *auto*
then show *?thesis* **by** *simp*
qed

lemma *refines-bind-left:*

assumes f : *refines* $f f' s s' Q$
assumes ll : $\bigwedge e e' t t'. Q (\text{Exception } e, t) (\text{Exception } e', t') \implies e \neq \text{default} \implies e' \neq \text{default} \implies R (\text{Exception } e, t) (\text{Exception } e', t')$
 $Q (\text{Exception } e, t) (\text{Exception } e', t') \implies e \neq \text{default} \implies e' \neq \text{default} \implies R (\text{Exception } e, t) (\text{Exception } e', t')$
assumes lr : $\bigwedge e v' t t'. Q (\text{Exception } e, t) (\text{Result } v', t') \implies e \neq \text{default} \implies R (\text{Exception } e, t) (\text{Result } v', t')$
assumes rl : $\bigwedge v e' t t'. Q (\text{Result } v, t) (\text{Exception } e', t') \implies e' \neq \text{default} \implies \text{refines } (g v) (\text{yield } (\text{Exception } e')) t t' R$
assumes rr : $\bigwedge v v' t t'. Q (\text{Result } v, t) (\text{Result } v', t') \implies \text{refines } (g v) (\text{return } v') t t' R$
shows *refines* $(\text{bind } f g) f' s s' R$
proof –
have *refines* $(\text{bind } f g) (\text{bind } f' \text{return}) s s' R$
using $ll lr rl rr$ **by** $(\text{intro } \text{refines-bind}[OF f]) \text{auto}$
then show *?thesis* **by** *simp*
qed

lemma *rel-spec-bind-right*:
assumes f : *rel-spec* $f f' s s' Q$
assumes ll : $\bigwedge e e' t t'. Q (\text{Exception } e, t) (\text{Exception } e', t') \implies e \neq \text{default} \implies e' \neq \text{default} \implies R (\text{Exception } e, t) (\text{Exception } e', t')$
 $Q (\text{Exception } e, t) (\text{Exception } e', t') \implies e \neq \text{default} \implies e' \neq \text{default} \implies R (\text{Exception } e, t) (\text{Exception } e', t')$
assumes lr : $\bigwedge e v' t t'. Q (\text{Exception } e, t) (\text{Result } v', t') \implies e \neq \text{default} \implies \text{rel-spec } (\text{yield } (\text{Exception } e)) (g' v') t t' R$
 $Q (\text{Exception } e, t) (\text{Result } v', t') \implies e \neq \text{default} \implies \text{rel-spec } (\text{yield } (\text{Exception } e)) (g' v') t t' R$
assumes rl : $\bigwedge v e' t t'. Q (\text{Result } v, t) (\text{Exception } e', t') \implies e' \neq \text{default} \implies R (\text{Result } v, t) (\text{Exception } e', t')$
 $Q (\text{Result } v, t) (\text{Exception } e', t') \implies e' \neq \text{default} \implies R (\text{Result } v, t) (\text{Exception } e', t')$
assumes rr : $\bigwedge v v' t t'. Q (\text{Result } v, t) (\text{Result } v', t') \implies \text{rel-spec } (\text{return } v) (g' v') t t' R$
 $Q (\text{Result } v, t) (\text{Result } v', t') \implies \text{rel-spec } (\text{return } v) (g' v') t t' R$
shows *rel-spec* $f (\text{bind } f' g') s s' R$
using *assms unfolding rel-spec-iff-refines*
apply $(\text{intro } \text{conjI})$
subgoal **by** $(\text{rule } \text{refines-bind-right}[\text{where } Q=Q]) \text{auto}$
subgoal **by** $(\text{rule } \text{refines-bind-left}[\text{where } Q=Q^{-1-1}]) \text{auto}$
done

lemma *refines-bind-handle-left'*:
 $f \cdot s \{ \lambda v s'. (\forall r. v = \text{Result } r \longrightarrow \text{refines } (g r) k s' t R) \wedge (\forall e. e \neq \text{default} \longrightarrow v = \text{Exception } e \longrightarrow \text{refines } (h e) k s' t R) \} \implies \text{refines } (\text{bind-handle } f g h) k s t R$
by $(\text{auto } \text{simp}: \text{runs-to.rep-eq } \text{refines.rep-eq } \text{run-bind-handle } \text{split-beta}' \text{ac-simps } \text{intro!}: \text{sim-bind-post-state-left } \text{split}: \text{prod.splits } \text{exception-or-result-splits})$

lemma *refines-bind-left-res*:
 $f \cdot s \{ \lambda \text{Res } r s'. \text{refines } (g r) h s' t R \} \implies \text{refines } (f >>= g) h s t R$
unfolding *bind-def* **by** $(\text{rule } \text{refines-bind-handle-left}') \text{simp}$

lemma *refines-bind-left-exn*:
 $f \cdot s \{ \lambda r s'. (\forall a. r = \text{Result } a \longrightarrow \text{refines } (g a) h s' t R) \wedge (\forall e. r = \text{Exn } e \longrightarrow \text{refines } (\text{throw } e) h s' t R) \} \implies$

$refines (f \gg= g) h s t R$
unfolding *bind-def*
by (*rule refines-bind-handle-left'*)
(auto simp add: Exn-def default-option-def imp-ex)

lemma *runs-to-partial-bind1*:
 $(\bigwedge r s. (g r) \cdot s \{P\}) \implies ((f \gg= g)::('a, 's) res-monad) \cdot s \{P\}$
apply (*rule runs-to-partial-bind*)
apply (*rule runs-to-partial-weaken[OF runs-to-partial-True]*)
apply *auto*
done

lemma *unknown-bind-const[simp]*: $unknown \gg= (\lambda x. f) = f$
by (*rule spec-monad-ext*) (*simp add: run-bind*)

lemma *bind-cong-left*:
fixes $f::('e::default, 'a, 's) spec-monad$
shows $(\bigwedge r. g r = g' r) \implies (f \gg= g) = (f \gg= g')$
by (*rule spec-monad-ext*) (*simp add: run-bind*)

lemma *bind-cong-right*:
fixes $f::('e::default, 'a, 's) spec-monad$
shows $f = f' \implies (f \gg= g) = (f' \gg= g)$
by (*rule spec-monad-ext*) (*simp add: run-bind*)

lemma *rel-spec-bind-res''*:
fixes $f::('a, 's) res-monad$
shows $f \cdot s \{ \lambda Res r t. rel-spec (g r) (g' r) t t R \} \implies rel-spec (f \gg= g) (f \gg= g') s s R$
by (*intro rel-spec-bind-res rel-spec-refl'*) (*auto simp: split-beta'*)

lemma *rel-spec-monad-bind-rel-exception-or-result*:
assumes $mn: rel-spec-monad R (rel-exception-or-result E P) m n$
and $fg: rel-fun P (rel-spec-monad R (rel-exception-or-result E Q)) f g$
shows $rel-spec-monad R (rel-exception-or-result E Q) (m \gg= f) (n \gg= g)$
by (*intro rel-spec-monadI rel-spec-bind-bind[OF rel-spec-monadD[OF mn]]*)
(simp-all add: fg[THEN rel-funD, THEN rel-spec-monadD])

lemma *rel-spec-monad-bind-rel-exception-or-result'*:
 $(\bigwedge x. rel-spec-monad (=) (rel-exception-or-result (=) Q) (f x) (g x)) \implies$
 $rel-spec-monad (=) (rel-exception-or-result (=) Q) (m \gg= f) (n \gg= g)$
apply (*rule rel-spec-monad-bind-rel-exception-or-result[where P=(=)]*)
apply (*auto simp add: rel-exception-or-result-eq-conv rel-spec-monad-eq-conv*)
done

lemma *rel-spec-monad-bind*:
 $rel-spec-monad R (\lambda Res v1 Res v2. P v1 v2) m n \implies rel-fun P (rel-spec-monad R Q) f g \implies$
 $rel-spec-monad R Q (m \gg= f) (n \gg= g)$

apply (*clarsimp simp add: rel-spec-monad-iff-rel-spec[abs-def] rel-fun-def*
intro!: rel-spec-bind-res)
subgoal premises *prems* **for** *s t*
by (*rule rel-spec-mono[OF - prems(1)[rule-format, OF prems(3)]]*)
(*clarsimp simp: le-fun-def prems(2)[rule-format]*)
done

lemma *rel-spec-monad-bind-left*:
assumes *mn: rel-spec-monad R (λRes v1 Res v2. P v1 v2) m n*
and *fg: rel-fun P (rel-spec-monad R Q) f return*
shows *rel-spec-monad R Q (m >>= f) n*
proof –
from *mn fg*
have *rel-spec-monad R Q (m >>= f) (n >>= return)*
by (*rule rel-spec-monad-bind*)
then show *?thesis*
by (*simp*)
qed

lemma *rel-spec-monad-bind'*:
fixes *m:: ('a, 's) res-monad*
shows $(\bigwedge x. \text{rel-spec-monad } (=) Q (f x) (g x)) \implies \text{rel-spec-monad } (=) Q (m >>= f) (m >>= g)$
by (*rule rel-spec-monad-bind[where R=(=) and P=(=)]*)
(*auto simp: rel-fun-def rel-spec-monad-iff-rel-spec intro!: rel-spec-refl*)

lemma *run-bind-cong-simple*:
 $\text{run } f s = \text{run } f' s \implies (\bigwedge v s. \text{run } (g v) s = \text{run } (g' v) s) \implies$
 $\text{run } (\text{bind } f g) s = \text{run } (\text{bind } f' g') s$
by (*simp add: run-bind*)

lemma *run-bind-cong-simple-same-head*: $(\bigwedge v s. \text{run } (g v) s = \text{run } (g' v) s) \implies$
 $\text{run } (\text{bind } f g) s = \text{run } (\text{bind } f g') s$
by (*rule run-bind-cong-simple*) *auto*

lemma *refines-bind-same*:
 $\text{refines } (f >>= g) (f >>= g') s s R \text{ if } f \cdot s \Downarrow \lambda \text{Res } y t. \text{refines } (g y) (g' y) t t R$
 \Downarrow
apply (*rule refines-bind-res*)
apply (*rule refines-same-runs-toI*)
apply (*rule runs-to-weaken[OF that]*)
apply *auto*
done

lemma *refines-bind-handle-right-runs-to-partialI*:
 $g \cdot t \Downarrow \lambda r t'. (\forall e. e \neq \text{default} \longrightarrow r = \text{Exception } e \longrightarrow \text{refines } f (k e) s t' R) \wedge$
 $(\forall a. r = \text{Result } a \longrightarrow \text{refines } f (h a) s t' R) \Downarrow \implies \text{always-progress } g \implies$
 $\text{refines } f (\text{bind-handle } g h k) s t R$
by *transfer*

(*auto simp: prod-eq-iff split-beta'*
intro!: sim-bind-post-state-right
split: exception-or-result-splits prod.splits)

lemma *refines-bind-right-runs-to-partialI*:

$g \cdot t \text{ ?}\{\} \lambda r t'. (\forall e. e \neq \text{default} \longrightarrow r = \text{Exception } e \longrightarrow$
 $\text{refines } f \text{ (throw-exception-or-result } e) s t' R) \wedge$
 $(\forall a. r = \text{Result } a \longrightarrow \text{refines } f \text{ (h } a) s t' R) \}\} \Longrightarrow \text{always-progress } g \Longrightarrow$
 $\text{refines } f \text{ (bind } g \text{ h) } s t R$

unfolding *bind-def* **by** (rule *refines-bind-handle-right-runs-to-partialI*)

lemma *refines-bind-right-runs-toI*:

$g \cdot t \{\} \lambda \text{Res } r t'. \text{refines } f \text{ (h } r) s t' R \}\} \Longrightarrow \text{always-progress } g \Longrightarrow$
 $\text{refines } f \text{ (g } \gg= \text{ h) } s t R$

unfolding *bind-def*

by (rule *refines-bind-handle-right-runs-to-partialI*)

(*auto intro: runs-to-partial-of-runs-to*)

lemma *refines-bind-left-refine*:

$\text{refines } (f \gg= g) h s t R$

if $\text{refines } f f' s s (=) \text{refines } (f' \gg= g) h s t R$

by (rule *refines-trans[OF refines-bind[OF that(1)] that(2), where R=(=)]*)

(*auto simp add: refines-refl*)

lemma *refines-bind-right-single*:

assumes x : *pure-post-state* ($\text{Result } x, u) \leq \text{run } g t$ **and** h : $\text{refines } f \text{ (h } x) s u R$

shows $\text{refines } f \text{ (g } \gg= \text{ h) } s t R$

apply (rule *refines-le-run-trans[OF h]*)

apply (*simp add: run-bind*)

apply (rule *order-trans[OF - mono-bind-post-state, OF - x order-refl]*)

apply *simp*

done

lemma *return-let-bind*: $(\text{return } (\text{let } v = f' \text{ in } (g' v))) = \text{do } \{v \leftarrow \text{return } f'; \text{return } (g' v)\}$

apply (rule *spec-monad-eqI*)

apply (*auto simp add: runs-to-iff*)

done

lemma *rel-spec-monad-rel-xval-bind*:

assumes $f-f'$: *rel-spec-monad* S (*rel-xval* $L P$) $f f'$

assumes Res-Res : $\bigwedge v v'. P v v' \Longrightarrow \text{rel-spec-monad } S \text{ (rel-xval } L R) (g v) (g' v')$

shows $\text{rel-spec-monad } S \text{ (rel-xval } L R) (f \gg= g) (f' \gg= g')$

apply (*intro rel-spec-monadI rel-spec-bind-exn*)

apply (rule $f-f'$ [*THEN rel-spec-monadD*], *assumption*)

using Res-Res [*THEN rel-spec-monadD*]

apply *auto*

done

lemma *rel-spec-monad-rel-xval-same-bind*:
assumes $f-f'$: *rel-spec-monad* S (*rel-xval* L R) f f'
assumes *Res-Res*: $\bigwedge v v'. R\ v\ v' \implies \text{rel-spec-monad } S\ (\text{rel-xval } L\ R)\ (g\ v)\ (g'\ v')$
shows *rel-spec-monad* S (*rel-xval* L R) $(f \ggg g)$ $(f' \ggg g')$
using *assms* **by** (*rule rel-spec-monad-rel-xval-bind*)

lemma *rel-spec-monad-rel-xval-result-eq-bind*:
assumes $f-f'$: *rel-spec-monad* S (*rel-xval* L $(=)$) f f'
assumes *Res-Res*: $\bigwedge v. \text{rel-spec-monad } S\ (\text{rel-xval } L\ (=))\ (g\ v)\ (g'\ v)$
shows *rel-spec-monad* S (*rel-xval* L $(=)$) $(f \ggg g)$ $(f' \ggg g')$
apply (*rule rel-spec-monad-rel-xval-bind* [*OF f-f'*])
subgoal using *Res-Res* **by** *auto*
done

lemma *rel-spec-monad-fail*: *rel-spec-monad* Q R *fail* *fail*
by (*auto simp add: rel-spec-monad-def rel-set-def*)

lemma *bind-fail[simp]*: *fail* \ggg X = *fail*
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind*)
done

16.9.17 *assert*

lemma *assert-simps[simp]*:
assert *True* = *return* $()$
assert *False* = *top*
by (*auto simp add: assert-def*)

lemma *always-progress-assert[always-progress-intros]*: *always-progress* (*assert* P)
by (*simp add: always-progress-def assert-def*)

lemma *run-assert[run-spec-monad]*:
run (*assert* P) s = (*if* P *then* *pure-post-state* (*Result* $()$, s) *else* \top)
by (*simp add: assert-def*)

lemma *runs-to-assert-iff[simp]*: *assert* $P \cdot s \{\!\!| Q \!\!\}$ $\longleftrightarrow P \wedge Q$ (*Result* $()$) s
by (*simp add: runs-to.rep-eq run-assert*)

lemma *runs-to-assert[runs-to-vcg]*: $P \implies Q$ (*Result* $()$) $s \implies \text{assert } P \cdot s \{\!\!| Q \!\!\}$
by *simp*

lemma *runs-to-partial-assert-iff[simp]*: *assert* $P \cdot s \{?\!\!| Q \!\!\} \longleftrightarrow (P \longrightarrow Q$ (*Result* $()$) s)
by (*simp add: runs-to-partial.rep-eq run-assert*)

lemma *refines-top-iff[simp]*: *refines* \top $g\ s\ t\ R \longleftrightarrow \text{run } g\ t = \top$
by *transfer auto*

lemma *refines-assert*:
refines (assert P) (assert Q) s t R \longleftrightarrow $(Q \longrightarrow P \wedge R \text{ (Result ()), } s \text{ (Result (), } t))$
by (*simp add: assert-def*)

16.9.18 *assume*

lemma *assume-simps*[*simp*]:
assume True = return ()
assume False = bot
by (*auto simp add: assume-def*)

lemma *always-progress-assume*[*always-progress-intros*]: $P \Longrightarrow \text{always-progress (assume P)}$
by (*simp add: always-progress-def assume-def*)

lemma *run-assume*[*run-spec-monad*]:
run (assume P) s = (if P then pure-post-state (Result (), s) else \perp)
by (*simp add: assume-def*)

lemma *run-assume-simps*[*run-spec-monad, simp*]:
 $P \Longrightarrow \text{run (assume P) s = pure-post-state (Result (), s)}$
 $\neg P \Longrightarrow \text{run (assume P) s = } \perp$
by (*simp-all add: run-assume*)

lemma *runs-to-assume-iff*[*simp*]: $\text{assume P} \cdot s \Downarrow Q \Downarrow \longleftrightarrow (P \longrightarrow Q \text{ (Result ()) } s)$
by (*simp add: runs-to.rep-eq run-assume*)

lemma *runs-to-partial-assume-iff*[*simp*]: $\text{assume P} \cdot s \Downarrow\!\!\Downarrow Q \Downarrow \longleftrightarrow (P \longrightarrow Q \text{ (Result ()) } s)$
by (*simp add: runs-to-partial.rep-eq run-assume*)

lemma *runs-to-assume*[*runs-to-vcg*]: $(P \Longrightarrow Q \text{ (Result ()) } s) \Longrightarrow \text{assume P} \cdot s \Downarrow Q \Downarrow$
by *simp*

16.9.19 *assume-outcome*

lemma *run-assume-outcome*[*run-spec-monad, simp*]: $\text{run (assume-outcome f) s = Success (f s)}$
apply *transfer*
apply *simp*
done

lemma *always-progress-assume-outcome*[*always-progress-intros*]:
 $(\bigwedge s. f s \neq \{\}) \Longrightarrow \text{always-progress (assume-outcome f)}$
by (*auto simp add: always-progress-def bot-post-state-def*)

lemma *runs-to-assume-outcome*[*runs-to-vcg*]:
 (*assume-outcome* f) · $s \{ \lambda r t. (r, t) \in f s \}$
by (*simp add: runs-to.rep-eq*)

lemma *runs-to-assume-outcome-iff*[*runs-to-iff*]:
 (*assume-outcome* f) · $s \{ Q \} \longleftrightarrow (\forall (r, t) \in f s. Q r t)$
by (*auto simp add: runs-to.rep-eq*)

lemma *runs-to-partial-assume-outcome*[*runs-to-vcg*]:
 (*assume-outcome* f) · $s ? \{ \lambda r t. (r, t) \in f s \}$
by (*simp add: runs-to-partial.rep-eq*)

lemma *assume-outcome-elementary*:
assume-outcome $f = \text{do } \{ s \leftarrow \text{get-state}; (r, t) \leftarrow \text{select } (f s); \text{set-state } t; \text{yield } r \}$
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind Sup-Success pure-post-state-def*)
done

16.9.20 *assume-result-and-state*

lemma *run-assume-result-and-state*[*run-spec-monad, simp*]:
run (*assume-result-and-state* f) $s = \text{Success } ((\lambda(v, t). (\text{Result } v, t)) \text{ ` } f s)$
by (*auto simp add: assume-result-and-state-def run-bind Sup-Success-pair pure-post-state-def*)

lemma *always-progress-assume-result-and-state*[*always-progress-intros*]:
 ($\bigwedge s. f s \neq \{ \}$) \implies *always-progress* (*assume-result-and-state* f)
by (*simp add: always-progress-def*)

lemma *runs-to-assume-result-and-state*[*runs-to-vcg*]:
assume-result-and-state $f \cdot s \{ \lambda r t. \exists v. r = \text{Result } v \wedge (v, t) \in f s \}$
by (*auto simp add: runs-to.rep-eq*)

lemma *runs-to-assume-result-and-state-iff*[*runs-to-iff*]:
assume-result-and-state $f \cdot s \{ Q \} \longleftrightarrow (\forall (v, t) \in f s. Q (\text{Result } v) t)$
by (*auto simp add: runs-to.rep-eq*)

lemma *runs-to-partial-assume-result-and-state*[*runs-to-vcg*]:
assume-result-and-state $f \cdot s ? \{ \lambda r t. \exists v. r = \text{Result } v \wedge (v, t) \in f s \}$
by (*auto simp add: runs-to-partial.rep-eq*)

lemma *refines-assume-result-and-state-right*:
 $f \cdot s \{ \lambda r s'. \exists r' t'. (r', t') \in g t \wedge R (r, s') (\text{Result } r', t') \} \implies$
refines f (*assume-result-and-state* g) $s t R$
by (*simp add: refines.rep-eq runs-to.rep-eq sim-post-state-Success2 split-beta' Bex-def*)

lemma *refines-assume-result-and-state*:
sim-set $R (P s) (Q t) \implies$
refines (*assume-result-and-state* P) (*assume-result-and-state* Q) $s t$
 ($\lambda(\text{Res } v, s) (\text{Res } w, t). R (v, s) (w, t)$)

by (force simp: refines.rep-eq sim-set-def)

lemma *refines-assume-result-and-state-iff*:

refines (assume-result-and-state *A*) (assume-result-and-state *B*) *s t Q* \longleftrightarrow
sim-set ($\lambda(v, s') (w, t'). Q (Result\ v, s') (Result\ w, t')$) (*A s*) (*B t*)

apply (simp add: refines.rep-eq, intro iffI)

subgoal

by (fastforce simp add: sim-set-def)

subgoal

by (force simp add: sim-set-def)

done

16.9.21 *gets*

lemma *run-gets*[*run-spec-monad, simp*]: *run* (*gets f*) *s* = *pure-post-state* (*Result* (*f s*), *s*)

by (simp add: gets-def run-bind)

lemma *always-progress-gets*[*always-progress-intros*]: *always-progress* (*gets f*)

by (simp add: always-progress-def)

lemma *runs-to-gets*[*runs-to-vcg*]: *gets f* · *s* $\{\!\!\{ \lambda r\ t. r = Result\ (f\ s) \wedge t = s \}\!\!\}$

by (simp add: runs-to.rep-eq)

lemma *runs-to-gets-iff*[*runs-to-iff*]: *gets f* · *s* $\{\!\!\{ Q \}\!\!\} \longleftrightarrow Q (Result\ (f\ s))\ s$

by (simp add: runs-to.rep-eq)

lemma *runs-to-partial-gets*[*runs-to-vcg*]: *gets f* · *s* $\{\!\!\{ \lambda r\ t. r = Result\ (f\ s) \wedge t = s \}\!\!\}$

by (simp add: runs-to-partial.rep-eq)

lemma *refines-gets*: *R* (*Result* (*f s*), *s*) (*Result* (*g t*), *t*) \implies *refines* (*gets f*) (*gets g*) *s t R*

by (auto simp add: refines.rep-eq)

lemma *rel-spec-gets*: *R* (*Result* (*f s*), *s*) (*Result* (*g t*), *t*) \implies *rel-spec* (*gets f*) (*gets g*) *s t R*

by (auto simp add: rel-spec-iff-refines intro: refines-gets)

lemma *runs-to-always-progress-to-gets*:

$(\bigwedge s. f \cdot s \{\!\!\{ \lambda r\ t. t = s \wedge r = Result\ (v\ s) \}\!\!\}) \implies$ *always-progress f* $\implies f = gets\ v$

apply (clarsimp simp: spec-monad-ext-iff runs-to.rep-eq always-progress.rep-eq)

subgoal **premises** *prems* **for** *s*

using *prems*[*rule-format*, of *s*]

by (cases *run f s*; auto simp add: pure-post-state-def Ball-def)

done

lemma *gets-let-bind*: (*gets* ($\lambda s. let\ v = f'\ s\ in\ (g'\ v\ s)$)) = *do* {*v* \leftarrow *gets f'*; *gets* (*g' v*)}

apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

16.9.22 *assert-result-and-state*

lemma *run-assert-result-and-state*[*run-spec-monad*]:

run (assert-result-and-state f) s =
(if f s = {} then \top else Success (($\lambda(v, t).$ (Result v, t)) ‘ f s))
by (*auto simp add: assert-result-and-state-def run-bind pure-post-state-def Sup-Success-pair*)

lemma *always-progress-assert-result-and-state*[*always-progress-intros*]:

always-progress (assert-result-and-state f)
by (*auto simp add: always-progress-def run-assert-result-and-state*)

lemma *runs-to-assert-result-and-state-iff*[*runs-to-iff*]:

assert-result-and-state f \cdot s $\{\!\!|$ Q $\!\!\}$ \longleftrightarrow (f s \neq {}) \wedge ($\forall (v, t) \in f s.$ Q (Result v) t))
by (*auto simp add: runs-to.rep-eq run-assert-result-and-state*)

lemma *runs-to-assert-result-and-state*[*runs-to-vcg*]:

f s \neq {} \implies assert-result-and-state f \cdot s $\{\!\!|$ $\lambda r t. \exists v. r = \text{Result } v \wedge (v, t) \in f s$ $\!\!\}$
by (*simp add: runs-to-assert-result-and-state-iff*)

lemma *runs-to-partial-assert-result-and-state*[*runs-to-vcg*]:

assert-result-and-state f \cdot s $\{\!\!|$ $\lambda r t. \exists v. r = \text{Result } v \wedge (v, t) \in f s$ $\!\!\}$
by (*auto simp add: runs-to-partial.rep-eq run-assert-result-and-state*)

lemma *runs-to-state-select*[*runs-to-vcg*]:

$\exists t. (s, t) \in R \implies \text{state-select } R \cdot s \{\!\!|$ $\lambda r t. r = \text{Result } () \wedge (s, t) \in R$ $\!\!\}$

by (*auto intro!: runs-to-assert-result-and-state[THEN runs-to-weaken]*)

lemma *runs-to-partial-state-select*[*runs-to-vcg*]:

state-select R \cdot s $\{\!\!|$ $\lambda r t. r = \text{Result } () \wedge (s, t) \in R$ $\!\!\}$
by (*simp add: runs-to-partial.rep-eq run-assert-result-and-state*)

lemma *refines-assert-result-and-state*:

assumes *sim*: ($\bigwedge r' s'. (r', s') \in f s \implies (\exists v' t'. (v', t') \in g t \wedge R (\text{Result } r', s') (\text{Result } v', t'))$)

assumes *emp*: $f s = \{\}$ \implies $g t = \{\}$

shows *refines* (*assert-result-and-state f*) (*assert-result-and-state g*) *s t R*

using *sim emp* **by** (*fastforce simp add: refines-iff-runs-to runs-to-iff*)

lemma *refines-state-select*:

assumes *sim*: ($\bigwedge s'. (s, s') \in f \implies (\exists t'. (t, t') \in g \wedge R (\text{Result } (), s') (\text{Result } (), t'))$)

assumes *emp*: $\nexists s'. (s, s') \in f \implies \nexists t'. (t, t') \in g$

shows *refines* (*state-select f*) (*state-select g*) *s t R*

using *sim emp* **by** (*intro refines-assert-result-and-state*) *auto*

16.9.23 *assuming*

lemma *run-assuming*[*run-spec-monad*]:

run (assuming g) s = (if g s then pure-post-state (Result (), s) else ⊥)

by (*simp add: assuming-def run-bind*)

lemma *always-progress-assuming*[*always-progress-intros*]: *always-progress (assuming g) ↔ (∀ s. g s)*

by (*auto simp add: always-progress-def run-assuming*)

lemma *runs-to-assuming*[*runs-to-vcg*]: *assuming g · s {λr t. g s ∧ r = Result () ∧ t = s}*

by (*simp add: runs-to.rep-eq run-assuming*)

lemma *runs-to-assuming-iff*[*runs-to-iff*]: *assuming g · s {Q} ↔ (g s → Q (Result ()) s)*

by (*auto simp add: runs-to.rep-eq run-assuming*)

lemma *runs-to-partial-assuming*[*runs-to-vcg*]:

assuming g · s ?{λr t. g s ∧ r = Result () ∧ t = s ∧ g s}

by (*simp add: runs-to-partial.rep-eq run-assuming*)

lemma *assuming-state-assume*:

assuming P = assume-result-and-state (λs. (if P s then {((), s)} else {}))

by (*simp add: spec-monad-eq-iff runs-to-iff*)

lemma *assuming-True*[*simp*]: *assuming (λs. True) = skip*

by (*simp add: spec-monad-eq-iff runs-to-iff*)

lemma *refines-assuming*:

*(P s ⇒ Q t) ⇒ (P s ⇒ Q t ⇒ R (Result (), s) (Result (), t)) ⇒
refines (assuming P) (assuming Q) s t R*

by (*auto simp add: refines.rep-eq run-assuming*)

lemma *rel-spec-assuming*:

*(Q t ↔ P s) ⇒ (P s ⇒ Q t ⇒ R (Result (), s) (Result (), t)) ⇒
rel-spec (assuming P) (assuming Q) s t R*

by (*auto simp add: rel-spec-def run-assuming rel-set-def*)

lemma *refines-bind-assuming-right*:

P t ⇒ (P t ⇒ refines f (g ()) s t R) ⇒ refines f (assuming P ≫= g) s t R

by (*simp add: refines.rep-eq run-assuming run-bind*)

lemma *refines-bind-assuming-left*:

(P s ⇒ refines (f ()) g s t R) ⇒ refines (assuming P >>= f) g s t R

by (*simp add: refines.rep-eq run-assuming run-bind*)

16.9.24 *guard*

lemma *run-guard*[*run-spec-monad*]:

run (guard *g*) *s* = (if *g s* then pure-post-state (Result (), *s*) else \top)
by (simp add: guard-def run-bind)

lemma *always-progress-guard*[*always-progress-intros*]: *always-progress* (guard *g*)
by (auto simp add: always-progress-def run-guard)

lemma *runs-to-guard*[*runs-to-vcg*]: $g\ s \Longrightarrow \text{guard } g \cdot s \ \{\!\! \{ \lambda r\ t.\ r = \text{Result } () \wedge t = s \}\!\!\}$
by (simp add: runs-to.rep-eq run-guard)

lemma *runs-to-guard-iff*[*runs-to-iff*]: $\text{guard } g \cdot s \ \{\!\! \{ Q \}\!\!\} \longleftrightarrow (g\ s \wedge Q\ (\text{Result } ())\ s)$
by (auto simp add: runs-to.rep-eq run-guard)

lemma *runs-to-partial-guard*[*runs-to-vcg*]: $\text{guard } g \cdot s \ ?\!\!\{ \lambda r\ t.\ r = \text{Result } () \wedge t = s \wedge g\ s \}$
by (simp add: runs-to-partial.rep-eq run-guard)

lemma *refines-guard*:
 $(Q\ t \Longrightarrow P\ s) \Longrightarrow (P\ s \Longrightarrow Q\ t \Longrightarrow R\ (\text{Result } ())\ s\ (\text{Result } ())\ t) \Longrightarrow$
refines (guard *P*) (guard *Q*) *s t R*
by (auto simp add: refines.rep-eq run-guard)

lemma *rel-spec-guard*:
 $(Q\ t \longleftrightarrow P\ s) \Longrightarrow (P\ s \Longrightarrow Q\ t \Longrightarrow R\ (\text{Result } ())\ s\ (\text{Result } ())\ t) \Longrightarrow$
rel-spec (guard *P*) (guard *Q*) *s t R*
by (auto simp add: rel-spec-def run-guard rel-set-def)

lemma *refines-bind-guard-right*:
refines *f* (guard *P* \ggg *g*) *s t R* **if** $P\ t \Longrightarrow \text{refines } f\ (g\ ())\ s\ t\ R$
using *that*
by (auto simp: refines.rep-eq run-guard run-bind)

lemma *guard-False-fail*: *guard* ($\lambda\ \cdot.\ \text{False}$) = *fail*
by (simp add: spec-monad-ext run-guard)

lemma *rel-spec-monad-bind-guard*:
shows $(\bigwedge x.\ \text{rel-spec-monad } (=)\ Q\ (f\ x)\ (g\ x)) \Longrightarrow$
rel-spec-monad (=) *Q* (guard *P* \ggg *f*) (guard *P* \ggg *g*)
by (auto simp add: rel-spec-monad-def run-bind run-guard)

lemma *runs-to-guard-bind-iff*: $((\text{guard } P\ \ggg\ f) \cdot s \ \{\!\! \{ Q \}\!\!\}) \longleftrightarrow P\ s \wedge ((f\ ()) \cdot s \ \{\!\! \{ Q \}\!\!\})$
by (simp add: runs-to-iff)

lemma *refines-bind-guard-right-iff*:
refines *f* (guard *P* \ggg *g*) *s t R* $\longleftrightarrow (P\ t \longrightarrow \text{refines } f\ (g\ ())\ s\ t\ R)$
by (auto simp: refines.rep-eq run-guard run-bind)

lemma *refines-bind-guard-right-end*:

assumes $f\text{-}g$: *refines* $f\ g\ s\ t\ R$
shows *refines* f (*do* { $res \leftarrow g$; *guard* G ; *return* res }) $s\ t$
 $(\lambda(r, s)\ (q, t). R\ (r, s)\ (q, t) \wedge$
 $(\text{case } q \text{ of } \text{Exception } e \Rightarrow \text{True} \mid \text{Result } - \Rightarrow G\ t))$
apply (*subst* *bind-return*[*symmetric*])
apply (*rule* *refines-bind*[*OF* $f\text{-}g$])
apply (*auto* *simp*: *refines-bind-guard-right-iff*)
done

lemma *refines-bind-guard-right-end'*:
assumes $f\text{-}g$: *refines* $f\ g\ s\ t\ R$
shows *refines* f (*do* { $res \leftarrow g$; *guard* ($G\ res$); *return* res }) $s\ t$
 $(\lambda(r, s)\ (q, t). R\ (r, s)\ (q, t) \wedge$
 $(\text{case } q \text{ of } \text{Exception } e \Rightarrow \text{True} \mid \text{Result } v \Rightarrow G\ v\ t))$
apply (*subst* *bind-return*[*symmetric*])
apply (*rule* *refines-bind*[*OF* $f\text{-}g$])
apply (*auto* *simp*: *refines-bind-guard-right-iff*)
done

16.9.25 *assert-opt*

lemma *run-assert-opt*[*run-spec-monad*, *simp*]:
 $\text{run } (\text{assert-opt } x) s = (\text{case } x \text{ of } \text{Some } v \Rightarrow \text{pure-post-state } (\text{Result } v, s) \mid \text{None}$
 $\Rightarrow \top)$
by (*simp* *add*: *assert-opt-def* *run-bind* *split*: *option.splits*)

lemma *always-progress-assert-opt*[*always-progress-intros*]: *always-progress* (*assert-opt* x)
by (*simp* *add*: *always-progress-intros* *assert-opt-def* *split*: *option.splits*)

lemma *runs-to-assert-opt*[*runs-to-vcg*]: $(\exists v. x = \text{Some } v \wedge Q\ (\text{Result } v)\ s) \Longrightarrow$
 $\text{assert-opt } x \cdot s \{Q\}$
by (*auto* *simp* *add*: *runs-to.rep-eq*)

lemma *runs-to-assert-opt-iff*[*runs-to-iff*]:
 $\text{assert-opt } x \cdot s \{Q\} \longleftrightarrow (\exists v. x = \text{Some } v \wedge Q\ (\text{Result } v)\ s)$
by (*auto* *simp* *add*: *runs-to.rep-eq* *split*: *option.split*)

lemma *runs-to-partial-assert-opt*[*runs-to-vcg*]:
 $(\bigwedge v. x = \text{Some } v \Longrightarrow Q\ (\text{Result } v)\ s) \Longrightarrow \text{assert-opt } x \cdot s \{?Q\}$
by (*auto* *simp* *add*: *runs-to-partial.rep-eq* *split*: *option.split*)

16.9.26 *gets-the*

lemma *run-gets-the'*:
 $\text{run } (\text{gets-the } f) s = (\text{case } f\ s \text{ of } \text{Some } v \Rightarrow \text{pure-post-state } (\text{Result } v, s) \mid \text{None}$
 $\Rightarrow \top)$
by (*simp* *add*: *gets-the-def* *run-bind* *top-post-state-def* *pure-post-state-def* *split*:
option.split)

lemma *run-gets-the*[*run-spec-monad, simp*]:
 $run\ (gets\ the\ f)\ s = (case\ (f\ s)\ of\ Some\ v \Rightarrow\ pure\ post\ state\ (Result\ v,\ s)\ | \ None \Rightarrow \top)$
by (*simp add: gets-the-def run-bind*)

lemma *always-progress-gets-the*[*always-progress-intros*]: *always-progress* (*gets-the f*)
by (*simp add: always-progress-intros gets-the-def split: option.splits*)

lemma *runs-to-gets-the*[*runs-to-vcg*]: $(\exists v. f\ s = Some\ v \wedge Q\ (Result\ v)\ s) \Longrightarrow gets\ the\ f \cdot s \ \{\!\! \{ Q \}\!\!\}$
by (*auto simp add: runs-to.rep-eq*)

lemma *runs-to-gets-the-iff*[*runs-to-iff*]:
 $gets\ the\ f \cdot s \ \{\!\! \{ Q \}\!\!\} \longleftrightarrow (\exists v. f\ s = Some\ v \wedge Q\ (Result\ v)\ s)$
by (*auto simp add: runs-to.rep-eq split: option.split*)

lemma *runs-to-partial-gets-the*[*runs-to-vcg*]:
 $(\bigwedge v. f\ s = Some\ v \Longrightarrow Q\ (Result\ v)\ s) \Longrightarrow gets\ the\ f \cdot s \ \{\!\! \{ Q \}\!\!\}$
by (*auto simp add: runs-to-partial.rep-eq split: option.split*)

16.9.27 *modify*

lemma *run-modify*[*run-spec-monad, simp*]: $run\ (modify\ f)\ s = pure\ post\ state\ (Result\ (),\ f\ s)$
by (*simp add: modify-def run-bind*)

lemma *always-progress-modify*[*always-progress-intros*]: *always-progress* (*modify f*)
by (*simp add: modify-def always-progress-intros*)

lemma *runs-to-modify*[*runs-to-vcg*]: $modify\ f \cdot s \ \{\!\! \{ \lambda r\ t. r = Result\ () \wedge t = f\ s \}\!\!\}$
by (*simp add: runs-to.rep-eq*)

lemma *runs-to-modify-res*[*runs-to-vcg*]: $((modify\ f)::(unit,\ 's)\ res\ monad) \cdot s \ \{\!\! \{ \lambda r\ t. t = f\ s \}\!\!\}$
by (*simp add: runs-to.rep-eq*)

lemma *runs-to-modify-iff*[*runs-to-iff*]: $modify\ f \cdot s \ \{\!\! \{ Q \}\!\!\} \longleftrightarrow Q\ (Result\ ())\ (f\ s)$
by (*simp add: runs-to.rep-eq*)

lemma *runs-to-partial-modify*[*runs-to-vcg*]: $modify\ f \cdot s \ \{\!\! \{ \lambda r\ t. r = Result\ () \wedge t = f\ s \}\!\!\}$
by (*simp add: runs-to-partial.rep-eq*)

lemma *runs-to-partial-modify-res*[*runs-to-vcg*]:
 $((modify\ f)::(unit,\ 's)\ res\ monad) \cdot s \ \{\!\! \{ \lambda r\ t. t = f\ s \}\!\!\}$
by (*simp add: runs-to-partial.rep-eq*)

lemma *refines-modify*:

R (*Result* (), f s) (*Result* (), g t) \implies *refines* (*modify* f) (*modify* g) s t R
by (*auto simp add: refines.rep-eq*)

lemma *rel-spec-modify*:

R (*Result* (), f s) (*Result* (), g t) \implies *rel-spec* (*modify* f) (*modify* g) s t R
by (*auto simp add: rel-spec-iff-refines intro: refines-modify*)

16.9.28 condition

lemma *run-condition[run-spec-monad]*: *run* (*condition* c f g) s = (*if* c s *then* *run* f s *else* *run* g s)

by (*simp add: condition-def run-bind*)

lemma *run-condition-True[run-spec-monad, simp]*: c s \implies *run* (*condition* c f g) s = *run* f s

by (*simp add: run-condition*)

lemma *run-condition-False[run-spec-monad, simp]*: $\neg c$ s \implies *run* (*condition* c f g) s = *run* g s

by (*simp add: run-condition*)

lemma *always-progress-condition[always-progress-intros]*:

always-progress f \implies *always-progress* g \implies *always-progress* (*condition* c f g)
by (*auto simp add: always-progress-def run-condition*)

lemma *condition-swap*: (*condition* C A B) = (*condition* ($\lambda s. \neg C$ s) B A)

by (*rule spec-monad-ext*) (*clarsimp simp add: run-condition*)

lemma *condition-fail-rhs*: (*condition* C X *fail*) = (*guard* C $>>=$ ($\lambda-. X$))

by (*rule spec-monad-ext*) (*simp add: run-bind run-guard run-condition*)

lemma *condition-fail-lhs*: (*condition* C *fail* X) = (*guard* ($\lambda s. \neg C$ s) $>>=$ ($\lambda-. X$))

by (*metis condition-fail-rhs condition-swap*)

lemma *condition-bind-fail[simp]*:

(*condition* C A B $>>=$ ($\lambda-. fail$)) = *condition* C (A $>>=$ ($\lambda-. fail$)) (B $>>=$ ($\lambda-. fail$))

apply (*rule spec-monad-ext*)

apply (*clarsimp simp add: run-condition run-bind*)

done

lemma *condition-True[simp]*: *condition* ($\lambda-. True$) f g = f

apply (*rule spec-monad-ext*)

apply (*clarsimp simp add: run-condition run-bind*)

done

lemma *condition-False[simp]*: *condition* ($\lambda-. False$) f g = g

apply (*rule spec-monad-ext*)

apply (*clarsimp simp add: run-condition run-bind*)

done

lemma *le-condition-runI*:

$(\bigwedge s. c\ s \implies \text{run } h\ s \leq \text{run } f\ s) \implies (\bigwedge s. \neg c\ s \implies \text{run } h\ s \leq \text{run } g\ s)$
 $\implies h \leq \text{condition } c\ f\ g$
by (*simp add: le-spec-monad-runI run-condition*)

lemma *mono-condition-spec-monad*:

$\text{mono } T \implies \text{mono } F \implies \text{mono } (\lambda x. \text{condition } C\ (F\ x)\ (T\ x))$
by (*auto simp: condition-def intro!: mono-bind-spec-monad mono-const*)

lemma *mono-condition*: $f \leq f' \implies g \leq g' \implies \text{condition } c\ f\ g \leq \text{condition } c\ f'\ g'$

by (*simp add: le-fun-def less-eq-spec-monad.rep-eq run-condition*)

lemma *monotone-condition-le*[*partial-function-mono*]:

$\text{monotone } R\ (\leq)\ (\lambda f'. f\ f') \implies (\text{monotone } R\ (\leq)\ (\lambda f'. g\ f'))$
 $\implies \text{monotone } R\ (\leq)\ (\lambda f'. \text{condition } c\ (f\ f')\ (g\ f'))$
by (*auto simp add: monotone-def intro!: mono-condition*)

lemma *monotone-condition-ge*[*partial-function-mono*]:

$\text{monotone } R\ (\geq)\ (\lambda f'. f\ f') \implies (\text{monotone } R\ (\geq)\ (\lambda f'. g\ f'))$
 $\implies \text{monotone } R\ (\geq)\ (\lambda f'. \text{condition } c\ (f\ f')\ (g\ f'))$
by (*auto simp add: monotone-def intro!: mono-condition*)

lemma *runs-to-condition*[*runs-to-vcg*]:

$(c\ s \implies f \cdot s \ \{\!\! \{ Q \}\!\! \}) \implies (\neg c\ s \implies g \cdot s \ \{\!\! \{ Q \}\!\! \}) \implies \text{condition } c\ f\ g \cdot s \ \{\!\! \{ Q \}\!\! \}$
by (*simp add: runs-to.rep-eq run-condition*)

lemma *runs-to-condition-iff*[*runs-to-iff*]:

$\text{condition } c\ f\ g \cdot s \ \{\!\! \{ Q \}\!\! \} \longleftrightarrow (\text{if } c\ s \text{ then } f \cdot s \ \{\!\! \{ Q \}\!\! \} \text{ else } g \cdot s \ \{\!\! \{ Q \}\!\! \})$
by (*simp add: runs-to.rep-eq run-condition*)

lemma *runs-to-partial-condition*[*runs-to-vcg*]:

$(c\ s \implies f \cdot s \ \{\!\! \{ Q \}\!\! \}) \implies (\neg c\ s \implies g \cdot s \ \{\!\! \{ Q \}\!\! \}) \implies \text{condition } c\ f\ g \cdot s \ \{\!\! \{ Q \}\!\! \}$
by (*simp add: runs-to-partial.rep-eq run-condition*)

lemma *refines-condition-iff*:

assumes $c' s' \longleftrightarrow c\ s$
shows $\text{refines } (\text{condition } c\ f\ g)\ (\text{condition } c'\ f'\ g')\ s\ s'\ R \longleftrightarrow$
 $(\text{if } c'\ s' \text{ then } \text{refines } f\ f'\ s\ s'\ R \text{ else } \text{refines } g\ g'\ s\ s'\ R)$
using *assms*
by (*auto simp add: refines.rep-eq run-condition*)

lemma *refines-condition*:

$P\ s \longleftrightarrow P'\ s' \implies$
 $(P\ s \implies P'\ s' \implies \text{refines } f\ f'\ s\ s'\ R) \implies$
 $(\neg P\ s \implies \neg P'\ s' \implies \text{refines } g\ g'\ s\ s'\ R) \implies$
 $\text{refines } (\text{condition } P\ f\ g)\ (\text{condition } P'\ f'\ g')\ s\ s'\ R$

using *refines-condition-iff*
by *metis*

lemma *refines-condition-TrueI*:

assumes $c' s' = c s$ **and** $c' s'$ *refines* $f f' s s' R$
shows *refines* (*condition* $c f g$) (*condition* $c' f' g'$) $s s' R$
by (*simp add: refines-condition-iff*[**where** $c'=c'$ **and** $c=c$ **and** $s'=s'$ **and** $s=s$,
OF assms(1)] *assms(2, 3)*)

lemma *refines-condition-FalseI*:

assumes $c' s' = c s$ **and** $\neg c' s'$ *refines* $g g' s s' R$
shows *refines* (*condition* $c f g$) (*condition* $c' f' g'$) $s s' R$
by (*simp add: refines-condition-iff*[**where** $c'=c'$ **and** $c=c$ **and** $s'=s'$ **and** $s=s$,
OF assms(1)] *assms(2, 3)*)

lemma *refines-condition-bind-left*:

refines (*condition* $C T F \gg X$) $Y s t R \longleftrightarrow$
 $(C s \longrightarrow \text{refines } (T \gg X) Y s t R) \wedge (\neg C s \longrightarrow \text{refines } (F \gg X) Y s t R)$
by (*simp add: refines.rep-eq run-bind run-condition*)

lemma *refines-condition-bind-right*:

refines X (*condition* $C T F \gg Y$) $s t R \longleftrightarrow$
 $(C t \longrightarrow \text{refines } X (T \gg Y) s t R) \wedge (\neg C t \longrightarrow \text{refines } X (F \gg Y) s t R)$
by (*simp add: refines.rep-eq run-bind run-condition*)

lemma *rel-spec-condition-iff*:

assumes $c' s' \longleftrightarrow c s$
shows *rel-spec* (*condition* $c f g$) (*condition* $c' f' g'$) $s s' R \longleftrightarrow$
(if $c' s'$ *then* *rel-spec* $f f' s s' R$ *else* *rel-spec* $g g' s s' R$ *)*
using *assms*
by (*auto simp add: rel-spec-def run-condition*)

lemma *rel-spec-condition*:

$P s \longleftrightarrow P' s' \implies$
 $(P s \implies P' s' \implies \text{rel-spec } f f' s s' R) \implies$
 $(\neg P s \implies \neg P' s' \implies \text{rel-spec } g g' s s' R) \implies$
 $\text{rel-spec } (\text{condition } P f g) (\text{condition } P' f' g') s s' R$
using *rel-spec-condition-iff*
by *metis*

lemma *rel-spec-condition-TrueI*:

assumes $c' s' = c s$ **and** $c' s'$ *rel-spec* $f f' s s' R$
shows *rel-spec* (*condition* $c f g$) (*condition* $c' f' g'$) $s s' R$
by (*simp add: rel-spec-condition-iff*[**where** $c'=c'$ **and** $c=c$ **and** $s'=s'$ **and** $s=s$,
OF assms(1)] *assms(2, 3)*)

lemma *rel-spec-condition-FalseI*:

assumes $c' s' = c s$ **and** $\neg c' s'$ *rel-spec* $g g' s s' R$
shows *rel-spec* (*condition* $c f g$) (*condition* $c' f' g'$) $s s' R$

by (*simp add: rel-spec-condition-iff*[**where** $c'=c'$ **and** $c=c$ **and** $s'=s'$ **and** $s=s$,
OF assms(1)] *assms(2, 3)*)

lemma *refines-condition-left*:

$(P\ s \Longrightarrow \text{refines } f\ h\ s\ t\ R) \Longrightarrow (\neg P\ s \Longrightarrow \text{refines } g\ h\ s\ t\ R) \Longrightarrow$
 $\text{refines } (\text{condition } P\ f\ g)\ h\ s\ t\ R$
by (*simp add: refines.rep-eq run-condition*)

lemma *rel-spec-condition-left*:

$(P\ s \Longrightarrow \text{rel-spec } f\ h\ s\ t\ R) \Longrightarrow (\neg P\ s \Longrightarrow \text{rel-spec } g\ h\ s\ t\ R) \Longrightarrow$
 $\text{rel-spec } (\text{condition } P\ f\ g)\ h\ s\ t\ R$
by (*auto simp add: rel-spec-def run-condition*)

lemma *refines-condition-true*:

$P\ t \Longrightarrow \text{refines } f\ g\ s\ t\ R \Longrightarrow \text{refines } f\ (\text{condition } P\ g\ h)\ s\ t\ R$
by (*simp add: refines.rep-eq run-condition*)

lemma *rel-spec-condition-true*:

$P\ t \Longrightarrow \text{rel-spec } f\ g\ s\ t\ R \Longrightarrow$
 $\text{rel-spec } f\ (\text{condition } P\ g\ h)\ s\ t\ R$
by (*auto simp add: rel-spec-def run-condition*)

lemma *refines-condition-false*:

$\neg P\ t \Longrightarrow \text{refines } f\ h\ s\ t\ R \Longrightarrow$
 $\text{refines } f\ (\text{condition } P\ g\ h)\ s\ t\ R$
by (*simp add: refines.rep-eq run-condition*)

lemma *rel-spec-condition-false*:

$\neg P\ t \Longrightarrow \text{rel-spec } f\ h\ s\ t\ R \Longrightarrow$
 $\text{rel-spec } f\ (\text{condition } P\ g\ h)\ s\ t\ R$
by (*auto simp add: rel-spec-def run-condition*)

lemma *condition-bind*:

$(\text{condition } P\ f\ g \gg\gg h) = \text{condition } P\ (f \gg\gg h)\ (g \gg\gg h)$
by (*simp add: spec-monad-ext-iff run-condition run-bind*)

lemma *rel-spec-monad-condition*:

assumes *rel-fun* $R (=) P\ P'$
and *rel-spec-monad* $R\ Q\ f\ f'$
and *rel-spec-monad* $R\ Q\ g\ g'$
shows *rel-spec-monad* $R\ Q\ (\text{condition } P\ f\ g)\ (\text{condition } P'\ f'\ g')$
using *assms*
by (*auto simp add: rel-spec-monad-def run-condition rel-fun-def*)

lemma *rel-spec-monad-condition-const*:

$P \longleftrightarrow P' \Longrightarrow (P \Longrightarrow \text{rel-spec-monad } R\ Q\ f\ f') \Longrightarrow$
 $(\neg P \Longrightarrow \text{rel-spec-monad } R\ Q\ g\ g') \Longrightarrow$
 $\text{rel-spec-monad } R\ Q\ (\text{condition } (\lambda\cdot. P)\ f\ g)\ (\text{condition } (\lambda\cdot. P')\ f'\ g')$
by (*cases P*) (*auto simp add: condition-def*)

16.9.29 *when*

lemma *run-when*[*run-spec-monad*]:

run (when c f) s = (if c then run f s else pure-post-state (Result (), s))

unfolding *when-def* **by** *simp*

lemma *always-progress-when*[*always-progress-intros*]:

always-progress f \implies always-progress (when c f)

unfolding *when-def* **by** (*simp add: always-progress-intros*)

lemma *runs-to-when*[*runs-to-vcg*]:

(c \implies f · s $\{ Q \}$) \implies (\neg c \implies Q (Result ()) s) \implies when c f · s $\{ Q \}$

by (*auto simp add: runs-to.rep-eq run-when*)

lemma *runs-to-when-iff*[*runs-to-iff*]:

(when c f) · s $\{ Q \} \longleftrightarrow$ (if c then f · s $\{ Q \}$ else Q (Result ()) s)

by (*auto simp add: runs-to.rep-eq run-when*)

lemma *runs-to-partial-when*[*runs-to-vcg*]:

(c \implies f · s $? \{ Q \}$) \implies (\neg c \implies Q (Result ()) s) \implies when c f · s $? \{ Q \}$

by (*auto simp add: runs-to-partial.rep-eq run-when*)

lemma *mono-when*: *f \leq f' \implies when c f \leq when c f'*

unfolding *when-def* **by** (*simp add: mono-condition*)

lemma *monotone-when-le*[*partial-function-mono*]:

monotone R (\leq) ($\lambda f'. f f'$)

\implies *monotone R (\leq) ($\lambda f'. \text{when } c (f f')$)*

unfolding *when-def* **by** (*simp add: monotone-condition-le*)

lemma *monotone-when-ge*[*partial-function-mono*]:

monotone R (\geq) ($\lambda f'. f f'$)

\implies *monotone R (\geq) ($\lambda f'. \text{when } c (f f')$)*

unfolding *when-def* **by** (*simp add: monotone-condition-ge*)

lemma *when-True*[*simp*]: *when True f = f*

apply (*rule spec-monad-ext*)

apply (*simp add: run-when*)

done

lemma *when-False*[*simp*]: *when False f = return ()*

apply (*rule spec-monad-ext*)

apply (*simp add: run-when*)

done

16.9.30 **While**

context *fixes* *C* :: '*a* \Rightarrow '*s* \Rightarrow *bool* **and** *B* :: '*a* \Rightarrow (*e*::*default*, '*a*, '*s*) *spec-monad*
begin

definition *whileLoop* :: 'a ⇒ ('e, 'a, 's) spec-monad **where**

whileLoop =

gfp (λW a. *condition* (C a) (*bind* (B a) W) (*return* a))

— Collapses to *Failure* in case of any non terminating computation.

definition *whileLoop-finite* :: 'a ⇒ ('e, 'a, 's) spec-monad **where**

whileLoop-finite =

lfp (λW a. *condition* (C a) (*bind* (B a) W) (*return* a))

— Does not collapse to *Failure* in presence of a non terminating computation.

Failure can still occur when the body fails in some iteration. It captures the outcomes of all terminating and thus finite computations.

inductive *whileLoop-terminates* :: 'a ⇒ 's ⇒ bool **where**

step: $\bigwedge a s. (C a s \implies B a \cdot s \ \S \ \lambda v s. \forall a. v = \text{Result } a \longrightarrow \text{whileLoop-terminates } a s \ \S) \implies$

whileLoop-terminates a s

— This is weaker than *run* (*whileLoop* a) s ≠ ⊤: as it uses partial correctness

lemma *mono-whileLoop-functional*:

mono (λW a. *condition* (C a) (*bind* (B a) W) (*return* a))

by (*intro mono-app mono-lam mono-const mono-bind-spec-monad mono-condition-spec-monad*)

lemma *whileLoop-unroll*:

whileLoop a =

condition (C a) (*bind* (B a) *whileLoop*) (*return* a)

unfolding *whileLoop-def*

by (*subst gfp-unfold[OF mono-whileLoop-functional]*) *simp*

lemma *whileLoop-finite-unfold*:

whileLoop-finite a =

condition (C a) (*bind* (B a) *whileLoop-finite*) (*return* a)

unfolding *whileLoop-finite-def*

by (*subst lfp-unfold[OF mono-whileLoop-functional]*) *simp*

lemma *whileLoop-ne-Failure*:

(C a s ⇒ B a · s § λx s. ∀ a. x = Result a ⇒ run (whileLoop a) s ≠ Failure §) ⇒

run (whileLoop a) s ≠ Failure

by (*subst whileLoop-unroll*)

(*simp add: run-condition run-bind-eq-top-iff flip: top-post-state-def*)

lemma *whileLoop-ne-top-induct*[*consumes 1, case-names step*]:

assumes a-s: run (whileLoop a) s ≠ ⊤

and *step*: $\bigwedge a s. (C a s \implies B a \cdot s \ \S \ \lambda x s. \forall a. x = \text{Result } a \longrightarrow P a s \ \S) \implies$

P a s

shows *P a s*

proof —

have (λa. *Spec* (λs. if *P a s* then ⊥ else ⊤)) ≤ *whileLoop*

unfolding *whileLoop-def*

by (intro gfp-upperbound)
 (auto intro: step runs-to-weaken
 simp: le-fun-def less-eq-spec-monad.rep-eq run-condition top-unique run-bind-eq-top-iff)
 with a-s show ?thesis
 by (auto simp add: le-fun-def less-eq-spec-monad.rep-eq top-unique top-post-state-def
 split: if-splits)

qed

lemma runs-to-whileLoop:
 assumes R : wf R
 assumes *: I (Result a) s
 assumes P -Result: $\bigwedge a s. \neg C a s \implies I$ (Result a) $s \implies P$ (Result a) s
 assumes P -Exception: $\bigwedge a s. a \neq \text{default} \implies I$ (Exception a) $s \implies P$ (Exception
 a) s
 assumes B : $\bigwedge a s. C a s \implies I$ (Result a) $s \implies$
 $B a \cdot s \{ \lambda r t. I r t \wedge (\forall b. r = \text{Result } b \longrightarrow ((b, t), (a, s)) \in R) \}$
 shows whileLoop $a \cdot s \{ P \}$
proof (use R * in <induction $x \equiv (a, s)$ arbitrary: $a s$)
 case (less $a s$)
 show ?case
 by (auto simp: P -Result P -Exception whileLoop-unroll[$of a$]
 runs-to-condition-iff runs-to-bind-iff less
 intro!: runs-to-weaken[$OF B$])

qed

lemma runs-to-whileLoop-finite:
 assumes *: I (Result a) s
 assumes P -Result: $\bigwedge a s. \neg C a s \implies I$ (Result a) $s \implies P$ (Result a) s
 assumes P -Exception: $\bigwedge a s. a \neq \text{default} \implies I$ (Exception a) $s \implies P$ (Exception
 a) s
 assumes B : $\bigwedge a s. C a s \implies I$ (Result a) $s \implies B a \cdot s \{ I \}$
 shows whileLoop-finite $a \cdot s \{ P \}$
proof –
 have whileLoop-finite $a \cdot s$
 $\{ \lambda x s. I x s \wedge (\forall a. x = \text{Result } a \longrightarrow C a s \longrightarrow P (\text{Result } a) s) \}$
 unfolding whileLoop-finite-def
 apply (rule runs-to-lfp[OF mono-whileLoop-functional, of $\lambda a. I$ (Result a), OF
 *])
 subgoal premises prems for $W x s$
 using prems(2)
 by (intro runs-to-condition runs-to-bind runs-to-while-iff
 runs-to-weaken[$OF B$] runs-to-weaken[OF prems(1)] conjI allI impI)
 simp-all
 done
 then show ?thesis
 apply (rule runs-to-weaken)
 subgoal for $r s$
 by (cases r ; cases $\exists a. r = \text{Result } a \wedge C a s$; simp add: P -Result P -Exception)
 done

qed

lemma *runs-to-partial-whileLoop-finite*:

assumes *: I (Result a) s

assumes B : $\bigwedge a s. C a s \implies I$ (Result a) $s \implies (B a) \cdot s \{I\}$

assumes P -Result: $\bigwedge a s. \neg C a s \implies I$ (Result a) $s \implies P$ (Result a) s

assumes P -Exception: $\bigwedge a s. a \neq \text{default} \implies I$ (Exception a) $s \implies P$ (Exception a) s

shows $\text{whileLoop-finite } a \cdot s \{P\}$

proof –

have $\text{whileLoop-finite } a \cdot s \{P\}$ **if** **: $\text{run } (\text{whileLoop-finite } a) s \neq \top$

apply (rule *runs-to-whileLoop-finite* **where**

$I = \lambda x s. I x s \wedge (\forall a. x = \text{Result } a \longrightarrow \text{run } (\text{whileLoop-finite } a) s \neq \top)$)

apply (auto simp: * ** P -Result P -Exception) $[\beta]$

subgoal for $a s$ **using** B [of $a s$]

by (subst (asm) *whileLoop-finite-unfold*)

(auto simp: *runs-to-conj* *run-bind-eq-top-iff* dest: *runs-to-of-runs-to-partial-runs-to*)

done

then show *?thesis*

by (auto simp: *runs-to-partial-alt* *top-post-state-def*)

qed

lemma *whileLoop-finite-eq-whileLoop-of-whileLoop-terminates*:

assumes *whileLoop-terminates* $a s$

shows $\text{run } (\text{whileLoop-finite } a) s = \text{run } (\text{whileLoop } a) s$

proof (use *assms* in *induction*)

case (step $a s$)

show *?case*

apply (subst *whileLoop-unroll*)

apply (subst *whileLoop-finite-unfold*)

apply (auto simp: *run-condition* *intro!*: *run-bind-cong* [*OF refl* *runs-to-partial-weaken*, *OF step*])

done

qed

lemma *whileLoop-terminates-of-succeeds*:

$\text{run } (\text{whileLoop } a) s \neq \top \implies \text{whileLoop-terminates } a s$

by (*induction rule*: *whileLoop-ne-top-induct*)

(auto *intro*: *whileLoop-terminates.step* *runs-to-partial-of-runs-to*)

lemma *whileLoop-finite-eq-whileLoop*:

$\text{run } (\text{whileLoop } a) s \neq \top \implies \text{run } (\text{whileLoop-finite } a) s = \text{run } (\text{whileLoop } a) s$

by (rule *whileLoop-finite-eq-whileLoop-of-whileLoop-terminates* [*OF* *whileLoop-terminates-of-succeeds*])

lemma *runs-to-whileLoop-of-runs-to-whileLoop-finite-if-terminates*:

$\text{whileLoop-terminates } i s \implies \text{whileLoop-finite } i \cdot s \{Q\} \implies \text{whileLoop } i \cdot s \{Q\}$

unfolding *runs-to.rep-eq* *whileLoop-finite-eq-whileLoop-of-whileLoop-terminates*

.

```

lemma whileLoop-finite-le-whileLoop: whileLoop-finite a ≤ whileLoop a
  using le-fun-def lfp-le-gfp mono-whileLoop-functional
  unfolding whileLoop-finite-def whileLoop-def
  by fastforce

lemma runs-to-partial-whileLoop-finite-whileLoop:
  whileLoop-finite i · s ?[Q] ⇒ whileLoop i · s ?[Q]
  by (cases run (whileLoop i) s = ⊤)
    (auto simp:
      runs-to.rep-eq runs-to-partial-alt whileLoop-finite-eq-whileLoop top-post-state-def)

lemma runs-to-partial-whileLoop:
  assumes I (Result a) s
  assumes  $\bigwedge a s. \neg C a s \implies I (Result a) s \implies P (Result a) s$ 
  assumes  $\bigwedge a s. a \neq default \implies I (Exception a) s \implies P (Exception a) s$ 
  assumes  $\bigwedge a s. C a s \implies I (Result a) s \implies (B a) \cdot s ?[I]$ 
  shows whileLoop a · s ?[P]
  using assms(1,4,2,3)
  by (rule runs-to-partial-whileLoop-finite-whileLoop[OF runs-to-partial-whileLoop-finite])

lemma always-progress-whileLoop[always-progress-intros]:
  assumes B: ( $\bigwedge v. always-progress (B v)$ )
  shows always-progress (whileLoop a)
  unfolding always-progress.rep-eq
proof
  fix s have run (whileLoop a) s ≠ ⊥ if *: run (whileLoop a) s ≠ ⊤
  proof (use * in <induction rule: whileLoop-ne-top-induct>)
    case (step a s) then show ?case
      apply (subst whileLoop-unroll)
      apply (simp add: run-condition run-bind bind-post-state-eq-bot prod-eq-iff
runs-to.rep-eq
        split: prod.splits exception-or-result-splits)
      apply clarsimp
      subgoal premises prems
        using holds-post-state-combine[OF prems(1,3), where R=λx. False] B
      by (auto simp: holds-post-state-False exception-or-result-nchotomy always-progress.rep-eq)
      done
    qed
  then show run (whileLoop a) s ≠ ⊥
    by (cases run (whileLoop a) s = Failure) (auto simp: bot-post-state-def)
  qed

lemma runs-to-partial-whileLoop-cond-false:
  (whileLoop I) · s ?[λr t. ∀ a. r = Result a ⟶ ¬ C a t]
  by (rule runs-to-partial-whileLoop[of λ-. True]) simp-all

end

```

context

fixes R

fixes $C :: 'a \Rightarrow 's \Rightarrow \text{bool}$ **and** $B :: 'a \Rightarrow ('e::\text{default}, 'a, 's)$ *spec-monad*

and $C' :: 'b \Rightarrow 't \Rightarrow \text{bool}$ **and** $B' :: 'b \Rightarrow ('f::\text{default}, 'b, 't)$ *spec-monad*

assumes $C: \bigwedge x s x' s'. R (\text{Result } x, s) (\text{Result } x', s') \Longrightarrow C x s \longleftrightarrow C' x' s'$

assumes $B: \bigwedge x s x' s'. R (\text{Result } x, s) (\text{Result } x', s') \Longrightarrow C x s \Longrightarrow C' x' s' \Longrightarrow$
refines $(B x) (B' x') s s' R$

assumes $R: \bigwedge v s v' s'. R (v, s) (v', s') \Longrightarrow (\exists x. v = \text{Result } x) \longleftrightarrow (\exists x'. v' = \text{Result } x')$

begin

lemma *refines-whileLoop-finite-strong*:

assumes $x-x': R (\text{Result } x, s) (\text{Result } x', s')$

shows *refines* $(\text{whileLoop-finite } C B x) (\text{whileLoop-finite } C' B' x') s s'$

$(\lambda(r, s) (r', s'). (\forall v. r = \text{Result } v \longrightarrow (\exists v'. r' = \text{Result } v' \wedge \neg C v s \wedge \neg C' v' s')) \wedge$

$R (r, s) (r', s'))$

(is *refines* $--- ?R$)

proof $-$

let $?P = \lambda p. \forall x s x' s'. R (\text{Result } x, s) (\text{Result } x', s') \longrightarrow$

refines $(p x) (\text{whileLoop-finite } C' B' x') s s' ?R$

have $?P (\text{whileLoop-finite } C B)$

unfolding *whileLoop-finite-def*[*of* $C B$]

proof (*rule* *lfp-ordinal-induct*[*OF* *mono-whileLoop-functional*], *intro* *allI* *impI*)

fix $S x s x' s'$ **assume** $S: ?P S$ **and** $x-x': R (\text{Result } x, s) (\text{Result } x', s')$

from $x-x'$ **show**

refines $(\text{condition } (C x) (B x \gg S) (\text{return } x)) (\text{whileLoop-finite } C' B' x') s s' ?R$

by (*subst* *whileLoop-finite-unfold*)

(*auto* *dest*: R *simp*: C [*OF* $x-x'$] *refines-condition-iff*

intro!: *refines-bind'*[*OF* *refines-mono*[*OF* $- B$ [*OF* $x-x'$]]] S [*rule-format*])

qed (*simp* *add*: *refines-Sup1*)

with $x-x'$ **show** *?thesis* **by** *simp*

qed

lemma *refines-whileLoop-finite*:

assumes $x-x': R (\text{Result } x, s) (\text{Result } x', s')$

shows *refines* $(\text{whileLoop-finite } C B x) (\text{whileLoop-finite } C' B' x') s s' R$

by (*rule* *refines-mono*[*OF* $- \text{refines-whileLoop-finite-strong}$, *OF* $- x-x'$]) *simp*

lemma *whileLoop-succeeds-terminates-of-refines*:

assumes *run* $(\text{whileLoop } C' B' x') s' \neq \top$

shows $R (\text{Result } x, s) (\text{Result } x', s') \Longrightarrow \text{run } (\text{whileLoop } C B x) s \neq \text{Failure}$

proof (*use* *assms* **in** *induction* *arbitrary*: $x s$ *rule*: *whileLoop-ne-top-induct*)

case (*step* $a' s' a s$)

show *?case*

proof (*rule* *whileLoop-ne-Failure*)

```

assume  $C\ a\ s$ 
with  $C[\text{of } a\ s\ a'\ s']\ \text{step.premis}$  have  $C'\ a'\ s'$  by simp

show  $B\ a \cdot s\ \{\!\! \{ \lambda x\ s.\ \forall a.\ x = \text{Result } a \longrightarrow \text{run } (\text{whileLoop } C\ B\ a)\ s \neq \text{Failure}$ 
 $\}\}$ 
  using  $\text{step.IH}[\text{OF } \langle C'\ a'\ s'\rangle]\ B[\text{OF } \text{step.premis } \langle C\ a\ s\rangle\ \langle C'\ a'\ s'\rangle]$ 
  by (rule runs-to-refines) (metis R)
qed
qed

```

lemma *refines-whileLoop-strong*:

```

assumes  $x\ x':\ R\ (\text{Result } x,\ s)\ (\text{Result } x',\ s')$ 
shows  $\text{refines } (\text{whileLoop } C\ B\ x)\ (\text{whileLoop } C'\ B'\ x')\ s\ s'$ 
   $(\lambda(r,\ s)\ (r',\ s').\ (\forall v.\ r = \text{Result } v \longrightarrow (\exists v'.\ r' = \text{Result } v' \wedge \neg C\ v\ s \wedge \neg C'$ 
 $v'\ s')) \wedge$ 
   $R\ (r,\ s)\ (r',\ s'))$ 
apply (cases run (whileLoop C' B' x') s' = Failure)
subgoal by (simp add: refines.rep-eq)
subgoal
  using
    whileLoop-succeeds-terminates-of-refines[OF - x-x']
    refines-whileLoop-finite-strong[OF x-x']
  by (simp add: refines.rep-eq whileLoop-finite-eq-whileLoop top-post-state-def)
done

```

lemma *refines-whileLoop*:

```

assumes  $x\ x':\ R\ (\text{Result } x,\ s)\ (\text{Result } x',\ s')$ 
shows  $\text{refines } (\text{whileLoop } C\ B\ x)\ (\text{whileLoop } C'\ B'\ x')\ s\ s'\ R$ 
by (rule refines-mono[OF - refines-whileLoop-strong, OF - x-x']) simp

```

end

lemma *runs-to-whileLoop-res'*:

```

assumes  $R:\ \text{wf } R$ 
assumes  $*$ :  $I\ a\ s$ 
assumes  $P\text{-Result}:\ \bigwedge a\ s.\ \neg C\ a\ s \Longrightarrow I\ a\ s \Longrightarrow P\ (\text{Result } a)\ s$ 
assumes  $B:\ \bigwedge a\ s.\ C\ a\ s \Longrightarrow I\ a\ s \Longrightarrow$ 
   $B\ a \cdot s\ \{\!\! \{ \lambda r\ t.\ (\forall b.\ r = \text{Result } b \longrightarrow I\ b\ t \wedge ((b,\ t),\ (a,\ s)) \in R) \}\}$ 
shows  $(\text{whileLoop } C\ B\ a::('a,\ 's)\ \text{res-monad}) \cdot s\ \{\!\! \{ P \}\}$ 
apply (rule runs-to-whileLoop[OF R, where I = \lambda Exception - \Rightarrow (\lambda-. False) |
 $\text{Result } v \Rightarrow I\ v$ ])
subgoal using  $*$  by auto
subgoal using  $P\text{-Result}$  by auto
subgoal by auto
subgoal for  $a\ b$  by (rule B[of a b, THEN runs-to-weaken]) auto
done

```

lemma *runs-to-whileLoop-res*:

```

assumes  $B:\ \bigwedge a\ s.\ C\ a\ s \Longrightarrow I\ a\ s \Longrightarrow$ 

```

$B a \cdot s \{ \lambda Res r t. I r t \wedge ((r, t), (a, s)) \in R \}$
assumes P -Result: $\bigwedge a s. I a s \implies \neg C a s \implies P a s$
assumes R : wf R
assumes *: $I a s$
shows $(whileLoop C B a::('a, 's) res-monad) \cdot s \{ \lambda Res r. P r \}$
apply $(rule runs-to-whileLoop-res' [where R = R and I = I and B = B, OF R *])$
using B P -Result **by** *auto*

lemma *runs-to-whileLoop-variant-res*:

assumes I : $\bigwedge r s c. I r s c \implies$
 $C r s \implies (B r) \cdot s \{ \lambda Res q t. \exists c'. I q t c' \wedge (c', c) \in R \}$
assumes Q : $\bigwedge r s c. I r s c \implies \neg C r s \implies Q r s$
assumes R : wf R
shows $I r s c \implies (whileLoop C B r::('a, 's) res-monad) \cdot s \{ \lambda Res r. Q r \}$
proof (*induction arbitrary*: $r s$ rule: wf-induct[$OF R$])
case $(1 c)$ **show** ?*case*
apply (*subst whileLoop-unroll*)
supply I [**where** $c=c$, *runs-to-vcg*]
apply (*runs-to-vcg*)
subgoal by (*simp add: 1*)
subgoal using 1 **by** *simp*
subgoal using $Q 1$ **by** *blast*
done

qed

lemma *runs-to-whileLoop-inc-res*:

assumes *: $\bigwedge i. i < M \implies (B (F i)) \cdot (S i) \{ \lambda Res r' t'. r' = (F (Suc i)) \wedge t' = (S (Suc i)) \}$
and [*simp*]: $\bigwedge i. i \leq M \implies C (F i) (S i) \longleftrightarrow i < M$
and [*simp*]: $i = F 0 \ si = S 0 \ t = F M \ st = S M$
shows $(whileLoop C B i::('a, 's) res-monad) \cdot si \{ \lambda Res r' t'. r' = t \wedge t' = st \}$
apply (*rule runs-to-whileLoop-variant-res*[**where** $I=\lambda r t i. r = F i \wedge t = S i \wedge i \leq M$ and $c=0$,
 $OF - - wf-nat-bound[of M]$])
apply *simp-all*
apply (*rule runs-to-weaken*[$OF *$])
apply *auto*
done

lemma *runs-to-whileLoop-dec-res*:

assumes *: $\bigwedge i::nat. i > 0 \implies i \leq M \implies$
 $(B (F i)) \cdot (S i) \{ \lambda Res r' t'. r' = (F (i - 1)) \wedge t' = (S (i - 1)) \}$
and [*simp*]: $\bigwedge i. C (F i) (S i) \longleftrightarrow i > 0$
and [*simp*]: $i = F M \ si = S M \ t = F 0 \ st = S 0$
shows $(whileLoop C B i::('a, 's) res-monad) \cdot si \{ \lambda Res r' t'. r' = t \wedge t' = st \}$
apply (*rule runs-to-whileLoop-variant-res*[**where** $I=\lambda r t i. r = F i \wedge t = S i \wedge$

$i \leq M$ and $c=M$,
 $OF - - wf\text{-measure}[of \lambda x. x]]$
apply (*fastforce intro!*: *runs-to-weaken*[$OF *$])+
done

lemma *runs-to-whileLoop-exn*:

assumes R : *wf* R
assumes $*$: $I (Result\ a)\ s$
assumes $P\text{-Result}$: $\bigwedge a\ s. \neg C\ a\ s \implies I (Result\ a)\ s \implies P (Result\ a)\ s$
assumes $P\text{-Exn}$: $\bigwedge a\ s. I (Exn\ a)\ s \implies P (Exn\ a)\ s$
assumes B : $\bigwedge a\ s. C\ a\ s \implies I (Result\ a)\ s \implies$
 $B\ a \cdot s \{ \lambda r\ t. I\ r\ t \wedge (\forall b. r = Result\ b \longrightarrow ((b, t), (a, s)) \in R) \}$
shows *whileLoop* $C\ B\ a \cdot s \{ P \}$
apply (*rule runs-to-whileLoop*[**where** $I = I, OF\ R\ *$])
subgoal using $P\text{-Result}$ **by** *auto*
subgoal using $P\text{-Exn}$ **by** (*auto simp add: default-option-def Exn-def*)
subgoal using B **by** *blast*
done

lemma *runs-to-whileLoop-exn'*:

assumes B : $\bigwedge a\ s. I (Result\ a)\ s \implies C\ a\ s \implies$
 $B\ a \cdot s \{ \lambda r\ t. I\ r\ t \wedge (\forall b. r = Result\ b \longrightarrow ((b, t), (a, s)) \in R) \}$
assumes $P\text{-Result}$: $\bigwedge a\ s. I (Result\ a)\ s \implies \neg C\ a\ s \implies P (Result\ a)\ s$
assumes $P\text{-Exn}$: $\bigwedge a\ s. I (Exn\ a)\ s \implies P (Exn\ a)\ s$
assumes R : *wf* R
assumes $*$: $I (Result\ a)\ s$
shows *whileLoop* $C\ B\ a \cdot s \{ P \}$
by (*rule runs-to-whileLoop-exn*[**where** $R=R$ and $I=I$ and $B=B$ and $C=C$ and
 $P=P$,
 $OF\ R\ * P\text{-Result}\ P\text{-Exn}\ B$])

lemma *runs-to-partial-whileLoop-res*:

assumes $*$: $I\ a\ s$
assumes $P\text{-Result}$: $\bigwedge a\ s. \neg C\ a\ s \implies I\ a\ s \implies P (Result\ a)\ s$
assumes B : $\bigwedge a\ s. C\ a\ s \implies I\ a\ s \implies$
 $(B\ a) \cdot s \{ \lambda r\ t. (\forall b. r = Result\ b \longrightarrow I\ b\ t) \}$
shows (*whileLoop* $C\ B\ a::('a, 's)\ res\ monad) \cdot s \{ P \}$
apply (*rule runs-to-partial-whileLoop*[**where** $I = \lambda Exception\ - \Rightarrow (\lambda-. False) |$
 $Result\ v \Rightarrow I\ v$])
subgoal using $*$ **by** *auto*
subgoal using $P\text{-Result}$ **by** *auto*
subgoal by *auto*
subgoal by (*rule* $B[THEN\ runs\ to\ partial\ weaken]$) *auto*
done

lemma *runs-to-partial-whileLoop-exn*:

assumes $*$: $I (Result\ a)\ s$
assumes $P\text{-Result}$: $\bigwedge a\ s. \neg C\ a\ s \implies I (Result\ a)\ s \implies P (Result\ a)\ s$

assumes $P\text{-Exn}$: $\bigwedge a s. I (Exn a) s \implies P (Exn a) s$
assumes B : $\bigwedge a s. C a s \implies I (Result a) s \implies$
 $(B a) \cdot s \text{ ?}\{\lambda r t. I r t\}$
shows $whileLoop C B a \cdot s \text{ ?}\{P\}$
apply (rule runs-to-partial-whileLoop[where $I = I, OF *$])
subgoal using $P\text{-Result}$ **by** *auto*
subgoal using $P\text{-Exn}$ **by** (auto simp add: default-option-def Exn-def)
subgoal using B **by** *blast*
done

notation (output)

$whileLoop ((whileLoop (-)// (-)) [1000, 1000] 1000)$

lemma $whileLoop\text{-mono}$: $b \leq b' \implies whileLoop c b i \leq whileLoop c b' i$
unfolding $whileLoop\text{-def}$
by (simp add: gfp-mono le-funD le-funI mono-bind mono-condition)

lemma $whileLoop\text{-finite-mono}$: $b \leq b' \implies whileLoop\text{-finite} c b i \leq whileLoop\text{-finite} c b' i$
unfolding $whileLoop\text{-finite-def}$
by (simp add: le-funD le-funI lfp-mono mono-bind mono-condition)

lemma $monotone\text{-whileLoop-le}$ [partial-function-mono]:
 $(\bigwedge x. monotone R (\leq) (\lambda f. b f x)) \implies monotone R (\leq) (\lambda f. whileLoop c (b f) i)$
using $whileLoop\text{-mono}$ **unfolding** $monotone\text{-def}$
by (metis le-fun-def)

lemma $monotone\text{-whileLoop-ge}$ [partial-function-mono]:
 $(\bigwedge x. monotone R (\geq) (\lambda f. b f x)) \implies monotone R (\geq) (\lambda f. whileLoop c (b f) i)$
using $whileLoop\text{-mono}$ **unfolding** $monotone\text{-def}$
by (metis le-fun-def)

lemma $monotone\text{-whileLoop-finite-le}$ [partial-function-mono]:
 $(\bigwedge x. monotone R (\leq) (\lambda f. b f x)) \implies monotone R (\leq) (\lambda f. whileLoop\text{-finite} c (b f) i)$
using $whileLoop\text{-finite-mono}$ **unfolding** $monotone\text{-def}$
by (metis le-fun-def)

lemma $monotone\text{-whileLoop-finite-ge}$ [partial-function-mono]:
 $(\bigwedge x. monotone R (\geq) (\lambda f. b f x)) \implies monotone R (\geq) (\lambda f. whileLoop\text{-finite} c (b f) i)$
using $whileLoop\text{-finite-mono}$ **unfolding** $monotone\text{-def}$
by (metis le-fun-def)

lemma $run\text{-whileLoop-le-invariant-cong}$:

assumes I : $I (Result i) s$
assumes $invariant$: $\bigwedge r s. C r s \implies I (Result r) s \implies B r \cdot s \text{ ?}\{I\}$
assumes C : $\bigwedge r s. I (Result r) s \implies C r s = C' r s$
assumes B : $\bigwedge r s. C r s \implies I (Result r) s \implies run (B r) s = run (B' r) s$

```

shows run (whileLoop C B i) s ≤ run (whileLoop C' B' i) s
unfolding le-run-refines-iff
apply (rule refines-mono[of λa b. a = b ∧ I (fst a) (snd a)])
apply simp
apply (rule refines-whileLoop)
subgoal using C by auto
subgoal for x s y t using invariant[of x s] B[of x t]
  by (cases run (B' y) t)
    (auto simp add: refines.rep-eq runs-to-partial-alt runs-to.rep-eq split-beta)
subgoal by simp
subgoal using I by simp
done

```

lemma run-whileLoop-invariant-cong:

```

assumes I: I (Result i) s
assumes invariant: ∧r s. C r s ⇒ I (Result r) s ⇒ B r · s ?{I}
assumes C: ∧r s. I (Result r) s ⇒ C r s = C' r s
assumes B: ∧r s. C r s ⇒ I (Result r) s ⇒ run (B r) s = run (B' r) s
shows run (whileLoop C B i) s = run (whileLoop C' B' i) s
proof (rule antisym)
  from I invariant C B
  show run (whileLoop C B i) s ≤ run (whileLoop C' B' i) s
    by (rule run-whileLoop-le-invariant-cong)
  next
  from B invariant have invariant-sym: ∧r s. C r s ⇒ I (Result r) s ⇒ B' r
  · s ?{I}
    by (simp add: runs-to-partial.rep-eq)
  show run (whileLoop C' B' i) s ≤ run (whileLoop C B i) s
    apply (rule run-whileLoop-le-invariant-cong
      [where I=I, OF I invariant-sym C[symmetric] B[symmetric]])
    apply (auto simp add: C)
  done
qed

```

lemma whileLoop-cong:

```

assumes C: ∧r s. C r s = C' r s
assumes B: ∧r s. C r s ⇒ run (B r) s = run (B' r) s
shows whileLoop C B = whileLoop C' B'
proof (rule ext)
  fix i
  show whileLoop C B i = whileLoop C' B' i
  proof (rule spec-monad-ext)
    fix s
    show run (whileLoop C B i) s = run (whileLoop C' B' i) s
      by (rule run-whileLoop-invariant-cong[where I = λ-. True])
      (simp-all add: C B)
  qed
qed

```

lemma *refines-whileLoop'*:
assumes $C: \bigwedge a s b t. R (\text{Result } a, s) (\text{Result } b, t) \implies C a s \longleftrightarrow C' b t$
and $B:$
 $\bigwedge a s b t. R (\text{Result } a, s) (\text{Result } b, t) \implies C a s \implies C' b t \implies \text{refines } (B a)$
 $(B' b) s t R$
and $I: R (\text{Result } I, s) (\text{Result } I', s')$
and $R:$
 $\bigwedge r s r' s'. R (r, s) (r', s') \implies \text{rel-exception-or-result } (\lambda - . \text{True}) (\lambda - .$
 $\text{True}) r r'$
shows $\text{refines } (\text{whileLoop } C B I) (\text{whileLoop } C' B' I') s s'$
 $(\lambda(r, s) (r', s'). (\forall v. r = \text{Result } v \longrightarrow (\exists v'. r' = \text{Result } v' \wedge \neg C v s \wedge \neg C'$
 $v' s')) \wedge$
 $R (r, s) (r', s'))$
apply (*rule refines-whileLoop-strong*)
subgoal using C **by** *simp*
subgoal using B **by** *simp*
subgoal for $v s v' s'$ **using** $R[\text{of } v s v' s']$ **by** (*auto elim!: rel-exception-or-result.cases*)
subgoal using I **by** *simp*
done

lemma *rel-spec-whileLoop'*:
assumes $C: \bigwedge a s b t. R (\text{Result } a, s) (\text{Result } b, t) \implies C a s \longleftrightarrow C' b t$
and $B: \bigwedge a s b t. R (\text{Result } a, s) (\text{Result } b, t) \implies C a s \implies C' b t \implies \text{rel-spec}$
 $(B a) (B' b) s t R$
and $I: R (\text{Result } I, s) (\text{Result } I', s')$
and $R: \bigwedge r s r' s'. R (r, s) (r', s') \implies \text{rel-exception-or-result } (\lambda - . \text{True}) (\lambda -$
 $. \text{True}) r r'$
shows $\text{rel-spec } (\text{whileLoop } C B I) (\text{whileLoop } C' B' I') s s'$
 $(\lambda(r, s) (r', s'). (\forall v. r = \text{Result } v \longrightarrow (\exists v'. r' = \text{Result } v' \wedge \neg C v s \wedge \neg C'$
 $v' s')) \wedge$
 $R (r, s) (r', s'))$
using $C B I R$
unfolding *rel-spec-iff-refines*
apply (*intro conjI*)
subgoal
by (*intro refines-whileLoop'*) *auto*
subgoal
apply (*intro refines-mono[OF - refines-whileLoop'[where R=R⁻¹⁻¹]]*)
subgoal by *simp* (*smt (verit) Exception-eq-Result rel-exception-or-result.cases*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *simp* (*smt (verit, best) rel-exception-or-result.simps*)
done
done

lemma *rel-spec-whileLoop*:
assumes $C: \bigwedge a s b t. R (\text{Result } a, s) (\text{Result } b, t) \implies C a s \longleftrightarrow C' b t$

and $B: \bigwedge a s b t. R (\text{Result } a, s) (\text{Result } b, t) \implies C a s \implies C' b t \implies \text{rel-spec}$
 $(B a) (B' b) s t R$
and $I: R (\text{Result } I, s) (\text{Result } I', s')$
and $R: \bigwedge r s r' s'. R (r, s) (r', s') \implies \text{rel-exception-or-result } (\lambda - . \text{True}) (\lambda - . \text{True}) r r'$
shows $\text{rel-spec } (\text{whileLoop } C B I) (\text{whileLoop } C' B' I') s s' R$
by $(\text{rule rel-spec-mono}[\text{OF - rel-spec-whileLoop}'[\text{OF assms}]]) \text{ auto}$

lemma *do-whileLoop-combine*:

$\text{do } \{ x1 \leftarrow \text{body } x0; \text{whileLoop } P \text{ body } x1 \} =$
 $\text{do } \{$
 $(b, y) \leftarrow \text{whileLoop } (\lambda(b, x) s. b \longrightarrow P x s) (\lambda(b, x). \text{do } \{ y \leftarrow \text{body } x; \text{return}$
 $(\text{True}, y) \})$
 $(\text{False}, x0);$
 $\text{return } y$
 $\}$
apply $(\text{subst } (2) \text{ whileLoop-unroll})$
apply $(\text{simp add: bind-assoc})$
apply $(\text{rule arg-cong}[\text{where } f=\text{bind } -, \text{OF ext}])$
apply $(\text{subst bind-return}[\text{symmetric}])$
apply $(\text{rule rel-spec-eqD})$
apply $(\text{rule rel-spec-bind-bind}[\text{where}$
 $Q=\text{rel-prod } (\text{rel-exception-or-result } (=) (\lambda x (b, y). b = \text{True} \wedge x = y)) (=)])$
subgoal for $x s$
apply $(\text{rule rel-spec-whileLoop})$
subgoal by *auto*
subgoal
apply $(\text{simp split: prod.splits}) [1]$
apply $(\text{subst bind-return}[\text{symmetric}])$
apply *clarify*
apply $(\text{rule rel-spec-bind-bind}[\text{where } Q=(=)])$
apply $(\text{simp-all add: rel-spec-refl})$
done
by $(\text{auto simp: rel-exception-or-result.simps split: prod.splits})$
apply $(\text{simp-all split: prod.splits})$
done

lemma *rel-spec-whileLoop-res*:

assumes $C: \bigwedge a s b t. R (\text{Result } a, s) (\text{Result } b, t) \implies C a s \iff C' b t$
and $B: \bigwedge a s b t. R (\text{Result } a, s) (\text{Result } b, t) \implies C a s \implies C' b t \implies \text{rel-spec}$
 $(B a) (B' b) s t R$
and $I: R (\text{Result } I, s) (\text{Result } I', s')$
shows $\text{rel-spec } ((\text{whileLoop } C B I)::('a, 's) \text{res-monad}) ((\text{whileLoop } C' B' I')::('b,$
 $'t) \text{res-monad}) s s' R$
by $(\text{rule rel-spec-whileLoop}[\text{OF } C B I]) \text{ auto}$

lemma *rel-spec-monad-whileLoop*:

assumes *init*: $R I I'$
assumes *cond*: $\bigwedge x y. R x y \implies (\text{rel-fun } S (=)) (C x) (C' y)$

assumes *body*: $\bigwedge x y. R x y \implies \text{rel-spec-monad } S \text{ (rel-exception-or-result } E R)$
 $(B x) (B' y)$
shows *rel-spec-monad* $S \text{ (rel-exception-or-result } E R) \text{ (whileLoop } C B I) \text{ (whileLoop } C' B' I')$
apply (*clarsimp simp add: rel-spec-monad-iff-rel-spec*)
apply (*rule rel-spec-whileLoop*)
subgoal for $s t x s' y t'$ **using** *cond* **by** (*simp add: rel-fun-def*)
subgoal for $s t x s' y t'$ **using** *body[of x y]* **by** (*simp add: rel-spec-monad-iff-rel-spec*)
subgoal using *init* **by** *simp*
subgoal by (*auto elim!: rel-exception-or-result.cases*)
done

lemma *rel-spec-monad-whileLoop-res'*:

assumes *init*: $R I I'$
assumes *cond*: $\bigwedge x y. R x y \implies (\text{rel-fun } S (=)) (C x) (C' y)$
assumes *body*: $\bigwedge x y. R x y \implies \text{rel-spec-monad } S (\lambda \text{Res } x \text{ Res } y. R x y) (B x)$
 $(B' y)$
shows *rel-spec-monad* $S (\lambda \text{Res } x \text{ Res } y. R x y)$
 $((\text{whileLoop } C B I)::('a, 's) \text{ res-monad})$
 $((\text{whileLoop } C' B' I')::('b, 't) \text{ res-monad})$
using *assms*
by (*auto intro: rel-spec-monad-whileLoop simp add: rel-exception-or-result-Res-val*)

lemma *rel-spec-monad-whileLoop-res*:

assumes *init*: $R I I'$
assumes *cond*: *rel-fun* $R \text{ (rel-fun } S (=)) C C'$
assumes *body*: *rel-fun* $R \text{ (rel-spec-monad } S (\lambda \text{Res } x \text{ Res } y. R x y)) B B'$
shows *rel-spec-monad* $S (\lambda \text{Res } x \text{ Res } y. R x y)$
 $((\text{whileLoop } C B I)::('a, 's) \text{ res-monad})$
 $((\text{whileLoop } C' B' I')::('b, 't) \text{ res-monad})$
using *assms*
by (*auto intro: rel-spec-monad-whileLoop-res' simp add: rel-fun-def*)

lemma *runs-to-whileLoop-finite-exn*:

assumes $B: \bigwedge r s. I (\text{Result } r) s \implies C r s \implies (B r) \cdot s \{ I \}$
assumes $Qr: \bigwedge r s. I (\text{Result } r) s \implies \neg C r s \implies Q (\text{Result } r) s$
assumes $Ql: \bigwedge e s. I (\text{Exn } e) s \implies Q (\text{Exn } e) s$
assumes $I: I (\text{Result } r) s$
shows $(\text{whileLoop-finite } C B r) \cdot s \{ Q \}$
using *assms* **by** (*intro runs-to-whileLoop-finite[of I]*) (*auto simp: Exn-def default-option-def*)

lemma *runs-to-whileLoop-finite-res*:

assumes $B: \bigwedge r s. I r s \implies C r s \implies (B r) \cdot s \{ \lambda \text{Res } r. I r \}$
assumes $Q: \bigwedge r s. I r s \implies \neg C r s \implies Q r s$
assumes $I: I r s$
shows $((\text{whileLoop-finite } C B r)::('a, 's) \text{ res-monad}) \cdot s \{ \lambda \text{Res } r. Q r \}$
using *assms* **by** (*intro runs-to-whileLoop-finite[of $\lambda \text{Res } r. I r$]*) *simp-all*

lemma *runs-to-whileLoop-eq-whileLoop-finite*:

run (whileLoop C B r) s $\neq \top \implies$
(whileLoop C B r) · s $\{ Q \}$ \longleftrightarrow *(whileLoop-finite C B r) · s* $\{ Q \}$
by (*simp add: whileLoop-finite-eq-whileLoop runs-to.rep-eq*)

lemma *whileLoop-finite-cond-fail*:

$\neg C r s \implies$ (*run (whileLoop-finite C B r) s*) = (*run (return r) s*)
apply (*subst whileLoop-finite-unfold*)
apply *simp*
done

lemma *runs-to-whileLoop-finite-cond-fail*:

$\neg C r s \implies$ (*whileLoop-finite C B r) · s* $\{ Q \}$ \longleftrightarrow (*return r) · s* $\{ Q \}$
apply (*subst whileLoop-finite-unfold*)
apply (*simp add: runs-to.rep-eq*)
done

lemma *runs-to-whileLoop-cond-fail*:

$\neg C r s \implies$ (*whileLoop C B r) · s* $\{ Q \}$ \longleftrightarrow (*return r) · s* $\{ Q \}$
apply (*subst whileLoop-unroll*)
apply (*simp add: runs-to.rep-eq*)
done

lemma *whileLoop-finite-unfold'*:

(whileLoop-finite C B r) =
((condition (C r) (B r) (return r)) >>= (whileLoop-finite C B))
apply (*rule spec-monad-ext*)
apply (*subst (1) whileLoop-finite-unfold*)
subgoal for *s*
apply (*cases C r s*)
subgoal by (*simp add: run-bind*)
subgoal by (*simp add: whileLoop-finite-cond-fail run-bind*)
done
done

lemma *runs-to-whileLoop-unroll*:

assumes $\neg C r s \implies P r s$
assumes [*runs-to-vcg*]: *C r s* \implies (*B r) · s* $\{ \lambda Res r t. ((whileLoop C B r) · t$
 $\{ \lambda Res r. P r \}) \}$
shows (*whileLoop C B r) · s* $\{ \lambda Res r. P r \}$
using *assms(1)*
by (*subst whileLoop-unroll runs-to-vcg*)

lemma *runs-to-partial-whileLoop-unroll*:

assumes $\neg C r s \implies P r s$
assumes *C r s* \implies (*B r) · s* $\{ \lambda Res r t. ((whileLoop C B r) · t$ $\{ \lambda Res r. P r$
 $\}) \}$
shows (*whileLoop C B r) · s* $\{ \lambda Res r. P r \}$
using *assms*

by (subst whileLoop-unroll) runs-to-vcg

lemma runs-to-whileLoop-unroll-exn:

assumes $\neg C r s \implies P (Result r) s$

assumes [runs-to-vcg]: $C r s \implies (B r) \cdot s \{ \lambda r t.$

$(\forall a. r = Result a \longrightarrow ((whileLoop C B a) \cdot t \{ P \})) \wedge$

$(\forall e. r = Exn e \longrightarrow P (Exn e) t) \}$

shows $(whileLoop C B r) \cdot s \{ \lambda r s'. P r s' \}$

using *assms(1)*

by (subst whileLoop-unroll) (runs-to-vcg, auto simp: Exn-def default-option-def)

lemma runs-to-partial-whileLoop-unroll-exn:

assumes $\neg C r s \implies P (Result r) s$

assumes $C r s \implies (B r) \cdot s \{ \lambda r t.$

$(\forall a. r = Result a \longrightarrow ((whileLoop C B a) \cdot t \{ P \})) \wedge$

$(\forall e. r = Exn e \longrightarrow P (Exn e) t) \}$

shows $(whileLoop C B r) \cdot s \{ \lambda r s'. P r s' \}$

using *assms*

by (subst whileLoop-unroll) runs-to-vcg

lemma refines-whileLoop-exn:

assumes $C: \bigwedge a s b t. R (Result a, s) (Result b, t) \implies C a s \longleftrightarrow C' b t$

and $B: \bigwedge a s b t. R (Result a, s) (Result b, t) \implies C a s \implies C' b t \implies \text{refines}$
 $(B a) (B' b) s t R$

and $I: R (Result I, s) (Result I', s')$

and $R1: \bigwedge a s b t. R (Exn a, s) (Result b, t) \implies \text{False}$

and $R2: \bigwedge a s b t. R (Result a, s) (Exn b, t) \implies \text{False}$

shows $\text{refines} (whileLoop C B I) (whileLoop C' B' I') s s' R$

proof –

have *ref*: $\text{refines} (whileLoop C B I) (whileLoop C' B' I') s s'$

$(\lambda(r, s) (r', s'). (\forall v. r = Result v \longrightarrow (\exists v'. r' = Result v' \wedge \neg C v s \wedge \neg$
 $C' v' s'))) \wedge$

$R (r, s) (r', s')$

apply (rule refines-whileLoop' [OF C])

subgoal by *assumption*

subgoal by (rule B)

subgoal by (rule I)

subgoal using *R1 R2*

by (auto simp add: rel-exception-or-result.simps Exn-def default-option-def)

(metis default-option-def exception-or-result-cases not-None-eq)+

done

show *?thesis*

apply (rule refines-mono [OF - ref])

apply (auto)

done

qed

lemma *refines-whileLoop''*:
assumes $C: \bigwedge a s b t. R (\text{Result } a, s) (\text{Result } b, t) \implies C a s \longleftrightarrow C' b t$
and $B: \bigwedge a s b t. R (\text{Result } a, s) (\text{Result } b, t) \implies C a s \implies C' b t \implies \text{refines}$
 $(B a) (B' b) s t R$
and $I: R (\text{Result } I, s) (\text{Result } I', s')$
and $R: \bigwedge r s r' s'. R (r, s) (r', s') \implies \text{rel-exception-or-result } (\lambda - . \text{True}) (\lambda - . \text{True}) r r'$
shows $\text{refines } (\text{whileLoop } C B I) (\text{whileLoop } C' B' I') s s' R$
proof –
have $\text{refines } (\text{whileLoop } C B I) (\text{whileLoop } C' B' I') s s'$
 $(\lambda(r, s) (r', s'). (\forall v. r = \text{Result } v \longrightarrow (\exists v'. r' = \text{Result } v' \wedge \neg C v s \wedge \neg C' v' s')) \wedge$
 $R (r, s) (r', s'))$
by $(\text{rule } \text{refines-whileLoop}' [\text{OF } C B I R])$
then show *?thesis*
apply $(\text{rule } \text{refines-weaken})$
apply *auto*
done
qed

lemma *rel-spec-monad-whileLoop-exn*:
assumes *init*: $R I I'$
assumes *cond*: $\bigwedge x y. R x y \implies (\text{rel-fun } S (=)) (C x) (C' y)$
assumes *body*: $\bigwedge x y. R x y \implies \text{rel-spec-monad } S (\text{rel-xval } E R) (B x) (B' y)$
shows $\text{rel-spec-monad } S (\text{rel-xval } E R) (\text{whileLoop } C B I) (\text{whileLoop } C' B' I')$
unfolding *rel-xval-rel-exception-or-result-conv*
by $(\text{rule } \text{rel-spec-monad-whileLoop} [\text{OF } \text{init } \text{cond } \text{body} [\text{simplified } \text{rel-xval-rel-exception-or-result-conv}]])$

lemma *refines-whileLoop-guard-right*:
assumes $\bigwedge x s x' s'. R (\text{Result } x, s) (\text{Result } x', s') \implies G' x' s' \implies C x s = C' x' s'$
assumes $\bigwedge x s x' s'. R (\text{Result } x, s) (\text{Result } x', s') \implies C x s \implies C' x' s' \implies$
 $G' x' s' \implies \text{refines } (B x) (B' x') s s' R$
assumes $\bigwedge v s v' s'. R (v, s) (v', s') \implies (\exists x. v = \text{Result } x) = (\exists x'. v' = \text{Result } x')$
assumes $R (\text{Result } x, s) (\text{Result } x', s')$
assumes $G s' = G' x' s'$
shows $\text{refines } (\text{whileLoop } C B x) (\text{guard } G \gg (\lambda -. \text{whileLoop } C' (\lambda r. \text{do } \{\text{res} <- B' r; \text{guard } (G' \text{res}); \text{return } \text{res}\} x')) s s' R$
apply $(\text{rule } \text{refines-bind-guard-right})$
apply $(\text{rule } \text{refines-mono} [\text{where } R = \lambda(r, s) (q, t). R (r, s) (q, t) \wedge$
 $(\text{case } q \text{ of } \text{Exception } e \Rightarrow \text{True} \mid \text{Result } v \Rightarrow G' v t)])$
subgoal by *simp*
subgoal
apply $(\text{rule } \text{refines-whileLoop})$
subgoal using *assms(1)* **by** *auto*
subgoal apply $(\text{rule } \text{refines-bind-guard-right-end}')$
using *assms(2)*

by *auto*
 subgoal using *assms(3)* by (*auto split: exception-or-result-splits*)
 subgoal using *assms(4,5)* by *auto*
 done
 done

16.9.31 *map-value*

lemma *map-value-lift-state*: $\text{map-value } f (\text{lift-state } R \ g) = \text{lift-state } R (\text{map-value } f \ g)$

apply *transfer*
 apply (*simp add: map-post-state-eq-lift-post-state lift-post-state-comp lift-post-state-Sup image-image OO-def prod-eq-iff rel-prod.simps*)
 apply (*simp add: ac-simps*)
 done

lemma *run-map-value[run-spec-monad]*: $\text{run } (\text{map-value } f \ g) \ s = \text{map-post-state } (\lambda(v, s). (f \ v, s)) (\text{run } g \ s)$
 by *transfer (auto simp add: apfst-def map-prod-def map-post-state-def split: post-state.splits prod.splits)*

lemma *always-progress-map-value[always-progress-intros]*:
 $\text{always-progress } g \implies \text{always-progress } (\text{map-value } f \ g)$
 by (*simp add: always-progress-def run-map-value*)

lemma *runs-to-map-value-iff[runs-to-iff]*: $\text{map-value } f \ g \cdot s \ \{\!\! \{ Q \}\!\! \} \longleftrightarrow g \cdot s \ \{\!\! \{ \lambda r \ t. Q \ (f \ r) \ t \}\!\! \}$
 by (*auto simp add: runs-to.rep-eq run-map-value split-beta'*)

lemma *runs-to-partial-map-value-iff[runs-to-iff]*:
 $\text{map-value } f \ g \cdot s \ \{\!\! \{ Q \}\!\! \} \longleftrightarrow g \cdot s \ \{\!\! \{ \lambda r \ t. Q \ (f \ r) \ t \}\!\! \}$
 by (*auto simp add: runs-to-partial.rep-eq run-map-value split-beta'*)

lemma *runs-to-map-value[runs-to-vcg]*: $g \cdot s \ \{\!\! \{ \lambda r \ t. Q \ (f \ r) \ t \}\!\! \} \implies \text{map-value } f \ g \cdot s \ \{\!\! \{ Q \}\!\! \}$
 by (*simp add: runs-to-map-value-iff*)

lemma *runs-to-partial-map-value[runs-to-vcg]*:
 $g \cdot s \ \{\!\! \{ \lambda r \ t. Q \ (f \ r) \ t \}\!\! \} \implies \text{map-value } f \ g \cdot s \ \{\!\! \{ Q \}\!\! \}$
 by (*auto simp add: runs-to-partial.rep-eq run-map-value split-beta'*)

lemma *mono-map-value*: $g \leq g' \implies \text{map-value } f \ g \leq \text{map-value } f \ g'$
 by *transfer (simp add: le-fun-def mono-map-post-state)*

lemma *monotone-map-value-le[partial-function-mono]*:
 $\text{monotone } R \ (\leq) \ (\lambda f'. g \ f') \implies \text{monotone } R \ (\leq) \ (\lambda f'. \text{map-value } f \ (g \ f'))$
 by (*auto simp: monotone-def mono-map-value*)

lemma *monotone-map-value-ge[partial-function-mono]*:

monotone $R (\geq) (\lambda f'. g f') \implies \text{monotone } R (\geq) (\lambda f'. \text{map-value } f (g f'))$
by (*auto simp: monotone-def mono-map-value*)

lemma *map-value-fail[simp]*: *map-value* $f \text{ fail} = \text{fail}$
by *transfer simp*

lemma *map-value-map-exn-gets[simp]*: *map-value* (*map-exn emb*) (*gets* x) = *gets* x
by (*simp add: spec-monad-ext-iff run-map-value*)

lemma *refines-map-value-right-iff*:
refines $f (\text{map-value } m g) s t R \iff \text{refines } f g s t (\lambda(x, s) (y, t). R (x, s) (m y, t))$
by *transfer (simp add: sim-post-state-map-post-state2 split-beta' apfst-def map-prod-def)*

lemma *refines-map-value-left-iff*:
refines (*map-value* $m f$) $g s t R \iff \text{refines } f g s t (\lambda(x, s) (y, t). R (m x, s) (y, t))$
by *transfer (simp add: sim-post-state-map-post-state1 split-beta' apfst-def map-prod-def)*

lemma *rel-spec-map-value-right-iff*:
rel-spec $f (\text{map-value } m g) s t R \iff \text{rel-spec } f g s t (\lambda(x, s) (y, t). R (x, s) (m y, t))$
by (*simp add: rel-spec-iff-refines refines-map-value-right-iff refines-map-value-left-iff conversep.simps[abs-def] split-beta'*)

lemma *rel-spec-map-value-left-iff*:
rel-spec (*map-value* $m f$) $g s t R \iff \text{rel-spec } f g s t (\lambda(x, s) (y, t). R (m x, s) (y, t))$
by (*simp add: rel-spec-iff-refines refines-map-value-right-iff refines-map-value-left-iff conversep.simps[abs-def] split-beta'*)

lemma *refines-map-value-right*:
refines $f (\text{map-value } m f) s s (\lambda(x, s) (y, t). y = m x \wedge s = t)$
by (*simp add: refines-map-value-right-iff refines-refl*)

lemma *refines-map-value*:
assumes *refines* $f f' s t Q$
assumes $\bigwedge r s' w t'. Q (r, s') (w, t') \implies R (g r, s') (g' w, t')$
shows *refines* (*map-value* $g f$) (*map-value* $g' f'$) $s t R$
apply (*simp add: refines-map-value-left-iff refines-map-value-right-iff*)
apply (*rule refines-weaken [OF assms(1)]*)
using *assms(2)* **by** *auto*

lemma *map-value-id[simp]*: *map-value* $(\lambda x. x) = (\lambda x. x)$
apply (*simp add: fun-eq-iff, intro allI iffI*)
apply (*rule spec-monad-eqI*)
apply (*rule runs-to-iff*)
done

16.9.32 *liftE*

lemma *run-liftE[run-spec-monad]*:

$run (liftE f) s =$
 $map-post-state (\lambda(v, s). (map-exception-or-result (\lambda x. undefined) id v, s)) (run$
 $f s)$
by (*simp add: run-map-value liftE-def*)

lemma *always-progress-liftE[always-progress-intros]*:

$always-progress f \implies always-progress (liftE f)$
by (*simp add: liftE-def always-progress-intros*)

lemma *runs-to-liftE-iff*:

$liftE f \cdot s \{ Q \} \longleftrightarrow f \cdot s \{ \lambda r t. Q (map-exception-or-result (\lambda x. undefined) id$
 $r) t \}$
by (*simp add: liftE-def runs-to-map-value-iff*)

lemma *runs-to-liftE-iff-Res[runs-to-iff]*:

$liftE f \cdot s \{ Q \} \longleftrightarrow f \cdot s \{ \lambda Res r. Q (Result r) \}$
by (*auto intro!: runs-to-cong-pred-only simp: runs-to-liftE-iff*)

lemma *runs-to-liftE'*:

$f \cdot s \{ \lambda r t. Q (map-exception-or-result (\lambda x. undefined) id r) t \} \implies liftE f \cdot s \{$
 $Q \}$
by (*simp add: runs-to-liftE-iff*)

lemma *runs-to-liftE[runs-to-vcg]: $f \cdot s \{ \lambda Res r. Q (Result r) \} \implies liftE f \cdot s \{$*

$Q \}$
by (*simp add: runs-to-liftE-iff-Res*)

lemma *refines-liftE-left-iff*:

$refines (liftE f) g s t R \longleftrightarrow$
 $refines f g s t (\lambda(x, s') (y, t'). \forall v. x = Result v \longrightarrow R (Result v, s') (y, t'))$
by (*auto simp add: liftE-def refines-map-value-left-iff intro!: refines-cong-cases*
del: iffI)

lemma *refines-liftE-right-iff*:

$refines f (liftE g) s t R \longleftrightarrow$
 $refines f g s t (\lambda(x, s') (y, t'). \forall v. y = Result v \longrightarrow R (x, s') (Result v, t'))$
by (*auto simp add: liftE-def refines-map-value-right-iff intro!: refines-cong-cases*
del: iffI)

lemma *rel-spec-liftE*:

$rel-spec (liftE f) g s t R \longleftrightarrow$
 $rel-spec f g s t (\lambda(x, s') (y, t'). \forall v. x = Result v \longrightarrow R (Result v, s') (y, t'))$
by (*simp add: rel-spec-iff-refines refines-liftE-right-iff refines-liftE-left-iff*
conversep.simps[abs-def] split-beta)

lemma *rel-spec-monad-bind-liftE*:

$rel-spec-monad R (\lambda Res v1 Res v2. P v1 v2) m n \implies rel-fun P (rel-spec-monad$

$R \ Q \ f \ g \implies$
 $\text{rel-spec-monad } R \ Q \ ((\text{liftE } m) \gg= f) \ ((\text{liftE } n) \gg= g)$
apply (*auto intro!*: *rel-bind-post-state'* *rel-post-state-refl*
simp: *rel-spec-monad-def run-bind run-liftE bind-post-state-map-post-state rel-fun-def*)[1]
subgoal premises *prems* **for** $s \ t$
by (*rule rel-post-state-weaken*[*OF prems*(1)[*rule-format*, *OF prems*(3)]])
(*auto simp add*: *rel-prod.simps prems*(2))
done

lemma *rel-spec-monad-bind-liftE'*:
 $(\bigwedge x. \text{rel-spec-monad } (=) \ Q \ (f \ x) \ (g \ x)) \implies \text{rel-spec-monad } (=) \ Q \ (\text{liftE } m \gg= f) \ (\text{liftE } m \gg= g)$
by (*auto simp add*: *rel-spec-monad-def run-bind run-liftE bind-post-state-map-post-state intro!*: *rel-bind-post-state'* *rel-post-state-refl*)

lemma *runs-to-partial-liftE'*:
 $f \cdot s \ ?\{\lambda r \ t. \ Q \ (\text{map-exception-or-result } (\lambda x. \text{undefined}) \ \text{id } r) \ t\} \implies \text{liftE } f \cdot s \ ?\{\ Q \}$
by (*simp add*: *runs-to-partial.rep-eq run-liftE split-beta'*)

lemma *runs-to-partial-liftE*[*runs-to-vcg*]:
assumes [*runs-to-vcg*]: $f \cdot s \ ?\{\lambda \text{Res } r. \ Q \ (\text{Result } r)\}$ **shows** $\text{liftE } f \cdot s \ ?\{\ Q \}$
supply *runs-to-partial-liftE'*[*runs-to-vcg*] **by** *runs-to-vcg*

lemma *mono-liftE*: $f \leq f' \implies \text{liftE } f \leq \text{liftE } f'$
unfolding *liftE-def* **by** (*rule mono-map-value*)

lemma *monotone-liftE-le*[*partial-function-mono*]:
 $\text{monotone } R \ (\leq) \ (\lambda f'. \ f \ f') \implies \text{monotone } R \ (\leq) \ (\lambda f'. \ \text{liftE } (f \ f'))$
unfolding *liftE-def* **by** (*rule monotone-map-value-le*)

lemma *monotone-liftE-ge*[*partial-function-mono*]:
 $\text{monotone } R \ (\geq) \ (\lambda f'. \ f \ f') \implies \text{monotone } R \ (\geq) \ (\lambda f'. \ \text{liftE } (f \ f'))$
unfolding *liftE-def* **by** (*rule monotone-map-value-ge*)

lemma *bind-handle-liftE*: $\text{bind-handle } (\text{liftE } f) \ g \ h = \text{bind } f \ g$
by (*simp add*: *spec-monad-eq-iff runs-to-iff*)

lemma *liftE-top*[*simp*]: $\text{liftE } \top = \top$
by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-bot*[*simp*]: $\text{liftE } \text{bot} = \text{bot}$
by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-fail*[*simp*]: $\text{liftE } \text{fail} = \text{fail}$
by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-return*[*simp*]: $\text{liftE } (\text{return } x) = \text{return } x$
by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-throw-Exception*[simp]:
 $\text{liftE } (\text{yield } (\text{Exception } x)) = \text{skip}$
by (rule *spec-monad-ext*) (simp add: *run-liftE*)

lemma *liftE-throw-exception-or-result*[simp]:
 $\text{liftE } (\text{throw-exception-or-result } x) = \text{return undefined}$
by (rule *spec-monad-ext*) (simp add: *run-liftE map-exception-or-result-def*)

lemma *liftE-get-state*[simp]: $\text{liftE } (\text{get-state}) = \text{get-state}$
by (rule *spec-monad-ext*) (simp add: *run-liftE*)

lemma *liftE-set-state*[simp]: $\text{liftE } (\text{set-state } s) = \text{set-state } s$
by (rule *spec-monad-ext*) (simp add: *run-liftE*)

lemma *liftE-select*[simp]: $\text{liftE } (\text{select } S) = \text{select } S$
by (rule *spec-monad-ext*) (simp add: *run-liftE*)

lemma *liftE-unknown*[simp]: $\text{liftE } (\text{unknown}) = \text{unknown}$
by (rule *spec-monad-ext*) (simp add: *run-liftE*)

lemma *liftE-lift-state*: $\text{liftE } (\text{lift-state } R f) = \text{lift-state } R (\text{liftE } f)$
unfolding *liftE-def* **by** (rule *map-value-lift-state*)

lemma *liftE-exec-concrete*: $\text{liftE } (\text{exec-concrete } st f) = \text{exec-concrete } st (\text{liftE } f)$
unfolding *exec-concrete-def* **by** (rule *liftE-lift-state*)

lemma *liftE-exec-abstract*: $\text{liftE } (\text{exec-abstract } st f) = \text{exec-abstract } st (\text{liftE } f)$
unfolding *exec-abstract-def* **by** (rule *liftE-lift-state*)

lemma *liftE-assert*[simp]: $\text{liftE } (\text{assert } P) = \text{assert } P$
by (rule *spec-monad-ext*) (simp add: *run-liftE run-assert*)

lemma *liftE-assume*[simp]: $\text{liftE } (\text{assume } P) = \text{assume } P$
by (rule *spec-monad-ext*) (simp add: *run-liftE run-assume*)

lemma *liftE-gets*[simp]: $\text{liftE } (\text{gets } f) = \text{gets } f$
by (rule *spec-monad-ext*) (simp add: *run-liftE*)

lemma *liftE-guard*[simp]: $\text{liftE } (\text{guard } P) = \text{guard } P$
by (rule *spec-monad-ext*) (simp add: *run-liftE run-guard*)

lemma *liftE-assert-opt*[simp]: $\text{liftE } (\text{assert-opt } v) = \text{assert-opt } v$
by (rule *spec-monad-ext*) (auto simp add: *run-liftE split: option.splits*)

lemma *liftE-gets-the*[simp]: $\text{liftE } (\text{gets-the } f) = \text{gets-the } f$
by (rule *spec-monad-ext*) (auto simp add: *run-liftE split: option.splits*)

lemma *liftE-modify*[simp]: $\text{liftE } (\text{modify } f) = \text{modify } f$

by (rule spec-monad-ext) (simp add: run-liftE)

lemma liftE-bind: liftE x >>= (λa. liftE (y a)) = liftE (x >>= y)
 by (simp add: spec-monad-eq-iff runs-to-iff)

lemma bindE-liftE-skip: liftE (f >>= (λy. skip)) >>= g = liftE f >>= (λ-. g ())
 by (auto simp add: spec-monad-eq-iff runs-to-iff intro!: arg-cong[**where** f=runs-to f -])

lemma liftE-state-select[simp]: liftE (state-select f) = state-select f
 apply (rule spec-monad-eqI)
 apply (auto simp add: runs-to-iff)
 done

lemma liftE-assume-result-and-state[simp]:
 liftE (assume-result-and-state f) = assume-result-and-state f
 apply (rule spec-monad-eqI)
 apply (auto simp add: runs-to-iff)
 done

lemma map-value-map-exn-liftE [simp]:
 map-value (map-exn emb) (liftE f) = liftE f
 by (simp add: spec-monad-eq-iff runs-to-iff)

lemma liftE-condition: liftE (condition c f g) = condition c (liftE f) (liftE g)
 by (simp add: spec-monad-eq-iff runs-to-iff)

lemma liftE-whileLoop: liftE (whileLoop C B I) = whileLoop C (λr. liftE (B r)) I
 apply (rule rel-spec-eqD)
 apply (subst rel-spec-liftE)
 apply (rule rel-spec-whileLoop)
 subgoal by simp
 apply (subst rel-spec-symm)
 apply (subst rel-spec-liftE)
 apply (simp add: rel-spec-refl)
 apply auto
 done

16.9.33 try

lemma run-try[run-spec-monad]:
 run (try f) s = map-post-state (λ(v, s). (unnest-exn v, s)) (run f s)
 by (simp add: try-def run-map-value)

lemma always-progress-try[always-progress-intros]: always-progress f ⇒ always-progress (try f)
 by (simp add: try-def always-progress-intros)

lemma runs-to-try[runs-to-vcg]: f · s {λr t. Q (unnest-exn r) t} ⇒ try f · s {

$Q \}$
by (*simp add: try-def runs-to-map-value*)

lemma *runs-to-try-iff*[*runs-to-iff*]: $try\ f \cdot s \ \{\!\! \{ Q \}\!\! \} \longleftrightarrow f \cdot s \ \{\!\! \{ \lambda r\ t.\ Q\ (unnest-exn\ r)\ t \}\!\! \}$
by (*auto simp add: try-def runs-to-iff*)

lemma *runs-to-partial-try*[*runs-to-vcg*]: $f \cdot s \ \{\!\! \{ \lambda r\ t.\ Q\ (unnest-exn\ r)\ t \}\!\! \} \implies try\ f \cdot s \ \{\!\! \{ Q \}\!\! \}$
by (*simp add: try-def runs-to-partial-map-value*)

lemma *mono-try*: $f \leq f' \implies try\ f \leq try\ f'$
unfolding *try-def*
by (*rule mono-map-value*)

lemma *monotone-try-le*[*partial-function-mono*]:
 $monotone\ R\ (\leq)\ (\lambda f'.\ f\ f') \implies monotone\ R\ (\leq)\ (\lambda f'.\ try\ (f\ f'))$
unfolding *try-def* **by** (*rule monotone-map-value-le*)

lemma *monotone-try-ge*[*partial-function-mono*]:
 $monotone\ R\ (\geq)\ (\lambda f'.\ f\ f') \implies monotone\ R\ (\geq)\ (\lambda f'.\ try\ (f\ f'))$
unfolding *try-def* **by** (*rule monotone-map-value-ge*)

lemma *refines-try-right*:
 $refines\ f\ (try\ f)\ s\ s\ (\lambda(x, s)\ (y, t).\ y = unnest-exn\ x \wedge s = t)$
by (*auto simp add: try-def refines-map-value-right*)

16.9.34 *finally*

lemma *run-finally*[*run-spec-monad*]:
 $run\ (finally\ f)\ s = map-post-state\ (\lambda(v, s).\ (unite\ v, s))\ (run\ f\ s)$
by (*simp add: finally-def run-map-value*)

lemma *always-progress-finally*[*always-progress-intros*]:
 $always-progress\ f \implies always-progress\ (finally\ f)$
by (*simp add: finally-def always-progress-intros*)

lemma *runs-to-finally-iff*[*runs-to-iff*]:
 $finally\ f \cdot s \ \{\!\! \{ Q \}\!\! \} \longleftrightarrow f \cdot s \ \{\!\! \{ \lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow Q\ (Result\ v)\ t) \wedge (\forall v.\ r = Exn\ v \longrightarrow Q\ (Result\ v)\ t) \}\!\! \}$
by (*auto simp: finally-def runs-to-map-value-iff unite-def intro!: arg-cong[where f=runs-to -] split: xval-splits*)

lemma *runs-to-finally'*: $f \cdot s \ \{\!\! \{ \lambda r\ t.\ Q\ (unite\ r)\ t \}\!\! \} \implies finally\ f \cdot s \ \{\!\! \{ Q \}\!\! \}$
by (*simp add: finally-def runs-to-map-value*)

lemma *runs-to-finally*[*runs-to-vcg*]:
 $f \cdot s \ \{\!\! \{ \lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow Q\ (Result\ v)\ t) \wedge (\forall v.\ r = Exn\ v \longrightarrow Q$

$(\text{Result } v) t\} \implies$
 $\text{finally } f \cdot s \{ Q \}$
by (*simp add: runs-to-finally-iff*)

lemma *runs-to-partial-finally'*: $f \cdot s \{ \lambda r t. Q (\text{unite } r) t \} \implies \text{finally } f \cdot s \{ Q \}$
by (*simp add: finally-def runs-to-partial-map-value*)

lemma *runs-to-partial-finally-iff*:
 $\text{finally } f \cdot s \{ Q \} \longleftrightarrow$
 $f \cdot s \{ \lambda r t. (\forall v. r = \text{Result } v \longrightarrow Q (\text{Result } v) t) \wedge (\forall v. r = \text{Exn } v \longrightarrow Q (\text{Result } v) t) \}$
by (*auto simp: finally-def runs-to-partial-map-value-iff unite-def intro!: arg-cong[where f=runs-to-partial -] split: xval-splits*)

lemma *runs-to-partial-finally[runs-to-vcg]*:
 $f \cdot s \{ \lambda r t. (\forall v. r = \text{Result } v \longrightarrow Q (\text{Result } v) t) \wedge (\forall v. r = \text{Exn } v \longrightarrow Q (\text{Result } v) t) \} \implies$
 $\text{finally } f \cdot s \{ Q \}$
by (*simp add: runs-to-partial-finally-iff*)

lemma *mono-finally*: $f \leq f' \implies \text{finally } f \leq \text{finally } f'$
unfolding *finally-def*
by (*rule mono-map-value*)

lemma *monotone-finally-le[partial-function-mono]*:
 $\text{monotone } R (\leq) (\lambda f'. f f') \implies \text{monotone } R (\leq) (\lambda f'. \text{finally } (f f'))$
unfolding *finally-def* **by** (*rule monotone-map-value-le*)

lemma *monotone-finally-ge[partial-function-mono]*:
 $\text{monotone } R (\geq) (\lambda f'. f f') \implies \text{monotone } R (\geq) (\lambda f'. \text{finally } (f f'))$
unfolding *finally-def* **by** (*rule monotone-map-value-ge*)

16.9.35 (*<catch>*)

lemma *run-catch[run-spec-monad]*:
 $\text{run } (f \text{ <catch> } h) s =$
 $\text{bind-post-state } (\text{run } f s)$
 $(\lambda(r, t). \text{case } r \text{ of } \text{Exn } e \Rightarrow \text{run } (h e) t \mid \text{Result } v \Rightarrow \text{run } (\text{return } v) t)$
apply (*simp add: catch-def run-bind-handle*)
apply (*rule arg-cong[where f=bind-post-state -, OF ext]*)
apply (*auto simp add: Exn-def default-option-def split: exception-or-result-splits xval-splits*)
done

lemma *always-progress-catch[always-progress-intros]*:
 $\text{always-progress } f \implies (\wedge e. \text{always-progress } (h e)) \implies \text{always-progress } (f \text{ <catch> } h)$
unfolding *catch-def*

by (auto intro: always-progress-intros)

lemma runs-to-catch-iff[runs-to-iff]:

$(f \langle \text{catch} \rangle h) \cdot s \Vdash Q \iff$
 $f \cdot s \Vdash \lambda r t. (\forall v. r = \text{Result } v \longrightarrow Q (\text{Result } v) t) \wedge (\forall e. r = \text{Exn } e \longrightarrow h e \cdot$
 $t \Vdash Q)$
by (simp add: runs-to.rep-eq run-catch split-beta' ac-simps split: xval-splits prod.splits)

lemma runs-to-catch[runs-to-vcg]:

$f \cdot s \Vdash \lambda r t. (\forall v. r = \text{Result } v \longrightarrow Q (\text{Result } v) t) \wedge$
 $(\forall e. r = \text{Exn } e \longrightarrow h e \cdot t \Vdash Q) \implies$
 $(f \langle \text{catch} \rangle h) \cdot s \Vdash Q$
by (simp add: runs-to-catch-iff)

lemma runs-to-partial-catch[runs-to-vcg]:

$f \cdot s \text{ ?}\Vdash \lambda r t. (\forall v. r = \text{Result } v \longrightarrow Q (\text{Result } v) t) \wedge$
 $(\forall e. r = \text{Exn } e \longrightarrow h e \cdot t \text{ ?}\Vdash Q) \implies$
 $(f \langle \text{catch} \rangle h) \cdot s \text{ ?}\Vdash Q$

unfolding runs-to-partial.rep-eq run-catch

by (rule holds-partial-bind-post-state)

(simp add: split-beta' ac-simps split: xval-splits prod.splits)

lemma mono-catch: $f \leq f' \implies h \leq h' \implies \text{catch } f h \leq \text{catch } f' h'$

unfolding catch-def

apply (rule mono-bind-handle)

by (simp-all add: le-fun-def)

lemma monotone-catch-le[partial-function-mono]:

$\text{monotone } R (\leq) (\lambda f'. f f') \implies (\bigwedge e. \text{monotone } R (\leq) (\lambda f'. h f' e))$
 $\implies \text{monotone } R (\leq) (\lambda f'. \text{catch } (f f') (h f'))$

unfolding catch-def

apply (rule monotone-bind-handle-le)

by auto

lemma monotone-catch-ge[partial-function-mono]:

$\text{monotone } R (\geq) (\lambda f'. f f') \implies (\bigwedge e. \text{monotone } R (\geq) (\lambda f'. h f' e))$
 $\implies \text{monotone } R (\geq) (\lambda f'. \text{catch } (f f') (h f'))$

unfolding catch-def

apply (rule monotone-bind-handle-ge)

by auto

lemma catch-liftE: $\text{catch } (\text{liftE } g) h = g$

by (simp add: catch-def bind-handle-liftE)

lemma refines-catch:

assumes f: refines f f' s s' Q

assumes ll: $\bigwedge e e' t t'. Q (\text{Exn } e, t) (\text{Exn } e', t') \implies$

refines (h e) (h' e') t t' R

assumes lr: $\bigwedge e v' t t'. Q (\text{Exn } e, t) (\text{Result } v', t') \implies$

refines (h e) (return v') t t' R
assumes rl: $\bigwedge v e' t t'. Q (\text{Result } v, t) (\text{Exn } e', t') \implies$
refines (return v) (h' e') t t' R
assumes rr: $\bigwedge v v' t t'. Q (\text{Result } v, t) (\text{Result } v', t') \implies$
 $R (\text{Result } v, t) (\text{Result } v', t')$
shows *refines* (catch f h) (catch f' h') s s' R
unfolding catch-def
apply (rule *refines-bind-handle-bind-handle-exn*[OF f])
subgoal using ll **by** auto
subgoal using lr **by** auto
subgoal using rl **by** auto
subgoal using rr **by** (auto simp add: *refines-yield*)
done

lemma *rel-spec-catch*:

assumes f: *rel-spec* f f' s s' Q
assumes ll: $\bigwedge e e' t t'. Q (\text{Exn } e, t) (\text{Exn } e', t') \implies$
rel-spec (h e) (h' e') t t' R
assumes lr: $\bigwedge e v' t t'. Q (\text{Exn } e, t) (\text{Result } v', t') \implies$
rel-spec (h e) (return v') t t' R
assumes rl: $\bigwedge v e' t t'. Q (\text{Result } v, t) (\text{Exn } e', t') \implies$
rel-spec (return v) (h' e') t t' R
assumes rr: $\bigwedge v v' t t'. Q (\text{Result } v, t) (\text{Result } v', t') \implies$
 $R (\text{Result } v, t) (\text{Result } v', t')$
shows *rel-spec* (catch f h) (catch f' h') s s' R
using *assms* **by** (auto simp: *rel-spec-iff-refines intro!*: *refines-catch*)

16.9.36 check

lemma *run-check*[*run-spec-monad*]: $\text{run } (\text{check } e \ p) \ s =$
(if ($\exists x. p \ s \ x$) *then* *Success* ($(\lambda x. (x, s)) \cdot (\text{Result } \cdot \{x. p \ s \ x\})$) *else*
pure-post-state ($\text{Exn } e, s$)
by (auto simp add: *check-def run-bind*)

lemma *always-progress-check*[*always-progress-intros, simp*]: *always-progress* (check e p)
by (auto simp add: *always-progress-def run-check*)

lemma *runs-to-check-iff*[*runs-to-iff*]:
 $(\text{check } e \ p) \cdot s \ \{\!\! \{ Q \}\!\!\} \longleftrightarrow (\text{if } (\exists x. p \ s \ x) \text{ then } (\forall x. p \ s \ x \longrightarrow Q (\text{Result } x) \ s) \text{ else } Q (\text{Exn } e) \ s)$
by (auto simp add: *runs-to.rep-eq run-check*)

lemma *runs-to-check*[*runs-to-vcg*]:
 $(\exists x. p \ s \ x \implies (\forall x. p \ s \ x \longrightarrow Q (\text{Result } x) \ s)) \implies (\forall x. \neg p \ s \ x) \implies Q (\text{Exn } e)$
 $s \implies$
 $\text{check } e \ p \cdot s \ \{\!\! \{ Q \}\!\!\}$
by (simp add: *runs-to-check-iff*)

lemma *runs-to-partial-check*[*runs-to-vcg*]:
 $(\exists x. p \ s \ x \Longrightarrow (\forall x. p \ s \ x \longrightarrow Q \ (Result \ x) \ s)) \Longrightarrow (\forall x. \neg p \ s \ x) \Longrightarrow Q \ (Exn \ e)$
 $s \Longrightarrow$
check e p · s ?{Q}
by (*simp add: runs-to-partial.rep-eq run-check*)

lemma *refines-check-right-ok*:
 $Q \ t \ a \Longrightarrow \text{refines } f \ (g \ a) \ s \ t \ R \Longrightarrow \text{refines } f \ (\text{check } q \ Q \ >>= \ g) \ s \ t \ R$
by (*rule refines-bind-right-single*[*of a t*])
(auto simp add: run-check)

lemma *refines-check-right-fail*:
 $(\forall a. \neg Q \ t \ a) \Longrightarrow f \cdot s \ \{\!| \lambda r \ s'. R \ (r, \ s') \ (Exn \ q, \ t) \ |\!\} \Longrightarrow \text{refines } f \ (\text{check } q \ Q \ >>= \ g) \ s \ t \ R$
apply (*rule refines-bind-right-runs-to-partialI*[*OF runs-to-partial-of-runs-to*])
apply (*auto simp: runs-to-check-iff Exn-def refines-yield-right-iff*)
done

lemma *refines-throwError-check*:
 $(\forall a. \neg P \ t \ a) \Longrightarrow R \ (Exn \ r, \ s) \ (Exn \ q, \ t) \Longrightarrow$
 $\text{refines } (\text{throw } r) \ (\text{check } q \ P \ >>= \ f) \ s \ t \ R$
by (*intro refines-check-right-fail auto*)

lemma *refines-condition-neg-check*:
 $P \ s \ \longleftrightarrow (\forall a. \neg Q \ t \ a) \Longrightarrow$
 $(P \ s \Longrightarrow \forall a. \neg Q \ t \ a \Longrightarrow R \ (Result \ r, \ s) \ (Exn \ q, \ t)) \Longrightarrow$
 $(\bigwedge a. \neg P \ s \Longrightarrow Q \ t \ a \Longrightarrow \text{refines } f \ (g \ a) \ s \ t \ R) \Longrightarrow$
 $\text{refines } (\text{condition } P \ (\text{return } r) \ f) \ (\text{check } q \ Q \ >>= \ g) \ s \ t \ R$
by (*intro refines-condition-left*)
(auto intro: refines-check-right-ok refines-check-right-fail)

16.9.37 ignoreE

named-theorems *ignoreE-simps* \langle *Rewrite rules to push **const** \langle ignoreE \rangle inside.* \rangle

lemma *ignoreE-eq*: *ignoreE f = vmap-value (map-exception-or-result (λx. undefined) id) f*
unfolding *ignoreE-def catch-def*
by *transfer*
(simp add: fun-eq-iff post-state-eq-iff eq-commute all-comm[where 'a='a]
split: prod.splits exception-or-result-splits)

lemma *run-ignoreE*: *run (ignoreE f) s =*
bind-post-state (run f s)
 $(\lambda(r, t). \text{case } r \ \text{of } Exn \ e \Rightarrow \perp \mid Result \ v \Rightarrow \text{pure-post-state } (Result \ v, \ t))$
unfolding *ignoreE-def* **by** (*simp add: run-catch*)

lemma *runs-to-ignoreE-iff*[*runs-to-iff*]:
 $(\text{ignoreE } f) \cdot s \ \{\!| \ Q \ |\!\} \ \longleftrightarrow f \cdot s \ \{\!| \lambda r \ t. (\forall v. r = Result \ v \longrightarrow Q \ (Result \ v) \ t) \ |\!\}$

unfolding *ignoreE-eq runs-to.rep-eq*
by *transfer*
(simp add: split-beta' eq-commute prod-eq-iff split: prod.splits exception-or-result-splits)

lemma *runs-to-ignoreE[runs-to-vcg]*:
 $f \cdot s \{ \lambda r t. (\forall v. r = \text{Result } v \longrightarrow Q (\text{Result } v) t) \} \Longrightarrow (\text{ignoreE } f) \cdot s \{ Q \}$
by *(simp add: runs-to-ignoreE-iff)*

lemma *liftE-le-iff-le-ignoreE*: $\text{liftE } f \leq g \longleftrightarrow f \leq \text{ignoreE } g$
unfolding *liftE-def ignoreE-eq[abs-def]*
by *transfer (simp add: le-fun-def vmap-post-state-le-iff-le-map-post-state id-def)*

lemma *runs-to-partial-ignoreE[runs-to-vcg]*:
 $f \cdot s \{ \lambda r t. (\forall v. r = \text{Result } v \longrightarrow Q (\text{Result } v) t) \} \Longrightarrow$
 $(\text{ignoreE } f) \cdot s \{ Q \}$
unfolding *ignoreE-def*
apply *(runs-to-vcg)*
apply *(fastforce simp add: runs-to-partial.rep-eq)*
done

lemma *mono-ignoreE*: $f \leq f' \Longrightarrow \text{ignoreE } f \leq \text{ignoreE } f'$
unfolding *ignoreE-def*
apply *(rule mono-catch)*
by *simp-all*

lemma *monotone-ignoreE-le[partial-function-mono]*:
 $\text{monotone } R (\leq) (\lambda f'. f f')$
 $\Longrightarrow \text{monotone } R (\leq) (\lambda f'. \text{ignoreE } (f f'))$
unfolding *ignoreE-def*
apply *(rule monotone-catch-le)*
by *simp-all*

lemma *monotone-ignoreE-ge[partial-function-mono]*:
 $\text{monotone } R (\geq) (\lambda f'. f f')$
 $\Longrightarrow \text{monotone } R (\geq) (\lambda f'. \text{ignoreE } (f f'))$
unfolding *ignoreE-def*
apply *(rule monotone-catch-ge)*
by *simp-all*

lemma *ignoreE-liftE [simp]*: $\text{ignoreE } (\text{liftE } f) = f$
by *(simp add: catch-liftE ignoreE-def)*

lemma *ignoreE-top[simp]*: $\text{ignoreE } \top = \top$
by *(rule spec-monad-eqI) (simp add: runs-to-iff)*

lemma *ignoreE-bot[simp]*: $\text{ignoreE } \text{bot} = \text{bot}$
by *(rule spec-monad-eqI) (simp add: runs-to-iff)*

lemma *ignoreE-fail[simp]*: $\text{ignoreE } \text{fail} = \text{fail}$

by (*rule spec-monad-eqI*) (*simp add: runs-to-iff*)

lemma *ignoreE-return[simp]*: *ignoreE (return x) = return x*
by (*rule spec-monad-eqI*) (*simp add: runs-to-iff*)

lemma *ignoreE-throw[simp]*: *ignoreE (throw x) = bot*
by (*rule spec-monad-eqI*) (*simp add: runs-to-iff*)

lemma *ignoreE-throw-Exception[simp]*: *e ≠ default ⇒ ignoreE (yield (Exception e)) = bot*
by (*rule spec-monad-eqI*) (*simp add: runs-to-iff*)

lemma *ignoreE-guard[simp]*: *ignoreE (guard P) = guard P*
by (*rule spec-monad-eqI*) (*simp add: runs-to-iff*)

lemma *ignoreE-get-state[simp]*: *ignoreE (get-state) = get-state*
by (*rule spec-monad-eqI*) (*simp add: runs-to-iff*)

lemma *ignoreE-set-state[simp]*: *ignoreE (set-state s) = set-state s*
by (*rule spec-monad-eqI*) (*simp add: runs-to-iff*)

lemma *ignoreE-select[simp]*: *ignoreE (select S) = select S*
by (*rule spec-monad-eqI*) (*simp add: runs-to-iff*)

lemma *ignoreE-unknown[simp]*: *ignoreE (unknown) = unknown*
by (*rule spec-monad-eqI*) (*simp add: runs-to-iff*)

lemma *ignoreE-exec-concrete[simp]*: *ignoreE (exec-concrete st f) = exec-concrete st (ignoreE f)*
by (*rule spec-monad-eqI*) (*auto simp add: runs-to-iff*)

lemma *ignoreE-assert[simp]*: *ignoreE (assert P) = assert P*
by (*rule spec-monad-eqI*) (*simp add: runs-to-iff*)

lemma *ignoreE-assume[simp]*: *ignoreE (assume P) = assume P*
by (*rule spec-monad-eqI*) (*simp add: runs-to-iff*)

lemma *ignoreE-gets[simp]*: *ignoreE (gets f) = gets f*
by (*rule spec-monad-eqI*) (*simp add: runs-to-iff*)

lemma *ignoreE-assert-opt[simp]*: *ignoreE (assert-opt v) = assert-opt v*
by (*rule spec-monad-eqI*) (*simp add: runs-to-iff*)

lemma *ignoreE-gets-the[simp]*: *ignoreE (gets-the f) = gets-the f*
by (*rule spec-monad-eqI*) (*auto simp add: runs-to-iff*)

lemma *ignoreE-modify[simp]*: *ignoreE (modify f) = modify f*
by (*rule spec-monad-eqI*) (*simp add: runs-to-iff*)

lemma *ignoreE-condition*[*ignoreE-simps*]:
ignoreE (condition *c f g*) = condition *c* (*ignoreE f*) (*ignoreE g*)
by (rule *spec-monad-eqI*) (simp add: *runs-to-iff*)

lemma *ignoreE-when*[*simp*]: *ignoreE* (when *P f*) = when *P* (*ignoreE f*)
by (simp add: *when-def ignoreE-condition*)

lemma *ignoreE-bind*[*ignoreE-simps*]:
ignoreE (bind *f g*) = bind (*ignoreE f*) ($\lambda v. (ignoreE (g v))$)
by (simp add: *spec-monad-eq-iff runs-to-ignoreE-iff runs-to-bind-iff*)

lemma *ignoreE-map-exn*[*simp*]: *ignoreE* (map-value (map-exn *f*) *g*) = *ignoreE g*
by (simp add: *spec-monad-eq-iff runs-to-iff*)

lemma *ignoreE-whileLoop*[*ignoreE-simps*]:
ignoreE (whileLoop *C B I*) = whileLoop *C* ($\lambda x. ignoreE (B x)$) *I*
proof –
have ($\lambda I. ignoreE (whileLoop C B I)$) = whileLoop *C* ($\lambda x. ignoreE (B x)$)
unfolding *whileLoop-def*
apply (rule *gfp-fusion*[*OF - mono-whileLoop-functional mono-whileLoop-functional*,
where *g*= $\lambda x a. liftE (x a)$])
subgoal by (simp add: *le-fun-def liftE-le-iff-le-ignoreE*)
subgoal by (simp add: *ignoreE-bind ignoreE-condition*)
done
then show *?thesis* **by** (simp add: *fun-eq-iff*)
qed

16.9.38 *on-exit'*

lemmas *bind-finally-def* = *bind-exception-or-result-def*

lemma *bind-exception-or-result-liftE-assoc*:
bind-exception-or-result (bind (*liftE f*) *g*) *h* = bind *f* ($\lambda v. bind-exception-or-result$
(*g v*) *h*)
by (rule *spec-monad-eqI*) (auto simp add: *runs-to-iff*)

lemma *bind-exception-or-result-bind-guard-assoc*:
bind-exception-or-result (bind (*guard P*) *g*) *h* =
bind (*guard P*) ($\lambda v. bind-exception-or-result (g v) h$)
by (rule *spec-monad-eqI*) (auto simp add: *runs-to-iff*)

lemma *on-exit-bind-exception-or-result-conv*:
on-exit f cleanup = *bind-exception-or-result f* ($\lambda x. do \{state-select cleanup; yield$
*x\})
by (simp add: *on-exit-def on-exit'-def*)*

lemma *guard-on-exit-bind-exception-or-result-conv*:
guard-on-exit f P cleanup =
bind-exception-or-result f ($\lambda x. do \{guard P; state-select cleanup; yield x\}$)

by (*simp add: on-exit'-def bind-assoc*)

lemma *assume-result-and-state-check-only-state*:

assume-result-and-state ($\lambda s. \{((), s'). s' = s \wedge P s\}$) = *assuming* *P*

by (*simp add: spec-monad-eq-iff runs-to-iff*)

lemma *assume-on-exit-bind-exception-or-result-conv*:

assume-on-exit *f P cleanup* = *bind-exception-or-result* *f*

($\lambda x. \text{do } \{ \text{assume-result-and-state } (\lambda s. \{((), s'). s' = s \wedge P s\}); \text{state-select cleanup; yield } x \}$)

by (*simp add: on-exit'-def bind-assoc assume-result-and-state-check-only-state*)

lemma *monotone-on-exit'-le[partial-function-mono]*:

monotone *R* (\leq) ($\lambda f'. f f'$) \implies *monotone* *R* (\leq) ($\lambda f'. g f'$) \implies

monotone *R* (\leq) ($\lambda f'. \text{on-exit}'(f f')(g f')$)

unfolding *on-exit'-def* **by** (*intro partial-function-mono*)

lemma *monotone-on-exit'-ge[partial-function-mono]*:

monotone *R* (\geq) ($\lambda f'. f f'$) \implies *monotone* *R* (\geq) ($\lambda f'. g f'$) \implies

monotone *R* (\geq) ($\lambda f'. \text{on-exit}'(f f')(g f')$)

unfolding *on-exit'-def* **by** (*intro partial-function-mono*)

lemma *runs-to-on-exit'-iff[runs-to-iff]*:

on-exit' *f c · s* $\{ Q \}$ \longleftrightarrow

f · s $\{ \lambda r t. c · t \{ \lambda q t. Q \text{ (case } q \text{ of Exception } e \Rightarrow \text{Exception } e \mid \text{Result } - \Rightarrow r) t \} \}$

by (*auto simp add: on-exit'-def runs-to-iff fun-eq-iff split: exception-or-result-split intro!: arg-cong[where f=runs-to - -]*)

lemma *runs-to-on-exit'[runs-to-vcg]*:

f · s $\{ \lambda r t. c · t \{ \lambda q t. Q \text{ (case } q \text{ of Exception } e \Rightarrow \text{Exception } e \mid \text{Result } - \Rightarrow r) t \} \}$

\implies *on-exit'* *f c · s* $\{ Q \}$

by (*simp add: runs-to-on-exit'-iff*)

lemma *runs-to-partial-on-exit'[runs-to-vcg]*:

f · s $\{ ? \{ \lambda r t. c · t \{ ? \{ \lambda q t. Q \text{ (case } q \text{ of Exception } e \Rightarrow \text{Exception } e \mid \text{Result } - \Rightarrow r) t \} \} \}$

\implies *on-exit'* *f c · s* $\{ ? \{ Q \}$

unfolding *on-exit'-def*

apply (*rule runs-to-partial-bind-exception-or-result*)

apply (*rule runs-to-partial-weaken, assumption*)

apply (*rule runs-to-partial-bind*)

apply (*rule runs-to-partial-weaken, assumption*)

apply *auto*

done

lemma *refines-on-exit'*:

assumes *f: refines f f' s s'*

($\lambda(r, t) (r', t'). \text{refines } c c' t t' (\lambda(q, t) (q', t')). R$

(*case* *q* of *Exception* *e* \Rightarrow *Exception* *e* \mid *Result* $- \Rightarrow r, t$)

(case q' of Exception $e' \Rightarrow$ Exception $e' \mid$ Result $- \Rightarrow r', t'$))
shows *refines* (on-exit' f c) (on-exit' f' c') s s' R
unfolding *on-exit'-def*
apply (rule *refines-bind-exception-or-result*[*OF* f [*THEN* *refines-weaken*]])
apply (*clarsimp* *intro!*: *refines-bind'*)
apply (rule *refines-weaken*, *assumption*)
apply *auto*
done

lemma *on-exit'-skip*: *on-exit' f skip = f*
by (*simp* *add*: *spec-monad-eq-iff* *runs-to-iff*)

16.9.39 *run-bind*

lemma *run-run-bind*: *run (run-bind f t g) s = bind-post-state (run f t) ($\lambda(r, t).$ run (g r t) s)*
by (*transfer*) *simp*

lemma *runs-to-run-bind-iff*[*runs-to-iff*]:
(run-bind f t g) \cdot s $\{\!\{Q\}\!\}$ \longleftrightarrow f \cdot t $\{\!\{\lambda r t. (g r t) \cdot s \{\!\{Q\}\!\}\!\}$
by (*simp* *add*: *runs-to.rep-eq* *run-run-bind* *split-beta'*)

lemma *runs-to-run-bind*[*runs-to-vcg*]:
f \cdot t $\{\!\{\lambda r t. (g r t) \cdot s \{\!\{Q\}\!\}\!\}$ \Longrightarrow (run-bind f t g) \cdot s $\{\!\{Q\}\!\}$
by (*simp* *add*: *runs-to-run-bind-iff*)

lemma *runs-to-partial-run-bind*[*runs-to-vcg*]:
f \cdot t $\{\!\{\lambda r t. (g r t) \cdot s \{\!\{Q\}\!\}\!\}$ \Longrightarrow (run-bind f t g) \cdot s $\{\!\{Q\}\!\}$
by (*auto* *simp*: *runs-to-partial.rep-eq* *run-run-bind* *split-beta'*
intro!: *holds-partial-bind-post-state*)

lemma *mono-run-bind*: *f \leq f' \Longrightarrow g \leq g' \Longrightarrow run-bind f t g \leq run-bind f' t g'*
apply (rule *le-spec-monad-runI*)
apply (*simp* *add*: *run-run-bind*)
apply (rule *mono-bind-post-state*)
apply (*auto* *simp* *add*: *le-fun-def* *less-eq-spec-monad.rep-eq* *split*: *exception-or-result-splits*)
done

lemma *monotone-run-bind-le*[*partial-function-mono*]:
*monotone R (\leq) ($\lambda f'. f f')$ \Longrightarrow ($\bigwedge r t.$ *monotone R* (\leq) ($\lambda f'. g f' r t$))*
 \Longrightarrow *monotone R* (\leq) ($\lambda f'.$ *run-bind* ($f f'$) t ($g f'$))
apply (*simp* *add*: *monotone-def*)
using *mono-run-bind*
by (*metis* *le-fun-def*)

lemma *monotone-run-bind-ge*[*partial-function-mono*]:
*monotone R (\geq) ($\lambda f'. f f')$ \Longrightarrow ($\bigwedge r t.$ *monotone R* (\geq) ($\lambda f'. g f' r t$))*
 \Longrightarrow *monotone R* (\geq) ($\lambda f'.$ *run-bind* ($f f'$) t ($g f'$))
apply (*simp* *add*: *monotone-def*)

using *mono-run-bind*
by (*metis le-fun-def*)

lemma *liftE-run-bind*: $\text{liftE } (\text{run-bind } f \ t \ g) = \text{run-bind } f \ t \ (\lambda r \ t. \text{liftE } (g \ r \ t))$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *exec-concrete-run-bind*: $\text{exec-concrete } st \ f =$
do {
 $s \leftarrow \text{get-state};$
 $t \leftarrow \text{select } \{t. \text{st } t = s\};$
 $\text{run-bind } f \ t \ (\lambda r' \ t'. \text{do } \{\text{set-state } (st \ t'); \text{yield } r' \})$
}
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *exec-abstract-run-bind*: $\text{exec-abstract } st \ f =$
do {
 $s \leftarrow \text{get-state};$
 $\text{run-bind } f \ (st \ s) \ (\lambda r' \ t'. \text{do } \{s' \leftarrow \text{select } \{s'. \text{t}' = st \ s'\}; \text{set-state } s'; \text{yield } r' \})$
}
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *refines-run-bind*:
 $\text{refines } f \ f' \ x \ y \ (\lambda(r,x') (r', y'). \text{refines } (g \ r \ x') (g' \ r' \ y') \ s \ t \ R) \implies$
 $\text{refines } (\text{run-bind } f \ x \ g) (\text{run-bind } f' \ y \ g') \ s \ t \ R$
apply *transfer*
apply (*rule sim-bind-post-state, assumption*)
apply *auto*
done

16.9.40 Iteration of monadic actions

fun *iter-spec-monad* **where**
 $\text{iter-spec-monad } f \ 0 = \text{skip} \mid$
 $\text{iter-spec-monad } f \ (\text{Suc } n) = \text{do } \{\text{iter-spec-monad } f \ n; f \ n \}$

lemma *iter-spec-monad-unfold*:
 $0 < n \implies \text{iter-spec-monad } f \ n = \text{do } \{\text{iter-spec-monad } f \ (n-1); f \ (n-1) \}$
by (*metis Suc-pred' iter-spec-monad.simps(2)*)

lemma *iter-spec-monad-cong*:
 $(\bigwedge j. j < i \implies f \ j = g \ j) \implies \text{iter-spec-monad } f \ i = \text{iter-spec-monad } g \ i$
by (*induction i*) *auto*

16.9.41 *forLoop*

definition *forLoop*:: $int \Rightarrow int \Rightarrow (int \Rightarrow ('a, 's) \text{ res-monad}) \Rightarrow (int, 's) \text{ res-monad}$
where

forLoop $z (m::int) B = \text{whileLoop } (\lambda i s. i < m) (\lambda i. \text{do } \{B\ i; \text{return } (i + 1)\}) z$

lemma *runs-to-forLoop*:

assumes $z \leq m$

assumes $I: \bigwedge r s. I r s \Longrightarrow z \leq r \Longrightarrow r < m \Longrightarrow (B r) \cdot s \{\!\! \{ \lambda t. I (r + 1) t \}\!\!\}$

assumes $Q: \bigwedge s. I m s \Longrightarrow Q m s$

shows $I z s \Longrightarrow (\text{forLoop } z m B) \cdot s \{\!\! \{ \lambda \text{Res } r. Q r \}\!\!\}$

unfolding *forLoop-def*

using $\langle z \leq m \rangle Q I$

by (*intro runs-to-whileLoop-res*[**where** $I = \lambda q t. I q t \wedge z \leq q \wedge q \leq m$

and $P = Q$

and $R = \text{measure } (\lambda(i, -). \text{nat } (m - i))$])

(*auto intro!*: *runs-to-bind*)

lemma *whileLoop-cong-inv*:

run $((\text{whileLoop } C f z)::('a, 's) \text{ res-monad}) s = \text{run } (\text{whileLoop } D g z) s$

if $I: \bigwedge i s. I i s \Longrightarrow C i s \Longrightarrow f i \cdot s \{\!\! \{ \lambda v' s'. \forall i'. v' = \text{Result } i' \longrightarrow I i' s' \}\!\!\}$

and $I\text{-eq}: \bigwedge i s. I i s \Longrightarrow C i s \Longrightarrow \text{run } (f i) s = \text{run } (g i) s$

and $C\text{-eq}: \bigwedge i s. I i s \Longrightarrow C i s \longleftrightarrow D i s$

and $I z s$

for $s::'s$ **and** $z::'a$ **and** $R::('a \times 's) \text{ rel}$

proof –

have *rel-spec* $(\text{whileLoop } C f z) (\text{whileLoop } D g z) s s$

$(\lambda(\text{Res } i, s) (\text{Res } i', s'). i = i' \wedge s = s' \wedge I i s)$

apply (*intro rel-spec-whileLoop-res*)

subgoal for $i s j t$ **using** $C\text{-eq}[of\ i\ s]$ **by** *auto*

subgoal premises *prems* **for** $i s j t$

using *prems* $I[of\ i\ s]$

apply (*clarsimp simp: rel-spec.rep-eq runs-to-partial.rep-eq I-eq*

intro!: rel-post-state-refl)

apply (*auto intro: holds-partial-post-state-weaken*)

done

subgoal using $\langle I z s \rangle$ **by** *simp*

done

from *rel-spec-mono*[*OF - this, of (=)*]

show *?thesis*

by (*auto simp: le-fun-def rel-spec-eq*)

qed

lemma *forLoop-skip*: *forLoop* $z m f = \text{return } z$ **if** $z \geq m$

using *that* **unfolding** *forLoop-def*

by (*subst whileLoop-unroll*) *simp*

lemma *forLoop-eq-whileLoop*: *forLoop* $z m f = \text{whileLoop } C g z$

if $\bigwedge i. z \leq i \Longrightarrow i < m \Longrightarrow g i = \text{do } \{ y \leftarrow f i; \text{return } (i + 1) \}$

$\bigwedge i s. z \leq i \Longrightarrow i \leq m \Longrightarrow C i s \longleftrightarrow i < m$

```

   $\bigwedge s. z > m \implies \neg C z s$ 
proof cases
  assume  $z \leq m$ 
  then show ?thesis
    unfolding forLoop-def
    apply (intro spec-monad-ext ext whileLoop-cong-inv[where  $I = \lambda i s. z \leq i \wedge i \leq m$ ])
    apply (auto simp: that intro!: runs-to-partial-bind)
    done
next
  assume  $\neg z \leq m$ 
  then show ?thesis
    apply (subst whileLoop-unroll)
    apply (simp add: forLoop-skip)
    apply (rule spec-monad-ext)
    apply (simp add: that)
    done
qed

```

```

lemma runs-to-partial-forLoopE: forLoop  $z m f \cdot s \text{ ?} \llbracket \lambda x s'. \forall v. x = \text{Result } v \longrightarrow v = m \rrbracket$ 
  if  $z \leq m$ 
proof -
  have  $*$ : forLoop  $z m f = \text{whileLoop } (\lambda i -. i \neq m) (\lambda i. \text{do } \{f\} i; \text{return } (i + 1)) z$ 
    using  $\langle z \leq m \rangle$ 
    by (auto intro!: forLoop-eq-whileLoop)
  show ?thesis unfolding  $*$ 
    by (rule runs-to-partial-whileLoop-cond-false[THEN runs-to-partial-weaken])
  simp
qed

```

16.9.42 whileLoop-unroll-reachable

```

context fixes  $C :: 'a \Rightarrow 's \Rightarrow \text{bool}$  and  $B :: 'a \Rightarrow ('e::\text{default}, 'a, 's) \text{ spec-monad}$ 
begin

```

```

inductive whileLoop-unroll-reachable  $:: 'a \Rightarrow 's \Rightarrow 'a \Rightarrow 's \Rightarrow \text{bool}$  for  $a s$  where
  initial[intro, simp]: whileLoop-unroll-reachable  $a s a s$ 
| step:  $\bigwedge b t X c u. \text{whileLoop-unroll-reachable } a s b t \implies C b t \implies \text{run } (B b) t = \text{Success } X \implies (\text{Result } c, u) \in X \implies \text{whileLoop-unroll-reachable } a s c u$ 

```

```

lemma whileLoop-unroll-reachable-trans:
  assumes  $a\text{-}b$ : whileLoop-unroll-reachable  $a s b t$  and  $b\text{-}c$ : whileLoop-unroll-reachable  $b t c u$ 
  shows whileLoop-unroll-reachable  $a s c u$ 
proof (use b-c in <induction arbitrary: >)
  case (step  $c u X d v$ )

```

show ?case
 by (rule whileLoop-unroll-reachable.step[OF step(5,2,3,4)])
qed (simp add: a-b)

end

lemma run-whileLoop-unroll-reachable-cong:

assumes eq:
 $\bigwedge b t. \text{whileLoop-unroll-reachable } C B a s b t \implies C b t \longleftrightarrow C' b t$
 $\bigwedge b t. \text{whileLoop-unroll-reachable } C B a s b t \implies C b t \implies \text{run } (B b) t = \text{run } (B' b) t$
shows $\text{run } (\text{whileLoop } C B a) s = \text{run } (\text{whileLoop } C' B' a) s$
proof –
have rel-spec (whileLoop C B a) (whileLoop C' B' a) s s
 ($\lambda(x', s') (y, t). y = x' \wedge t = s' \wedge$
 $(\forall a'. x' = \text{Result } a' \longrightarrow \text{whileLoop-unroll-reachable } C B a s a' s')$)
apply (rule rel-spec-whileLoop)
subgoal by (simp add: eq)
subgoal for a' s'
 using whileLoop-unroll-reachable.step[of C B a s a' s']
 by (cases run (B' a') s') (auto simp add: rel-spec-def eq intro!: rel-set-refl)
subgoal by simp
subgoal for a' s' b' t' by (cases a') (auto simp add: rel-exception-or-result.simps)
done
then have rel-spec (whileLoop C B a) (whileLoop C' B' a) s s (=)
 by (rule rel-spec-mono[rotated]) auto
then show ?thesis
 by (simp add: rel-spec-eq)
qed

16.9.43 on-exit

lemma refines-rel-prod-on-exit:

assumes f: refines f_c f_a s_c s_a (rel-prod R S')
assumes cleanup: $\bigwedge s_c s_a t_c. S' s_c s_a \implies (s_c, t_c) \in \text{cleanup}_c \implies \exists t_a. (s_a, t_a) \in \text{cleanup}_a \wedge S t_c t_a$
assumes emp: $\bigwedge s_c s_a. S' s_c s_a \implies \nexists t_c. (s_c, t_c) \in \text{cleanup}_c \implies \nexists t_a. (s_a, t_a) \in \text{cleanup}_a$
shows refines (on-exit f_c cleanup_c) (on-exit f_a cleanup_a) s_c s_a (rel-prod R S)
unfolding on-exit-bind-exception-or-result-conv
apply (rule refines-bind-exception-or-result)
apply (rule refines-mono [OF - f])
apply (clarsimp)
apply (rule refines-bind')
apply (rule refines-state-select)
using cleanup emp
apply auto
done

lemma *refines-runs-to-partial-rel-prod-on-exit*:
assumes *f*: *refines* $f_c f_a s_c s_a$ (*rel-prod* $R S'$)
assumes *runs-to*: $f_c \cdot s_c \{ \lambda r t. P t \}$
assumes *cleanup*: $\bigwedge s_c s_a t_c. S' s_c s_a \implies (s_c, t_c) \in \text{cleanup}_c \implies P s_c \implies \exists t_a. (s_a, t_a) \in \text{cleanup}_a \wedge S t_c t_a$
assumes *emp*: $\bigwedge s_c s_a. S' s_c s_a \implies P s_c \implies \nexists t_c. (s_c, t_c) \in \text{cleanup}_c \implies \nexists t_a. (s_a, t_a) \in \text{cleanup}_a$
shows *refines* (*on-exit* $f_c \text{ cleanup}_c$) (*on-exit* $f_a \text{ cleanup}_a$) $s_c s_a$ (*rel-prod* $R S$)
proof –
from *refines-runs-to-partial-fuse* [*OF* *f runs-to*]
have *refines* $f_c f_a s_c s_a (\lambda(r, t) (r', t'). \text{rel-prod } R S' (r, t) (r', t') \wedge P t)$.
moreover have $(\lambda(r, t) (r', t'). \text{rel-prod } R S' (r, t) (r', t') \wedge P t) = \text{rel-prod } R (\lambda t t'. S' t t' \wedge P t)$
by (*auto simp add: rel-prod-conv*)
ultimately have *refines* $f_c f_a s_c s_a (\text{rel-prod } R (\lambda t t'. S' t t' \wedge P t))$ **by** *simp*
then show *thesis*
apply (*rule refines-rel-prod-on-exit*)
subgoal using *cleanup* **by** *blast*
subgoal using *emp* **by** *blast*
done
qed

lemma *rel-spec-monad-mono*:
assumes *Q*: *rel-spec-monad* $R Q f g$ **and** QQ' : $\bigwedge x y. Q x y \implies Q' x y$
shows *rel-spec-monad* $R Q' f g$
proof –
have *rel-spec-monad* $R Q \leq \text{rel-spec-monad } R Q'$
unfolding *rel-spec-monad-def* **using** QQ'
by (*auto simp add: rel-post-state.simps rel-set-def*)
(*metis rel-prod-sel*)
with *Q* **show** *thesis* **by** *auto*
qed

lemma *gets-return*: *gets* $(\lambda-. x) = \text{return } x$
by (*rule spec-monad-eqI*) (*auto simp add: runs-to-iff*)

lemma *bind-handle-bind-exception-or-result-conv*:
bind-handle $f g h =$
bind-exception-or-result f
 $(\lambda \text{Exception } e \Rightarrow h e \mid \text{Result } v \Rightarrow g v)$
apply (*rule spec-monad-eqI*)
by (*auto simp add: runs-to-iff*)
(*auto elim!: runs-to-weaken split: exception-or-result-splits*)

lemma *bind-handle-bind-exception-or-result-conv-exn*: *bind-handle* $f g (\lambda \text{Some } e \Rightarrow h e) =$
bind-exception-or-result f
 $(\lambda \text{Exn } e \Rightarrow h e \mid \text{Result } v \Rightarrow g v)$

by (simp add: bind-handle-bind-exception-or-result-conv case-xval-def)

lemma *try-nested-bind-exception-or-result-conv*:

shows $try\ (f\ >>= g) =$

(bind-exception-or-result f
 $(\lambda Exn\ e \Rightarrow (case\ e\ of\ Inl\ l \Rightarrow throw\ l\ |\ Inr\ r \Rightarrow return\ r)$
 $|\ Result\ v \Rightarrow try\ (g\ v)))$)

apply (rule spec-monad-eqI)

by (auto simp add: runs-to-iff)

(auto elim!: runs-to-weaken simp add: runs-to-iff unnest-exn-def
split: xval-splits sum.splits)

lemma *try-nested-bind-handle-conv*:

shows $try\ (f\ >>= g) =$

(bind-handle $f\ (\lambda v. try\ (g\ v))$
 $(\lambda Some\ e \Rightarrow (case\ e\ of\ Inl\ l \Rightarrow throw\ l\ |\ Inr\ r \Rightarrow return\ r)))$)

by (simp add: bind-handle-bind-exception-or-result-conv-exn try-nested-bind-exception-or-result-conv)

definition *no-fail*:: $('s \Rightarrow bool) \Rightarrow ('e::default, 'a, 's)\ spec-monad \Rightarrow bool$ **where**

no-fail $P\ f \equiv \forall s. P\ s \longrightarrow run\ f\ s \neq \top$

definition *no-throw*:: $('s \Rightarrow bool) \Rightarrow ('e::default, 'a, 's)\ spec-monad \Rightarrow bool$ **where**

no-throw $P\ f \equiv \forall s. P\ s \longrightarrow f \cdot s\ ?\{\} \lambda r\ t. \exists v. r = Result\ v\}$

definition *no-return*:: $('s \Rightarrow bool) \Rightarrow ('e::default, 'a, 's)\ spec-monad \Rightarrow bool$ **where**

no-return $P\ f \equiv \forall s. P\ s \longrightarrow f \cdot s\ ?\{\} \lambda r\ t. \exists e. r = Exception\ e \wedge e \neq default\}$

lemma *no-return-exn-def*: $no-return\ P\ f \longleftrightarrow (\forall s. P\ s \longrightarrow f \cdot s\ ?\{\} \lambda r\ t. \exists e. r = Exn\ e\}$)

by (auto simp add: no-return-def Exn-def default-option-def elim!: runs-to-partial-weaken)

lemma *no-throw-gets[simp]*: $no-throw\ P\ (gets\ f)$

by (auto simp add: no-throw-def runs-to-partial-def)

lemma *no-throw-modify[simp]*: $no-throw\ P\ (modify\ f)$

by (auto simp add: no-throw-def runs-to-partial-def)

lemma *no-throw-select[simp]*: $no-throw\ P\ (select\ f)$

by (auto simp add: no-throw-def runs-to-partial-def)

lemma *always-progress-select-UNIV[simp]*: $always-progress\ (select\ UNIV)$

by (auto simp add: always-progress-def bot-post-state-def)

lemma *rel-spec-monad-rel-xval-catch*:

assumes *fh*: $rel-spec-monad\ R\ (rel-xval\ E\ Q)\ f\ h$

assumes *gi*: $rel-fun\ E\ (rel-spec-monad\ R\ (rel-xval\ E2\ Q))\ g\ i$

shows $rel-spec-monad\ R\ (rel-xval\ E2\ Q)\ (f\ <catch>\ g)\ (h\ <catch>\ i)$

using *assms* **unfolding** *rel-spec-monad-iff-rel-spec*

by (auto intro!: rel-spec-catch simp: rel-fun-def)

16.10 Setup for Tagging

lemma *runs-to-tag*:

($\llbracket \text{tag} \implies f \cdot s \ \{ P \} \rrbracket$) \implies ($\text{tag} \mid f$) $\cdot s \ \{ P \}$
unfolding *TAG-def ASM-TAG-def* **by** *auto*

lemma *runs-to-tag-guard*:

fixes $g :: 'a \Rightarrow \text{bool}$
and $s :: 'a$
assumes $\text{tag} \mid g \ s$
assumes $P \ (\text{Result } ()) \ s$
shows ($\text{tag} \mid \text{guard } g$) $\cdot s \ \{ P \}$
using *assms unfolding TAG-def* **by** – (rule *runs-to-weaken*, rule *runs-to-guard*;
simp)

bundle *runs-to-vcg-tagging-setup*
begin

unbundle *basic-vcg-tagging-setup*

lemmas [*runs-to-vcg*] = *runs-to-tag runs-to-tag-guard*

end

end

theory *Reaches*

imports *Spec-Monad*

begin

The core notions on spec monads are

- Properties of a monad: *runs-to*, *runs-to-partial*
- Refinement / Simulation of monads: *refines*, *rel-spec* and *rel-spec-monad*.

It is considered good style to use these more abstract concepts as much as possible.

Next we introduce some notions that are more in a 'pointwise' spirit in the sense that they talk about a particular outcome of a monadic computation, e.g. that a particular state is a reachable outcome of a monadic computation.

There is nothing *wrong* with these notions but we consider them as less elegant and more 'brute force'. So we encourage to think twice before using them and prefer the more abstract notions.

primrec *outcomes* :: 'a post-state \Rightarrow 'a set **where**
outcomes Failure = {}
| *outcomes (Success X)* = X

lemma *le-post-state-iff*:
 $p \leq q \longleftrightarrow (q \neq \text{Failure} \longrightarrow (p \neq \text{Failure} \wedge \text{outcomes } p \subseteq \text{outcomes } q))$
by (cases q; cases p) simp-all

lemma *outcomes-map[simp]*: *outcomes (map-post-state f x)* = f ' *outcomes x*
by (cases x) simp-all

lemma *map-post-state-eq-Failure[simp]*:
map-post-state f x = Failure \longleftrightarrow $x = \text{Failure}$
by (cases x; simp)

lemma *outcomes-Sup[simp]*: *Failure* \notin $F \implies \text{outcomes (Sup } F) = (\bigcup_{f \in F} \text{outcomes } f)$
by (auto simp: Sup-post-state-def)
(metis *outcomes.simps(2)* *post-state.exhaust vimage-eq*)

lemmas *runs-to-holds-def = runs-to.rep-eq*
lemmas *runs-to-partial-holds-partial-def = runs-to-partial.rep-eq*

16.11 *succeeds and reaches*

definition *succeeds* :: ('e::default, 'a, 's) spec-monad \Rightarrow 's \Rightarrow bool **where**
succeeds f s \longleftrightarrow *run f s* $\neq \top$

definition *reaches* :: ('e::default, 'a, 's) spec-monad \Rightarrow 's \Rightarrow ('e, 'a) exception-or-result
 \Rightarrow 's \Rightarrow bool **where**
reaches f s r' s' \longleftrightarrow $(r', s') \in \text{outcomes (run f s)}$

There seems to a similarity between what happens in HOL with the dualism of sets as type vs. the characterisation as predicate as expressed in *Collect*. Similar 'a post-state has a set in the *Success* case. In particular with *holds-post-state* we switch to the predicate view. Which is then continued in *runs-to* which is the predicate view and the set view with *reaches*, which is more the element wise set view similar to $x \in X$ and finally culminates in $(?f = ?g) = (\forall P s. (?f \cdot s \ \S P \ \S) = (?g \cdot s \ \S P \ \S))$. The predicate view seems to be the one we finally aim for. So maybe we could get completely rid of the set like things and start of with a predicate already in 'a post-state?

lemma *runs-to-partial-def-old*:
runs-to-partial f s Q \longleftrightarrow (*succeeds f s* \longrightarrow $(\forall r t. \text{reaches } f s r t \longrightarrow Q r t)$)
unfolding *succeeds-def reaches-def runs-to-partial.rep-eq*
by (cases run f s) (auto simp: top-post-state-def)

lemma *runs-to-def-old*: $runs\text{-}to\ f\ s\ Q \longleftrightarrow succeeds\ f\ s \wedge (\forall r\ t. reaches\ f\ s\ r\ t \longrightarrow Q\ r\ t)$

unfolding *succeeds-def reaches-def runs-to.rep-eq*
by (*cases run f s*) (*auto simp: top-post-state-def*)

lemma *runs-to-succeeds-runsto-partial-conv*:

$runs\text{-}to\ f\ s\ Q \longleftrightarrow (succeeds\ f\ s \wedge runs\text{-}to\text{-}partial\ f\ s\ Q)$
by (*auto simp add: runs-to-def-old runs-to-partial-def-old*)

lemma *run-Success-succeeds*: $run\ f\ s = Success\ x \Longrightarrow succeeds\ f\ s$

by (*auto simp add: succeeds-def top-post-state-def*)

lemma *runs-to-partial-le-outcomes-conv*:

$runs\text{-}to\text{-}partial\ f\ s\ Q \longleftrightarrow (outcomes\ (run\ f\ s)) \leq \{(r,t). Q\ r\ t\}$

apply (*simp add: runs-to-partial-def-old succeeds-def outcomes-def reaches-def top-post-state-def*)

apply *transfer*

apply *standard*

subgoal

by (*auto split: post-state.splits*)

subgoal

by *blast*

done

lemma *not-succeeds-empty-outcomes*: $\neg succeeds\ f\ s \Longrightarrow outcomes\ (run\ f\ s) = \{\}$

by (*simp add: succeeds-def top-post-state-def*)

lemma *reaches-succeeds*: $reaches\ f\ s\ r\ t \Longrightarrow succeeds\ f\ s$

apply (*simp add: reaches-def succeeds-def top-post-state-def*)

apply *transfer*

apply *auto*

done

lemma *always-progress-succeeds-reaches-conv*:

$always\text{-}progress\ f \longleftrightarrow (\forall s. succeeds\ f\ s \longrightarrow (\exists r\ t. reaches\ f\ s\ r\ t))$

apply (*simp add: always-progress-def succeeds-def reaches-def*)

apply *transfer*

apply (*clarsimp simp add: top-post-state-def bot-post-state-def, safe*)

apply (*metis ex-in-conv outcomes.simps(2) post-state.exhaust surj-pair*)

by (*metis empty-iff post-state.distinct(1) outcomes.simps(2)*)

lemma *le-succeedsD*: $g \leq f \Longrightarrow succeeds\ f\ s \Longrightarrow succeeds\ g\ s$

apply (*simp add: succeeds-def*)

apply *transfer*

by (*auto simp add: le-fun-def less-eq-post-state.simps top-post-state-def*)

lemma *outcomes-succeeds-run-conv*: $outcomes\ (run\ f\ s) = X \Longrightarrow succeeds\ f\ s \Longrightarrow run\ f\ s = Success\ X$

by (*cases run f s*) (*auto simp add: succeeds-def top-post-state-def*)

lemma *succeeds-outcomes-run-eqI*: $\text{succeeds } f \ s \longleftrightarrow \text{succeeds } g \ s \implies$
 $(\text{succeeds } f \ s \implies \text{succeeds } g \ s \implies (\text{outcomes } (\text{run } f \ s) = \text{outcomes } (\text{run } g \ s)))$
 \implies
 $\text{run } f \ s = \text{run } g \ s$
by (*cases run f s; cases run g s*) (*auto simp add: succeeds-def top-post-state-def*)

lemma *succeeds-outcomes-spec-monad-eqI*:
 $(\bigwedge s. \text{succeeds } f \ s \longleftrightarrow \text{succeeds } g \ s) \implies$
 $(\bigwedge s. \text{succeeds } f \ s \implies \text{succeeds } g \ s \implies (\text{outcomes } (\text{run } f \ s) = \text{outcomes } (\text{run } g \ s)))$
 \implies
 $f = g$
apply (*rule spec-monad-ext*)
apply (*auto intro: succeeds-outcomes-run-eqI*)
done

lemma *succeeds-reaches-spec-monad-eqI*:
 $(\bigwedge s. \text{succeeds } f \ s \longleftrightarrow \text{succeeds } g \ s) \implies$
 $(\bigwedge s \ r \ t. \text{succeeds } f \ s \implies \text{succeeds } g \ s \implies (\text{reaches } f \ s \ r \ t \longleftrightarrow \text{reaches } g \ s \ r \ t)) \implies$
 $f = g$
apply (*rule succeeds-outcomes-spec-monad-eqI*)
apply *simp*
apply (*auto simp add: reaches-def*)
done

lemma *succeeds-runs-to-iff*: $\text{succeeds } f \ s \longleftrightarrow \text{runs-to } f \ s \ (\lambda - . \text{True})$
by (*simp add: runs-to-def-old*)

named-theorems *outcomes-spec-monad* *Simplification rules for outcomes of Spec monad*

16.11.1 Relational rewriting for Monads

lemma *refines-def-old*:
 $\text{refines } f \ g \ s \ t \ R \longleftrightarrow$
 $(\text{succeeds } g \ t \longrightarrow \text{succeeds } f \ s \wedge$
 $(\forall r \ s'. \text{reaches } f \ s \ r \ s' \longrightarrow (\exists x \ t'. \text{reaches } g \ t \ x \ t' \wedge R \ (r, \ s') \ (x, \ t'))))$
by (*cases run g t; cases run f s*)
 $(\text{force } \text{simp: } \text{refines.rep-eq } \text{succeeds-def } \text{reaches-def } \text{top-post-state-def})+$

lemma *rel-specI*:
assumes $\text{succeeds } f \ s \longleftrightarrow \text{succeeds } g \ t$
assumes $\bigwedge r \ s'. \text{reaches } f \ s \ r \ s' \implies \exists q \ t'. \text{reaches } g \ t \ q \ t' \wedge R \ (r, \ s') \ (q, \ t')$
assumes $\bigwedge q \ t'. \text{reaches } g \ t \ q \ t' \implies \exists r \ s'. \text{reaches } f \ s \ r \ s' \wedge R \ (r, \ s') \ (q, \ t')$
shows $\text{rel-spec } f \ g \ s \ t \ R$
using *assms*
by (*cases run f s; cases run g t;*
fastforce simp: rel-spec-def rel-prod.simps rel-set-def
reaches-def succeeds-def top-post-state-def)

lemma *rel-specD-succeeds*:
 $rel\text{-spec } f g s t R \implies succeeds f s \longleftrightarrow succeeds g t$
by (*cases run f s*; *cases run g t*;
simp add: rel-post-state.simps rel-spec-def rel-prod.simps succeeds-def top-post-state-def)

lemma *rel-specD-reaches1*:
 $rel\text{-spec } f g s t R \implies reaches f s r s' \implies \exists q t'. reaches g t q t' \wedge R (r, s') (q, t')$
by (*cases run f s*; *cases run g t*;
fastforce simp: rel-post-state.simps rel-spec-def rel-prod.simps rel-set-def reaches-def top-post-state-def)

lemma *rel-specD-reaches2*:
 $rel\text{-spec } f g s t R \implies reaches g t q t' \implies \exists r s'. reaches f s r s' \wedge R (r, s') (q, t')$
by (*cases run f s*; *cases run g t*;
fastforce simp: rel-post-state.simps rel-spec-def rel-prod.simps rel-set-def reaches-def top-post-state-def)

lemma *refines-iff*:
 $refines f g s t R \longleftrightarrow$
 $((succeeds g t \longrightarrow succeeds f s) \wedge$
 $(succeeds g t \longrightarrow succeeds f s \longrightarrow$
 $(\forall r s'. reaches f s r s' \longrightarrow (\exists q t'. reaches g t q t' \wedge R (r, s') (q, t')))))$
by (*simp add: refines-def-old blast*)

lemma *refinesI*:
assumes $succeeds g t \implies succeeds f s$
assumes $\bigwedge r s'. succeeds g t \implies succeeds f s \implies reaches f s r s' \implies$
 $(\exists q t'. reaches g t q t' \wedge R (r, s') (q, t'))$
shows $refines f g s t R$
using *assms*
by (*auto simp add: refines-def-old*)

lemma *refinesD-succeeds*:
 $refines f g s t R \implies succeeds g t \implies succeeds f s$
by (*auto simp: refines-def-old*)

lemma *refinesD-reaches*:
 $refines f g s t R \implies reaches f s r s' \implies succeeds g t$
 $\implies \exists q t'. reaches g t q t' \wedge R (r, s') (q, t')$
by (*fastforce simp: refines-def-old*)

lemma *refines-strengthen'*:
assumes $refines f g s t R$
assumes $\bigwedge r u q v. reaches f s r u \implies reaches g t q v \implies succeeds f s \implies$
 $succeeds g t$
 $\implies R (r, u) (q, v) \implies Q (r, u) (q, v)$
shows $refines f g s t Q$
using *assms* **by** (*fastforce simp: refines-def-old*)

lemma *always-progressD*: $\text{always-progress } f \implies \text{succeeds } f s \implies \exists r t. \text{reaches } f s r t$

by (*auto simp: always-progress-succeeds-reaches-conv*)

lemma *Ex-reaches*: $f \cdot s \{ \! \! \! \} P \{ \! \! \! \} \implies \text{always-progress } f \implies \exists r t. \text{reaches } f s r t$

by (*auto simp: runs-to-def-old always-progress-succeeds-reaches-conv*)

lemma *witness-outcomes-succeeds*: $x \in \text{outcomes } (\text{run } f s) \implies \text{succeeds } f s$

using *not-succeeds-empty-outcomes* **by** *fastforce*

lemma *runs-toD2*: $f \cdot s \{ \! \! \! \} P \{ \! \! \! \} \implies \text{reaches } f s r t \implies P r t$

by (*simp add: runs-to-def-old*)

lemma *runs-toD2-res*: **fixes** $f :: ('a, 's) \text{res-monad}$

shows $f \cdot s \{ \! \! \! \} \lambda Res r. P r \{ \! \! \! \} \implies \text{reaches } f s (\text{Result } r) t \implies P r t$

by (*simp add: runs-to-def-old*)

lemma *rel-post-state-runI*:

assumes $\text{succeeds } f s \longleftrightarrow \text{succeeds } g t$

assumes $\text{succeeds } f s \implies \text{succeeds } g t \implies \text{rel-set } R (\text{outcomes } (\text{run } f s)) (\text{outcomes } (\text{run } g t))$

shows $\text{rel-post-state } R (\text{run } f s) (\text{run } g t)$

using *assms*

by (*cases run f s; cases run g t*)

(*auto simp add: rel-post-state.simps succeeds-def top-post-state-def*)

lemma *rel-post-state-runI'*:

assumes $\text{succeeds } f s \longleftrightarrow \text{succeeds } g t$

assumes $\text{succeeds } f s \implies \text{succeeds } g t \implies \text{rel-post-state } R (\text{run } f s) (\text{run } g t)$

shows $\text{rel-post-state } R (\text{run } f s) (\text{run } g t)$

using *assms*

by (*auto simp add: rel-post-state.simps succeeds-def top-post-state-def*)

lemma *rel-post-state-runD*:

assumes $\text{rel-post-state } R (\text{run } f s) (\text{run } g t)$

shows $(\text{succeeds } f s \longleftrightarrow \text{succeeds } g t) \wedge$

$(\text{succeeds } f s \longrightarrow \text{succeeds } g t \longrightarrow \text{rel-set } R (\text{outcomes } (\text{run } f s)) (\text{outcomes } (\text{run } g t)))$

using *assms*

by (*auto simp add: rel-post-state.simps succeeds-def top-post-state-def*)

lemma *rel-post-state-run-iff*:

$\text{rel-post-state } R (\text{run } f s) (\text{run } g t) \longleftrightarrow$

$((\text{succeeds } f s \longleftrightarrow \text{succeeds } g t) \wedge$

$(\text{succeeds } f s \longrightarrow \text{succeeds } g t \longrightarrow \text{rel-set } R (\text{outcomes } (\text{run } f s)) (\text{outcomes } (\text{run } g t))))$

using *rel-post-state-runI rel-post-state-runD* **by** *metis*

lemma *rel-spec-monad-succeeds-iff*: $rel\text{-spec-monad } R \ Q \ f \ g \implies R \ s \ t \implies \text{succeeds } f \ s \longleftrightarrow \text{succeeds } g \ t$

by (*meson rel-specD-succeeds rel-spec-monad-iff-rel-spec*)

lemma *outcomes-top-ps[simp]*: $outcomes \top = \{ \}$ $outcomes (pure\text{-post-state } x) = \{x\}$

outcomes $\perp = \{ \}$

by (*auto simp: top-post-state-def pure-post-state-def bot-post-state-def*)

16.11.2 \top

lemma *outcomes-top[outcomes-spec-monad]*: $outcomes (run \top s) = \{ \}$

by *transfer (simp add: top-spec-monad-def top-post-state-def)*

lemma *succeeds-top[simp]*: $\text{succeeds } \top \ s \longleftrightarrow \text{False}$

unfolding *succeeds-def*

by *transfer (simp add: top-spec-monad-def top-post-state-def)*

lemma *reaches-top[simp]*: $\text{reaches } \top \ s \ r \ t \longleftrightarrow \text{False}$

unfolding *reaches-def*

by (*simp add: outcomes-top top-post-state-def*)

16.11.3 \perp

lemma *outcomes-bot[outcomes-spec-monad]*: $outcomes (run \perp s) = \{ \}$

by *transfer (simp add: bot-spec-monad-def bot-post-state-def)*

lemma *succeeds-bot[simp]*: $\text{succeeds } \perp \ s \longleftrightarrow \text{True}$

unfolding *succeeds-def*

by *transfer (simp add: bot-spec-monad-def bot-post-state-def top-post-state-def)*

lemma *reaches-bot[simp]*: $\text{reaches } \perp \ s \ r \ t \longleftrightarrow \text{False}$

unfolding *reaches-def*

by (*simp add: outcomes-bot bot-post-state-def*)

16.11.4 *fail*

lemma *outcomes-fail[outcomes-spec-monad]*: $outcomes (run \text{fail } s) = \{ \}$

by *transfer simp*

lemma *succeeds-fail-iff[iff]*: $\neg (\text{succeeds } \text{fail } f)$

by (*simp add: succeeds-def*)

lemma *reaches-fail-iff[iff]*: $\neg \text{reaches } \text{fail } s \ r \ t$

by (*simp add: reaches-def outcomes-spec-monad top-post-state-def*)

16.11.5 *yield*

lemma *succeeds-yield[simp]*: $\text{succeeds } (\text{yield } r) \ s$

unfolding *succeeds-def*

apply *transfer*
apply (*simp add: pure-post-state-def top-post-state-def*)
done

lemma *outcomes-yield* [*outcomes-spec-monad*]: *outcomes* (*run* (*yield* *r*) *s*) = {(*r*,
s)}
by (*simp add: pure-post-state-def*)

lemma *reaches-yield*[*simp*]: *reaches* (*yield* *v*) *s* *r* *t* \longleftrightarrow (*r* = *v* \wedge (*t* = *s*))
by (*simp add: reaches-def outcomes-spec-monad pure-post-state-def*)

16.11.6 *return*

lemma *outcomes-return* [*outcomes-spec-monad*]:
outcomes (*run* (*return* *v*) *s*) = {(*Result* *v*, *s*)}
by *transfer* (*simp add: pure-post-state-def*)

lemma *succeeds-return* [*iff*]: *succeeds* (*return* *v*) *s*
by (*simp add: succeeds-def top-post-state-def*)

lemma *reaches-return*[*simp*]: *reaches* (*return* *v*) *s* *r* *t* \longleftrightarrow (*r* = *Result* *v* \wedge (*t* =
s))
by (*simp add: reaches-def outcomes-spec-monad pure-post-state-def*)

lemma *refines-yield-left*:
refines (*yield* *x*) *f* *s* *t* *R*
if *succeeds* *f* *t* \implies *reaches* *f* *t* *x'* *s'* \wedge *R* (*x*, *s*) (*x'*, *s'*)
using *that*
unfolding *refines-def-old*
by (*auto simp: succeeds-def top-post-state-def intro!: bexI[where x=(x', s')]*)

16.11.7 *skip*

lemma *outcomes-skip* [*outcomes-spec-monad*]:
outcomes (*run* (*skip*) *s*) = {(*Result* (), *s*)}
by (*simp add: outcomes-spec-monad pure-post-state-def*)

lemma *succeeds-skip*: *succeeds* *skip* *s*
by *simp*

lemma *reaches-skip*: *reaches* (*return* *v*) *s* *r* *t* \longleftrightarrow (*r* = *Result* () \wedge (*t* = *s*))
by (*simp add: reaches-def outcomes-spec-monad pure-post-state-def*)

lemma *runs-to-skip*[*runs-to-vcg*]: *skip* \cdot *s* $\{\lambda$ - *t*. *t* = *s* $\}$
by *simp*

lemma *runs-to-partial-skip*[*runs-to-vcg*]: *skip* \cdot *s* $\{?$ λ - *t*. *t* = *s* $\}$
by *simp*

lemma *runs-to-skip-iff*[*runs-to-iff*]: *skip* \cdot *s* $\{Q\}$ \longleftrightarrow *Q* (*Result* ()) *s*

by (simp add: runs-to-def-old)

16.11.8 throw-exception-or-result

lemma *outcomes-throw-exception-or-result* [outcomes-spec-monad]:
outcomes (run (throw-exception-or-result x) s) = {(Exception x, s)}
by transfer (simp add: pure-post-state-def)

lemma *succeeds-throw-exception-or-result* [iff]: succeeds (throw-exception-or-result v) s
by (simp add: succeeds-def top-post-state-def)

lemma *reaches-throw-exception-or-result*[simp]:
reaches (throw-exception-or-result x) s r t \longleftrightarrow (r = Exception x \wedge (t = s))
by (simp add: reaches-def outcomes-spec-monad pure-post-state-def)

16.11.9 throw

lemma *outcomes-throw* [outcomes-spec-monad]:
outcomes (run (throw e) s) = {(Exn e, s)}
by (simp add: pure-post-state-def)

16.11.10 get-state

lemma *outcomes-get-state* [outcomes-spec-monad]:
outcomes (run get-state s) = {(Result s, s)}
by transfer (simp add: pure-post-state-def)

lemma *succeeds-get-state* [iff]: succeeds get-state s
by (simp add: succeeds-def)

lemma *reaches-get-state*[simp]:
reaches get-state s r t \longleftrightarrow (r = Result s \wedge (t = s))
by (simp add: reaches-def outcomes-spec-monad pure-post-state-def)

16.11.11 set-state

lemma *outcomes-set-state* [outcomes-spec-monad]:
outcomes (run (set-state t) s) = {(Result (), t)}
by transfer (simp add: pure-post-state-def)

lemma *succeeds-set-state* [iff]: succeeds (set-state t) s
by (simp add: succeeds-def)

lemma *reaches-set-state*[simp]: reaches (set-state s') s r t \longleftrightarrow (r = Result () \wedge (t = s'))
by (simp add: reaches-def outcomes-spec-monad pure-post-state-def)

16.11.12 *select*

lemma *succeeds-holds*: *succeeds f s* \longleftrightarrow *holds-post-state* ($\lambda x. \text{True}$) (*run f s*)
by (*cases run f s*) (*simp-all add: succeeds-def top-post-state-def*)

lemma *succeeds-select*[*simp*]: *succeeds (select S) s*
apply (*simp add: succeeds-holds select-def*)
apply *transfer*
apply (*auto simp add: pure-post-state-def*)
done

lemma *select-outcomes*[*outcomes-spec-monad*]: *outcomes (run (select S) s) =* ($\lambda v. (\text{Result } v, s)$) ‘ *S*
apply (*simp add: select-def*)
apply *transfer*
apply (*force simp add: pure-post-state-def outcomes-def Inf-set-def Sup-post-state-def split: post-state.splits prod.splits*)
done

lemma *reaches-select* [*simp*]: *reaches (select S) s r t* \longleftrightarrow ($r \in \text{Result } ‘ S \wedge t = s$)
apply (*auto simp add: reaches-def outcomes-spec-monad*)
done

16.11.13 *unknown*

lemma *succeeds-unknown*[*simp*]: *succeeds unknown s*
unfolding *unknown-def* by *simp*

lemma *unknown-outcomes*[*outcomes-spec-monad*]: *outcomes (run unknown s) =* ($\lambda v. (\text{Result } v, s)$) ‘ *UNIV*
by *simp*

lemma *reaches-unknown* [*simp*]: *reaches unknown s r t* \longleftrightarrow ($r \in \text{Result } ‘ UNIV \wedge t = s$)
unfolding *unknown-def* by *simp*

16.11.14 *lift-state*

lemma *succeeds-lift-state-iff*: *succeeds (lift-state R f) s* \longleftrightarrow ($\forall s'. R s s' \longrightarrow \text{succeeds } f s'$)
by (*simp add: succeeds-holds lift-state-def Spec-inverse*)

16.11.15 *const exec-concrete*

lemma *succeeds-exec-concrete-iff*: *succeeds (exec-concrete st f) s* \longleftrightarrow ($\forall t. s = st \longrightarrow \text{succeeds } f t$)
by (*simp add: exec-concrete-def succeeds-lift-state-iff*)

lemma *reaches-exec-concrete*:
assumes *succeeds*: *succeeds (exec-concrete st f) s*

shows $\text{reaches } (\text{exec-concrete } st \ f) \ s \ r \ s' \longleftrightarrow (\exists t \ t'. \ s = st \ t \wedge \text{reaches } f \ t \ r \ t' \wedge s' = st \ t')$
apply (*simp add: reaches-def run-exec-concrete*)
apply *standard*
subgoal
by (*auto simp add: map-post-state-def Sup-post-state-def image-def vimage-def split: prod.splits post-state.splits if-split-asm*)
(metis outcomes.simps(2))
subgoal
using *succeeds*
apply (*simp add: succeeds-def run-exec-concrete*)
by (*auto simp add: map-post-state-def Sup-post-state-def image-def vimage-def top-post-state-def split: prod.splits post-state.splits if-split-asm*)
(metis apsnd-conv outcomes.simps(2) post-state.exhaust)
done

16.11.16 *const exec-abstract*

lemma *succeeds-exec-abstract-iff*[*simp*]: $\text{succeeds } (\text{exec-abstract } st \ f) \ s \longleftrightarrow \text{succeeds } f \ (st \ s)$
by (*simp add: exec-abstract-def succeeds-lift-state-iff*)

lemma *reaches-exec-abstract*:

shows $\text{reaches } (\text{exec-abstract } st \ f) \ s \ r \ s' \longleftrightarrow (\exists t \ t'. \ \text{reaches } f \ (st \ s) \ r \ t' \wedge t' = st \ s')$
by (*cases run f (st s) (simp-all add: reaches-def run-exec-abstract)*)

16.11.17 *bind-handle*

lemma *succeeds-bind-handle*:

$\text{succeeds } (\text{bind-handle } f \ g \ h) \ s \longleftrightarrow$
 $(\text{succeeds } f \ s \wedge$
 $(\forall x \ s'. \ \text{reaches } f \ s \ x \ s' \longrightarrow$
 $(\text{case } x \ \text{of } \text{Exception } e \Rightarrow \text{succeeds } (h \ e) \ s' \mid \text{Result } v \Rightarrow \text{succeeds } (g \ v) \ s')))$

apply (*cases run f s*)

apply (*simp-all add:*

succeeds-def run-bind-handle bot-post-state-def top-post-state-def image-iff eq-commute[of Failure] reaches-def split: prod.splits exception-or-result-splits)

apply *auto*

done

lemma *succeeds-bind-handle-res*:

$\text{succeeds } (\text{bind-handle } (f::('s, 'a) \ \text{res-monad}) \ g \ h) \ s \longleftrightarrow$
 $(\text{succeeds } f \ s \wedge$
 $(\forall v \ s'. \ \text{reaches } f \ s \ (\text{Result } v) \ s' \longrightarrow$
 $\text{succeeds } (g \ v) \ s'))$

by (*simp add: succeeds-bind-handle*)

lemma *outcomes-bind-handle-succeeds*: $\text{succeeds } (\text{bind-handle } f g h) s \implies$
 $\text{outcomes } (\text{run } (\text{bind-handle } f g h) s) =$
 $\bigcup ((\lambda(r, s'). \text{ case } r \text{ of } \text{Exception } e \implies \text{outcomes } (\text{run } (h e) s')$
 $| \text{Result } v \implies \text{outcomes } (\text{run } (g v) s'))$
 $\quad '(\text{outcomes } (\text{run } f s)))$
apply (*cases run f s*)
apply (*simp-all add: succeeds-bind-handle succeeds-def run-bind-handle bind-post-state-eq-top*)
apply (*simp add: top-post-state-def image-iff split-beta'*
 $\quad \text{split: exception-or-result-splits}$)
apply (*intro SUP-cong refl*)
subgoal for $X x$ **by** (*cases fst x auto*)
done

lemma *outcomes-bind-handle-succeeds-res*: $\text{succeeds } (\text{bind-handle } (f::('a, 's) \text{ res-monad})$
 $g h) s \implies$
 $\text{outcomes } (\text{run } (\text{bind-handle } f g h) s) =$
 $\bigcup ((\lambda(r, s'). \text{ case } r \text{ of } \text{Exception } e \implies \{ \} | \text{Result } v \implies \text{outcomes } (\text{run } (g v)$
 $s'))$
 $\quad '(\text{outcomes } (\text{run } f s)))$
apply (*simp add: outcomes-bind-handle-succeeds*)
apply (*auto split: exception-or-result-splits*)
done

lemma *reaches-bind-handle*: $\text{reaches } (\text{bind-handle } f g h) s r t \iff$
 $(\text{succeeds } (\text{bind-handle } f g h) s \wedge$
 $(\exists r' s'. \text{ reaches } f s r' s' \wedge$
 $\quad (\text{case } r' \text{ of}$
 $\quad \quad \text{Exception } e \implies \text{reaches } (h e) s' r t$
 $\quad \quad | \text{Result } v \implies \text{reaches } (g v) s' r t)))$
apply *standard*
subgoal
apply (*frule reaches-succeeds*)
apply (*simp add: reaches-def outcomes-bind-handle-succeeds*)
apply (*clarsimp split: exception-or-result-splits*)
apply (*metis (no-types, opaque-lifting) Exception-eq-Result the-Exception-simp*)
apply (*metis (no-types, opaque-lifting) Result-eq-Result the-Exception-simp*
 $\text{the-Exception-Result}$)
done
subgoal
apply (*simp add: reaches-def outcomes-bind-handle-succeeds*)
by (*auto split: exception-or-result-splits*)
 $(\text{metis } (\text{mono-tags}) \text{ case-exception-or-result-Exception case-exception-or-result-Result}$
 $\text{case-prod-conv exception-or-result-cases})$
done

lemma *reaches-bind-handle-res*: $\text{reaches } ((\text{bind-handle } (f::('a, 's) \text{ res-monad}) g h))$
 $s r t \iff$
 $(\text{succeeds } (\text{bind-handle } f g h) s \wedge$
 $(\exists v s'. \text{ reaches } f s (\text{Result } v) s' \wedge \text{reaches } (g v) s' r t))$

by (simp add: reaches-bind-handle)

16.11.18 (\gg)

lemma *succeeds-bind*:

succeeds (bind f g) s \longleftrightarrow

(*succeeds* f s \wedge ($\forall v s'$. *reaches* f s (Result v) s' \longrightarrow *succeeds* (g v) s'))

apply (simp add: bind-def succeeds-bind-handle)

apply (auto split: exception-or-result-splits)

done

lemma *outcomes-bind-succeeds*: *succeeds* (bind f g) s \implies *outcomes* (run (bind f g) s) =

\bigcup (($\lambda(r, s')$. *case* r of *Exception* e \implies {(*Exception* e, s')} | *Result* v \implies *outcomes* (run (g v) s'))
' (*outcomes* (run f s)))

by (simp add: bind-def outcomes-bind-handle-succeeds outcomes-spec-monad pure-post-state-def)

lemma *outcomes-bind-succeeds-res*: *succeeds* (bind (f::('a, 's) res-monad) g) s \implies *outcomes* (run (bind f g) s) =

\bigcup (($\lambda(r, s')$. *case* r of *Exception* e \implies {} | *Result* v \implies *outcomes* (run (g v) s'))
' (*outcomes* (run f s)))

by (simp add: bind-def outcomes-bind-handle-succeeds-res outcomes-spec-monad)

lemma *reaches-bind*: *reaches* (bind f g) s r t \longleftrightarrow

(*succeeds* (bind f g) s \wedge

($\exists r' s'$. *reaches* f s r' s' \wedge

(*case* r' of

Exception e \implies r = *Exception* e \wedge t = s'

| *Result* v \implies *reaches* (g v) s' r t))

by (simp add: bind-def reaches-bind-handle)

lemma *reaches-bind-res*: *reaches* ((bind f g)::('a, 's) res-monad) s r t \longleftrightarrow

(*succeeds* (bind f g) s \wedge

($\exists v s'$. *reaches* f s (Result v) s' \wedge *reaches* (g v) s' r t))

by (simp add: bind-def reaches-bind-handle-res)

lemma *runs-toD-outcomes*: f \cdot s $\llbracket P \rrbracket \implies$ (x, s) \in *outcomes* (run f s) \implies P x s

by (cases run f s) (auto simp: runs-to.rep-eq)

lemma *run-bind-reaches-cong*:

assumes f: run f s = run f' s

assumes g: $\bigwedge v s'$. *succeeds* f s \implies *succeeds* f' s \implies

reaches f' s (Result v) s' \implies run (g v) s' = run (g' v) s'

shows run (bind f g) s = run (bind f' g') s

using f g **apply** (cases run f' s) **apply** (auto simp add: run-bind succeeds-def

reaches-def intro!: SUP-cong

split : exception-or-result-splits)

done

lemma *refines-bind-right'*:
succeeds $g\ t \implies \text{reaches } g\ t\ (\text{Result } a)\ t' \implies$
refines $f\ (h\ a)\ s\ t'\ R \implies$
refines $f\ (g\ \gg = h)\ s\ t\ R$
by (*force simp: refines-iff succeeds-bind reaches-bind*)

lemma *outcomes-empty-bind*:
assumes *emp*: *outcomes* $(\text{run } f\ s) = \{\}$
shows *outcomes* $(\text{run } (f\ \gg = g)\ s) = \{\}$
proof (*cases run f s*)
case *Failure*
then show *?thesis* **by** (*auto simp add: run-bind*)
next
case (*Success x2*)
then show *?thesis using emp*
by (*auto simp add: run-bind bot-post-state-def*)
qed

lemma *refines-bind-bind-exn*:
assumes *f*: *refines* $f\ f'\ s\ s'\ Q$
assumes *ll*: $\bigwedge e\ e'\ t\ t'. Q\ (\text{Exn } e,\ t)\ (\text{Exn } e',\ t') \implies$
 $R\ (\text{Exn } e,\ t)\ (\text{Exn } e',\ t')$
assumes *lr*: $\bigwedge e\ v'\ t\ t'. Q\ (\text{Exn } e,\ t)\ (\text{Result } v',\ t') \implies$
refines $(\text{throw } e)\ (g'\ v')\ t\ t'\ R$
assumes *rl*: $\bigwedge v\ e'\ t\ t'. Q\ (\text{Result } v,\ t)\ (\text{Exn } e',\ t') \implies$
refines $(g\ v)\ (\text{throw } e')\ t\ t'\ R$
assumes *rr*: $\bigwedge v\ v'\ t\ t'. Q\ (\text{Result } v,\ t)\ (\text{Result } v',\ t') \implies$
refines $(g\ v)\ (g'\ v')\ t\ t'\ R$
shows *refines* $(f\ \gg = g)\ (f'\ \gg = g')\ s\ s'\ R$
apply (*rule refines-bind'*)
apply (*rule refines-mono [OF - f]*)
using *assms*
apply (*auto simp add: default-option-def Exn-def*)
done

16.11.19 *assert*

lemma *outcomes-assert[outcomes-spec-monad]*:
outcomes $(\text{run } (\text{assert } P)\ s) = (\text{if } P\ \text{then } \{(\text{Result } (),\ s)\}\ \text{else } \{\})$
by (*simp add: assert-def outcomes-bind-succeeds outcomes-spec-monad*)

lemma *succeeds-assert[simp]*: *succeeds* $(\text{assert } P)\ s \longleftrightarrow P$
by (*simp add: assert-def*)

lemma *reaches-assert*[simp]: $\text{reaches } (\text{assert } P) \ s \ r \ t \longleftrightarrow (P \wedge r = \text{Result } () \wedge t = s)$

by (*simp add: assert-def*)

16.11.20 *assume*

lemma *outcomes-assume*[*outcomes-spec-monad*]:

$\text{outcomes } (\text{run } (\text{assume } P) \ s) = (\text{if } P \text{ then } \{(\text{Result } (), s)\} \text{ else } \{\})$

by (*simp add: assume-def outcomes-spec-monad*)

lemma *succeeds-assume*[simp]: $\text{succeeds } (\text{assume } P) \ s$

by (*simp add: assume-def*)

lemma *reaches-assume*[simp]: $\text{reaches } (\text{assume } P) \ s \ r \ t \longleftrightarrow (P \wedge r = \text{Result } () \wedge t = s)$

by (*simp add: assume-def*)

16.11.21 *assume-outcome*

lemma *outcomes-assume-outcome*[*outcomes-spec-monad*]:

$\text{outcomes } (\text{run } (\text{assume-outcome } f) \ s) = f \ s$

by *simp*

lemma *succeeds-assume-outcome*[simp]:

$\text{succeeds } (\text{assume-outcome } f) \ s$

by (*simp add: succeeds-def top-post-state-def*)

lemma *reaches-assume-outcome*[simp]:

$\text{reaches } (\text{assume-outcome } f) \ s \ r \ t \longleftrightarrow (r, t) \in f \ s$

by (*simp add: reaches-def*)

16.11.22 *assume-result-and-state*

lemma *outcomes-assume-result-and-state*[*outcomes-spec-monad*]:

$\text{outcomes } (\text{run } (\text{assume-result-and-state } f) \ s) = ((\lambda(v, t). (\text{Result } v, t)) \ ' \ f \ s)$

by *simp*

lemma *succeeds-assume-result-and-state*[simp]: $\text{succeeds } (\text{assume-result-and-state } f) \ s$

by (*simp add: assume-result-and-state-def succeeds-bind*)

lemma *Union-outcomes-split*:

$(\bigcup x \in f \ s.$

$\text{outcomes } (\text{run } (\text{case } x \text{ of } (v, y) \Rightarrow g \ v \ y) \ s))) = (\bigcup (v, y) \in f \ s. \text{outcomes } (\text{run } (g \ v \ y) \ s))$

using *Sup.SUP-cong* **by** *auto*

lemma *reaches-assume-result-and-state* [simp]: $\text{reaches } (\text{assume-result-and-state } f) \ s \ r \ t \longleftrightarrow (\exists v. r = \text{Result } v \wedge (v, t) \in f \ s)$

by (*auto simp add: reaches-def outcomes-spec-monad*)

16.11.23 *gets*

lemma *succeeds-gets*[simp]: *succeeds (gets f) s*
by (*simp add: gets-def succeeds-bind*)

lemma *outcomes-gets*[simp]: *outcomes (run (gets f) s) = {(Result (f s), s)}*
by *simp*

lemma *reaches-gets*[simp]: *reaches (gets f) s r t \longleftrightarrow (r = Result (f s) \wedge t = s)*
by (*simp add: reaches-def*)

16.11.24 *assert-result-and-state*

lemma *succeeds-assert-result-and-state*[simp]: *succeeds (assert-result-and-state f) s*
 \longleftrightarrow *f s \neq {}*
by (*simp add: assert-result-and-state-def succeeds-bind*)

lemma *outcomes-assert-result-and-state*[*outcomes-spec-monad*]:
outcomes (run (assert-result-and-state f) s) = (if f s = {} then {} else ((λ (v, t). (Result v, t)) ‘ f s))

proof (*cases f s = {}*)

case *True*

hence \neg *succeeds (assert-result-and-state f) s* **by** *simp*

hence *outcomes (run (assert-result-and-state f) s) = {}*

by (*simp add: not-succeeds-empty-outcomes*)

thus *?thesis* **by** (*simp add: True*)

next

case *False*

then show *?thesis*

by (*auto simp add: assert-result-and-state-def run-bind Sup-Success-pair pure-post-state-def*)

qed

lemma *reaches-assert-result-and-state* [simp]: *reaches (assert-result-and-state f) s r t \longleftrightarrow (\exists v. r = Result v \wedge (v, t) \in f s)*
by (*auto simp add: reaches-def outcomes-spec-monad*)

16.11.25 *assuming*

lemma *succeeds-assuming*[simp]: *succeeds (assuming g) s*
by (*simp add: assuming-def succeeds-bind*)

lemma *outcomes-assuming*[*outcomes-spec-monad*]: *outcomes (run (assuming g) s)*
 $=$ (*if g s then {(Result (), s)} else {}*)
by (*simp add: run-assuming*)

lemma *reaches-assuming*[simp]: *reaches (assuming g) s r t \longleftrightarrow (g s \wedge r = Result () \wedge t = s)*
by (*simp add: reaches-def outcomes-assuming*)

16.11.26 *guard*

lemma *succeeds-guard*[*simp*]: *succeeds* (*guard* *g*) *s* \longleftrightarrow *g* *s*
by (*simp* *add*: *guard-def* *succeeds-bind*)

lemma *outcomes-guard*[*outcomes-spec-monad*]: *outcomes* (*run* (*guard* *g*) *s*) = (*if* *g* *s* *then* $\{(Result\ (),\ s)\}$ *else* $\{\}$)
by (*simp* *add*: *run-guard*)

lemma *reaches-guard*[*simp*]: *reaches* (*guard* *g*) *s* *r* *t* \longleftrightarrow (*g* *s* \wedge *r* = *Result* () \wedge *t* = *s*)
by (*simp* *add*: *reaches-def* *outcomes-guard*)

16.11.27 *assert-opt*

lemma *succeeds-assert-opt*[*simp*]: *succeeds* (*assert-opt* *x*) *s* \longleftrightarrow $(\exists v. x = Some\ v)$
by (*simp* *add*: *assert-opt-def* *split*: *option.splits*)

lemma *outcomes-assert-opt*[*simp*]:
outcomes (*run* (*assert-opt* *x*) *s*) = (*case* *x* *of* *Some* *v* \Rightarrow $\{(Result\ v,\ s)\}$ | *None* \Rightarrow $\{\}$)
by (*simp* *split*: *option.splits*)

lemma *reaches-assert-opt*[*simp*]: *reaches* (*assert-opt* *x*) *s* *r* *t* \longleftrightarrow $(\exists v. x = Some\ v \wedge r = Result\ v \wedge t = s)$
by (*simp* *add*: *reaches-def* *split*: *option.splits*)

16.11.28 *gets-the*

lemma *succeeds-gets-the*[*simp*]: *succeeds* (*gets-the* *f*) *s* \longleftrightarrow $(\exists v. f\ s = Some\ v)$
by (*simp* *add*: *gets-the-def* *succeeds-bind*)

lemma *outcomes-gets-the*[*simp*]:
outcomes (*run* (*gets-the* *f*) *s*) = (*case* *f* *s* *of* *Some* *v* \Rightarrow $\{(Result\ v,\ s)\}$ | *None* \Rightarrow $\{\}$)
by (*simp* *split*: *option.splits*)

lemma *reaches-gets-the*[*simp*]: *reaches* (*gets-the* *f*) *s* *r* *t* \longleftrightarrow $(\exists v. f\ s = Some\ v \wedge r = Result\ v \wedge t = s)$
by (*simp* *add*: *reaches-def* *split*: *option.splits*)

16.11.29 *modify*

lemma *outcomes-run-modify*[*outcomes-spec-monad*]: *outcomes* (*run* (*modify* *f*) *s*) = $\{(Result\ (),\ f\ s)\}$
by *simp*

lemma *succeeds-modify*[*simp*]: *succeeds* (*modify* *f*) *s*
by (*simp* *add*: *modify-def* *succeeds-bind*)

lemma *reaches-modify[simp]*: $\text{reaches } (\text{modify } f) \ s \ r \ t \longleftrightarrow (r = \text{Result } () \wedge t = f \ s)$

by (*auto simp add: modify-def reaches-bind succeeds-bind*)

16.11.30 condition

lemma *outcomes-condition*:

$\text{outcomes } (\text{run } (\text{condition } c \ f \ g) \ s) = (\text{if } c \ s \ \text{then } \text{outcomes } (\text{run } f \ s) \ \text{else } \text{outcomes } (\text{run } g \ s))$

proof (*cases succeeds (condition c f g) s*)

case *True*

then show *?thesis* **by** (*auto simp add: condition-def outcomes-bind-succeeds outcomes-spec-monad*)

split: exception-or-result-splits)

next

case *False*

then show *?thesis*

by (*simp add: condition-def run-bind*)

qed

lemma *succeeds-condition-iff*:

$\text{succeeds } (\text{condition } c \ f \ g) \ s \longleftrightarrow \text{succeeds } (\text{if } c \ s \ \text{then } f \ \text{else } g) \ s$

by (*simp add: condition-def succeeds-bind*)

lemma *reaches-condition-iff*:

$\text{reaches } (\text{condition } c \ f \ g) \ s \ r' \ s' \longleftrightarrow \text{reaches } (\text{if } c \ s \ \text{then } f \ \text{else } g) \ s \ r' \ s'$

by (*simp add: reaches-def outcomes-condition*)

lemma *reaches-condition-True[simp]*:

$c \ s \implies \text{reaches } (\text{condition } c \ f \ g) \ s \ r' \ s' \longleftrightarrow \text{reaches } f \ s \ r' \ s'$

by (*simp add: reaches-condition-iff*)

lemma *reaches-condition-False[simp]*:

$\neg c \ s \implies \text{reaches } (\text{condition } c \ f \ g) \ s \ r' \ s' \longleftrightarrow \text{reaches } g \ s \ r' \ s'$

by (*simp add: reaches-condition-iff*)

lemma *succeeds-condition-True[simp]*:

$c \ s \implies \text{succeeds } (\text{condition } c \ f \ g) \ s \longleftrightarrow \text{succeeds } f \ s$

by (*simp add: succeeds-condition-iff*)

lemma *succeeds-condition-False[simp]*:

$\neg(c \ s) \implies \text{succeeds } (\text{condition } c \ f \ g) \ s \longleftrightarrow \text{succeeds } g \ s$

by (*simp add: succeeds-condition-iff*)

16.11.31 when

lemma *succeeds-when*: $\text{succeeds } (\text{when } c \ f) \ s \longleftrightarrow \neg c \vee \text{succeeds } f \ s$

unfolding *when-def* **by** (*simp add: succeeds-condition-iff*)

lemma *outcomes-when[outcomes-spec-monad]*:

outcomes (*run* (*when* *c f*) *s*) = (*if* *c* *then* *outcomes* (*run* *f s*) *else* {(*Result* (), *s*)})
by (*simp* *add*: *run-when*)

lemma *reaches-when-iff*:

reaches (*when* *c f*) *s r' s'* \longleftrightarrow (*if* *c* *then* *reaches* *f s r' s'* *else* (*r' = Result* () \wedge *s' = s*))
by (*simp* *add*: *reaches-def outcomes-when*)

16.11.32 While

lemma *reaches-whileLoop-cond-false*:

reaches (*whileLoop* *C B r*) *s (Result r')* *s'* $\implies \neg C r' s'$
using *runs-to-partial-whileLoop-cond-false*[*of C B r s*]
by (*auto simp*: *reaches-succeeds runs-to-partial-def-old*)

lemma *bind-post-state-cong*:

($\bigwedge r . r \in \text{outcomes } x \implies g r = g' r$) $\implies \text{bind-post-state } x g = \text{bind-post-state } x g'$
by (*cases* *x*) *simp-all*

16.11.33 map-value

lemma *succeeds-map-value*[*simp*]: *succeeds* (*map-value* *f g*) *s* \longleftrightarrow *succeeds* *g s*

by (*simp* *add*: *succeeds-def run-map-value top-post-state-def bot-post-state-def split: post-state.splits*)

lemma *outcomes-map-value*[*outcomes-spec-monad*]:

outcomes (*run* (*map-value* *f g*) *s*) = (($\lambda(v, s) . (f v, s)$) ‘ (*outcomes* (*run* *g s*)))
by (*auto simp* *add*: *always-progress-def run-map-value top-post-state-def bot-post-state-def split: post-state.splits*)

lemma *reaches-map-value*: *reaches* (*map-value* *f g*) *s r t* \longleftrightarrow ($\exists r' . \text{reaches } g s r' t \wedge r = f r'$)

by (*auto simp* *add*: *reaches-def outcomes-map-value*)

16.11.34 liftE

lemma *succeeds-liftE*[*simp*]: *succeeds* (*liftE* *f*) *s* \longleftrightarrow *succeeds* *f s*

by (*simp* *add*: *liftE-def*)

lemma *outcomes-liftE*[*outcomes-spec-monad*]:

outcomes (*run* (*liftE* *f*) *s*) = (($\lambda(v, s) . (\text{map-exception-or-result } (\lambda x . \text{undefined}) \text{id } v, s)$) ‘ (*outcomes* (*run* *f s*)))
by (*simp* *add*: *liftE-def outcomes-spec-monad*)

lemma *reaches-liftE*: *reaches* (*liftE* *f*) *s r t* \longleftrightarrow ($\exists r' . \text{reaches } f s (\text{Result } r') t \wedge r = \text{Result } r'$)

by (*auto simp* *add*: *liftE-def reaches-map-value*)

lemma *bind-cong1*:
fixes $f::('a, 's) \text{ res-monad}$
shows $\llbracket f = f'; \bigwedge v s s'. \text{reaches } f s (\text{Result } v) s' \implies g v = g' v \rrbracket \implies f >>= g$
 $= f' >>= g'$
apply (*rule spec-monad-ext*)
apply (*rule run-bind-reaches-cong*)
apply *auto*
done

lemma *bind-liftE-cong1*:
fixes $f::('a, 's) \text{ res-monad}$
shows $\llbracket f = f'; \bigwedge v s s'. \text{reaches } f s (\text{Result } v) s' \implies g v = g' v \rrbracket \implies$
 $(\text{liftE } f) >>= g = (\text{liftE } f') >>= g'$
apply (*rule spec-monad-ext*)
apply (*rule run-bind-reaches-cong*)
apply (*auto simp add: reaches-liftE*)
done

16.11.35 *try*

lemma *succeeds-try[simp]*: $\text{succeeds } (\text{try } f) s \longleftrightarrow \text{succeeds } f s$
by (*simp add: try-def*)

lemma *outcomes-try[outcomes-spec-monad]*:
 $\text{outcomes } (\text{run } (\text{try } f) s) = ((\lambda(v, s). (\text{unnest-exn } v, s)) \text{ ` } (\text{outcomes } (\text{run } f s)))$
by (*simp add: try-def outcomes-spec-monad*)

lemma *reaches-try*: $\text{reaches } (\text{try } f) s r t \longleftrightarrow (\exists r'. \text{reaches } f s r' t \wedge r = \text{unnest-exn } r')$
by (*simp add: try-def reaches-map-value*)

16.11.36 *finally*

lemma *succeeds-finally[simp]*: $\text{succeeds } (\text{finally } f) s \longleftrightarrow \text{succeeds } f s$
by (*simp add: finally-def*)

lemma *outcomes-finally[outcomes-spec-monad]*:
 $\text{outcomes } (\text{run } (\text{finally } f) s) = ((\lambda(v, s). (\text{unite } v, s)) \text{ ` } (\text{outcomes } (\text{run } f s)))$
by (*simp add: finally-def outcomes-spec-monad*)

lemma *reaches-finally*: $\text{reaches } (\text{finally } f) s r t \longleftrightarrow (\exists r'. \text{reaches } f s r' t \wedge r = \text{unite } r')$
by (*simp add: finally-def reaches-map-value*)

16.11.37 (*<catch>*)

lemma *succeeds-catch*: $\text{succeeds } (f \text{ <catch> } h) s \longleftrightarrow$
 $(\text{succeeds } f s \wedge$
 $(\forall x s'. \text{reaches } f s x s' \longrightarrow (\text{case } x \text{ of Exn } e \Rightarrow \text{succeeds } (h e) s' \mid \text{Result } v \Rightarrow \text{True})))$

unfolding *catch-def*

by (*fastforce simp add: succeeds-bind-handle default-option-def Exn-def split: exception-or-result-splits xval-splits*)

lemma *outcomes-catch-succeeds*: $\text{succeeds } (f \langle \text{catch} \rangle h) s \implies$

$\text{outcomes } (\text{run } (f \langle \text{catch} \rangle h) s) =$
 $\bigcup ((\lambda(r, s'). \text{case } r \text{ of } \text{Exn } e \implies \text{outcomes } (\text{run } (h e) s') \mid \text{Result } v \implies \{(\text{Result } v, s')\})$
 $\quad \text{'(outcomes } (\text{run } f s))$

unfolding *catch-def*

by (*auto simp add: outcomes-bind-handle-succeeds default-option-def Exn-def split: prod.splits exception-or-result-splits xval-splits*)

(*metis Exn-def Exn-neq-Result the-Exception-simp fst-conv option.sel snd-conv*

Pair-inject Result-eq-Result) $+$

lemma *reaches-catch*: $\text{reaches } (f \langle \text{catch} \rangle h) s r t \iff$

$(\text{succeeds } (f \langle \text{catch} \rangle h) s \wedge$
 $(\exists r' s'. \text{reaches } f s r' s' \wedge$
 $(\text{case } r' \text{ of}$
 $\quad \text{Exn } e \implies \text{reaches } (h e) s' r t$
 $\mid \text{Result } v \implies r = \text{Result } v \wedge t = s'))$

unfolding *catch-def*

by (*auto simp add: reaches-bind-handle default-option-def Exn-def split: prod.splits exception-or-result-splits xval-splits*)

(*metis option.sel*) $+$

16.11.38 *check*

lemma *succeeds-check[simp]*: $\text{succeeds } (\text{check } e p) s$

by (*simp add: succeeds-def run-check top-post-state-def*)

lemma *outcomes-check[outcomes-spec-monad]*: $\text{outcomes } (\text{run } (\text{check } e p) s) =$
 $(\text{if } (\exists x. p s x) \text{ then } ((\lambda x. (x, s)) \text{'(Result ' } \{x. p s x\})) \text{ else } \{(\text{Exn } e, s)\})$

by (*simp add: run-check*)

lemma *reaches-check*: $\text{reaches } (\text{check } e p) s r t \iff$

$(t = s) \wedge$
 $(\text{if } (\exists x. p s x) \text{ then } (\exists x. r = \text{Result } x \wedge p s x) \text{ else } r = \text{Exn } e)$

by (*auto simp add: reaches-def outcomes-check*)

lemma *refines-bind-ok*:

$\text{succeeds } g t \implies \text{reaches } g t (\text{Result } a) t' \implies$

$\text{refines } f (h a) s t' R \implies$

$\text{refines } f (g \gg= h) s t R$

by (*fastforce simp: refines-iff succeeds-bind reaches-bind*)

16.11.39 *ignoreE*

lemma *succeeds-ignoreE[simp]*:

succeeds (*ignoreE* *f*) *s* \longleftrightarrow (*succeeds* *f* *s*)
unfolding *ignoreE-def*
by (*auto simp add: succeeds-catch split: xval-splits*)

lemma *outcomes-ignoreE-succeeds*: *succeeds* *f* *s* \implies *outcomes* (*run* (*ignoreE* *f*) *s*)
 $=$
 $\bigcup ((\lambda(r, s'). \text{case } r \text{ of } \text{Exn } e \Rightarrow \{\} \mid \text{Result } v \Rightarrow \{(\text{Result } v, s')\})$
 $\quad \text{'(outcomes (run f s))})$
apply (*subst (asm) succeeds-ignoreE [symmetric]*)
by (*simp add: ignoreE-def outcomes-catch-succeeds*)

lemma *reaches-ignoreE*:
reaches (*ignoreE* *f*) *s* *r* *t* \longleftrightarrow (*succeeds* *f* *s*) \wedge ($\exists v. r = \text{Result } v \wedge \text{reaches } f \text{ s (Result } v) \text{ t}$)
apply (*simp add: reaches-catch succeeds-catch ignoreE-def*)
apply (*fastforce split: xval-splits*)
done

16.11.40 *bind-exception-or-result*

lemma *succeeds-bind-exception-or-result*:
succeeds (*bind-exception-or-result* *f* *g*) *s* \longleftrightarrow
succeeds *f* *s* \wedge ($\forall v \text{ s}'. \text{reaches } f \text{ s } v \text{ s}' \longrightarrow \text{succeeds } (g \text{ v}) \text{ s}'$)
by (*cases run f s*;
force simp: succeeds-def bind-exception-or-result.rep-eq
bind-post-state-eq-top top-post-state-def reaches-def)

lemma *reaches-bind-exception-or-result*:
reaches (*bind-exception-or-result* *f* *g*) *s* *r* *t* \longleftrightarrow
(*succeeds* (*bind-exception-or-result* *f* *g*) *s* \wedge
 $(\exists r' \text{ s}'. \text{reaches } f \text{ s } r' \text{ s}' \wedge \text{reaches } (g \text{ r}') \text{ s}' \text{ r } \text{ t})$)
apply (*cases run f s*)
apply (*simp-all add: reaches-def succeeds-def bind-exception-or-result.rep-eq*)
subgoal for *X*
apply (*cases* $\exists x \in X. \text{case } x \text{ of } (v, x) \Rightarrow \text{run } (g \text{ v}) \text{ x} = \top$)
apply (*simp-all add: Sup-post-state-def image-iff split-beta' Bex-def Ball-def*
eq-commute
flip: top-post-state-def)
by (*metis outcomes.simps(2) post-state.exhaust top-post-state-def*)
done

lemma *refines-bind-exception-or-result-strong*:
assumes *f*: *refines* *f* *f'* *s* *s'* *Q*
assumes *g*: $\bigwedge r \text{ t } r' \text{ t}'. Q (r, t) (r', t') \implies \text{reaches } f \text{ s } r \text{ t} \implies \text{reaches } f' \text{ s}' \text{ r}' \text{ t}'$
 $\implies \text{refines } (g \text{ r}) (g' \text{ r}') \text{ t } \text{ t}' \text{ R}$
shows *refines* (*bind-exception-or-result* *f* *g*) (*bind-exception-or-result* *f'* *g'*) *s* *s'* *R*
using *refines-bind-exception-or-result refines-mono f g*
by (*smt (verit, ccfu-threshold) case-prod-conv refines-strengthen'*)

16.11.41 *bind-finally*

lemma *succeeds-bind-finally*:

$succeeds (bind\text{-}finally\ f\ g)\ s = (succeeds\ f\ s \wedge (\forall v\ s'.\ reaches\ f\ s\ v\ s' \longrightarrow succeeds\ (g\ v)\ s'))$

by (*rule succeeds-bind-exception-or-result*)

lemma *reaches-bind-finally*: $reaches (bind\text{-}finally\ f\ g)\ s\ r\ t \longleftrightarrow (succeeds (bind\text{-}finally\ f\ g)\ s \wedge (\exists r'\ s'.\ reaches\ f\ s\ r'\ s' \wedge reaches\ (g\ r')\ s'\ r\ t))$

by (*rule reaches-bind-exception-or-result*)

16.11.42 *run-bind*

lemma *succeeds-run-bind*:

$succeeds (run\text{-}bind\ f\ t\ g)\ s \longleftrightarrow succeeds\ f\ t \wedge (\forall r\ t'.\ reaches\ f\ t\ r\ t' \longrightarrow succeeds\ (g\ r\ t')\ s)$

by (*cases run f t*)

(*force simp: succeeds-def run-run-bind top-post-state-def reaches-def split: post-state.splits*)⁺

lemma *reaches-run-bind*: $reaches (run\text{-}bind\ f\ t\ g)\ s\ r\ s' \longleftrightarrow$

$(succeeds (run\text{-}bind\ f\ t\ g)\ s) \wedge$

$(\exists r'\ t'.\ reaches\ f\ t\ r'\ t' \wedge reaches\ (g\ r'\ t')\ s\ r\ s')$

apply (*cases run f t*)

apply (*simp-all add: reaches-def run-run-bind succeeds-run-bind*)

apply (*simp add: succeeds-def split-beta'*)

subgoal for A

apply (*cases $\forall r\ t'. (r, t') \in A \longrightarrow run (g\ r\ t')\ s \neq \top$*)

subgoal by (*subst outcomes-Sup; force simp flip: top-post-state-def*)

subgoal

apply (*subst Sup-eq-Failure[THEN iffD2]*)

apply (*force simp flip: top-post-state-def*)

apply *auto*

done

done

done

lemma *runs-to-partial-reaches*: $f \cdot s \ ?\{\!\! \{ reaches\ f\ s \}\!\!\}$

apply (*cases run f s*)

apply (*auto simp add: runs-to-partial-def reaches-def*)

done

lemma *refines-strengthen-reaches*:

assumes *f-g: refines f g s t R*

assumes *reach: $(\bigwedge x\ s'\ y\ t'.\ R(x, s')(y, t') \implies reaches\ f\ s\ x\ s' \implies reaches\ g\ t\ y\ t' \implies Q(x, s')(y, t'))$*

shows *refines f g s t Q*

apply (*rule refines-strengthen [OF f-g runs-to-partial-reaches runs-to-partial-reaches]*)
)

using reach
by blast

lemma *refines-bind-bind-strong'*:
assumes *f*: *refines f f' s s' Q*
assumes *Ex-Ex*: $(\bigwedge e e' t t'. Q (\text{Exception } e, t) (\text{Exception } e', t') \implies e \neq \text{default} \implies e' \neq \text{default} \implies \text{reaches } f s (\text{Exception } e) t \implies \text{reaches } f' s' (\text{Exception } e') t' \implies R (\text{Exception } e, t) (\text{Exception } e', t'))$
assumes *Res-Ex*: $(\bigwedge e v' t t'. Q (\text{Exception } e, t) (\text{Result } v', t') \implies e \neq \text{default} \implies \text{reaches } f s (\text{Exception } e) t \implies \text{reaches } f' s' (\text{Result } v') t' \implies \text{refines } (\text{yield } (\text{Exception } e)) (g' v') t t' R)$
assumes *Ex-Ex*: $(\bigwedge v e' t t'. Q (\text{Result } v, t) (\text{Exception } e', t') \implies e' \neq \text{default} \implies \text{reaches } f s (\text{Result } v) t \implies \text{reaches } f' s' (\text{Exception } e') t' \implies \text{refines } (g v) (\text{yield } (\text{Exception } e')) t t' R)$
assumes *Res-Res*: $(\bigwedge v v' t t'. Q (\text{Result } v, t) (\text{Result } v', t') \implies \text{reaches } f s (\text{Result } v) t \implies \text{reaches } f' s' (\text{Result } v') t' \implies \text{refines } (g v) (g' v') t t' R)$
shows *refines (f >>= g) (f' >>= g') s s' R*
apply (*rule refines-bind'*)
apply (*rule refines-strengthen-reaches [OF f]*)
apply (*auto intro: assms*)
done

lemma *rel-spec-monad-bind-strong*:
assumes *f-f'*: *rel-spec-monad S P f f'*
assumes *Ex-Ex*: $\bigwedge e e' s s' t t'. S s s' \implies S t t' \implies P (\text{Exception } e) (\text{Exception } e') \implies e \neq \text{default} \implies e' \neq \text{default} \implies \text{reaches } f s (\text{Exception } e) t \implies \text{reaches } f' s' (\text{Exception } e') t' \implies Q (\text{Exception } e) (\text{Exception } e')$
assumes *Ex-Res*: $\bigwedge e v' s s' t t'. S s s' \implies S t t' \implies P (\text{Exception } e) (\text{Result } v') \implies e \neq \text{default} \implies \text{reaches } f s (\text{Exception } e) t \implies \text{reaches } f' s' (\text{Result } v') t' \implies \text{rel-spec-monad } S Q (\text{yield } (\text{Exception } e)) (g' v')$
assumes *Res-Ex*: $\bigwedge v e' s s' t t'. S s s' \implies S t t' \implies P (\text{Result } v) (\text{Exception } e') \implies e' \neq \text{default} \implies$

$reaches\ f\ s\ (Result\ v)\ t \implies reaches\ f'\ s'\ (Exception\ e')\ t' \implies$
 $rel-spec-monad\ S\ Q\ (g\ v)\ (yield\ (Exception\ e'))$
assumes *Res-Res*: $\bigwedge v\ v'\ s\ s'\ t\ t'. S\ s\ s' \implies S\ t\ t' \implies P\ (Result\ v)\ (Result\ v')$
 \implies
 $reaches\ f\ s\ (Result\ v)\ t \implies reaches\ f'\ s'\ (Result\ v')\ t' \implies$
 $rel-spec-monad\ S\ Q\ (g\ v)\ (g'\ v')$
shows $rel-spec-monad\ S\ Q\ (f \ggg g)\ (f' \ggg g')$
apply (*simp add: rel-spec-monad-iff-refines*)
apply (*clarify, intro conjI*)
subgoal for $s\ t$
apply (*rule refines-bind-bind-strong' [where Q=(rel-prod P S)]*)
subgoal using $f-f'$ **by** (*auto simp add: rel-spec-monad-iff-refines*)
subgoal
using $Ex-Ex$ **by** *auto*
subgoal
using $Ex-Res$ **by** (*auto simp add: rel-spec-monad-iff-refines*)
subgoal
using $Res-Ex$ **by** (*auto simp add: rel-spec-monad-iff-refines*)
subgoal
using $Res-Res$ **by** (*auto simp add: rel-spec-monad-iff-refines*)
done
subgoal for $s\ t$
apply (*rule refines-bind-bind-strong' [where Q=(rel-prod P⁻¹⁻¹ S⁻¹⁻¹)]*)
subgoal using $f-f'$ **by** (*auto simp add: rel-spec-monad-iff-refines*)
subgoal
using $Ex-Ex$ **by** *auto*
subgoal
using $Res-Ex$ **by** (*auto simp add: rel-spec-monad-iff-refines*)
subgoal
using $Ex-Res$ **by** (*auto simp add: rel-spec-monad-iff-refines*)
subgoal
using $Res-Res$ **by** (*auto simp add: rel-spec-monad-iff-refines*)
done
done

lemma *rel-spec-monad-bind-strong-exn*:
assumes $f-f'$: $rel-spec-monad\ S\ P\ f\ f'$
assumes $Ex-Ex$: $\bigwedge e\ e'\ s\ s'\ t\ t'. S\ s\ s' \implies S\ t\ t' \implies P\ (Exn\ e)\ (Exn\ e') \implies$
 $reaches\ f\ s\ (Exn\ e)\ t \implies reaches\ f'\ s'\ (Exn\ e')\ t' \implies$
 $Q\ (Exn\ e)\ (Exn\ e')$
assumes $Ex-Res$: $\bigwedge e\ v'\ s\ s'\ t\ t'. S\ s\ s' \implies S\ t\ t' \implies P\ (Exn\ e)\ (Result\ v') \implies$
 $reaches\ f\ s\ (Exn\ e)\ t \implies reaches\ f'\ s'\ (Result\ v')\ t' \implies$
 $rel-spec-monad\ S\ Q\ (throw\ e)\ (g'\ v')$
assumes $Res-Ex$: $\bigwedge v\ e'\ s\ s'\ t\ t'. S\ s\ s' \implies S\ t\ t' \implies P\ (Result\ v)\ (Exn\ e') \implies$
 $reaches\ f\ s\ (Result\ v)\ t \implies reaches\ f'\ s'\ (Exn\ e')\ t' \implies$
 $rel-spec-monad\ S\ Q\ (g\ v)\ (throw\ e')$
assumes $Res-Res$: $\bigwedge v\ v'\ s\ s'\ t\ t'. S\ s\ s' \implies S\ t\ t' \implies P\ (Result\ v)\ (Result\ v')$
 \implies
 $reaches\ f\ s\ (Result\ v)\ t \implies reaches\ f'\ s'\ (Result\ v')\ t' \implies$

```

    rel-spec-monad S Q (g v) (g' v')
  shows rel-spec-monad S Q (f  $\gg$  g) (f'  $\gg$  g')
  apply (rule rel-spec-monad-bind-strong [OF f-f'])
  subgoal using Ex-Ex by (auto simp add: Exn-def default-option-def)
  subgoal using Ex-Res by (auto simp add: Exn-def default-option-def)
  subgoal using Res-Ex by (auto simp add: Exn-def default-option-def)
  subgoal using Res-Res by (auto)
  done

```

```

lemma rel-spec-monad-rel-xval-bind-strong:
  assumes f-f': rel-spec-monad S (rel-xval L P) f f'
  assumes Res-Res:  $\bigwedge v v' s s' t t'. S s s' \implies S t t' \implies P v v' \implies$ 
    reaches f s (Result v) t  $\implies$  reaches f' s' (Result v') t'  $\implies$ 
    rel-spec-monad S (rel-xval L R) (g v) (g' v')
  shows rel-spec-monad S (rel-xval L R) (f  $\gg$  g) (f'  $\gg$  g')
  apply (rule rel-spec-monad-bind-strong-exn [OF f-f'])
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal using Res-Res by auto
  done

```

```

lemma rel-spec-monad-bind-exception-or-result-strong:
  assumes f-f': rel-spec-monad S P f f'
  assumes g-g':  $\bigwedge r r' s s' t t'. S s s' \implies S t t' \implies P r r' \implies$ 
    reaches f s r t  $\implies$  reaches f' s' r' t'  $\implies$ 
    rel-spec-monad S Q (g r) (g' r')
  shows rel-spec-monad S Q (bind-exception-or-result f g) (bind-exception-or-result
    f' g')
  apply (simp add: rel-spec-monad-iff-refines)
  apply (clarify, intro conjI)
  subgoal for s t
    apply (rule refines-bind-exception-or-result-strong [where Q=rel-prod P S])
    subgoal
      using f-f' by (auto simp add: rel-spec-monad-iff-refines)
    subgoal
      using g-g' by (auto simp add: rel-spec-monad-iff-refines)
    done
  subgoal for s t
    apply (rule refines-bind-exception-or-result-strong [where Q=rel-prod P-1-1
    S-1-1])
    subgoal
      using f-f' by (auto simp add: rel-spec-monad-iff-refines)
    subgoal
      using g-g' by (auto simp add: rel-spec-monad-iff-refines)
    done
  done

```


end

```
theory Simp-Trace  
  imports AutoCorres-Utils  
begin
```

ATTENTION: to activate these methods use the following line:

```
setup  $\langle$ Raw-Simplifier.set-trace-ops Simp-Trace.trace-ops $\rangle$ 
```

Provide a tactic wrapper to activate simplifier tracing and produce a statistic how many conditional rules were tried for how long. Also provides a shorthand for simp trace activation by adding *T* to the method name: *simpT simp-allT autoT*

ML \langle

```
structure Simp-Trace =  
struct
```

```
type trace-statistics =  
{  
  conditional-rules : (Time.time * int) Symtab.table,  
  rules : int Symtab.table,  
  steps : int,  
  max-steps : int,  
  depth : int,  
  max-depth : int  
}
```

```
structure Data = Proof-Data  
( type T = trace-statistics Synchronized.var option  
  fun init - = NONE  
)
```

```
val trace-data : Proof.context  $\rightarrow$  trace-statistics Synchronized.var option = Data.get
```

```
fun initial n : trace-statistics =  
{  
  conditional-rules = Symtab.empty,  
  rules = Symtab.empty,  
  steps = 0,  
  max-steps = n,  
  depth = 0,  
  max-depth = 0  
}
```

```
fun activate n = Data.put (SOME (Synchronized.var trace-data (initial n)))
```

```

fun print ({conditional-rules, rules, steps, max-depth, ...} : trace-statistics) =
  let
    val - = writeln (=== SIMP STATISTICS ( ^
      string-of-int steps ^ steps, ^ string-of-int max-depth ^ max depth) ===)
    val - = writeln Conditional Rules:
    val - = conditional-rules |> Symtab.dest
    |> sort (make-ord (fn ((-, (t1, -)), (-, (t2, -))) => Time.< (t2, t1)))
    |> app (fn (name, (t, c)) =>
      writeln ( ^ name ^ : ^ Time.toString t ^ / ^ string-of-int c ^ = ^
        Real.toString (Time.toReal t / Real.fromInt c)))
    val - = writeln Rules:
    val - = rules |> Symtab.dest
    |> sort (make-ord (fn ((-, c1), (-, c2)) => c2 < c1))
    |> app (fn (name, c) => writeln ( ^ name ^ : ^ string-of-int c))
    val - = writeln (=====)
  in
    ()
  end

fun trace-apply var {unconditional, rrule, thm, ...} ctxt cont =
  let
    val d = Synchronized.value var
    val - = #max-steps d <= #steps d andalso error (simp tracer: max steps
reaches)
    fun store time = Synchronized.change var (fn
      { conditional-rules, rules, steps, depth, max-steps, max-depth } =>
      {
        conditional-rules = (if unconditional then conditional-rules else
          conditional-rules |>
            Symtab.map-default (#name rrule, (Time.zeroTime, 0))
              (fn (t, c) => (Time.+ (t, #cpu time), c + 1))),
          rules = rules |> Symtab.map-default (#name rrule, 0) (fn c => c + 1),
          steps = steps + 1,
          max-steps = max-steps,
          depth = depth,
          max-depth = max-depth
        }
      )
  in
    Timing.timing (Exn.capture cont) ctxt
    |> (fn (t, x) => (store t; Exn.release x))
  end

fun map-depth f
  ({ conditional-rules, rules, steps, depth, max-steps, max-depth } : trace-statistics)
:
  trace-statistics =
  let
    val d = f depth
  end

```

```

in
  { conditional-rules = conditional-rules,
    rules = rules,
    steps = steps,
    depth = d,
    max-depth = if max-depth < d then d else max-depth,
    max-steps = max-steps }
end

fun increase-depth var ctxt =
  (Synchronized.change var (map-depth (fn d => d + 1))); ctxt

val trace-ops : Raw-Simplifier.trace-ops = {
  trace-invoke = fn - => fn ctxt =>
    (case trace-data ctxt of SOME var => increase-depth var ctxt
     | NONE => ctxt),
  trace-rrule = fn rule => fn ctxt => fn cont =>
    case trace-data ctxt of
      SOME var => trace-apply var rule ctxt cont
    | NONE => cont ctxt,
  trace-simproc = K (K (K NONE))
}

fun wrapper n tac ctxt st =
  let
    val ctxt' = ctxt
      |> activate n
      |> Config.put Raw-Simplifier.simp-trace true
      |> Config.put Raw-Simplifier.simp-trace-depth-limit 4
    val res = Exn.capture (fn () => (tac ctxt' st |> Seq.pull)) ()
    val - = print (Synchronized.value (the (trace-data ctxt')))
  in
    case Exn.release res of
      SOME (x, -) => Seq.cons x (Seq.make (fn () =>
        error (stat simp tracer does not allow back ^ Position.here here)))
    | NONE => Seq.empty
  end

fun wrapper' n tac ctxt = wrapper n (fn ctxt => tac ctxt 1) ctxt

fun wrapper-method method ctxt' (-, thm) =
  method (ctxt', thm)

fun method-wrapper (n : int option) (m : Proof.context -> Method.method) ctxt
: Method.method =
  fn thms => Method.RUNTIME
    (wrapper (the-default 1000 n) (fn ctxt => wrapper-method (m ctxt thms) ctxt)
  ctxt)

```

```

val no-asmN = no-asm;
val no-asm-useN = no-asm-use;
val no-asm-simpN = no-asm-simp;
val asm-lrN = asm-lr;

val simp-options =
  (Args.parens (Scan.option Parse.nat -- Args.$$$ no-asmN) >> apsnd (K simp-tac)
  ||
  Args.parens (Scan.option Parse.nat -- Args.$$$ no-asm-simpN) >> apsnd (K
  asm-simp-tac) ||
  Args.parens (Scan.option Parse.nat -- Args.$$$ no-asm-useN) >> apsnd (K
  full-simp-tac) ||
  Args.parens (Scan.option Parse.nat -- Args.$$$ asm-lrN) >> apsnd (K asm-lr-simp-tac)
  ||
  Args.parens Parse.nat >> (fn n => (SOME n, asm-full-simp-tac)) ||
  Scan.succeed (NONE, asm-full-simp-tac));

val wrapper : (Proof.context -> Method.method) context-parser =
  (Scan.lift (Scan.option (Args.parens Parse.nat)) -- Method.text-closure)
  >> (fn (n, text) => method-wrapper n (fn ctxt => Method.evaluate-runtime text
  ctxt))

fun simpT-method meth =
  Scan.lift simp-options --|
  Method.sections Simplifier.simp-modifiers' >>
  (fn (n, tac) => method-wrapper n (fn ctxt => METHOD (fn facts => meth
  ctxt tac facts)))

val simp-wrapper =
  simpT-method (fn ctxt => fn tac => fn facts =>
  HEADGOAL (Method.insert-tac ctxt facts THEN'
  (CHANGED-PROP oo tac) ctxt))

val simp-all-wrapper =
  simpT-method (fn ctxt => fn tac => fn facts =>
  ALLGOALS (Method.insert-tac ctxt facts) THEN
  (CHANGED-PROP o PARALLEL-ALLGOALS o tac) ctxt)

val auto-wrapper =
  (Scan.lift (Scan.option (Args.parens Parse.nat)) --
  Scan.lift (Scan.option (Parse.nat -- Parse.nat))) --|
  Method.sections clasimp-modifiers >>
  (fn (i, NONE) =>
  method-wrapper i (SIMPLE-METHOD o CHANGED-PROP o auto-tac)
  | (i, SOME (m, n)) =>
  method-wrapper i (fn ctxt => SIMPLE-METHOD (CHANGED-PROP (mk-auto-tac
  ctxt m n))));

end

```

>

setup *<Method.setup binding <stat> Simp-Trace.wrapper
(Simplifier statistics: wraps a method call and computes statistics from simplifier
calls. ^
Needs active Simp-Trace.trace-ops)>*

setup *<Method.setup binding <simpT> Simp-Trace.simp-wrapper
(simplification with tracing ^
Needs active Simp-Trace.trace-ops)>*

setup *<Method.setup binding <simp-allT> Simp-Trace.simp-all-wrapper
(simplification on all goals with tracing ^
Needs active Simp-Trace.trace-ops)>*

setup *<Method.setup binding <autoT> Simp-Trace.auto-wrapper
(auto with tracing ^
Needs active Simp-Trace.trace-ops)>*

end

Chapter 17

Basic Stuff

theory *AutoCorres-Base*

imports

TypHeapLib

LemmaBucket-C

CTranslation

Synthesize

Cong-Tactic

Reaches

Mutual-CCPO-Recursion

Simp-Trace

begin

definition *THIN* :: *prop* \Rightarrow *prop* **where** *PROP THIN* (*PROP P*) \equiv *PROP P*

lemma *THIN-I*: *PROP P* \Longrightarrow *PROP THIN* (*PROP P*)

by (*simp add: THIN-def*)

ML-file \langle *utils.ML* \rangle

named-theorems *corres-admissible* **and** *corres-top*

named-theorems *funp-intros* **and** *fun-of-rel-intros*

definition *fun-of-rel*:: (*'a* \Rightarrow *'b* \Rightarrow *bool*) \Rightarrow (*'a* \Rightarrow *'b*) \Rightarrow *bool* **where**
fun-of-rel *r f* = ($\forall x y. r\ x\ y \longrightarrow y = f\ x$)

definition *funp*:: (*'a* \Rightarrow *'b* \Rightarrow *bool*) \Rightarrow *bool* **where**
funp *r* = ($\exists f. fun-of-rel\ r\ f$)

lemma *funp-witness*: *fun-of-rel* *r f* \Longrightarrow *funp* *r*

by (*auto simp add: funp-def*)

lemma *funp-to-single-valuedp*: *funp* *r* \Longrightarrow *single-valuedp* *r*

by (auto simp add: single-valuedp-def funp-def fun-of-rel-def)

lemma fun-of-rel-xval[fun-of-rel-intros]:
 fun-of-rel L f-l \implies fun-of-rel R f-r \implies fun-of-rel (rel-xval L R) (map-xval f-l f-r)
 by (auto simp add: fun-of-rel-def rel-xval.simps)

lemma funp-rel-xval[funp-intros, corres-admissible]:
 assumes L: funp L
 assumes R: funp R
 shows funp (rel-xval L R)
proof –
 from L obtain f-l where f-l: $\bigwedge x y. L x y \longrightarrow y = f-l x$ by (auto simp add: funp-def fun-of-rel-def)
 from R obtain f-r where f-r: $\bigwedge x y. R x y \longrightarrow y = f-r x$ by (auto simp add: funp-def fun-of-rel-def)
 define f where f = map-xval f-l f-r
 {
 fix x y
 have rel-xval L R x y $\longrightarrow y = f x$
 using f-l f-r
 by (auto simp add: rel-xval.simps f-def)
 }
 then show ?thesis
 by (auto simp add: funp-def fun-of-rel-def)
qed

lemma fun-of-rel-eq[fun-of-rel-intros]: fun-of-rel (=) ($\lambda x. x$)
 by (auto simp add: fun-of-rel-def)

lemma fun-of-rel-bot[fun-of-rel-intros]: fun-of-rel ($\lambda - . False$) f
 by (auto simp add: fun-of-rel-def)

lemma funp-eq[funp-intros, corres-admissible]: funp (=)
 by (auto simp add: funp-def fun-of-rel-def)

lemma admissible-mem: ccpo.admissible Inf (\geq) ($\lambda A. x \in A$)
 by (auto simp: ccpo.admissible-def)

lemma funp-rel-prod[funp-intros, corres-admissible]:
 assumes L: funp L
 assumes R: funp R
 shows funp (rel-prod L R)
proof –
 from L obtain f-l where f-l: $\bigwedge x y. L x y \longrightarrow y = f-l x$ by (auto simp add: funp-def fun-of-rel-def)
 from R obtain f-r where f-r: $\bigwedge x y. R x y \longrightarrow y = f-r x$ by (auto simp add: funp-def fun-of-rel-def)
 define f where f = map-prod f-l f-r
 {

```

    fix x y
    have rel-prod L R x y  $\longrightarrow$  y = f x
      using f-l f-r
      by (auto simp add: rel-prod.simps f-def map-prod-def split: prod.splits)
  }
  then show ?thesis
    by (auto simp add: funp-def fun-of-rel-def)
qed

lemma funp-rel-sum[funp-intros, corres-admissible]:
  assumes L: funp L
  assumes R: funp R
  shows funp (rel-sum L R)
proof -
  from L obtain f-l where f-l:  $\bigwedge x y. L x y \longrightarrow y = f-l x$  by (auto simp add:
  funp-def fun-of-rel-def)
  from R obtain f-r where f-r:  $\bigwedge x y. R x y \longrightarrow y = f-r x$  by (auto simp add:
  funp-def fun-of-rel-def)
  define f where f = sum-map f-l f-r
  {
    fix x y
    have rel-sum L R x y  $\longrightarrow$  y = f x
      using f-l f-r
      by (auto simp add: rel-sum.simps f-def)
  }
  then show ?thesis
    by (auto simp add: funp-def fun-of-rel-def)
qed

lemma fun-of-rel-bottom: fun-of-rel (( $\lambda - . False$ )) f
  by (auto simp add: fun-of-rel-def)

lemma funp-bottom [funp-intros, corres-admissible]: funp ( $\lambda - . False$ )
  using fun-of-rel-bottom
  by (metis funp-def)

lemma fun-of-rel-prod[fun-of-rel-intros]:
  fun-of-rel L f-l  $\implies$  fun-of-rel R f-r  $\implies$  fun-of-rel (rel-prod L R) (map-prod f-l
  f-r)
  by (auto simp add: fun-of-rel-def)

lemma fun-of-rel-sum[fun-of-rel-intros]:
  fun-of-rel L f-l  $\implies$  fun-of-rel R f-r  $\implies$  fun-of-rel (rel-sum L R) (sum-map f-l
  f-r)
  by (auto simp add: fun-of-rel-def rel-sum.simps)

lemma fun-of-relcompp[fun-of-rel-intros]: fun-of-rel F f  $\implies$  fun-of-rel G g  $\implies$ 
  fun-of-rel (F OO G) (g o f)
  by (auto simp add: fun-of-rel-def)

```


lemma *funp-relcompp*[*funp-intros, corres-admissible*]: *funp F* \implies *funp G* \implies *funp (F OO G)*
using *fun-of-relcompp*
by (*force simp add: funp-def*)

lemma *fun-of-rel-rel-map*[*fun-of-rel-intros*]: *fun-of-rel (rel-map f) f*
by (*auto simp add: fun-of-rel-def rel-map-def*)

lemma *funp-rel-map*[*funp-intros, corres-admissible*]: *funp (rel-map f)*
using *fun-of-rel-rel-map*
by (*auto simp add: funp-def*)

lemma *ccpo-prod-gfp-gfp*:
class.ccpo
(*prod-lub Inf Inf* :: (*'a*::*complete-lattice* * *'b*::*complete-lattice*) *set* \implies -))
(*rel-prod* (\geq) (\geq)) (*mk-less* (*rel-prod* (\geq) (\geq)))
by (*rule ccpo-rel-prodI ccpo-Inf*)**+**

lemma *runs-to-partial-top*[*corres-top*]: $\top \cdot s \ ?\{ Q \}$
by (*simp add: runs-to-partial-def-old*)

lemma *refines-top*[*corres-top*]: *refines C* \top *s t R*
by (*auto simp add: refines-def-old*)

lemma *pred-andE*[*elim!*]: $\llbracket (A \text{ and } B) x; \llbracket A x; B x \rrbracket \implies R \rrbracket \implies R$
by(*simp add: pred-conj-def*)

lemma *pred-andI*[*intro!*]: $\llbracket A x; B x \rrbracket \implies (A \text{ and } B) x$
by(*simp add: pred-conj-def*)

definition *measure'* :: (*'a* \implies *'b*::*wellorder*) \implies (*'a* \times *'a*) *set*
where *measure'* = ($\lambda f. \{(a, b). f a < f b\}$)

lemma *in-measure'*[*simp, code-unfold*]:
 $((x, y) : \text{measure}' f) = (f x < f y)$
by (*simp add: measure'-def*)

lemma *wf-measure'* [*iff*]: *wf (measure' f)*
apply (*clarsimp simp: measure'-def*)
apply (*insert wf-inv-image [OF wellorder-class.wf, where f=f]*)
apply (*clarsimp simp: inv-image-def*)
done

lemma *wf-wellorder-measure*: *wf* $\{(a, b). (M a :: 'a :: \text{wellorder}) < M b\}$
apply (*subgoal-tac wf (inv-image (\{(a, b). a < b\}) M)*)
apply (*clarsimp simp: inv-image-def*)

```

apply (rule wf-inv-image)
apply (rule wellorder-class.wf)
done

lemma wf-custom-measure:
   $\llbracket \bigwedge a b. (a, b) \in R \implies f a < (f :: 'a \Rightarrow \text{nat}) b \rrbracket \implies \text{wf } R$ 
  by (metis in-measure wf-def wf-measure)

lemma rel-fun-conversep':  $\text{rel-fun } R \ Q \ f \ g \longleftrightarrow \text{rel-fun } R^{-1-1} \ Q^{-1-1} \ g \ f$ 
  unfolding rel-fun-def by auto

end

theory SimplBucket
  imports AutoCorres-Base
begin

lemma Normal-resultE:
   $\llbracket \Gamma \vdash \langle c, s \rangle \Rightarrow \text{Normal } t'; \bigwedge t. \llbracket \Gamma \vdash \langle c, \text{Normal } t \rangle \Rightarrow \text{Normal } t'; s = \text{Normal } t \rrbracket$ 
 $\implies P \rrbracket \implies P$ 
  by (metis noAbrupt-startD noFault-startD noStuck-startD xstate.exhaust)

lemma exec-While-final-cond':
   $\llbracket \Gamma \vdash \langle b, s \rangle \Rightarrow s'; b = \text{While } C \ B; s = \text{Normal } v; s' = \text{Normal } x \rrbracket \implies x \notin C$ 
  apply (induct arbitrary: v x rule: exec.induct)
  apply simp-all
  apply (atomize, clarsimp)
  apply (erule exec-elim-cases, auto)
  done

lemma exec-While-final-cond:
   $\llbracket \Gamma \vdash \langle \text{While } C \ B, s \rangle \Rightarrow \text{Normal } s' \rrbracket \implies s' \notin C$ 
  apply (erule Normal-resultE)
  apply (erule exec-While-final-cond', auto)
  done

lemma exec-While-final-inv':
   $\llbracket \Gamma \vdash \langle b, s \rangle \Rightarrow s'; b = \text{While } C \ B; s = \text{Normal } v; s' = \text{Normal } x; I v; \bigwedge s s'. \llbracket I$ 
 $s; \Gamma \vdash \langle B, \text{Normal } s \rangle \Rightarrow \text{Normal } s' \rrbracket \implies I s' \rrbracket \implies I x$ 
  apply atomize
  apply (induct arbitrary: v x rule: exec.induct)

  apply (clarify | (atomize, erule Normal-resultE, metis))+
  done

```

lemma *exec-While-final-inv*:

$\llbracket \Gamma \vdash \langle \text{While } C \ B, \text{Normal } s \rangle \Rightarrow \text{Normal } s'; I \ s; \bigwedge s \ s'. \llbracket I \ s; \Gamma \vdash \langle B, \text{Normal } s \rangle \Rightarrow \text{Normal } s' \rrbracket \Longrightarrow I \ s' \rrbracket \Longrightarrow I \ s'$

apply (*erule exec-While-final-inv'*, *auto*)

done

primrec

exceptions-thrown :: ('a, 'p, 'e) com \Rightarrow bool

where

exceptions-thrown Skip = False

| *exceptions-thrown* (Seq a b) = (*exceptions-thrown* a \vee *exceptions-thrown* b)

| *exceptions-thrown* (Basic a) = False

| *exceptions-thrown* (Language.Spec a) = False

| *exceptions-thrown* (Cond a b c) = (*exceptions-thrown* b \vee *exceptions-thrown* c)

| *exceptions-thrown* (Catch a b) = (*exceptions-thrown* a \wedge *exceptions-thrown* b)

| *exceptions-thrown* (While a b) = *exceptions-thrown* b

| *exceptions-thrown* Throw = True

| *exceptions-thrown* (Call p) = True

| *exceptions-thrown* (Guard f g a) = *exceptions-thrown* a

| *exceptions-thrown* (DynCom a) = ($\exists s. \text{exceptions-thrown } (a \ s)$)

primrec

exceptions-unresolved :: ('a, 'p, 'e) com list \Rightarrow bool

where

exceptions-unresolved [] = True

| *exceptions-unresolved* (x#xs) = (*exceptions-thrown* x \wedge *exceptions-unresolved* xs)

lemma *exceptions-thrown-not-abrupt*:

$\llbracket \Gamma \vdash \langle p, s \rangle \Rightarrow s'; \neg \text{exceptions-thrown } p; \neg \text{isAbr } s \rrbracket \Longrightarrow \neg \text{isAbr } s'$

apply (*induct rule: exec.induct*, *auto*)

done

end

theory CCorresE

imports

SimplBucket

begin

definition

$$\begin{aligned}
\text{ccorresE} &:: ('t \Rightarrow 's) \Rightarrow \text{bool} \Rightarrow 'ee \text{ set} \Rightarrow ('p \Rightarrow ('t, 'p, 'ee) \text{ com option}) \\
&\Rightarrow ('s \Rightarrow \text{bool}) \Rightarrow ('t \text{ set}) \\
&\Rightarrow (\text{unit}, \text{unit}, 's) \text{ exn-monad} \Rightarrow ('t, 'p, 'ee) \text{ com} \Rightarrow \text{bool}
\end{aligned}$$
where

$$\begin{aligned}
\text{ccorresE st check-term AF } \Gamma \ G \ G' &\equiv \\
\lambda m \ c. \forall s. G \ (st \ s) \wedge (s \in G') \wedge \text{succeeds } m \ (st \ s) &\longrightarrow \\
((\forall t. \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow t \longrightarrow & \\
(\text{case } t \text{ of} & \\
\quad \text{Normal } s' \Rightarrow \text{reaches } m \ (st \ s) \ (\text{Result } ()) \ (st \ s') & \\
\quad | \text{Abrupt } s' \Rightarrow \text{reaches } m \ (st \ s) \ (\text{Exn } ()) \ (st \ s') & \\
\quad | \text{Fault } e \Rightarrow e \in AF & \\
\quad | - \Rightarrow \text{False})) & \\
\wedge (\text{check-term} \longrightarrow \Gamma \vdash c \downarrow \text{Normal } s)) &
\end{aligned}$$
lemma ccorresE-cong:

$$\begin{aligned}
&[[\bigwedge s. P \ s = P' \ s; \\
&\quad \bigwedge s. (s \in Q) = (s \in Q'); \\
&\quad \bigwedge s. P' \ s \Longrightarrow \text{run } f \ s = \text{run } f' \ s; \\
&\quad \bigwedge s \ x. s \in Q' \Longrightarrow \Gamma \vdash \langle g, \text{Normal } s \rangle \Rightarrow x = \Gamma \vdash \langle g', \text{Normal } s \rangle \Rightarrow x \\
&]] \Longrightarrow
\end{aligned}$$

$$\text{ccorresE st ct AF } \Gamma \ P \ Q \ f \ g = \text{ccorresE st ct AF } \Gamma \ P' \ Q' \ f' \ g$$
apply *atomize***apply** (*clarsimp simp: ccorresE-def split: xstate.splits*)**by** (*auto simp add: succeeds-def reaches-def*)**lemma ccorresE-guard-imp:**

$$[[\text{ccorresE st ct AF } \Gamma \ Q \ Q' \ A \ B; \bigwedge s. P \ s \Longrightarrow Q \ s; \bigwedge t. t \in P' \Longrightarrow t \in Q']] \Longrightarrow \text{ccorresE st ct AF } \Gamma \ P \ P' \ A \ B$$
apply *atomize***apply** (*clarsimp simp: ccorresE-def split: xstate.splits*)**done****lemma ccorresE-guard-imp-stronger:**

$$\begin{aligned}
&[[\text{ccorresE st ct AF } \Gamma \ Q \ Q' \ A \ B; \\
&\quad \bigwedge s. [[P \ (st \ s); s \in P']] \Longrightarrow Q \ (st \ s); \\
&\quad \bigwedge s. [[P \ (st \ s); s \in P']] \Longrightarrow s \in Q']] \Longrightarrow
\end{aligned}$$

$$\text{ccorresE st ct AF } \Gamma \ P \ P' \ A \ B$$
apply *atomize***apply** (*clarsimp simp: ccorresE-def split-def split: xstate.splits*)**done****lemma ccorresE-assume-pre:**

$$\begin{aligned}
&[[\bigwedge s. [[G \ (st \ s); s \in G']] \Longrightarrow \\
&\quad \text{ccorresE st ct AF } \Gamma \ (G \ \text{and } (\lambda s'. s' = st \ s)) \ (G' \cap \{t'. t' = s\}) \ A \ B]] \Longrightarrow \\
&\quad \text{ccorresE st ct AF } \Gamma \ G \ G' \ A \ B
\end{aligned}$$
apply *atomize***apply** (*simp add: ccorresE-def pred-conj-def*)

done

lemma *ccorresE-Seq*:

```
[[ ccorresE st ct AF  $\Gamma \top$  UNIV L L';
   ccorresE st ct AF  $\Gamma \top$  UNIV R R' ]]  $\implies$ 
ccorresE st ct AF  $\Gamma \top$  UNIV (do {-  $\leftarrow$  L; R }) (L' ;; R')
apply (clarsimp simp: ccorresE-def)
apply (rule conjI)
apply clarsimp
apply (erule exec-Normal-elim-cases)
subgoal
apply (clarsimp split: xstate.splits
        simp add: succeeds-bind reaches-bind, intro conjI)
by (metis (mono-tags, lifting) Normal-resultE case-exception-or-result-Result)
    (metis (mono-tags, lifting) case-exception-or-result-Exn
        case-exception-or-result-Result exec-Normal-elim-cases(1)
        exec-Normal-elim-cases(3) xstate.exhaust)+
subgoal
apply (clarsimp split: xstate.splits
        simp add: succeeds-bind reaches-bind)
by (metis Abrupt Fault terminates.Seq xstate.exhaust)
done
```

lemma *ccorresE-Cond*:

```
[[ ccorresE st ct AF  $\Gamma \top$  C A L';
   ccorresE st ct AF  $\Gamma \top$  (UNIV - C) A R' ]]  $\implies$ 
ccorresE st ct AF  $\Gamma \top$  UNIV A (Cond C L' R')
apply (clarsimp simp: ccorresE-def pred-neg-def)
subgoal for s
apply (rule conjI)
apply clarsimp
apply (erule exec-Normal-elim-cases)
apply (erule-tac x=s in allE, erule impE, fastforce, fastforce)
apply (erule-tac x=s in allE, erule impE, fastforce, fastforce)
apply clarsimp
apply (cases s  $\in$  C)
apply (rule terminates.CondTrue, assumption)
apply (erule allE, erule impE, fastforce)
apply clarsimp
apply (rule terminates.CondFalse, assumption)
apply (erule allE, erule impE, fastforce)
apply clarsimp
done
done
```

lemma *ccorresE-Cond-match*:

```
[[ ccorresE st ct AF  $\Gamma$  C C' L L';
   ccorresE st ct AF  $\Gamma$  (not C) (UNIV - C') R R';
    $\bigwedge s. C (st s) = (s \in C')$  ]]  $\implies$ 
```

```

    ccorresE st ct AF  $\Gamma \top$  UNIV (condition C L R) (Cond C' L' R')
  apply atomize
  apply (simp add: ccorresE-def pred-neg-def)
  apply clarify
  apply (intro conjI allI impI)

```

```

subgoal for s t
  apply (auto elim!: exec-Normal-elim-cases split: xstate.splits)
  done
subgoal for s
  apply (cases s  $\in$  C')
  subgoal
    by (simp add: terminates.CondTrue)
  subgoal
    by (simp add: terminates.CondFalse)
  done
done

```

lemma *ccorresE-Guard*:

```

[[ ccorresE st ct AF  $\Gamma \top$  G X Y ]]  $\implies$  ccorresE st ct AF  $\Gamma \top$  G X (Guard F G Y)
  apply (clarsimp simp: ccorresE-def)
  apply (rule conjI)
  apply clarsimp
  apply (erule exec-Normal-elim-cases, auto)[1]
  apply clarsimp
  apply (rule terminates.Guard, assumption)
  apply force
  done

```

lemma *ccorresE-Catch*:

```

[[ ccorresE st ct AF  $\Gamma \top$  UNIV A A'; ccorresE st ct AF  $\Gamma \top$  UNIV B B' ]]  $\implies$ 
  ccorresE st ct AF  $\Gamma \top$  UNIV (A <catch> ( $\lambda$ -. B)) (TRY A' CATCH B' END)
  apply (clarsimp simp: ccorresE-def)
  apply (rule conjI)
  apply clarsimp
  apply (erule-tac x=s in allE)
  apply (erule exec-Normal-elim-cases)
  subgoal for s t s'
    apply (cases t)
    subgoal
      using reaches-catch
      by (metis case-xval-simps(1) succeeds-catch xstate.simps(16) xstate.simps(17))
    subgoal
      using reaches-catch
      by (metis case-xval-simps(1) succeeds-catch xstate.simps(17))
    subgoal
      using reaches-catch
      by (metis case-xval-simps(1) succeeds-catch xstate.simps(17) xstate.simps(18))

```

```

subgoal
  using reaches-catch
  by (metis case-xval-simps(1) succeeds-catch xstate.simps(17) xstate.simps(19))
done
subgoal for s t

  apply (cases t)
  apply (fastforce simp add: reaches-catch succeeds-catch)+
done
subgoal for s
  by (metis case-xval-simps(1) succeeds-catch terminates.Catch xstate.simps(17))
done

```

lemma *ccorresE-Call*:

```

 $\llbracket \Gamma X' = \text{Some } Z'; \text{ccorresE } st \text{ ct } AF \Gamma \top UNIV Z Z' \rrbracket \implies$ 
 $\text{ccorresE } st \text{ ct } AF \Gamma \top UNIV Z (\text{Call } X')$ 
apply (clarsimp simp: ccorresE-def)
apply (rule conjI)
apply clarsimp
apply (erule exec-Normal-elim-cases)
apply (clarsimp)
apply clarsimp
apply clarify
apply (erule terminates.Call)
apply (erule allE, erule (1) impE)
apply clarsimp
done

```

lemma *ccorresE-exec-Normal*:

```

 $\llbracket \text{ccorresE } st \text{ ct } AF \Gamma G G' B B'; \Gamma \vdash \langle B', \text{Normal } s \rangle \Rightarrow \text{Normal } t; s \in G'; G$ 
 $(st \ s); \text{succeeds } B (st \ s) \rrbracket$ 
 $\implies \text{reaches } B (st \ s) (\text{Result } ()) (st \ t)$ 
apply (clarsimp simp: ccorresE-def)
apply force
done

```

lemma *ccorresE-exec-Abrupt*:

```

 $\llbracket \text{ccorresE } st \text{ ct } AF \Gamma G G' B B'; \Gamma \vdash \langle B', \text{Normal } s \rangle \Rightarrow \text{Abrupt } t; s \in G'; G$ 
 $(st \ s); \text{succeeds } B (st \ s) \rrbracket$ 
 $\implies \text{reaches } B (st \ s) (\text{Exn } ()) (st \ t)$ 
apply (clarsimp simp: ccorresE-def)
apply force
done

```

lemma *ccorresE-exec-Fault*:

```

 $\llbracket \text{ccorresE } st \text{ ct } AF \Gamma G G' B B'; \Gamma \vdash \langle B', \text{Normal } s \rangle \Rightarrow \text{Fault } f; f \notin AF; s \in$ 
 $G'; G (st \ s); \text{succeeds } B (st \ s) \rrbracket \implies P$ 
apply (clarsimp simp: ccorresE-def)

```

apply *force*
done

lemma *ccorresE-exec-Stuck*:

$\llbracket \text{ccorresE } st \text{ ct } AF \ \Gamma \ G \ G' \ B \ B'; \Gamma \vdash \langle B', \text{Normal } s \rangle \Rightarrow \text{Stuck}; s \in G'; G \ (st \ s); \text{succeeds } B \ (st \ s) \rrbracket \Longrightarrow P$

apply (*clarsimp simp: ccorresE-def*)
apply *force*
done

lemma *ccorresE-exec-cases* [*consumes 5*]:

$\llbracket \text{ccorresE } st \text{ ct } AF \ \Gamma \ G \ G' \ B \ B'; \Gamma \vdash \langle B', \text{Normal } s \rangle \Rightarrow s'; s \in G'; G \ (st \ s); \text{succeeds } B \ (st \ s);$

$\bigwedge t'. \llbracket s' = \text{Normal } t'; \text{reaches } B \ (st \ s) \ (\text{Result } ()) \ (st \ t') \rrbracket \Longrightarrow R;$
 $\bigwedge t'. \llbracket s' = \text{Abrupt } t'; \text{reaches } B \ (st \ s) \ (\text{Exn } ()) \ (st \ t') \rrbracket \Longrightarrow R;$
 $\bigwedge f. \llbracket s' = \text{Fault } f; f \in AF \rrbracket \Longrightarrow R$
 $\rrbracket \Longrightarrow R$

apply *atomize*
apply (*cases s'*)
apply (*drule ccorresE-exec-Normal, auto*)[1]
apply (*drule ccorresE-exec-Abrupt, auto*)[1]
subgoal for *f*
apply (*cases f ∈ AF*)
subgoal
by *auto*
subgoal
by (*drule ccorresE-exec-Fault, auto*)[1]
done
apply (*drule ccorresE-exec-Stuck, auto*)[1]
done

lemma *ccorresE-terminates*:

$\llbracket \text{ccorresE } st \text{ ct } AF \ \Gamma \ \top \ UNIV \ B \ B'; \text{succeeds } B \ (st \ s); ct \rrbracket \Longrightarrow \Gamma \vdash B' \downarrow \text{Normal } s$

by (*clarsimp simp: ccorresE-def*)

lemma *exec-While-final-inv'*:

assumes *exec*: $\Gamma \vdash \langle b, x \rangle \Rightarrow s'$

shows

$\llbracket b = \text{While } C \ B; x = \text{Normal } s; \bigwedge s. \llbracket s \notin C \rrbracket \Longrightarrow I \ s \ (\text{Normal } s);$
 $\bigwedge t \ t'. \llbracket t \in C; \Gamma \vdash \langle B, \text{Normal } t \rangle \Rightarrow \text{Normal } t'; I \ t' \ s' \rrbracket \Longrightarrow I \ t \ s';$
 $\bigwedge t \ t'. \llbracket t \in C; \Gamma \vdash \langle B, \text{Normal } t \rangle \Rightarrow \text{Abrupt } t' \rrbracket \Longrightarrow I \ t \ (\text{Abrupt } t');$
 $\bigwedge t. \llbracket t \in C; \Gamma \vdash \langle B, \text{Normal } t \rangle \Rightarrow \text{Stuck} \rrbracket \Longrightarrow I \ t \ \text{Stuck};$
 $\bigwedge t \ f. \llbracket t \in C; \Gamma \vdash \langle B, \text{Normal } t \rangle \Rightarrow \text{Fault } f \rrbracket \Longrightarrow I \ t \ (\text{Fault } f) \rrbracket$
 $\Longrightarrow I \ s \ s'$

using *exec*

apply (*induct arbitrary: s rule: exec.induct, simp-all*)

apply *clarsimp*

apply *atomize*
apply *clarsimp*
apply (*erule allE, erule (1) impE*)
apply (*erule exec-elim-cases, auto*)
done

lemma *exec-While-final-inv*:

$\llbracket \Gamma \vdash \langle \text{While } C \ B, \text{Normal } s \rangle \Rightarrow s' \rrbracket$;
 $\bigwedge s. \llbracket s \notin C \rrbracket \Longrightarrow I \ s \ (\text{Normal } s)$;
 $\bigwedge t \ t'. \llbracket t \in C; \Gamma \vdash \langle B, \text{Normal } t \rangle \Rightarrow \text{Normal } t'; I \ t' \ s' \rrbracket \Longrightarrow I \ t \ s'$;
 $\bigwedge t \ t'. \llbracket t \in C; \Gamma \vdash \langle B, \text{Normal } t \rangle \Rightarrow \text{Abrupt } t' \rrbracket \Longrightarrow I \ t \ (\text{Abrupt } t')$;
 $\bigwedge t. \llbracket t \in C; \Gamma \vdash \langle B, \text{Normal } t \rangle \Rightarrow \text{Stuck} \rrbracket \Longrightarrow I \ t \ \text{Stuck}$;
 $\bigwedge t \ f. \llbracket t \in C; \Gamma \vdash \langle B, \text{Normal } t \rangle \Rightarrow \text{Fault } f \rrbracket \Longrightarrow I \ t \ (\text{Fault } f) \rrbracket$
 $\Longrightarrow I \ s \ s'$
apply (*erule exec-While-final-inv', (rule refl)+, simp-all*)
done

lemma *ccorresE-termination'*:

assumes *no-fail: succeeds (whileLoop CC BB r) s*
and *s-match: s = st s' \wedge CC = (λ -. C) \wedge BB = (λ -. B)*
and *corres: ccorresE st ct AF Γ \top UNIV B B'*
and *cond-match: $\bigwedge s. C \ (st \ s) = (s \in C')$*
and *ct: ct*
shows $\Gamma \vdash \text{While } C' \ B' \downarrow \text{Normal } s'$
proof –
from *no-fail have run (whileLoop CC BB r) s \neq \top*
by (*simp add: succeeds-def*)
from this show ?thesis
using *s-match*
proof (*induct arbitrary: s' rule: whileLoop-ne-top-induct*)
case (*step a s*)
then show ?case using corres cond-match ct
apply (*cases s' \notin C'*)
apply (*simp add: terminates.WhileFalse*)
apply (*rule terminates.WhileTrue*)
apply *simp*
subgoal
by (*simp add: runs-to-def-old ccorresE-terminates*)
subgoal
apply (*clarsimp simp: runs-to-def-old*)
by (*metis Abrupt Fault ccorresE-exec-Normal ccorresE-exec-Stuck iso-tuple-UNIV-I top1I xstate.exhaust*)
done
qed
qed

lemma *ccorresE-termination*:

assumes *no-fail: succeeds (whileLoop (λ -. C) (λ -. B) r) s*
and *s-match: s = st s'*

and *corres*: *ccorresE st ct AF $\Gamma \top UNIV B B'$*
and *cond-match*: $\bigwedge s. C (st s) = (s \in C')$
and *ct*: *ct*
shows $\Gamma \vdash \text{While } C' B' \downarrow \text{Normal } s'$
apply (*auto intro*: *ccorresE-termination'* [*OF no-fail - corres - ct*] *simp*: *s-match cond-match*)
done

lemma *ccorresE-While*:

assumes *body-refines*: *ccorresE st ct AF $\Gamma \top UNIV B B'$*
and *cond-match*: $\bigwedge s. C (st s) = (s \in C')$
shows *ccorresE st ct AF $\Gamma G G'$* (*whileLoop* ($\lambda-. C$) ($\lambda-. B$) ()) (*While* *C' B'*)

proof –

{

fix *s t*

assume *guard-abs*: *G (st s)*

assume *guard-conc*: *s $\in G'$*

assume *no-fail*: *succeeds* (*whileLoop* ($\lambda-. C$) ($\lambda-. B$) ()) (*st s*)

assume *conc-steps*: $\Gamma \vdash \langle \text{While } C' B', \text{Normal } s \rangle \Rightarrow t$

have *case t of*

Normal s' \Rightarrow reaches (*whileLoop* ($\lambda-. C$) ($\lambda-. B$) ()) (*st s*) (*Result* ()) (*st s'*)

| *Abrupt s' \Rightarrow reaches* (*whileLoop* ($\lambda-. C$) ($\lambda-. B$) ()) (*st s*) (*Exn* ()) (*st s'*)

| *Fault e $\Rightarrow e \in AF$*

| *- \Rightarrow False*

apply (*insert no-fail, erule rev-mp*)

apply (*rule exec-While-final-inv* [*OF conc-steps*])

subgoal

using *cond-match*

apply *clarsimp*

apply (*subst whileLoop-unroll*)

apply (*simp add: reaches-condition-iff*)

done

subgoal for *t1 t'*

apply (*subst (1 2 3) whileLoop-unroll*)

apply (*clarsimp simp add: cond-match*)

using *ccorresE-exec-Normal* [*OF body-refines, of t1 t'*]

using *cond-match*

apply (*force split: xstate.splits simp add: succeeds-bind reaches-bind*)

done

subgoal for *t1 t'*

apply (*subst (1 2 3) whileLoop-unroll*)

apply (*clarsimp simp add: cond-match*)

using *ccorresE-exec-Abrupt* [*OF body-refines, of t1 t'*]

using *cond-match*

apply (*clarsimp simp add: succeeds-bind reaches-bind*)

using *Exn-def by force*

subgoal for *t*

apply (*subst (1 2 3) whileLoop-unroll*)

```

    apply (clarsimp simp add: cond-match)
    using ccorresE-exec-Stuck [OF body-refines, of t]
    apply (metis UNIV-I succeeds-bind top1I)
  done
subgoal for t
  apply (subst (1 2 3) whileLoop-unroll)
  apply (clarsimp simp add: cond-match)
  using ccorresE-exec-Fault [OF body-refines, of t]
  apply (metis UNIV-I succeeds-bind top1I)
  done
done
}
moreover
{
  fix s
  assume guard-abs: G (st s)
  assume guard-conc: s ∈ G'
  assume no-fail: succeeds (whileLoop (λ-. C) (λ-. B) ()) (st s)
  have ct → Γ ⊢ While C' B' ↓ Normal s
  apply clarify
  apply (rule ccorresE-termination [OF no-fail])
    apply (rule refl)
    apply (rule body-refines)
    apply (rule cond-match)
  apply simp
  done
}
ultimately show ?thesis
  by (auto simp: ccorresE-def)
qed

```

lemma *ccorresE-get*:

```

(∧ s. ccorresE st ct AF Γ (P and (λ s'. s' = s)) Q (L s) R) ⇒ ccorresE st ct AF
Γ P Q ((get-state) >>= L) R
  apply atomize
  apply (auto simp add: ccorresE-def succeeds-bind reaches-bind pred-conj-def split:
xstate.splits )
  done

```

lemma *ccorresE-fail*:

```

ccorresE st ct AF Γ P Q fail R
  apply (clarsimp simp: ccorresE-def)
  done

```

lemma *ccorresE-DynCom*:

```

[[ ∧ t. [[ t ∈ P' ]] ⇒ ccorresE st ct AF Γ P (P' ∩ {t'. t' = t}) A (B t) ]] ⇒
ccorresE st ct AF Γ P P' A (DynCom B)
  apply atomize
  apply (clarsimp simp: ccorresE-def)

```

```

apply (rule conjI)
apply clarsimp
apply (erule exec-Normal-elim-cases)
apply (erule allE, erule(1) impE)
apply clarsimp
apply clarify
apply (rule terminates.DynCom)
apply clarsimp
done

```

lemma *ccorresE-Catch-nothrow*:

```

 $\llbracket \text{ccorresE } st \text{ ct } AF \Gamma \top UNIV A A'; \neg \text{exceptions-thrown } A \rrbracket \implies$ 
 $\text{ccorresE } st \text{ ct } AF \Gamma \top UNIV A (TRY A' CATCH B' END)$ 
apply (clarsimp simp: ccorresE-def)
apply (rule conjI)
apply clarsimp
apply (erule exec-Normal-elim-cases)
apply (frule exceptions-thrown-not-abrupt, simp, simp)
apply simp
apply simp
apply clarify
apply (rule terminates.Catch)
apply clarsimp
apply clarsimp
apply (drule (1) exceptions-thrown-not-abrupt)
apply simp
apply simp
done

```

context *stack-heap-state*

begin

definition *with-fresh-stack-ptr* :: $nat \Rightarrow ('s \Rightarrow 'a \text{ list set}) \Rightarrow ('a::\text{mem-type } ptr \Rightarrow ('e::\text{default}, 'v, 's) \text{ spec-monad}) \Rightarrow ('e::\text{default}, 'v, 's) \text{ spec-monad}$

where

```

with-fresh-stack-ptr  $n \ I \ c \equiv$ 
  do {
     $p \leftarrow \text{assume-result-and-state } (\lambda s. \{(p, t). \exists d \text{ vs } bs. (p, d) \in \text{stack-allocs } n \ \mathcal{S} \text{ TYPE}('a::\text{mem-type}) \text{ (htd } s) \wedge$ 
       $vs \in I \ s \wedge \text{length } vs = n \wedge \text{length } bs = n * \text{size-of } \text{TYPE}('a) \wedge$ 
       $t = \text{hmem-upd } (\text{fold } (\lambda i. \text{heap-update-padding } (p +_p \text{ int } i) \text{ (vs!i) (take$ 
       $(\text{size-of } \text{TYPE}('a)) (\text{drop } (i * \text{size-of } \text{TYPE}('a)) \text{ bs})) [0..<n]) \text{ (htd-upd } (\lambda-. d)$ 
       $s)\});$ 
     $\text{on-exit } (c \ p)$ 
     $(\{(s, t). \exists bs. \text{length } bs = n * \text{size-of } \text{TYPE}('a) \wedge t = \text{hmem-upd } (\text{heap-update-list}$ 
     $(\text{ptr-val } p) \text{ bs) (htd-upd } (\text{stack-releases } n \ p) \ s)\})$ 
  }

```

```

lemma monotone-with-fresh-stack-ptr-le[partial-function-mono]:
  assumes [partial-function-mono]:  $\bigwedge p. \text{monotone } R (\leq) (\lambda f. c f p)$ 
  shows monotone  $R (\leq) (\lambda f. \text{with-fresh-stack-ptr } n I (c f))$ 
  unfolding with-fresh-stack-ptr-def on-exit-def
  by (intro partial-function-mono)

```

```

lemma monotone-with-fresh-stack-ptr-ge[partial-function-mono]:
  assumes [partial-function-mono]:  $\bigwedge p. \text{monotone } R (\geq) (\lambda f. c f p)$ 
  shows monotone  $R (\geq) (\lambda f. \text{with-fresh-stack-ptr } n I (c f))$ 
  unfolding with-fresh-stack-ptr-def on-exit-def
  by (intro partial-function-mono)

```

ML ‹

```

structure with-fresh-stack-ptr =
  struct

```

```

  type data = {
    match: term -> {n:term, init:term, c:term, ct: term, instantiate: {n:term,
init:term, c:term} -> term},
    cterm-match: cterm -> {n:cterm, init:cterm, c:cterm, ct: cterm, instantiate:
{n:cterm, init:cterm, c:cterm} -> cterm},
    term: typ -> term}

```

```

  fun map-match f ({match, cterm-match, term}:data) =
    {match = f match, cterm-match = cterm-match, term = term}:data
  fun map-cterm-match f ({match, cterm-match, term}:data) =
    {match = match, cterm-match = f cterm-match, term = term}:data
  fun map-term f ({match, cterm-match, term}:data) =
    {match = match, cterm-match = cterm-match, term = f term}:data

```

```

  fun merge-match (f, g) = Utils.fast-merge (fn (f, g) => Utils.first-match [f, g])
  (f, g)

```

```

  structure Data = Generic-Data (
    type T = data;
    val empty = {match = fn - => raise Match, cterm-match = fn - => raise
Match, term = fn - => raise Match}
    val merge = Utils.fast-merge (fn ({match = f1, cterm-match = g1, term = t1},
{match = f2, cterm-match = g2, term = t2}) =>
      {match = merge-match (f1, f2), cterm-match = merge-match (g1, g2),
term = merge-match (t1, t2)});
  )

```

```

  fun match ctxt = #match (Data.get (Context.Proof ctxt))

```

```

fun cterm-match ctxt = #cterm-match (Data.get (Context.Proof ctxt))
fun term ctxt = #term (Data.get (Context.Proof ctxt))

fun add-match match = Data.map (map-match (Utils.add-match match))
fun add-cterm-match cterm-match = Data.map (map-cterm-match (Utils.add-match
cterm-match))
fun add-term match = Data.map (map-term (Utils.add-match match))

end
>

declaration <
fn phi => fn context =>
  let
    val T = Morphism.typ phi @ {typ 's}
    val t = Morphism.term phi @ {term with-fresh-stack-ptr}
    val thy = Context.theory-of context
    fun term T' =
      let
        in
          if can (Sign.typ-match thy (T, T')) Vartab.empty then t else raise Match
        end
      fun match t = @ {morph-match (fo) <with-fresh-stack-ptr ?n ?init ?c>} phi
      (Context.theory-of context) t
      handle Pattern.MATCH => raise Match
      fun cterm-match ct = @ {cterm-morph-match (fo) <with-fresh-stack-ptr ?n ?init
      ?c>} phi ct
      handle Pattern.MATCH => raise Match
    in
      context
      |> with-fresh-stack-ptr.add-match match
      |> with-fresh-stack-ptr.add-cterm-match cterm-match
      |> with-fresh-stack-ptr.add-term term
    end
  end
>

end

end

```

Chapter 18

L1 phase

```
theory L1Defs
imports CCorresE
begin
```

```
type-synonym 's L1-monad = (unit, unit, 's) exn-monad
```

```
definition L1-seq (A :: 's L1-monad) (B :: 's L1-monad) ≡ (A >>= (λ-. B)) :: 's L1-monad
```

```
definition L1-skip ≡ return () :: 's L1-monad
```

```
definition L1-modify m ≡ (modify m) :: 's L1-monad
```

```
definition L1-condition c (A :: 's L1-monad) (B :: 's L1-monad) ≡ condition c A B
```

```
definition L1-catch (A :: 's L1-monad) (B :: 's L1-monad) ≡ (A <catch> (λ-. B))
```

```
definition L1-while c (A :: 's L1-monad) ≡ (whileLoop (λ-. c) (λ-. A) ())
```

```
definition L1-throw ≡ throw () :: 's L1-monad
```

```
definition L1-spec r ≡ state-select r :: 's L1-monad
```

```
definition L1-assume f ≡ assume-result-and-state f :: 's L1-monad
```

```
definition L1-guard c ≡ guard c :: 's L1-monad
```

```
definition L1-init v ≡ (do { x ← select UNIV; modify (v (λ-. x)) }) :: 's L1-monad
```

```
definition L1-call scope-setup (dest-fn :: 's L1-monad) scope-teardown result-exn return-xf ≡
```

```
do {
  s ← get-state;
  ((do { modify scope-setup;
         dest-fn })
   <catch> (λ-. L1-seq (modify (λt. result-exn (scope-teardown s t) t))
             L1-throw));
  t ← get-state;
  modify (scope-teardown s);
  modify (return-xf t)
}
```

definition $L1\text{-fail} \equiv \text{fail} :: 's \text{ L1-monad}$

definition $L1\text{-set-to-pred } S \equiv \lambda s. s \in S$

definition $L1\text{-rel-to-fun } R = (\lambda s. \text{Pair } () \text{ ' Image } R \{s\})$

lemma $L1\text{-rel-to-fun-alt}: L1\text{-rel-to-fun } R = (\lambda s. \text{Pair } () \text{ ' } \{s'. (s, s') \in R\})$

by (*auto simp: L1-rel-to-fun-def*)

lemmas $L1\text{-defs} = L1\text{-seq-def } L1\text{-skip-def } L1\text{-modify-def } L1\text{-condition-def}$

$L1\text{-catch-def } L1\text{-while-def } L1\text{-throw-def } L1\text{-spec-def } L1\text{-assume-def } L1\text{-guard-def}$

$L1\text{-fail-def } L1\text{-init-def } L1\text{-set-to-pred-def } L1\text{-rel-to-fun-def}$

lemmas $L1\text{-defs}' =$

$L1\text{-seq-def } L1\text{-skip-def } L1\text{-modify-def } L1\text{-condition-def } L1\text{-catch-def}$

$L1\text{-while-def } L1\text{-throw-def } L1\text{-spec-def } L1\text{-assume-def}$

$L1\text{-guard-def}$

$L1\text{-fail-def } L1\text{-init-def } L1\text{-set-to-pred-def } L1\text{-rel-to-fun-def}$

declare $L1\text{-set-to-pred-def}$ [*simp*]

declare $L1\text{-rel-to-fun-def}$ [*simp*]

definition $L1\text{-guarded}:: ('s \Rightarrow \text{bool}) \Rightarrow 's \text{ L1-monad} \Rightarrow 's \text{ L1-monad}$

where

$L1\text{-guarded } g \ f = L1\text{-seq } (L1\text{-guard } g) \ f$

locale $L1\text{-functions} =$

fixes $\mathcal{P}:: \text{unit ptr} \Rightarrow 's \text{ L1-monad}$

begin

definition $L1\text{-dyn-call } g \ \text{scope-setup} \ (\text{dest} :: 's \Rightarrow \text{unit ptr}) \ \text{scope-teardown} \ \text{result-exn} \ \text{return-xf} \equiv$

$L1\text{-guarded } g \ (\text{gets } \text{dest} \ >>= (\lambda p. L1\text{-call } \text{scope-setup} \ (\mathcal{P} \ p) \ \text{scope-teardown} \ \text{result-exn} \ \text{return-xf}))$

end

definition

$L1\text{corres} :: \text{bool} \Rightarrow ('p \Rightarrow ('s, 'p, \text{strict-error-type}) \text{ com option})$

$\Rightarrow 's \text{ L1-monad} \Rightarrow ('s, 'p, \text{strict-error-type}) \text{ com} \Rightarrow \text{bool}$

where

$L1\text{corres } \text{check-term } \Gamma \equiv$

$\lambda A \ C. \forall s. \text{succeeds } A \ s \longrightarrow$

$((\forall t. \Gamma \vdash \langle C, \text{Normal } s \rangle \Rightarrow t \longrightarrow$

(*case t of*

$\text{Normal } s' \Rightarrow \text{reaches } A \ s \ (\text{Result } ()) \ s'$

$| \text{Abrupt } s' \Rightarrow \text{reaches } A \ s \ (\text{Exn } ()) \ s'$

$| \text{Fault } e \Rightarrow e \in \{\text{AssumeError}, \text{StackOverflow}\}$

$| - \Rightarrow \text{False}))$

$\wedge (\text{check-term} \longrightarrow \Gamma \vdash C \downarrow \text{Normal } s))$

definition

$L1corres' :: \text{bool} \Rightarrow ('p \Rightarrow ('s, 'p, \text{strictc-errortype}) \text{ com option}) \Rightarrow ('s \Rightarrow \text{bool})$
 $\Rightarrow 's \text{ L1-monad} \Rightarrow ('s, 'p, \text{strictc-errortype}) \text{ com} \Rightarrow \text{bool}$

where

$L1corres' \text{ check-term } \Gamma \ P \equiv$
 $\lambda A \ C. \forall s. (P \ s) \wedge \text{succeeds } A \ s \longrightarrow$
 $((\forall t. \Gamma \vdash \langle C, \text{Normal } s \rangle \Rightarrow t \longrightarrow$
 $(\text{case } t \text{ of}$
 $\quad \text{Normal } s' \Rightarrow \text{reaches } A \ s \ (\text{Result } ()) \ s'$
 $\quad | \text{Abrupt } s' \Rightarrow \text{reaches } A \ s \ (\text{Exn } ()) \ s'$
 $\quad | \text{Fault } e \Rightarrow e \in \{\text{AssumeError}, \text{StackOverflow}\}$
 $\quad | - \Rightarrow \text{False}))$
 $\wedge (\text{check-term} \longrightarrow \Gamma \vdash C \downarrow \text{Normal } s))$

lemma $L1corres\text{-alt-def}$: $L1corres \ ct \ \Gamma = \text{ccorresE} \ (\lambda x. \ x) \ ct \ \{\text{AssumeError}, \text{StackOverflow}\} \ \Gamma \ \top \ \text{UNIV}$

apply $(\text{rule ext})+$
apply $(\text{clarsimp simp: } L1corres\text{-def} \ \text{ccorresE}\text{-def})$
done

lemma $L1corres'\text{-alt-def}$: $L1corres' \ ct \ \Gamma \ P = \text{ccorresE} \ (\lambda x. \ x) \ ct \ \{\text{AssumeError}, \text{StackOverflow}\} \ \Gamma \ P \ \text{UNIV}$

apply $(\text{rule ext})+$
apply $(\text{clarsimp simp: } L1corres'\text{-def} \ \text{ccorresE}\text{-def})$
done

lemma $\text{admissible-nondet-ord-L1corres}$ [corres-admissible]:

$\text{ccpo.admissible Inf } (\geq) \ (\lambda A. \ L1corres \ ct \ \Gamma \ A \ C)$

unfolding $L1corres\text{-def} \ \text{imp-conjR} \ \text{disj-imp}[\text{symmetric}]$

apply $(\text{intro admissible-all})$

apply $(\text{intro admissible-conj})$

apply $(\text{rule ccpo.admissibleI})$

apply $(\text{clarsimp split: } \text{xstate.splits})$

apply $(\text{intro conjI admissible-mem})$

subgoal

apply $(\text{simp add: } \text{ccpo.admissible-def} \ \text{chain-def}$

$\text{succeeds-def} \ \text{reaches-def} \ \text{split: } \text{prod.splits})$

apply transfer

apply $(\text{clarsimp simp add: } \text{Inf-post-state-def} \ \text{top-post-state-def} \ \text{image-def} \ \text{vim-age-def}$

$\text{split: } \text{if-split-asm})$

by $(\text{metis } \text{outcomes.simps}(2) \ \text{post-state.simps}(3))$

subgoal

apply $(\text{simp add: } \text{ccpo.admissible-def} \ \text{chain-def})$

```

    succeeds-def reaches-def split: prod.splits)
  apply transfer
  apply (clarsimp simp add: Inf-post-state-def top-post-state-def image-def vimage-def
    age-def
    split: if-split-asm)
  by (metis outcomes.simps(2) post-state.simps(3))
  subgoal
  apply (simp add: ccpo.admissible-def chain-def
    succeeds-def reaches-def split: prod.splits)
  apply transfer
  apply (clarsimp simp add: Inf-post-state-def top-post-state-def image-def vimage-def
    age-def
    split: if-split-asm)
  by (metis outcomes.simps(2) post-state.simps(3))
  subgoal
  apply (simp add: ccpo.admissible-def chain-def
    succeeds-def reaches-def split: prod.splits)
  apply transfer
  apply (auto simp add: Inf-post-state-def top-post-state-def image-def vimage-def
    split: if-split-asm)
done
apply (rule ccpo.admissibleI)
apply (simp split: xstate.splits)
apply (clarsimp simp add: ccpo.admissible-def chain-def
  succeeds-def reaches-def split: prod.splits)
apply transfer
apply (auto simp add: Inf-post-state-def top-post-state-def image-def vimage-def
  split: if-split-asm)
done

```

lemma *L1corres-top* [*corres-top*]: *L1corres ct P \top C*
 by (*auto simp add: L1corres-def*)

lemma *L1corres-guard-DynCom*:

```

[[ $\bigwedge s. s \in g \implies L1corres\ ct\ \Gamma\ (B\ s)\ (B'\ s)$ ]]  $\implies$ 
  L1corres ct  $\Gamma$  (L1-seq (L1-guard ( $\lambda s. s \in g$ )) (gets B  $\ggg$  ( $\lambda b. b$ ))) (Guard f g
(DynCom B'))
  apply (clarsimp simp: L1corres-def L1-defs)
  apply (force elim!: exec-Normal-elim-cases
    intro: terminates.intros
    split: xstate.splits
    simp add: succeeds-bind reaches-bind)
done

```

lemma *L1corres'-guard-DynCom-conseq*:

```

assumes conseq:  $\bigwedge s. P\ s \implies g\ s \implies s \in g'$ 
assumes corres:  $\bigwedge s. P\ s \implies g\ s \implies L1corres'\ ct\ \Gamma\ (\lambda s. P\ s \wedge g\ s)\ (B\ s)\ (B'\ s)$ 
shows L1corres' ct  $\Gamma$  P (L1-seq (L1-guard g) (gets B  $\ggg$  ( $\lambda b. b$ ))) (Guard f g'

```

```

(DynCom B')
  using conseq corres
  apply (clarsimp simp: L1corres'-def L1-defs)
  apply (force elim!: exec-Normal-elim-cases
        intro: terminates.intros
        split: xstate.splits
        simp add: succeeds-bind reaches-bind)

done

```

```

lemma L1corres'-guard-DynCom:
  [[ $\bigwedge s. P s \implies s \in g \implies L1corres' ct \Gamma (\lambda s. P s \wedge s \in g) (B s) (B' s)$ ]]  $\implies$ 
   $L1corres' ct \Gamma P (L1seq (L1guard (\lambda s. s \in g)) (gets B \gg (\lambda b. b))) (Guard$ 
   $f g (DynCom B'))$ 
  apply (rule L1corres'-guard-DynCom-conseq [where B=B])
  apply simp
  apply simp
done

```

```

lemma L1corres-DynCom:
  assumes corres-f:  $\bigwedge s. L1corres ct \Gamma (g s) (f s)$ 
  shows  $L1corres ct \Gamma (gets g \gg (\lambda b. b)) (DynCom f)$ 
  using corres-f
  apply (clarsimp simp: L1corres-def L1-defs)
  apply (force elim!: exec-Normal-elim-cases
        intro: terminates.intros
        split: xstate.splits
        simp add: succeeds-bind reaches-bind)

done

```

```

lemma L1corres'-DynCom:
  assumes corres-f:  $\bigwedge s. P s \implies L1corres' ct \Gamma P (g s) (f s)$ 
  shows  $L1corres' ct \Gamma P (gets g \gg (\lambda b. b)) (DynCom f)$ 
  using corres-f
  apply (clarsimp simp: L1corres'-def L1-defs)
  apply (force elim!: exec-Normal-elim-cases
        intro: terminates.intros
        split: xstate.splits
        simp add: succeeds-bind reaches-bind)

done

```

```

lemma L1corres'-DynCom-fix-state:
  assumes corres-f:  $\bigwedge s. P s \implies L1corres' ct \Gamma (\lambda s'. P s' \wedge s' = s) (g s) (f s)$ 
  shows  $L1corres' ct \Gamma P (gets g \gg (\lambda b. b)) (DynCom f)$ 
  using corres-f
  apply (clarsimp simp: L1corres'-def L1-defs)
  apply (force elim!: exec-Normal-elim-cases)

```

intro: terminates.intros
split: xstate.splits
simp add: succeeds-bind reaches-bind
done

lemma *L1corres'-guard'*:

$\llbracket L1corres' \text{ ct } \Gamma (\lambda s. P \ s \wedge s \in g) \ B \ B' \rrbracket \implies$
 $L1corres' \text{ ct } \Gamma \ P \ (L1seq \ (L1guard \ (\lambda s. s \in g)) \ B) \ (Guard \ f \ g \ B')$
apply (*clarsimp simp: L1corres'-def L1-defs*)
apply (*erule-tac x=s in allE*)
apply (*rule conjI*)
apply *clarsimp*
apply (*erule exec-Normal-elim-cases*)
subgoal
by (*auto simp: succeeds-bind reaches-bind split: xstate.splits*)
subgoal
by (*auto simp: succeeds-bind reaches-bind split: xstate.splits*)
subgoal
by (*auto simp: succeeds-bind intro: terminates.intros*)
done

lemma *L1corres'-guarded*:

$\llbracket L1corres' \text{ ct } \Gamma (\lambda s. P \ s \wedge s \in g) \ B \ B' \rrbracket \implies$
 $L1corres' \text{ ct } \Gamma \ P \ (L1guarded \ (\lambda s. s \in g) \ B) \ (Guard \ f \ g \ B')$
unfolding *L1guarded-def* **by** (*rule L1corres'-guard'*)

lemma *L1corres'-Guard-maybe-guard*:

$L1corres' \text{ ct } \Gamma \ P \ B \ (Guard \ f \ g \ B') \implies L1corres' \text{ ct } \Gamma \ P \ B \ (maybe-guard \ f \ g \ B')$
apply (*simp add: L1corres'-def maybe-guard-def*)
by (*meson exec.Guard iso-tuple-UNIV-I terminates-Normal-elim-cases(2)*)

lemma *L1corres'-guarded-DynCom-conseq*:

assumes *conseq: $\bigwedge s. P \ s \implies g \ s \implies s \in g'$*
assumes *corres-B: $\bigwedge s. P \ s \implies g \ s \implies L1corres' \text{ ct } \Gamma (\lambda s. P \ s \wedge g \ s) (B \ s) (B' \ s)$*
shows $L1corres' \text{ ct } \Gamma \ P \ (L1guarded \ g \ (gets \ B \ \ggg (\lambda b. b))) \ (maybe-guard \ f \ g' \ (DynCom \ B'))$
apply (*rule L1corres'-Guard-maybe-guard*)
unfolding *L1guarded-def L1-set-to-pred-def*
using *corres-B*
by (*simp add: L1corres'-guard-DynCom-conseq [OF conseq]*)

lemma *L1corres'-guarded-DynCom*:

assumes *corres-B: $\bigwedge s. P \ s \implies s \in g \implies L1corres' \text{ ct } \Gamma (\lambda s. P \ s \wedge s \in g) (B \ s) (B' \ s)$*
shows $L1corres' \text{ ct } \Gamma \ P \ (L1guarded \ (L1set-to-pred \ g) \ (gets \ B \ \ggg (\lambda b. b))) \ (maybe-guard \ f \ g \ (DynCom \ B'))$
apply (*rule L1corres'-guarded-DynCom-conseq [where B=B]*)

```

apply simp
using corres-B
apply simp
done

```

```

lemma L1corres'-conseq:
assumes corres-Q:  $L1corres' \text{ ct } \Gamma \ Q \ B \ B'$ 
assumes conseq:  $\bigwedge s. P \ s \implies Q \ s$ 
shows  $L1corres' \text{ ct } \Gamma \ P \ B \ B'$ 
using conseq corres-Q
by (auto simp add: L1corres'-def)

```

```

lemma L1corres-to-L1corres':  $L1corres \text{ ct } \Gamma = L1corres' \text{ ct } \Gamma \ \top$ 
by (simp add: L1corres-def L1corres'-def)

```

```

lemma L1corres-guarded-DynCom-conseq:
assumes conseq:  $\bigwedge s. g \ s \implies s \in g'$ 
assumes corres-B:  $\bigwedge s. g \ s \implies L1corres \text{ ct } \Gamma \ (B \ s) \ (B' \ s)$ 
shows  $L1corres \text{ ct } \Gamma \ (L1\text{-guarded } g \ (\text{gets } B \ \gg\ (\lambda b. b))) \ (\text{maybe-guard } f \ g' \ (DynCom \ B'))$ 
using corres-B unfolding L1corres-to-L1corres'
thm L1corres'-guarded-DynCom
apply –
apply (rule L1corres'-guarded-DynCom-conseq [where  $B=B$  ] )
using conseq apply simp
apply (rule L1corres'-conseq [where  $Q = \top$  ] )
apply (simp)
apply simp
done

```

```

lemma L1corres-guarded-DynCom:
assumes corres-B:  $\bigwedge s. s \in g \implies L1corres \text{ ct } \Gamma \ (B \ s) \ (B' \ s)$ 
shows  $L1corres \text{ ct } \Gamma \ (L1\text{-guarded } (L1\text{-set-to-pred } g) \ (\text{gets } B \ \gg\ (\lambda b. b))) \ (\text{maybe-guard } f \ g \ (DynCom \ B'))$ 
using corres-B unfolding L1corres-to-L1corres'
apply –
apply (rule L1corres'-guarded-DynCom [where  $B=B$  ] )
apply (rule L1corres'-conseq [where  $Q = \top$  ] )
apply simp
apply simp
done

```

definition

```

L1-call-simpl check-term Gamma proc
= do {  $s \leftarrow \text{get-state}$ ;
       $\text{assert } (\text{check-term} \longrightarrow \text{Gamma} \vdash \text{Call } \text{proc} \downarrow \text{Normal } s)$ ;
       $xs \leftarrow \text{select } \{t. \text{Gamma} \vdash \langle \text{Call } \text{proc}, \text{Normal } s \rangle \Rightarrow t\}$ ;

```

```

    case xs :: (-, strictc-errortype) xstate of
      Normal s ⇒ set-state s
    | Abrupt s ⇒ do {set-state s; throw () }
    | Fault ft ⇒ fail
    | Stuck ⇒ fail
  }

```

lemma *L1corres-call-simpl:*

```

L1corres ct Γ (L1-call-simpl ct Γ proc) (Call proc)
apply (clarsimp simp: L1corres-def L1-call-simpl-def)
apply safe
subgoal for s t
  apply (cases t)
  apply (fastforce elim!: exec-Normal-elim-cases
    intro: terminates.intros exec.intros

    simp add: succeeds-bind reaches-bind Exn-def )+
  done
subgoal for s
  apply (auto intro!: terminates.intros simp add: succeeds-bind reaches-bind)
  done
done

```

lemma *L1corres-skip:*

```

L1corres ct Γ L1-skip SKIP
unfolding L1corres-def L1-defs
by (force elim!: exec-Normal-elim-cases
  intro: terminates.intros
  split: xstate.splits
  simp add: succeeds-bind reaches-bind)

```

lemma *L1corres-throw:*

```

L1corres ct Γ L1-throw Throw
unfolding L1corres-def L1-defs
by (auto elim!: exec-Normal-elim-cases
  intro: terminates.intros
  split: xstate.splits
  simp add: succeeds-bind reaches-bind Exn-def)

```

lemma *L1corres-seq:*

```

[[ L1corres ct Γ L L'; L1corres ct Γ R R' ]] ⇒
  L1corres ct Γ (L1-seq L R) (L' ;; R')
apply (clarsimp simp: L1corres-alt-def)
apply (clarsimp simp: L1-seq-def)
apply (rule ccorresE-Seq)
apply auto
done

```

lemma *L1corres-modify*:
L1corres ct Γ (*L1-modify* *m*) (*Basic m*)
apply (*clarsimp simp*: *L1corres-def L1-defs*)
apply (*auto elim!*: *exec-Normal-elim-cases*
intro: *terminates.intros*
split: *xstate.splits*
simp add: *succeeds-bind reaches-bind Exn-def*)
done

lemma *L1corres-condition*:
 $\llbracket L1corres\ ct\ \Gamma\ L\ L';\ L1corres\ ct\ \Gamma\ R\ R' \rrbracket \implies$
L1corres ct Γ (*L1-condition* (*L1-set-to-pred* *c*) *L R*) (*Cond c L' R'*)
apply (*clarsimp simp*: *L1corres-alt-def*)
apply (*clarsimp simp*: *L1-defs*)
apply (*rule ccorresE-Cond-match*)
apply (*erule ccorresE-guard-imp, simp+*)[1]
apply (*erule ccorresE-guard-imp, simp+*)[1]
apply *simp*
done

lemma *L1corres-catch*:
 $\llbracket L1corres\ ct\ \Gamma\ L\ L';\ L1corres\ ct\ \Gamma\ R\ R' \rrbracket \implies$
L1corres ct Γ (*L1-catch* *L R*) (*Catch L' R'*)
apply (*clarsimp simp*: *L1corres-alt-def*)
apply (*clarsimp simp*: *L1-catch-def*)
apply (*erule ccorresE-Catch*)
apply *force*
done

lemma *L1corres-while*:
 $\llbracket L1corres\ ct\ \Gamma\ B\ B' \rrbracket \implies$
L1corres ct Γ (*L1-while* (*L1-set-to-pred* *c*) *B*) (*While c B'*)
apply (*clarsimp simp*: *L1corres-alt-def*)
apply (*clarsimp simp*: *L1-defs*)
apply (*rule ccorresE-While*)
apply *clarsimp*
apply *simp*
done

lemma *L1corres-guard*:
 $\llbracket L1corres\ ct\ \Gamma\ B\ B' \rrbracket \implies$
L1corres ct Γ (*L1-seq* (*L1-guard* (*L1-set-to-pred* *c*)) *B*) (*Guard f c B'*)
apply (*clarsimp simp*: *L1corres-alt-def*)
apply (*clarsimp simp*: *ccorresE-def L1-defs*)
apply (*erule-tac x=s in allE*)
apply (*auto elim!*: *exec-Normal-elim-cases*
intro: *terminates.intros*
split: *xstate.splits*
simp add: *succeeds-bind reaches-bind Exn-def*)

done

lemma *L1corres-spec*:

L1corres ct Γ (*L1-spec* *x*) (*com.Spec* *x*)
apply (*clarsimp simp*: *L1corres-def L1-defs*)
apply (*auto elim!*: *exec-Normal-elim-cases*
 intro: *terminates.intros*
 split: *xstate.splits*
 simp add: *succeeds-bind reaches-bind Exn-def*)
done

lemma *L1-init-alt-def*:

L1-init upd \equiv *L1-spec* $\{(s, t). \exists v. t = \text{upd } (\lambda-. v) s\}$
apply (*rule eq-reflection*)
apply (*clarsimp simp*: *L1-defs*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L1corres-init*:

L1corres ct Γ (*L1-init upd*) (*lvar-nondet-init upd*)
by (*auto simp*: *L1-init-alt-def lvar-nondet-init-def intro*: *L1corres-spec*)

lemma *L1corres-guarded-spec*:

L1corres ct Γ (*L1-spec* *R*) (*guarded-spec-body* *F* *R*)
apply (*clarsimp simp*: *L1corres-alt-def ccorresE-def L1-spec-def guarded-spec-body-def*)
apply (*force simp*: *liftE-def bind-def*
 elim: *exec-Normal-elim-cases intro*: *terminates.Guard terminates.Spec*)
done

lemma *L1corres-assume*:

L1corres ct Γ (*L1-assume* (*L1-rel-to-fun* *R*)) (*guarded-spec-body* *AssumeError* *R*)
apply (*clarsimp simp*: *L1corres-alt-def ccorresE-def L1-assume-def L1-rel-to-fun-alt*
guarded-spec-body-def)
apply (*auto elim!*: *exec-Normal-elim-cases*
 intro: *terminates.intros*
 split: *xstate.splits*
 simp add: *succeeds-bind reaches-bind Exn-def*)
done

lemma *pred-conj-apply[simp]*: (*P and Q*) *s* \longleftrightarrow *P s* \wedge *Q s*

by (*auto simp add*: *pred-conj-def*)

lemma *L1corres-call*:

\llbracket *L1corres ct* Γ *dest-fn* (*Call* *dest*) $\rrbracket \implies$
L1corres ct Γ
 (*L1-call* *scope-setup* *dest-fn* *scope-teardown-norm* *scope-teardown-exn* *f*)
 (*call-exn* *scope-setup* *dest* *scope-teardown-norm* *scope-teardown-exn* $(\lambda- t.$


```

Basic (f t))
  apply (clarsimp simp: L1corres-alt-def)
  apply (unfold call-exn-def block-exn-def L1-call-def)
  apply (rule ccorresE-DynCom)
  apply clarsimp
  apply (rule ccorresE-get)
  apply (rule ccorresE-assume-pre, clarsimp)
  apply (rule ccorresE-guard-imp-stronger)
    apply (rule ccorresE-Seq)
    apply (rule ccorresE-Catch)
    apply (rule ccorresE-Seq)
      apply (rule L1corres-modify[unfolded L1-modify-def L1corres-alt-def])
    apply assumption
    apply (rule L1corres-seq[unfolded L1corres-alt-def])
      apply (rule L1corres-modify[unfolded L1-modify-def L1corres-alt-def])
    apply (rule L1corres-throw [unfolded L1corres-alt-def])
  apply (rule ccorresE-DynCom)
  apply (rule ccorresE-get)
  apply (rule ccorresE-assume-pre, clarsimp)
  apply (rule ccorresE-guard-imp)
    apply (rule ccorresE-Seq)
      apply (rule L1corres-modify[unfolded L1-modify-def L1corres-alt-def])
      apply (rule L1corres-modify[unfolded L1-modify-def L1corres-alt-def])
    apply simp
  apply simp
  apply simp
  apply simp
done

lemma (in L1-functions) L1corres-dyn-call-conseq:
  assumes conseq:  $\bigwedge s. g s \implies s \in g'$ 
  assumes corres-dest:  $\bigwedge s. g s \implies L1corres\ ct\ \Gamma\ (\mathcal{P}\ (dest\ s))\ (Call\ (dest\ s))$ 
  shows
    L1corres\ ct\ \Gamma
      (L1-dyn-call\ g\ scope-setup\ dest\ scope-teardown-norm\ scope-teardown-exn
result)
      (dynCall-exn\ f\ g'\ scope-setup\ dest\ scope-teardown-norm\ scope-teardown-exn
result)
    ( $\lambda t. Basic\ (result\ t)$ )
  proof -
    have eq: (gets ( $\lambda s. L1-call\ scope-setup\ (\mathcal{P}\ (dest\ s))\ scope-teardown-norm\ scope-teardown-exn$ 
result)  $\ggg$  ( $\lambda b. b$ )) =
      (gets dest  $\ggg$  ( $\lambda p. L1-call\ scope-setup\ (\mathcal{P}\ p)\ scope-teardown-norm\ scope-teardown-exn$ 
result))
    apply (rule spec-monad-ext)
    apply (simp add: run-bind)
    done
  show ?thesis
    apply (simp add: L1-dyn-call-def dynCall-exn-def)
    apply (rule L1corres-guarded-DynCom-conseq [where B =  $\lambda s. L1-call\ scope-setup$ 

```

$(\mathcal{P} (dest\ s))$ *scope-teardown-norm scope-teardown-exn result, simplified eq*)
apply (*simp add: conseq*)
apply (*rule L1corres-call*)
apply (*rule corres-dest*)
by *simp*
qed

lemma (*in L1-functions*) *L1corres-dyn-call-same-guard*:
assumes *eq: L1-set-to-pred g \equiv g'*
assumes *corres-dest: $\bigwedge s. g' s \implies L1corres\ ct\ \Gamma\ (\mathcal{P}\ (dest\ s))\ (Call\ (dest\ s))$*
shows
 $L1corres\ ct\ \Gamma$
 $(L1-dyn-call\ g'\ scope-setup\ dest\ scope-teardown-norm\ scope-teardown-exn$
result)
 $(dynCall-exn\ f\ g\ scope-setup\ dest\ scope-teardown-norm\ scope-teardown-exn$
 $(\lambda\ t. Basic\ (result\ t)))$
apply (*rule L1corres-dyn-call-conseq*)
apply (*simp add: eq [symmetric]*)
by (*rule corres-dest*)

lemma (*in L1-functions*) *L1corres-dyn-call-add-and-select-guard*:
assumes *eq: L1-set-to-pred g \equiv g'*
assumes *corres-dest: $\bigwedge s. G\ s \implies L1corres\ ct\ \Gamma\ (\mathcal{P}\ (dest\ s))\ (Call\ (dest\ s))$*
shows
 $L1corres\ ct\ \Gamma$
 $(L1-dyn-call\ (G\ and\ g')\ scope-setup\ dest\ scope-teardown-norm\ scope-teardown-exn$
result)
 $(dynCall-exn\ f\ g\ scope-setup\ dest\ scope-teardown-norm\ scope-teardown-exn$
 $(\lambda\ t. Basic\ (result\ t)))$
apply (*rule L1corres-dyn-call-conseq*)
apply (*simp add: eq [symmetric]*)
apply (*rule corres-dest*)
apply (*simp*)
done

lemma *L1-seq-guard-merge: L1-seq (L1-guard P) (L1-seq (L1-guard Q) c) = L1-seq*
 $(L1-guard\ (P\ and\ Q))\ c$
unfolding *L1-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *and-unfold: (and) = $(\lambda P\ Q\ s. P\ s \wedge Q\ s)$*
by (*auto simp add: fun-eq-iff*)

lemma *L1-seq-guard-eq: $(\bigwedge s. P\ s = Q\ s) \implies L1-seq\ (L1-guard\ P)\ c = L1-seq$*
 $(L1-guard\ Q)\ c$

unfolding *L1-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *foldr-and-commute*: $\bigwedge s. \text{foldr } (\text{and}) \text{ gs } (P \text{ and } g) s = (g s \wedge \text{foldr } (\text{and}) \text{ gs } P s)$
by (*induct gs arbitrary: P g*) (*auto simp add:*)

lemma *L1corres-fail*:
L1corres ct Γ L1-fail X
apply (*clarsimp simp: L1corres-alt-def*)
apply (*clarsimp simp: L1-fail-def*)
apply (*rule ccorresE-fail*)
done

lemma *L1corres-prepend-unknown-var'*:
 $\llbracket L1corres \text{ ct } \Gamma A C; \bigwedge s::'s::\text{type}. X (\lambda-::'a::\text{type}. (X' s)) s = s \rrbracket \implies L1corres \text{ ct } \Gamma (L1\text{-seq } (L1\text{-init } X) A) C$
unfolding *L1corres-def L1-defs*
by (*auto elim!: exec-Normal-elim-cases*
intro: terminates.intros
split: xstate.splits
simp add: succeeds-bind reaches-bind Exn-def)
metis+

lemma *L1-catch-seq-join*: $\text{no-throw } (\lambda-. \text{True}) A \implies L1\text{-seq } A (L1\text{-catch } B C) = (L1\text{-catch } (L1\text{-seq } A B) C)$
unfolding *no-throw-def L1-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add: runs-to-iff*)
apply (*clarsimp simp add: runs-to-def-old runs-to-partial-def-old Exn-def default-option-def*)
apply (*intro conjI impI iffI*)
apply (*metis Exn-def Exn-neq-Result*)
done

lemma *no-throw-L1-init [simp]*: $\text{no-throw } P (L1\text{-init } f)$
unfolding *L1-init-def no-throw-def*
apply (*clarsimp*)
apply (*runs-to-vcg*)
done

lemma *L1corres-prepend-unknown-var*:
 $\llbracket L1corres \text{ ct } \Gamma (L1\text{-catch } A B) C; \bigwedge s. X (\lambda-::'d::\text{type}. (X' s)) s = s \rrbracket$

$\implies L1corres\ ct\ \Gamma\ (L1catch\ (L1seq\ (L1init\ X)\ A)\ B)\ C$
unfolding $L1corres-def\ L1-defs$
by (*auto elim!*: *exec-Normal-elim-cases*
intro: terminates.intros
split: xstate.splits
simp add: succeeds-bind reaches-bind Exn-def succeeds-catch reaches-catch)
metis+

lemma $L1corres-Call$:
 $\llbracket \Gamma\ X' = Some\ Z';\ L1corres\ ct\ \Gamma\ Z\ Z' \rrbracket \implies$
 $L1corres\ ct\ \Gamma\ Z\ (Call\ X')$
apply (*clarsimp simp: L1corres-alt-def*)
apply (*erule (1) ccorresE-Call*)
done

lemma $L1-call-corres\ [fundef-cong]$:
 $\llbracket scope-setup = scope-setup';$
 $dest-fn = dest-fn';$
 $scope-teardown = scope-teardown';$
 $return-xf = return-xf' \rrbracket \implies$
 $L1call\ scope-setup\ dest-fn\ scope-teardown\ return-xf =$
 $L1call\ scope-setup'\ dest-fn'\ scope-teardown'\ return-xf'$
apply (*clarsimp simp: L1-call-def*)
done

lemma $L1-corres-cleanup$:
 $L1corres\ ct\ \Gamma\ (do\ \{y <- state-select\ \{(s,t).\ t = cleanup\ s\};$
 $return\ ()$
 $\})$
(Basic cleanup)
unfolding $L1corres-def\ L1-defs$
by (*auto elim!*: *exec-Normal-elim-cases*
intro: terminates.intros
split: xstate.splits
simp add: succeeds-bind reaches-bind Exn-def)

lemma $L1-corres-spec-cleanup$:
 $L1corres\ ct\ \Gamma\ (do\ \{y <- state-select\ cleanup;$
 $return\ ()$
 $\})$
(com.Spec cleanup)
unfolding $L1corres-def\ L1-defs$
by (*auto elim!*: *exec-Normal-elim-cases*
intro: terminates.intros
split: xstate.splits
simp add: succeeds-bind reaches-bind Exn-def)

lemma $L1-corres-cleanup-throw$:

L1corres $ct \Gamma$ (do { - <- state-select {(s,t). t = cleanup s};
 throw ()
 })
 (*Basic cleanup;; THROW*)
unfolding *L1corres-def L1-defs*
by (*auto elim!:* *exec-Normal-elim-cases*
intro: *terminates.intros*
split: *xstate.splits*
simp add: *succeeds-bind reaches-bind Exn-def default-option-def*)

lemma *L1-corres-spec-cleanup-throw:*
L1corres $ct \Gamma$ (do{ - <- state-select cleanup;
 throw ()
 })
 (*com.Spec cleanup;; THROW*)
unfolding *L1corres-def L1-defs*
by (*auto elim!:* *exec-Normal-elim-cases*
intro: *terminates.intros*
split: *xstate.splits*
simp add: *succeeds-bind reaches-bind Exn-def default-option-def*)

lemma *on-exit-unit-def:* (*on-exit f cleanup::(unit, unit, 's) exn-monad*) =
bind-handle f
 ($\lambda v.$ *bind (state-select cleanup) ($\lambda -.$ return ())*)
 ($\lambda e.$ *bind (state-select cleanup) ($\lambda -.$ throw ())*)
unfolding *on-exit-def on-exit'-def bind-handle-eq*
by (*intro arg-cong2[where f=bind-exception-or-result]*)
 (*auto simp:* *fun-eq-iff default-option-def Exn-def*
split: *exception-or-result-split*)

lemma *on-exit-catch-conv:* *on-exit f cleanup* =
 do {
 r \leftarrow (*f <catch> ($\lambda e.$ state-select cleanup >>= ($\lambda -.$ throw e))*);
 state-select cleanup;
 return r
 }
unfolding *on-exit-def on-exit'-def*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add:* *runs-to-iff fun-eq-iff Exn-def default-option-def*
intro!: *arg-cong[where f=runs-to - -]*)
by (*metis default-option-def exception-or-result-cases not-None-eq*)

lemma *L2corres-on-exit':*
assumes *m-c:* *L1corres ct Γ m c*
shows *L1corres ct Γ (on-exit m {(s,t). t = cleanup s}) (On-Exit c (Basic cleanup))*
unfolding *on-exit-catch-conv*
apply (*simp add:* *On-Exit-def*)

```

apply (rule L1corres-seq [simplified L1-seq-def])
apply (simp add: L1-catch-def [symmetric])
apply (rule L1corres-catch [OF m-c])
apply (rule L1-corres-cleanup-throw [simplified])
apply (rule L1-corres-cleanup [simplified])
done

```

lemma *L2corres-on-exit*:

```

assumes m-c: L1corres ct  $\Gamma$  m c
shows L1corres ct  $\Gamma$  (on-exit m cleanup) (On-Exit c (com.Spec cleanup))
apply (simp add: on-exit-catch-conv On-Exit-def)
apply (rule L1corres-seq [simplified L1-seq-def])
apply (simp add: L1-catch-def [symmetric])
apply (rule L1corres-catch [OF m-c])
apply (rule L1-corres-spec-cleanup-throw[simplified])
apply (rule L1-corres-spec-cleanup[simplified])
done

```

definition

```

refines-simpl :: bool  $\Rightarrow$  ('p  $\Rightarrow$  ('s, 'p, strictc-errortype) com option)  $\Rightarrow$ 
  ('s, 'p, strictc-errortype) com  $\Rightarrow$ 
  (('e::default, 'a, 't) spec-monad)  $\Rightarrow$ 
  ('s  $\Rightarrow$  't  $\Rightarrow$  (('s, strictc-errortype) xstate  $\Rightarrow$  (('e, 'a) exception-or-result * 't)  $\Rightarrow$ 
bool)  $\Rightarrow$  bool where
refines-simpl ct  $\Gamma$  c m s t R  $\equiv$ 
  succeeds m t  $\longrightarrow$ 
  (( $\forall s'$ .  $\Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow s' \longrightarrow$ 
    ( $s' \in \{\text{Fault AssumeError}, \text{Fault StackOverflow}\} \vee$ 
    ( $\exists r t'. \text{reaches } m \ t \ r \ t' \wedge R \ s' \ (r, t')$ )))  $\wedge$ 
    (ct  $\longrightarrow$   $\Gamma \vdash c \downarrow \text{Normal } s$ ))

```

lemma *refines-simplI*:

```

assumes termi: succeeds m t  $\Longrightarrow$  ct  $\Longrightarrow$   $\Gamma \vdash c \downarrow \text{Normal } s$ 
assumes sim:  $\bigwedge s'. \text{succeeds } m \ t \Longrightarrow \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow s' \Longrightarrow s' \notin \{\text{Fault}$ 
AssumeError}, \text{Fault StackOverflow}\}
   $\Longrightarrow \exists r t'. \text{reaches } m \ t \ r \ t' \wedge R \ s' \ (r, t')$ 
shows refines-simpl ct  $\Gamma$  c m s t R
using termi sim unfolding refines-simpl-def
by blast

```

definition

```

rel-L1 :: ('s, strictc-errortype) xstate  $\Rightarrow$  ('e, 'a) xval  $\times$  's  $\Rightarrow$  bool where
rel-L1  $\equiv \lambda s \ (r, t). \text{(case } s \text{ of}$ 
  Normal s'  $\Rightarrow (\exists x. r = \text{Result } x) \wedge t = s'$ 
  | Abrupt s'  $\Rightarrow (\exists x. r = \text{Exn } x) \wedge t = s'$ 
  | Fault e  $\Rightarrow \text{False}$ 
  | Stuck  $\Rightarrow \text{False}$ )

```

lemma *rel-L1-unit*:

rel-L1 = ($\lambda s (r, t).$ (case *s* of
 Normal *s'* $\Rightarrow r = \text{Result } () \wedge t = s'$
 | *Abrupt* *s'* $\Rightarrow r = \text{Exn } () \wedge t = s'$
 | *Fault* *e* $\Rightarrow \text{False}$
 | *Stuck* $\Rightarrow \text{False}$))
by (*auto simp add: rel-L1-def split: xstate.splits*)

lemma *rel-L1-conv* [*simp*]:

rel-L1 (*Normal* *s*) (*r*, *t*) = ($\exists x. r = \text{Result } x \wedge t = s$)
rel-L1 (*Abrupt* *s*) (*r*, *t*) = ($\exists x. r = \text{Exn } x \wedge t = s$)
rel-L1 (*Fault* *e*) *x* = *False*
rel-L1 *Stuck* *x* = *False*
by (*auto simp add: rel-L1-def*)

lemma *refines-simpl-rel-L1I*:

assumes *termi*: *succeeds* *m* *t* $\Longrightarrow ct \Longrightarrow \Gamma \vdash c \downarrow \text{Normal } s$
assumes *sim-Normal*: $\bigwedge s'. \text{succeeds } m \ t \Longrightarrow \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \text{Normal } s'$
 $\Longrightarrow \exists r. \text{reaches } m \ t \ (\text{Result } r) \ s'$
assumes *sim-Abrupt*: $\bigwedge s'. \text{succeeds } m \ t \Longrightarrow \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \text{Abrupt } s'$
 $\Longrightarrow \exists r. \text{reaches } m \ t \ (\text{Exn } r) \ s'$
assumes *sim-Fault*: $\bigwedge e. \text{succeeds } m \ t \Longrightarrow \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \text{Fault } e$
 $\Longrightarrow e \in \{\text{AssumeError}, \text{StackOverflow}\}$
assumes *sim-Stuck*: *succeeds* *m* *t* $\Longrightarrow \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \text{Stuck}$
 $\Longrightarrow \text{False}$
shows *refines-simpl* *ct* $\Gamma \ c \ m \ s \ t \ \text{rel-L1}$
apply (*rule* *refines-simplI* [*OF termi*], *assumption*, *assumption*)
apply (*simp add: rel-L1-def*)
subgoal for *s'*
 using *sim-Normal sim-Abrupt sim-Fault sim-Stuck*
 apply (*cases s'*)
 apply (*fastforce split: prod.splits sum.splits*)
 done
done

lemma *L1corres-refines-simpl*:

L1corres *ct* $\Gamma \ m \ c \Longrightarrow \text{refines-simpl } ct \ \Gamma \ c \ m \ s \ s \ \text{rel-L1}$
apply (*clarsimp simp add: L1corres-def refines-simpl-def rel-L1-def split: xstate.splits prod.splits sum.splits*)
by (*smt (verit) Inl-Inr-False xstate.distinct(1) xstate.inject(1) xstate.inject(2)*)

lemma *refines-simpl-L1corres*:

assumes $\bigwedge s. \text{refines-simpl } ct \ \Gamma \ c \ m \ s \ s \ \text{rel-L1}$
shows *L1corres* *ct* $\Gamma \ m \ c$
using *assms*
apply (*force simp add: L1corres-def refines-simpl-def rel-L1-def split: xstate.splits prod.splits sum.splits*)
done

theorem *L1corres-refines-simpl-conv*:

*L1corres ct Γ m c \longleftrightarrow ($\forall s$. *refines-simpl ct Γ c m s s rel-L1*)*

using *L1corres-refines-simpl refines-simpl-L1corres*

by *metis*

lemma *refines-simpl-DynCom*:

refines-simpl ct Γ (c s) m s t R \implies refines-simpl ct Γ (DynCom c) m s t R

by (*auto simp add: refines-simpl-def terminates.DynCom elim: exec-Normal-elim-cases(12)*)

lemma *refines-simpl-StackOverflow*:

assumes *c: s \in g \implies refines-simpl ct Γ c m s t R*

shows *refines-simpl ct Γ (Guard StackOverflow g c) m s t R*

using *c*

by (*auto simp add: refines-simpl-def elim: exec-Normal-elim-cases intro: terminates.intros*)

lemma *refines-simpl-rel-L1-bind*:

fixes *m1:: ('e, 'a, 's) exn-monad*

fixes *m2:: 'a \Rightarrow ('e, 'b, 's) exn-monad*

assumes *c1: refines-simpl ct Γ c1 m1 s s rel-L1*

assumes *c2: $\bigwedge r$ s'. *succeeds m1 s \implies $\Gamma \vdash \langle c1, Normal s \rangle \Rightarrow Normal s' \implies$ reaches m1 s (Result r) s' \implies**

refines-simpl ct Γ c2 (m2 r) s' s' rel-L1

shows *refines-simpl ct Γ (c1;;c2) (m1 >>= m2) s s rel-L1*

proof (*rule refines-simpl-rel-L1I*)

assume *nofail: succeeds (m1 >>= m2) s*

assume *ct: ct*

from *nofail have nofail-m1: succeeds m1 s*

by (*simp add: succeeds-bind*)

with *ct c1 have term-c1: $\Gamma \vdash c1 \downarrow Normal s$*

by (*simp add: refines-simpl-def*)

then

show *$\Gamma \vdash c1;;c2 \downarrow Normal s$*

proof (*rule terminates.intros, intro allI impI*)

fix *s'*

assume *exec-c1: $\Gamma \vdash \langle c1, Normal s \rangle \Rightarrow s'$*

show *$\Gamma \vdash c2 \downarrow s'$*

proof (*cases s'*)

case (*Normal s1*)

with *exec-c1 c1 nofail-m1 term-c1 ct obtain r where sim1: reaches m1 s (Result r) s1*

by (*force simp add: rel-L1-def refines-simpl-def*)

from *c2 [OF nofail-m1 exec-c1 [simplified Normal] this]*

have *refines-simpl ct Γ c2 (m2 r) s1 s1 rel-L1 .*

with *ct Normal nofail sim1*

show *?thesis*

by (*simp add: refines-simpl-def reaches-bind succeeds-bind*)


```

    qed simp-all
  qed
next
fix s'
assume nofail: succeeds (m1 >>= m2) s
from nofail have nofail-m1: succeeds m1 s
  by (simp add: succeeds-bind)
assume exec:  $\Gamma \vdash \langle c1;;c2, Normal\ s \rangle \Rightarrow Normal\ s'$ 
then show  $\exists r. reaches\ (m1\ >>= m2)\ s\ (Result\ r)\ s'$ 
proof (cases)
  case (1 s1)
  hence exec-c1:  $\Gamma \vdash \langle c1, Normal\ s \rangle \Rightarrow s1$  and
    exec-c2:  $\Gamma \vdash \langle c2, s1 \rangle \Rightarrow Normal\ s'$  .
  from exec-c2 obtain s1' where s1':  $s1 = Normal\ s1'$ 
  by (meson Normal-resultE)
  from exec-c1 c1 nofail-m1 obtain r1 where sim1: reaches m1 s (Result r1)
s1'
  by (metis (no-types, lifting) empty-iff insertE rel-L1-conv(1) s1' refines-simpl-def
xstate.simps(7))

  from c2 [OF nofail-m1 exec-c1[simplified s1'  $\uparrow$  sim1]]
  have refines-simpl ct  $\Gamma\ c2\ (m2\ r1)\ s1'\ s1'\ rel-L1$  .
  with nofail exec-c2 obtain r2 where reaches (m2 r1) s1' (Result r2) s'
  by (smt (verit) empty-iff insertE rel-L1-conv(1) s1' sim1 refines-simpl-def
succeeds-bind
reaches-bind xstate.simps(7))
  with sim1 nofail show ?thesis
  by (fastforce simp add: reaches-bind succeeds-bind)
qed
next
fix s'
assume nofail: succeeds (m1 >>= m2) s
from nofail have nofail-m1: succeeds m1 s
  by (simp add: succeeds-bind)
assume exec:  $\Gamma \vdash \langle c1;;c2, Normal\ s \rangle \Rightarrow Abrupt\ s'$ 
then show  $\exists r. reaches\ (m1\ >>= m2)\ s\ (Exn\ r)\ s'$ 
proof (cases)
  case (1 s1)
  hence exec-c1:  $\Gamma \vdash \langle c1, Normal\ s \rangle \Rightarrow s1$  and
    exec-c2:  $\Gamma \vdash \langle c2, s1 \rangle \Rightarrow Abrupt\ s'$  .

  show ?thesis
proof (cases s1)
  case (Normal s1')
  with exec-c1 c1 nofail-m1 obtain r1 where sim1: reaches m1 s (Result r1)
s1'
  by (metis (no-types, lifting) empty-iff insertE rel-L1-conv(1) refines-simpl-def
xstate.simps(7))

```

```

from  $c2$  [OF nofail-m1 exec-c1[simplified Normal]  $sim1$ ]
have refines-simpl  $ct \Gamma c2 (m2 r1) s1' s1' rel-L1$  .
with nofail exec-c2 sim1 obtain  $r2$  where reaches  $(m2 r1) s1' (Exn r2) s'$ 
  by (smt (verit) Normal empty-iff insertE rel-L1-conv(2) refines-simpl-def
succeeds-bind xstate.simps(11))
  with sim1 nofail show ?thesis
  by (fastforce simp add: reaches-bind succeeds-bind)
next
  case (Abrupt s1')
  with exec-c1 c1 nofail-m1 obtain  $r1$  where sim1: reaches  $m1 s (Exn r1)$ 
s1'
  by (metis (no-types, lifting) empty-iff insertE rel-L1-conv(2) refines-simpl-def
xstate.distinct(7))
  from Abrupt exec-c2 have  $s' = s1'$ 
  by (meson Abrupt-end xstate.inject(2))
  with nofail
  show ?thesis
  apply (clarsimp simp add: reaches-bind succeeds-bind Exn-def)
  using sim1 by fastforce
next
  case (Fault x)
  with exec-c2 show ?thesis
  by (meson Fault-end xstate.distinct(7))
next
  case Stuck
  with exec-c2 show ?thesis
  using noStuck-startD' by blast
qed
qed
next
fix  $e$ 
assume nofail: succeeds  $(m1 >>= m2) s$ 
from nofail have nofail-m1: succeeds  $m1 s$ 
  by (simp add: succeeds-bind)
assume exec:  $\Gamma \vdash \langle c1;;c2, Normal s \rangle \Rightarrow Fault e$ 
then show  $e \in \{AssumeError, StackOverflow\}$ 
proof (cases)
  case ( $1 s1$ )
  hence exec-c1:  $\Gamma \vdash \langle c1, Normal s \rangle \Rightarrow s1$  and
    exec-c2:  $\Gamma \vdash \langle c2, s1 \rangle \Rightarrow Fault e$  .
show ?thesis
proof (cases s1)
  case (Normal s1')
  with exec-c1 c1 nofail-m1 obtain  $r1$  where sim1: reaches  $m1 s (Result r1)$ 
s1'
  by (metis (no-types, lifting) empty-iff insertE rel-L1-conv(1) refines-simpl-def
xstate.simps(7))
  from  $c2$  [OF nofail-m1 exec-c1[simplified Normal]  $sim1$ ]
  have refines-simpl  $ct \Gamma c2 (m2 r1) s1' s1' rel-L1$  .

```

```

    with nofail exec-c2 sim1 show e ∈ {AssumeError, StackOverflow}
    by (metis (no-types, lifting) Normal insert-iff rel-L1-conv(3) refines-simpl-def
singleton-iff
      succeeds-bind xstate.inject(3))
  next
  case (Abrupt s1')
  with exec-c2 show ?thesis
  by (metis Abrupt-end xstate.distinct(7))
  next
  case (Fault x)
  with exec-c2 c1 exec-c1 show ?thesis
  by (metis exec-Normal-elim-cases(1) insert-iff nofail-m1 rel-L1-conv(3)
refines-simpl-def singleton-iff xstate.inject(3))
  next
  case Stuck
  with exec-c2 show ?thesis
  using noStuck-startD' by blast
qed
qed
next
assume nofail: succeeds (m1 >>= m2) s
from nofail have nofail-m1: succeeds m1 s
  by (simp add: succeeds-bind)
assume exec:  $\Gamma \vdash \langle c1;;c2, Normal \ s \rangle \Rightarrow Stuck$ 
then show False
proof (cases)
  case (1 s1)
  hence exec-c1:  $\Gamma \vdash \langle c1, Normal \ s \rangle \Rightarrow s1$  and
    exec-c2:  $\Gamma \vdash \langle c2, s1 \rangle \Rightarrow Stuck$  .
  show ?thesis
proof (cases s1)
  case (Normal s1')
  with exec-c1 c1 nofail-m1 obtain r1 where sim1: reaches m1 s (Result r1)
s1'
  by (metis (no-types, lifting) empty-iff insertE rel-L1-conv(1) refines-simpl-def
xstate.simps(7))
  from c2 [OF nofail-m1 exec-c1[simplified Normal] sim1]
  have refines-simpl ct  $\Gamma \ c2 \ (m2 \ r1) \ s1' \ s1' \ rel-L1$  .
  with nofail exec-c2 sim1 show False
  by (metis Normal insertE rel-L1-conv(4) refines-simpl-def singletonD suc-
ceeds-bind xstate.simps(15))
  next
  case (Abrupt s1')
  with exec-c2 show ?thesis
  by (metis Abrupt-end xstate.distinct(10))
  next
  case (Fault x)
  with exec-c2 show ?thesis

```

```

    by (metis Fault-end isFault-simps(4) not-isFault-iff)
  next
  case Stuck
  with exec-c2 c1 exec-c1 show ?thesis
    by (metis insert-iff nofail-m1 rel-L1-conv(4) refines-simpl-def singletonD
xstate.simps(15))
  qed
  qed
  qed

```

lemma *refines-simpl-rel-L1-catch*:

```

  assumes L: refines-simpl ct  $\Gamma$  L' L s s rel-L1
  assumes R:  $\bigwedge s$ . refines-simpl ct  $\Gamma$  R' R s s rel-L1
  shows refines-simpl ct  $\Gamma$  (Catch L' R') (L1-catch L R) s s rel-L1
  proof (rule refines-simpl-rel-L1I)
    assume nofault: succeeds (L1-catch L R) s
    assume ct: ct
    show  $\Gamma \vdash \text{TRY } L' \text{ CATCH } R' \text{ END} \downarrow \text{Normal } s$ 
    proof (rule terminates.intros, safe)
      show  $\Gamma \vdash L' \downarrow \text{Normal } s$ 
      using nofault ct L
      by (simp add: refines-simpl-def L1-catch-def rel-L1-def succeeds-catch)
    next
    fix s'
    assume  $\Gamma \vdash \langle L', \text{Normal } s \rangle \Rightarrow \text{Abrupt } s'$ 
    then show  $\Gamma \vdash R' \downarrow \text{Normal } s'$ 
    using nofault ct L R
    by (fastforce simp add: refines-simpl-def L1-catch-def rel-L1-def succeeds-catch)
  qed
  next
  fix s'
  assume nofault: succeeds (L1-catch L R) s
  assume exec:  $\Gamma \vdash \langle \text{TRY } L' \text{ CATCH } R' \text{ END}, \text{Normal } s \rangle \Rightarrow \text{Normal } s'$ 
  then show  $\exists r$ . reaches (L1-catch L R) s (Result r) s'
  proof (cases)
    case (1 s1)
    hence exec-L':  $\Gamma \vdash \langle L', \text{Normal } s \rangle \Rightarrow \text{Abrupt } s1$  and
      exec-R':  $\Gamma \vdash \langle R', \text{Normal } s1 \rangle \Rightarrow \text{Normal } s'$ .
    from nofault L R exec-L' obtain
      nofault-L: succeeds L s and
      nofault-R: succeeds R s1
    by (smt (verit, best) L1-defs(5) case-xval-simps(1) insertE refines-simpl-def
rel-L1-conv(2) singletonD succeeds-catch xstate.simps(11))
    from nofault-L exec-L' L obtain r1 where r1: reaches L s (Exn r1) s1
    by (metis (no-types, lifting) insertE rel-L1-conv(2) refines-simpl-def
singletonD xstate.distinct(7))
    from r1 exec-R' R obtain r2 where reaches R s1 (Result r2) s'
    by (metis (no-types, lifting) insertE nofault-R rel-L1-conv(1))
  qed

```

```

      refines-simpl-def singletonD xstate.simps(7))
with r1 nofault show ?thesis
  by (fastforce simp add: reaches-catch succeeds-catch L1-defs )
next
case 2
have exec-L':  $\Gamma \vdash \langle L', Normal\ s \rangle \Rightarrow Normal\ s'$  by fact
from nofault L R exec-L' obtain
  nofault-L: succeeds L s
  by (simp add: L1-defs(5) succeeds-catch)
with L exec-L' obtain r1 where reaches L s (Result r1) s'
by (metis (no-types, lifting) insertE rel-L1-conv(1) refines-simpl-def singletonD
xstate.distinct(3))
then show ?thesis using nofault
  by (fastforce simp add: L1-catch-def succeeds-catch reaches-catch)
qed
next
fix s'
assume nofault: succeeds (L1-catch L R) s
assume exec:  $\Gamma \vdash \langle TRY\ L'\ CATCH\ R'\ END, Normal\ s \rangle \Rightarrow Abrupt\ s'$ 
then show  $\exists r. reaches\ (L1-catch\ L\ R)\ s\ (Exn\ r)\ s'$ 
proof (cases)
  case (1 s1)
  hence exec-L':  $\Gamma \vdash \langle L', Normal\ s \rangle \Rightarrow Abrupt\ s1$  and
    exec-R':  $\Gamma \vdash \langle R', Normal\ s1 \rangle \Rightarrow Abrupt\ s'$  .
  from nofault L R exec-L' obtain
    nofault-L: succeeds L s and
    nofault-R: succeeds R s1
  by (fastforce simp add: L1-catch-def refines-simpl-def reaches-catch suc-
ceeds-catch)

  from nofault-L exec-L' L obtain r1 where r1: reaches L s (Exn r1) s1
  by (metis (no-types, lifting) insertE rel-L1-conv(2) refines-simpl-def
singletonD xstate.distinct(7))
  from nofault-R r1 exec-R' R obtain r2 where reaches R s1 (Exn r2) s'
  by (metis (no-types, lifting) empty-iff insertE rel-L1-conv(2) refines-simpl-def
xstate.distinct(7))
  with r1 nofault show ?thesis
  by (fastforce simp add: L1-catch-def succeeds-catch reaches-catch)
next
case 2
then show ?thesis by simp
qed
next
fix e
assume nofault: succeeds (L1-catch L R) s
assume exec:  $\Gamma \vdash \langle TRY\ L'\ CATCH\ R'\ END, Normal\ s \rangle \Rightarrow Fault\ e$ 
then show  $e \in \{AssumeError, StackOverflow\}$ 
proof (cases)
  case (1 s1)

```

hence $exec-L': \Gamma \vdash \langle L', Normal\ s \rangle \Rightarrow Abrupt\ s1$ **and**
 $exec-R': \Gamma \vdash \langle R', Normal\ s1 \rangle \Rightarrow Fault\ e$.
from $nofault\ L\ R\ exec-L'$ **obtain**
 $nofault-L: succeeds\ L\ s$ **and**
 $nofault-R: succeeds\ R\ s1$
by (*fastforce simp add: L1-catch-def refines-simpl-def reaches-catch succeeds-catch*)
from $nofault-L\ exec-L'\ L$ **obtain** $r1$ **where** $r1: reaches\ L\ s\ (Exn\ r1)\ s1$
by (*metis (no-types, lifting) insertE rel-L1-conv(2) refines-simpl-def singletonD xstate.distinct(7)*)
from $nofault-R\ r1\ exec-R'\ R$ **show** *?thesis*
by (*metis insert-iff rel-L1-conv(3) refines-simpl-def singletonD xstate.inject(3)*)
next
case 2
have $exec-L': \Gamma \vdash \langle L', Normal\ s \rangle \Rightarrow Fault\ e$ **by fact**
from $nofault\ L\ R\ exec-L'$ **obtain**
 $nofault-L: succeeds\ L\ s$
by (*simp add: L1-defs(5) succeeds-catch*)
with $L\ exec-L'$ **show** *?thesis*
by (*metis insertCI insertE rel-L1-conv(3) refines-simpl-def singleton-iff xstate.inject(3)*)
qed
next
assume $nofault: succeeds\ (L1-catch\ L\ R)\ s$
assume $exec: \Gamma \vdash \langle TRY\ L'\ CATCH\ R'\ END, Normal\ s \rangle \Rightarrow Stuck$
then show *False*
proof (*cases*)
case (1 $s1$)
hence $exec-L': \Gamma \vdash \langle L', Normal\ s \rangle \Rightarrow Abrupt\ s1$ **and**
 $exec-R': \Gamma \vdash \langle R', Normal\ s1 \rangle \Rightarrow Stuck$.
from $nofault\ L\ R\ exec-L'$ **obtain**
 $nofault-L: succeeds\ L\ s$ **and**
 $nofault-R: succeeds\ R\ s1$
by (*fastforce simp add: L1-catch-def refines-simpl-def reaches-catch succeeds-catch*)

from $nofault-R\ exec-R'\ R$ **show** *?thesis*
by (*metis empty-iff insertE rel-L1-conv(4) refines-simpl-def xstate.distinct(11)*)
next
case 2
have $exec-L': \Gamma \vdash \langle L', Normal\ s \rangle \Rightarrow Stuck$ **by fact**
from $nofault\ L\ R\ exec-L'$ **obtain**
 $nofault-L: succeeds\ L\ s$
by (*simp add: L1-defs(5) succeeds-catch*)
with $L\ exec-L'$ **show** *?thesis*
by (*metis insertE rel-L1-conv(4) refines-simpl-def singletonD xstate.distinct(11)*)
qed
qed

lemmas *refines-simpl-cleanup* = *L1corres-refines-simpl* [*OF L1-corres-cleanup*]
lemmas *refines-simpl-cleanup-throw* = *L1corres-refines-simpl* [*OF L1-corres-cleanup-throw*]
lemmas *refines-simpl-spec-cleanup* = *L1corres-refines-simpl* [*OF L1-corres-spec-cleanup*]
lemmas *refines-simpl-spec-cleanup-throw* = *L1corres-refines-simpl* [*OF L1-corres-spec-cleanup-throw*]

lemma *refines-simpl-rel-L1-on-exit'*:
fixes *m*:: 's *L1-monad*
assumes *m-c*: *refines-simpl ct* Γ *c m s s rel-L1*
shows *refines-simpl ct* Γ (*On-Exit c* (*Basic cleanup*)) (*on-exit m* $\{(s,t). t =$
*cleanup s\}) *s s rel-L1*
unfolding *on-exit-catch-conv*
apply (*simp add*: *On-Exit-def*)
apply (*rule refines-simpl-rel-L1-bind*)
apply (*simp add*: *L1-catch-def* [*symmetric*])
apply (*rule refines-simpl-rel-L1-catch* [*OF m-c*])
apply (*rule refines-simpl-cleanup-throw* [*simplified*])
apply (*rule refines-simpl-cleanup* [*simplified*])
done*

lemma *refines-simpl-rel-L1-on-exit*:
fixes *m*:: 's *L1-monad*
assumes *m-c*: *refines-simpl ct* Γ *c m s s rel-L1*
shows *refines-simpl ct* Γ (*On-Exit c* (*com.Spec cleanup*)) (*on-exit m cleanup*) *s s*
rel-L1
apply (*simp add*: *on-exit-catch-conv On-Exit-def*)
apply (*rule refines-simpl-rel-L1-bind*)
apply (*simp add*: *L1-catch-def* [*symmetric*])
apply (*rule refines-simpl-rel-L1-catch* [*OF m-c*])
apply (*rule refines-simpl-spec-cleanup-throw* [*simplified*])
apply (*rule refines-simpl-spec-cleanup* [*simplified*])
done

named-theorems *L1corres-with-fresh-stack-ptr*

context *stack-heap-state*

begin

lemma *refines-simpl-rel-L1-with-fresh-stack-ptr*:
fixes *m*:: 'a::mem-type *ptr* \Rightarrow 's *L1-monad*
assumes *c-m*: $\bigwedge p s. \text{refines-simpl ct } \Gamma (c p) (m p) s s \text{ rel-L1}$
shows *refines-simpl ct* Γ (*With-Fresh-Stack-Ptr n I c*) (*with-fresh-stack-ptr n I*
m) *s s rel-L1*
apply (*simp add*: *with-fresh-stack-ptr-def With-Fresh-Stack-Ptr-def*)
apply (*rule refines-simpl-StackOverflow*)
apply (*rule refines-simpl-DynCom*)
apply (*rule refines-simpl-rel-L1-bind*)

```

subgoal
  apply (rule refines-simpl-rel-L1I)
subgoal
  by (simp add: terminates.Spec)
subgoal for s'
  apply (erule exec-Normal-elim-cases)
  subgoal for t
    by (auto simp add: succeeds-assume-result-and-state)
  by auto
subgoal for s'
  by (meson exec-Normal-elim-cases(7) xstate.distinct(9) xstate.simps(5))
subgoal for e
  by (meson exec-Normal-elim-cases(7) xstate.distinct(11) xstate.simps(7))
subgoal
  apply (erule exec-Normal-elim-cases)
  using Ex-list-of-length by auto blast
done
apply (rule refines-simpl-DynCom)
apply (clarsimp)
apply (simp add: stack-allocs-allocated-ptrs)
apply (rule refines-simpl-rel-L1-on-exit[OF c-m])
done

```

```

lemma L1corres-with-fresh-stack-ptr[L1corres-with-fresh-stack-ptr]:
  fixes m:: 'a::mem-type ptr ⇒ 's L1-monad
  assumes c-m: ∧p. L1corres ct Γ (m p) (c p)
  shows L1corres ct Γ (with-fresh-stack-ptr n I m) (With-Fresh-Stack-Ptr n I c)
  apply (rule refines-simpl-L1corres)
  apply (rule refines-simpl-rel-L1-with-fresh-stack-ptr)
  apply (rule L1corres-refines-simpl)
  apply (rule c-m)
done
end

```

definition *UNDEFINED-FUNCTION* \equiv *False*

definition

undefined-function-body $::$ (*'a, int, strict-error-type*) *com*

where

undefined-function-body \equiv *Guard UndefinedFunction {x. UNDEFINED-FUNCTION}*
SKIP

definition

init-return-undefined-function-body::((*'a* ⇒ *'a*) ⇒ ((*'g*, *'l*, *'e*, *'z*) *state-scheme* ⇒ (*'g*, *'l*, *'e*, *'z*) *state-scheme*))

⇒ ((*'g*, *'l*, *'e*, *'z*) *state-scheme*, *int*, *strictc-errortype*) *com*

where

init-return-undefined-function-body *ret* ≡ *Seq* (*lvar-nondet-init* *ret*) (*Guard UndefinedFunction* {*x*. *UNDEFINED-FUNCTION*} *SKIP*)

lemma *L1corres-undefined-call*:

L1corres *ct* Γ ((*L1-seq* (*L1-guard* (*L1-set-to-pred* {*x*. *UNDEFINED-FUNCTION*}))) *L1-skip*)) (*Call* *X'*)

by (*clarsimp simp*: *L1corres-def* *L1-defs* *UNDEFINED-FUNCTION-def*)

lemma *L1-UNDEFINED-FUNCTION-fail*: (*L1-guard* (*L1-set-to-pred* {*x*. *UNDEFINED-FUNCTION*})) = *L1-fail*

apply (*clarsimp simp add*: *L1-defs* *UNDEFINED-FUNCTION-def*)

by (*simp add*: *run-guard spec-monad-ext*)

lemma *L1-seq-fail*: *L1-seq* *L1-fail* *X* = *L1-fail*

apply (*clarsimp simp add*: *L1-defs*)

done

lemma *L1-seq-init-fail*: (*L1-seq* (*L1-init* *ret*) *L1-fail*) = *L1-fail*

apply (*clarsimp simp add*: *L1-defs*)

apply (*rule spec-monad-eqI*)

apply (*auto simp add*: *runs-to-iff*)

done

lemma *L1-corres-L1-fail*: *L1corres* *ct* Γ *L1-fail* *X*

by (*simp add*: *L1corres-def* *L1-defs*)

lemma *L1corres-init-return-undefined-call*:

L1corres *ct* Γ (*L1-seq* (*L1-init* *ret*) ((*L1-seq* (*L1-guard* (*L1-set-to-pred* {*x*. *UNDEFINED-FUNCTION*}))) *L1-skip*))) (*Call* *X'*)

by (*simp only*: *L1-UNDEFINED-FUNCTION-fail* *L1-seq-fail* *L1-seq-init-fail* *L1-corres-L1-fail*)

named-theorems *L1unfold*

named-theorems *L1except*

lemma *signed-bounds-one-to-nat*: $n < s \ 1 \implies 0 \leq s \ n \implies \text{unat } n = 0$

by (*metis signed.leD unat-1-0 unat-gt-0 unsigned-eq-0-iff word-msb-0 word-sle-msb-le*)

lemma *signed-bounds-to-nat-boundsF*: $n < s \ \text{numeral } B \implies 0 \leq s \ n \implies \text{unat } n < \text{numeral } B$

by (*metis linorder-not-less of-nat-numeral signed.leD unat-less-helper word-msb-0 word-sle-msb-le*)

lemma *word-bounds-to-nat-boundsF*: $(n::'a::\text{len word}) < \text{numeral } B \implies 0 \leq_s n \implies \text{unat } n < \text{numeral } B$
by (*simp add: unat-less-helper*)

lemma *word-bounds-one-to-nat*: $(n::'a::\text{len word}) < 1 \implies 0 \leq_s n \implies \text{unat } n = 0$
by (*simp add: unat-less-helper*)

lemma *monotone-L1-seq-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone* $R (\leq) X$
assumes *mono-Y*: *monotone* $R (\leq) Y$
shows *monotone* $R (\leq)$
 $(\lambda f. (L1\text{-seq } (X f) (Y f)))$
unfolding *L1-defs*
apply (*intro partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-L1-seq-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone* $R (\geq) X$
assumes *mono-Y*: *monotone* $R (\geq) Y$
shows *monotone* $R (\geq)$
 $(\lambda f. (L1\text{-seq } (X f) (Y f)))$
unfolding *L1-defs*
apply (*intro partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-L1-catch-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone* $R (\leq) X$
assumes *mono-Y*: *monotone* $R (\leq) Y$
shows *monotone* $R (\leq)$
 $(\lambda f. (L1\text{-catch } (X f) (Y f)))$
unfolding *L1-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-L1-catch-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone* $R (\geq) X$
assumes *mono-Y*: *monotone* $R (\geq) Y$
shows *monotone* $R (\geq)$
 $(\lambda f. (L1\text{-catch } (X f) (Y f)))$
unfolding *L1-defs*
apply (*rule partial-function-mono*)

```

apply (rule mono-X)
apply (rule mono-Y)
done

```

```

lemma monotone-L1-condition-le [partial-function-mono]:
assumes mono-X: monotone  $R (\leq) X$ 
assumes mono-Y: monotone  $R (\leq) Y$ 
shows monotone  $R (\leq)$ 
  ( $\lambda f. (L1-condition\ C\ (X\ f)\ (Y\ f))$ )
unfolding L1-defs
apply (rule partial-function-mono)
apply (rule mono-X)
apply (rule mono-Y)
done

```

```

lemma monotone-L1-condition-ge [partial-function-mono]:
assumes mono-X: monotone  $R (\geq) X$ 
assumes mono-Y: monotone  $R (\geq) Y$ 
shows monotone  $R (\geq)$ 
  ( $\lambda f. (L1-condition\ C\ (X\ f)\ (Y\ f))$ )
unfolding L1-defs
apply (rule partial-function-mono)
apply (rule mono-X)
apply (rule mono-Y)
done

```

```

lemma monotone-L1-guarded-le [partial-function-mono]:
assumes mono-X [partial-function-mono]: monotone  $R (\leq) X$ 
shows monotone  $R (\leq)$ 
  ( $\lambda f. (L1-guarded\ C\ (X\ f))$ )
unfolding L1-guarded-def
apply (rule partial-function-mono)+
done

```

```

lemma monotone-L1-guarded-ge [partial-function-mono]:
assumes mono-X [partial-function-mono]: monotone  $R (\geq) X$ 
shows monotone  $R (\geq)$ 
  ( $\lambda f. (L1-guarded\ C\ (X\ f))$ )
unfolding L1-guarded-def
apply (rule partial-function-mono)+
done

```

```

lemma monotone-L1-while-le [partial-function-mono]:
assumes mono-B: monotone  $R (\leq) (\lambda f. B\ f)$ 
shows monotone  $R (\leq) (\lambda f. L1-while\ C\ (B\ f))$ 
unfolding L1-defs

```

```

apply (rule partial-function-mono)
apply (rule mono-B)
done

```

```

lemma monotone-L1-while-ge [partial-function-mono]:
assumes mono-B: monotone  $R (\geq)$  ( $\lambda f. B f$ )
shows monotone  $R (\geq)$  ( $\lambda f. L1\text{-while } C (B f)$ )
unfolding L1-defs
apply (rule partial-function-mono)
apply (rule mono-B)
done

```

```

lemma monotone-L1-call-le [partial-function-mono]:
assumes X[partial-function-mono]: monotone  $R (\leq)$   $X$ 
shows monotone  $R (\leq)$ 
  ( $\lambda f. L1\text{-call scope-setup } (X f) \text{ scope-teardown result-exn return-xf}$ )
unfolding L1-call-def
apply (rule partial-function-mono)+
done

```

```

lemma monotone-L1-call-ge [partial-function-mono]:
assumes X[partial-function-mono]: monotone  $R (\geq)$   $X$ 
shows monotone  $R (\geq)$ 
  ( $\lambda f. L1\text{-call scope-setup } (X f) \text{ scope-teardown result-exn return-xf}$ )
unfolding L1-call-def
apply (rule partial-function-mono)+
done

```

end

18.1 Peep-hole L1 optimisations

```

theory L1Peephole
imports L1Defs
begin

```

```

named-theorems L1opt

```

```

lemma L1-seq-assoc [L1opt]:  $(L1\text{-seq } (L1\text{-seq } X Y) Z) = (L1\text{-seq } X (L1\text{-seq } Y Z))$ 
apply (clarsimp simp: L1-seq-def bind-assoc)
done

```

```

lemma L1-seq-skip [L1opt]:
   $L1\text{-seq } A L1\text{-skip} = A$ 
   $L1\text{-seq } L1\text{-skip } A = A$ 
apply (clarsimp simp: L1-seq-def L1-skip-def)+
done

```

lemma *L1-condition-true* [L1opt]: *L1-condition* ($\lambda\cdot$. True) $A B = A$
apply (*clarsimp simp: L1-condition-def condition-def*)
done

lemma *L1-condition-false* [L1opt]: *L1-condition* ($\lambda\cdot$. False) $A B = B$
apply (*clarsimp simp: L1-condition-def condition-def*)
done

lemma *L1-condition-same* [L1opt]: *L1-condition* $C A A = A$
apply (*clarsimp simp: L1-defs condition-def spec-monad-ext run-bind*)
done

lemma *L1-fail-seq* [L1opt]: *L1-seq L1-fail* $X = L1-fail$
apply (*clarsimp simp: L1-seq-def L1-fail-def spec-monad-ext run-bind*)
done

lemma *L1-throw-seq* [L1opt]: *L1-seq L1-throw* $X = L1-throw$
apply (*clarsimp simp: L1-seq-def L1-throw-def*)
done

lemma *L1-fail-propagates* [L1opt]:
L1-seq L1-skip L1-fail = L1-fail
L1-seq (L1-modify M) L1-fail = L1-fail
L1-seq (L1-spec S) L1-fail = L1-fail
L1-seq (L1-guard G) L1-fail = L1-fail
L1-seq (L1-init I) L1-fail = L1-fail
L1-seq L1-fail L1-fail = L1-fail
apply (*unfold L1-defs*)
apply (*rule spec-monad-eqI; auto simp add: runs-to-iff*)
done

lemma *L1-condition-distrib*:
L1-seq (L1-condition C L R) X = L1-condition C (L1-seq L X) (L1-seq R X)
apply (*unfold L1-defs*)
apply (*rule spec-monad-eqI; auto simp add: runs-to-iff*)
done

lemmas *L1-fail-propagate-condition* [L1opt] = *L1-condition-distrib* [**where** $X=L1-fail$]

lemma *L1-fail-propagate-catch* [L1opt]:
(L1-seq (L1-catch L R) L1-fail) = (L1-catch (L1-seq L L1-fail) (L1-seq R L1-fail))
apply (*unfold L1-defs*)
apply (*rule spec-monad-eqI; auto simp add: runs-to-iff*)
apply (*fastforce simp add: runs-to-def-old Exn-def*)
done

lemma *L1-guard-false* [*L1opt*]:
L1-guard ($\lambda\cdot$. *False*) = *L1-fail*
unfolding *L1-defs*
by (*simp add: guard-False-fail*)

lemma *L1-guard-true* [*L1opt*]:
L1-guard ($\lambda\cdot$. *True*) = *L1-skip*
unfolding *L1-defs*
by (*simp add: spec-monad-ext run-guard*)

lemma *L1-condition-fail-lhs* [*L1opt*]:
L1-condition *C* *L1-fail* *A* = *L1-seq* (*L1-guard* (λs . \neg *C* *s*)) *A*
apply (*unfold L1-defs*)
apply (*rule spec-monad-eqI; auto simp add: runs-to-iff*)
done

lemma *L1-condition-fail-rhs* [*L1opt*]:
L1-condition *C* *A* *L1-fail* = *L1-seq* (*L1-guard* *C*) *A*
apply (*unfold L1-defs*)
apply (*rule spec-monad-eqI; auto simp add: runs-to-iff*)
done

lemma *L1-catch-fail* [*L1opt*]: *L1-catch* *L1-fail* *A* = *L1-fail*
unfolding *L1-defs*
apply (*rule spec-monad-eqI; auto simp add: runs-to-iff*)
done

lemma *L1-while-fail* [*L1opt*]: *L1-while* *C* *L1-fail* = *L1-guard* (λs . \neg *C* *s*)
unfolding *L1-defs*
apply (*subst whileLoop-unroll*)
apply (*rule spec-monad-ext*)
apply (*auto simp add: run-condition run-bind run-guard*)
done

lemma *whileLoop-succeeds-terminates-infinite*:
assumes *run* (*whileLoop* ($\lambda\cdot$. *C*) ($\lambda\cdot$. *skip*) ()) *s* \neq \top
shows *C* *s* \implies *False*
using *assms* **by** (*induct rule: whileLoop-ne-top-induct*) *auto*

lemma *run-whileLoop-infinite*: *run* (*whileLoop* ($\lambda\cdot$. *C*) ($\lambda\cdot$. *skip*) ()) *s* = *run* (*guard* (λs . \neg *C* *s*)) *s*
proof (*cases C s*)
case *True*
show *?thesis*
apply (*rule antisym*)
subgoal

```

    apply (subst whileLoop-unroll)
    apply (simp add: run-guard)
  done
subgoal
proof -
  have run (whileLoop (λ-. C) (λ-. skip) ()) s = ⊤
    using True whileLoop-succeeds-terminates-infinite[of C s] by auto
  hence ¬ succeeds (whileLoop (λ-. C) (λ-. skip) ()) s
    by (simp add: succeeds-def)
  then show ?thesis
    by (simp add: run-guard succeeds-def True top-post-state-def)
qed
done
next
case False
then show ?thesis apply (subst whileLoop-unroll) by (simp add: run-guard)
qed

lemma whileLoop-infinite: whileLoop (λ-. C) (λ-. skip) () = guard (λs. ¬ C s)
  apply (rule spec-monad-ext)
  apply (rule run-whileLoop-infinite)
  done

lemma L1-while-infinite [L1opt]: L1-while C L1-skip = L1-guard (λs. ¬ C s)
  unfolding L1-defs
  apply (rule whileLoop-infinite)
  done

lemma L1-while-false [L1opt]:
  L1-while (λ-. False) B = L1-skip
  apply (clarsimp simp: L1-while-def L1-skip-def)
  apply (subst whileLoop-unroll)
  apply clarsimp
  done

declare ucast-id [L1opt]
declare scast-id [L1opt]
declare L1-set-to-pred-def [L1opt]
declare L1-rel-to-fun-def [L1opt]

lemma in-set-to-pred [L1opt]: (λs. s ∈ {x. P x}) = P
  apply simp
  done

lemma in-set-if-then [L1opt]: (s ∈ (if P then A else B)) = (if P then (s ∈ A) else

```

```

(s ∈ B)
  apply simp
done

lemma Pair-unit-Image[L1opt]: Pair () ‘ S ‘ {x} = {(u, x′). (x, x′) ∈ S}
  by auto

lemmas if-simps =
  if-x-Not if-Not-x if-cancel if-True if-False if-bool-simps

declare empty-iff [L1opt]
declare UNIV-I [L1opt]
declare singleton-iff [L1opt]
declare if-simps [L1opt]
declare simp-thms [L1opt]

lemma L1-call-stop-cong: (L1-call f n g r) = (L1-call f n g r)
  by simp

lemma L1-merge-assignments : (L1-seq (L1-modify f)
  (L1-seq (L1-modify g) X)) ≡ L1-seq (L1-modify (λs. g (f
s))) X
  apply (subst atomize-eq)
  apply (unfold L1-defs)
  apply (rule spec-monad-eqI; auto simp add: runs-to-iff)
done

end

theory SimplConv
imports L1Peephole
begin

named-theorems L1unfold
declare creturn-def [L1unfold]
declare creturn-void-def [L1unfold]
declare cbreak-def [L1unfold]
declare cgoto-def [L1unfold]
declare whileAnno-def [L1unfold]
declare ccatchbrk-def [L1unfold]
declare ccatchgoto-def [L1unfold]
declare ccatchreturn-def [L1unfold]
declare cexit-def [L1unfold]

```


lemma *switch-alt-defs* [*L1unfold*]:
switch $x \square \equiv SKIP$
switch $v ((a, b) \# vs) \equiv Cond \{s. v s \in a\} b (switch\ v\ vs)$
by *auto*

lemma *sless-positive* [*simp*]:
 $\llbracket a < n; n \leq (2 \wedge (len-of\ TYPE('a) - 1)) - 1 \rrbracket \implies (a :: ('a::\{len\})\ word) <_s n$
apply (*subst signed.less-le*)
apply *safe*
apply (*subst word-sle-msb-le*)
apply *safe*
apply (*force simp: not-msb-from-less*)
apply *simp*
apply *simp*
done

lemma *sle-positive* [*simp*]:
 $\llbracket a \leq n; n \leq (2 \wedge (len-of\ TYPE('a) - 1)) - 1 \rrbracket \implies (a :: ('a::\{len\})\ word) \leq_s n$
apply (*clarsimp simp: signed.le-less*)
done

lemmas [*L1except*] =
L1-set-to-pred-def in-set-to-pred in-set-if-then
L1-rel-to-fun-def Pair-unit-Image
L1-seq-assoc

end

theory *CorresXF*
imports
CCorresE
begin

definition *corresXF-simple* $st\ xf\ P\ M\ M' \equiv$
 $\forall s. (P\ s \wedge\ succeeds\ M\ (st\ s)) \longrightarrow (\forall r'\ t'.\ reaches\ M'\ s\ r'\ t' \longrightarrow$
 $\quad reaches\ M\ (st\ s)\ (xf\ r'\ t')\ (st\ t')) \wedge\ succeeds\ M'\ s$

definition *corresXF* $st\ ret\ xf\ ex\ xf\ P\ A\ C \equiv$

$$\begin{aligned} & \forall s. P s \wedge \text{succeeds } A (st s) \longrightarrow \\ & \quad (\forall r t. \text{reaches } C s r t \longrightarrow \\ & \quad \quad (\text{case } r \text{ of} \\ & \quad \quad \quad \text{Exn } r \Rightarrow \text{reaches } A (st s) (\text{Exn } (ex\text{-}xf r t)) (st t) \\ & \quad \quad \quad | \text{Result } r \Rightarrow \text{reaches } A (st s) (\text{Result } (ret\text{-}xf r t)) (st t))) \\ & \quad \wedge \text{succeeds } C s \end{aligned}$$

definition $rel\text{-}XF \text{ st } ret\text{-}xf \text{ ex}\text{-}xf Q \equiv \lambda(r, t) (r', t').$

$$\begin{aligned} & t' = st t \wedge \\ & rel\text{-}xval (\lambda e e'. e' = ex\text{-}xf e t) (\lambda v v'. v' = ret\text{-}xf v t) r r' \wedge \\ & Q r t \end{aligned}$$

lemma $corresXF\text{-refines}\text{-}iff:$

$$\begin{aligned} & corresXF \text{ st } ret\text{-}xf \text{ ex}\text{-}xf P A C \longleftrightarrow \\ & \quad (\forall s. P s \longrightarrow \text{refines } C A s (st s) (rel\text{-}XF \text{ st } ret\text{-}xf \text{ ex}\text{-}xf (\lambda - . True))) \end{aligned}$$

apply *standard*

subgoal

apply (*clarsimp simp add: corresXF-def refines-def-old rel-XF-def rel-xval.simps split: xval-splits*)

by (*metis Exn-def default-option-def exception-or-result-cases not-Some-eq*)

subgoal

apply (*fastforce simp add: corresXF-def refines-def-old rel-XF-def rel-xval.simps split: xval-splits*)

done

done

definition $corresXF\text{-post} \text{ st } ret\text{-}xf \text{ ex}\text{-}xf P Q A C \equiv$

$$\begin{aligned} & \forall s. P s \wedge \text{succeeds } A (st s) \longrightarrow \\ & \quad (\forall r t. \text{reaches } C s r t \longrightarrow Q s r t \wedge \\ & \quad \quad (\text{case } r \text{ of} \\ & \quad \quad \quad \text{Exn } r \Rightarrow \text{reaches } A (st s) (\text{Exn } (ex\text{-}xf r t)) (st t) \\ & \quad \quad \quad | \text{Result } r \Rightarrow \text{reaches } A (st s) (\text{Result } (ret\text{-}xf r t)) (st t))) \\ & \quad \wedge \text{succeeds } C s \end{aligned}$$

lemma $corresXF\text{-post}\text{-refines}\text{-}iff:$

$$\begin{aligned} & corresXF\text{-post} \text{ st } ret\text{-}xf \text{ ex}\text{-}xf P Q A C \longleftrightarrow \\ & \quad (\forall s. P s \longrightarrow \text{refines } C A s (st s) (rel\text{-}XF \text{ st } ret\text{-}xf \text{ ex}\text{-}xf (Q s))) \end{aligned}$$

apply *standard*

subgoal

apply (*clarsimp simp add: corresXF-post-def refines-def-old rel-XF-def rel-xval.simps split: xval-splits*)

by (*metis Exn-def default-option-def exception-or-result-cases not-Some-eq*)

subgoal

apply (*fastforce simp add: corresXF-post-def refines-def-old rel-XF-def rel-xval.simps split: xval-splits*)

done

done

lemma *corresXF-post-to-corresXF*:
 $corresXF\text{-}post\ st\ ret\text{-}xf\ ex\text{-}xf\ P\ Q\ A\ C \implies corresXF\ st\ ret\text{-}xf\ ex\text{-}xf\ P\ A\ C$
by (*auto simp add: corresXF-def corresXF-post-def*)

lemma *corresXF-corres-XF-post-conv*:
 $corresXF\ st\ ret\text{-}xf\ ex\text{-}xf\ P\ A\ C = corresXF\text{-}post\ st\ ret\text{-}xf\ ex\text{-}xf\ P\ (\lambda\ -\ \cdot\ \cdot\ True)\ A\ C$
by (*auto simp add: corresXF-def corresXF-post-def*)

lemma *corresXF-simple-corresXF*:
(*corresXF-simple st*
 $(\lambda x\ s.\ case\ x\ of$
 $\quad Exn\ r \Rightarrow Exn\ (ex\text{-}state\ r\ s)$
 $\quad | Result\ r \Rightarrow (Result\ (ret\text{-}state\ r\ s))\ P\ M\ M')$
 $=\ (corresXF\ st\ ret\text{-}state\ ex\text{-}state\ P\ M\ M')$
unfolding *corresXF-simple-def corresXF-def*
by (*auto split: xval-splits*)

lemma *corresXF-simpleI*: \llbracket
 $\bigwedge s'\ t'\ r'. \llbracket P\ s';\ succeeds\ M\ (st\ s');\ reaches\ M'\ s'\ r'\ t' \rrbracket$
 $\implies reaches\ M\ (st\ s')\ (xf\ r'\ t')\ (st\ t');$
 $\bigwedge s'. \llbracket P\ s';\ succeeds\ M\ (st\ s') \rrbracket \implies succeeds\ M'\ s'$
 $\rrbracket \implies corresXF\text{-}simple\ st\ xf\ P\ M\ M'$
apply *atomize*
apply (*clarsimp simp: corresXF-simple-def*)
done

lemma *corresXF-I*: \llbracket
 $\bigwedge s'\ t'\ r'. \llbracket P\ s';\ succeeds\ M\ (st\ s');\ reaches\ M'\ s'\ (Result\ r')\ t' \rrbracket$
 $\implies reaches\ M\ (st\ s')\ (Result\ (ret\text{-}state\ r'\ t'))\ (st\ t');$
 $\bigwedge s'\ t'\ r'. \llbracket P\ s';\ succeeds\ M\ (st\ s');\ reaches\ M'\ s'\ (Exn\ r')\ t' \rrbracket$
 $\implies reaches\ M\ (st\ s')\ (Exn\ (ex\text{-}state\ r'\ t'))\ (st\ t');$
 $\bigwedge s'. \llbracket P\ s';\ succeeds\ M\ (st\ s') \rrbracket \implies succeeds\ M'\ s'$
 $\rrbracket \implies corresXF\ st\ ret\text{-}state\ ex\text{-}state\ P\ M\ M'$
apply *atomize*
apply (*clarsimp simp: corresXF-def*)
subgoal for *s r t*
apply (*erule-tac x=s in alle, erule (1) impE*)
apply (*erule-tac x=s in alle, erule (1) impE*)
apply (*erule-tac x=s in alle, erule (1) impE*)
apply (*clarsimp split: xval-splits*)
apply *auto*
done
done

```

lemma ccpo-prod-gfp-gfp:
  class.ccpo
  (prod-lub Inf Inf :: (('a::complete-lattice * 'b :: complete-lattice) set  $\Rightarrow$  -))
  (rel-prod ( $\geq$ ) ( $\geq$ )) (mk-less (rel-prod ( $\geq$ ) ( $\geq$ )))
  by (rule ccpo-rel-prodI ccpo-Inf)+

lemma admissible-mem: ccpo.admissible Inf ( $\geq$ ) ( $\lambda A. x \in A$ )
  by (auto simp: ccpo.admissible-def)

lemma admissible-nondet-ord-corresXF:
  ccpo.admissible Inf ( $\geq$ ) ( $\lambda A. \text{corresXF } st \ R \ E \ P \ A \ C$ )
  unfolding corresXF-def imp-conjL imp-conjR
  apply (intro admissible-all admissible-conj admissible-imp)
  subgoal for s
  apply (rule ccpo.admissibleI)
  apply (clarsimp simp add: ccpo.admissible-def chain-def
    succeeds-def reaches-def split: prod.splits xval-splits)
  apply (intro conjI allI impI)
  subgoal
  apply transfer
  by (auto simp add: Inf-post-state-def top-post-state-def)
  (metis outcomes.simps(2) post-state.simps(3))
  subgoal
  apply transfer
  by (auto simp add: Inf-post-state-def top-post-state-def)
  (metis outcomes.simps(2) post-state.simps(3))
  done
  subgoal
  apply (rule ccpo.admissibleI)
  apply (clarsimp simp add: ccpo.admissible-def chain-def
    succeeds-def reaches-def split: prod.splits xval-splits)
  apply transfer
  apply (auto simp add: Inf-post-state-def top-post-state-def image-def vimage-def

    split: if-split-asm)
  done
done

```

```

lemma corresXF-top: corresXF st ret-xf ex-xf P  $\top$  C
  by (auto simp add: corresXF-def)

```

```

lemma admissible-nondet-ord-corresXF-post:
  ccpo.admissible Inf ( $\geq$ ) ( $\lambda A. \text{corresXF-post } st \ R \ E \ P \ Q \ A \ C$ )
  unfolding corresXF-post-def imp-conjL imp-conjR
  apply (intro admissible-all admissible-conj admissible-imp)
  subgoal
  apply (rule ccpo.admissibleI)

```

```

apply (clarsimp simp add: ccpo.admissible-def chain-def
        succeeds-def reaches-def split: prod.splits xval-splits)
apply (intro conjI allI impI)
subgoal
  apply transfer
  apply (clarsimp simp add: Inf-post-state-def top-post-state-def)
apply (metis (no-types, lifting) INF-top-conv(1) Inf-post-state-def top-post-state-def)
done
subgoal
  apply transfer
  by (auto simp add: Inf-post-state-def top-post-state-def)
      (metis outcomes.simps(2) post-state.simps(3))
subgoal
  apply transfer
  apply (clarsimp simp add: Inf-post-state-def top-post-state-def)
apply (metis (no-types, lifting) INF-top-conv(1) Inf-post-state-def top-post-state-def)
done
subgoal
  apply transfer
  apply (clarsimp simp add: Inf-post-state-def top-post-state-def image-def vim-
age-def
        split: if-split-asm)
  by (metis outcomes.simps(2) post-state.simps(3))
done
subgoal
  apply (rule ccpo.admissibleI)
  apply (clarsimp simp add: ccpo.admissible-def chain-def
        succeeds-def reaches-def split: prod.splits xval-splits)
  apply transfer
  apply (auto simp add: Inf-post-state-def top-post-state-def image-def vimage-def
        split: if-split-asm)
done
done

```

lemma *corresXF-post-top: corresXF-post st ret-xf ex-xf P Q \top C*
by (auto simp add: corresXF-post-def)

lemma *corresXF-assume-pre:*
 $\llbracket \bigwedge s s'. \llbracket P s'; s = st s' \rrbracket \implies \text{corresXF } st \text{ xf-normal } \text{xf-exception } P L R \rrbracket \implies$
corresXF st xf-normal xf-exception P L R
apply atomize
apply (clarsimp simp: corresXF-def)
apply force
done

lemma *corresXF-assume-fix-pre:*

$\llbracket \bigwedge s s'. \llbracket P s'; s = st s' \rrbracket \implies \text{corresXF } st \text{ xf-normal } \text{xf-exception } (\lambda s. s = s' \wedge P s) L R \rrbracket \implies \text{corresXF } st \text{ xf-normal } \text{xf-exception } P L R$
apply *atomize*
apply (*clarsimp simp: corresXF-def*)
done

lemma *corresXF-guard-imp*:
 $\llbracket \text{corresXF } st \text{ xf-normal } \text{xf-exception } Q f g; \bigwedge s. P s \implies Q s \rrbracket$
 $\implies \text{corresXF } st \text{ xf-normal } \text{xf-exception } P f g$
apply (*clarsimp simp: corresXF-def*)
done

lemma *corresXF-return*:
 $\llbracket \bigwedge s. \llbracket P s \rrbracket \implies \text{xf-normal } b s = a \rrbracket \implies$
 $\text{corresXF } st \text{ xf-normal } \text{xf-exception } P (\text{return } a) (\text{return } b)$
apply (*clarsimp simp: corresXF-def*)
done

lemma *corresXF-gets*:
 $\llbracket \bigwedge s. P s \implies \text{ret } (g s) s = f (st s) \rrbracket \implies$
 $\text{corresXF } st \text{ ret ex } P (\text{gets } f) (\text{gets } g)$
apply (*clarsimp simp: corresXF-def*)
done

lemma *corresXF-insert-guard*:
 $\llbracket \text{corresXF } st \text{ ret ex } Q A C; \bigwedge s. \llbracket P s \rrbracket \implies G (st s) \longrightarrow Q s \rrbracket \implies$
 $\text{corresXF } st \text{ ret ex } P (\text{guard } G \gg = (\lambda-. A)) C$
apply (*auto simp: corresXF-def succeeds-guard succeeds-bind reaches-bind reaches-guard*)
done

lemma *corresXF-exec-abs-guard*:
 $\text{corresXF } st \text{ ret-xf ex-xf } (\lambda s. P s \wedge G (st s)) (A ()) C \implies \text{corresXF } st \text{ ret-xf ex-xf}$
 $P (\text{guard } G \gg = A) C$
apply (*auto simp: corresXF-def succeeds-guard succeeds-bind reaches-bind reaches-guard*)
done

lemma *corresXF-simple-exec*:
 $\llbracket \text{corresXF-simple } st \text{ xf } P A B; \text{reaches } B s r' s'; \text{succeeds } A (st s); P s \rrbracket$
 $\implies \text{reaches } A (st s) (\text{xf } r' s') (st s')$
apply (*fastforce simp: corresXF-simple-def*)
done

lemma *corresXF-simple-fail*:
 $\llbracket \text{corresXF-simple } st \text{ xf } P A B; \neg \text{succeeds } B s; P s \rrbracket$
 $\implies \neg \text{succeeds } A (st s)$
apply (*fastforce simp: corresXF-simple-def*)
done

lemma *corresXF-simple-no-fail*:

$\llbracket \text{corresXF-simple } st \text{ } xf \text{ } P \text{ } A \text{ } B; \text{ succeeds } A \text{ } (st \text{ } s); P \text{ } s \rrbracket$
 $\implies \text{ succeeds } B \text{ } s$
apply (*fastforce simp: corresXF-simple-def*)
done

lemma *corresXF-exec-normal*:

$\llbracket \text{corresXF } st \text{ } ret \text{ } ex \text{ } P \text{ } A \text{ } B; \text{ reaches } B \text{ } s \text{ } (Result \text{ } r') \text{ } s'; \text{ succeeds } A \text{ } (st \text{ } s); P \text{ } s \rrbracket$
 $\implies \text{ reaches } A \text{ } (st \text{ } s) \text{ } (Result \text{ } (ret \text{ } r' \text{ } s')) \text{ } (st \text{ } s')$
by (*auto simp add: corresXF-def split: xval-splits*)

lemma *corresXF-exec-exception*:

$\llbracket \text{corresXF } st \text{ } ret \text{ } ex \text{ } P \text{ } A \text{ } B; \text{ reaches } B \text{ } s \text{ } (Exn \text{ } r') \text{ } s'; \text{ succeeds } A \text{ } (st \text{ } s); P \text{ } s \rrbracket$
 $\implies \text{ reaches } A \text{ } (st \text{ } s) \text{ } (Exn \text{ } (ex \text{ } r' \text{ } s')) \text{ } (st \text{ } s')$
by (*auto simp add: corresXF-def split: xval-splits*)

lemma *corresXF-exec-fail*:

$\llbracket \text{corresXF } st \text{ } ret \text{ } ex \text{ } P \text{ } A \text{ } B; \neg \text{ succeeds } B \text{ } s; P \text{ } s \rrbracket$
 $\implies \neg \text{ succeeds } A \text{ } (st \text{ } s)$
by (*auto simp add: corresXF-def split: xval-splits*)

lemma *corresXF-intermediate*:

$\llbracket \text{corresXF } st \text{ } ret \text{ } xf \text{ } ex \text{ } xf \text{ } P \text{ } A' \text{ } C;$
 $\text{corresXF } id \text{ } (\lambda r \text{ } s. \text{ } r) \text{ } (\lambda r \text{ } s. \text{ } r) \text{ } (\lambda s. \exists x. s = st \text{ } x \wedge P \text{ } x) \text{ } A \text{ } A' \rrbracket \implies$
 $\text{corresXF } st \text{ } ret \text{ } xf \text{ } ex \text{ } xf \text{ } P \text{ } A \text{ } C$

apply (*clarsimp simp: corresXF-def split: xval-splits*)
apply *fast*
done

lemma *corresXF-join*:

$\llbracket \text{corresXF } st \text{ } V \text{ } E \text{ } P \text{ } L \text{ } L'; \bigwedge x \text{ } y. \text{corresXF } st \text{ } V' \text{ } E \text{ } (P' \text{ } x \text{ } y) \text{ } (R \text{ } x) \text{ } (R' \text{ } y);$
 $\bigwedge s. Q \text{ } s \implies L' \cdot s \text{ } ?\llbracket \lambda r \text{ } t. \text{ case } r \text{ of } Exn \text{ } - \Rightarrow \top \mid Result \text{ } v \Rightarrow P' \text{ } (V \text{ } v \text{ } t) \text{ } v \text{ } t \rrbracket;$
 $\bigwedge s. Q \text{ } s \implies P \text{ } s \rrbracket \implies$
 $\text{corresXF } st \text{ } V' \text{ } E \text{ } Q \text{ } (L \text{ } >>= R) \text{ } (L' \text{ } >>= R')$

apply (*clarsimp simp add: corresXF-refines-iff split: xval-splits*)
apply (*rule refines-bind-bind-exn[where Q=rel-XF st V E (λ- -. True) □*
 $(\lambda(r, t) x. \text{ case } r \text{ of } Exn \text{ } - \Rightarrow \top \mid Result \text{ } v \Rightarrow P' \text{ } (V \text{ } v \text{ } t) \text{ } v \text{ } t)]$)

subgoal

apply (*rule refines-strengthen1[where R=rel-XF st V E (λ- -. True)]*)
by *auto*

by (*auto simp add: rel-XF-def*)

lemma *corresXF-join-xf-state-independent-same-state*:

$\llbracket \text{corresXF } (\lambda s. \text{ } s) \text{ } (\lambda r \text{ } s. \text{ } V \text{ } r) \text{ } (\lambda r \text{ } s. \text{ } E \text{ } r) \text{ } P \text{ } L \text{ } L';$
 $\bigwedge y. \text{corresXF } (\lambda s. \text{ } s) \text{ } (\lambda r \text{ } s. \text{ } V' \text{ } r) \text{ } (\lambda r \text{ } s. \text{ } E \text{ } r) \text{ } (P' \text{ } (V \text{ } y)) \text{ } (R \text{ } (V \text{ } y)) \text{ } (R' \text{ } y);$
 $\bigwedge s. Q \text{ } s \implies L \cdot s \text{ } ?\llbracket \lambda r \text{ } t. \text{ case } r \text{ of } Exn \text{ } - \Rightarrow \top \mid Result \text{ } v \Rightarrow P' \text{ } v \text{ } t \rrbracket; \bigwedge s. Q \text{ } s$
 $\implies P \text{ } s \rrbracket \implies$
 $\text{corresXF } (\lambda s. \text{ } s) \text{ } (\lambda r \text{ } s. \text{ } V' \text{ } r) \text{ } (\lambda r \text{ } s. \text{ } E \text{ } r) \text{ } Q \text{ } (L \text{ } >>= R) \text{ } (L' \text{ } >>= R')$

apply (*clarsimp simp add: corresXF-refines-iff*)
apply (*rule refines-bind-bind-exn[where*

$Q = \text{rel-XF } (\lambda s. s) (\lambda r s. V r) (\lambda r s. E r) (\lambda -. \text{True}) \sqcap$
 $(\lambda x (r, t). \text{case } r \text{ of } \text{Exn } - \Rightarrow \top \mid \text{Result } v \Rightarrow P' v t))$

subgoal
apply (*rule refines-strengthen2* [**where** $R = \text{rel-XF } (\lambda s. s) (\lambda r s. V r) (\lambda r s. E r) (\lambda -. \text{True})$])
by *auto*
by (*auto simp add: rel-XF-def*)

lemma *corresXF-except*:
 $\llbracket \text{corresXF } st \ V \ E \ P \ L \ L'; \bigwedge x y. \text{corresXF } st \ V \ E' \ (P' \ x \ y) \ (R \ x) \ (R' \ y);$
 $\bigwedge s. Q \ s \Longrightarrow L' \cdot s \ ?\llbracket \lambda r s. \text{case } r \text{ of } \text{Exn } r \Rightarrow P' \ (E \ r \ s) \ r \ s \mid \text{Result } - \Rightarrow \top \rrbracket;$
 $\bigwedge s. Q \ s \Longrightarrow P \ s \rrbracket \Longrightarrow$
 $\text{corresXF } st \ V \ E' \ Q \ (L \ <\text{catch}> \ R) \ (L' \ <\text{catch}> \ R')$
apply (*clarsimp simp add: corresXF-refines-iff*)
apply (*rule refines-catch* [**where** $Q = (\text{rel-XF } st \ V \ E \ (\lambda -. \text{True})) \sqcap$
 $(\lambda (r, s) x. \text{case } r \text{ of } \text{Exn } r \Rightarrow P' \ (E \ r \ s) \ r \ s \mid \text{Result } - \Rightarrow \top)$])
subgoal
apply (*rule refines-strengthen1* [**where** $R = \text{rel-XF } st \ V \ E \ (\lambda -. \text{True})$])
by *auto*
apply (*auto simp add: rel-XF-def*)
done

lemma *corresXF-cond*:
 $\llbracket \text{corresXF } st \ V \ E \ P \ L \ L'; \text{corresXF } st \ V \ E \ P \ R \ R'; \bigwedge s. P \ s \Longrightarrow A \ (st \ s) = A' \ s$
 $\rrbracket \Longrightarrow$
 $\text{corresXF } st \ V \ E \ P \ (\text{condition } A \ L \ R) \ (\text{condition } A' \ L' \ R')$
apply (*clarsimp simp add: corresXF-refines-iff*)
using *refines-condition by metis*

lemma *refines-assume-succeeds*: $(\text{succeeds } g \ t \Longrightarrow \text{refines } f \ g \ s \ t \ R) \Longrightarrow \text{refines } f \ g \ s \ t \ R$
by (*auto simp add: refines-def-old*)

lemma *corresXF-while*:
assumes *body-corres*: $\bigwedge x y. \text{corresXF } st \ \text{ret } ex \ (\lambda s. P \ x \ s \wedge y = \text{ret } x \ s) \ (A \ y) \ (B \ x)$
and *cond-match*: $\bigwedge s r. P \ r \ s \Longrightarrow C \ r \ s = C' \ (\text{ret } r \ s) \ (st \ s)$
and *pred-inv*: $\bigwedge r s. P \ r \ s \Longrightarrow C \ r \ s \Longrightarrow \text{succeeds } (\text{whileLoop } C' \ A \ (\text{ret } r \ s)) \ (st \ s) \Longrightarrow$
 $B \ r \cdot s \ ?\llbracket \lambda r s. \text{case } r \text{ of } \text{Exn } - \Rightarrow \text{True} \mid \text{Result } r \Rightarrow P \ r \ s \rrbracket$
and *init-match*: $\bigwedge s. P' \ x \ s \Longrightarrow y = \text{ret } x \ s$
and *pred-imply*: $\bigwedge s. P' \ x \ s \Longrightarrow P \ x \ s$
shows $\text{corresXF } st \ \text{ret } ex \ (P' \ x) \ (\text{whileLoop } C' \ A \ y) \ (\text{whileLoop } C \ B \ x)$
apply (*clarsimp simp add: corresXF-refines-iff*)
apply (*rule refines-assume-succeeds*)
subgoal for s
apply (*rule refines-mono* [*OF* - *refines-whileLoop'*
[**where** $R = \text{rel-XF } st \ \text{ret } ex \ (\lambda r s. \text{case } r \text{ of } \text{Exn } - \Rightarrow \text{True} \mid \text{Result } r \Rightarrow P \ r \ s)$]) \sqcap


```

      ( $\lambda(r, s) \cdot \text{case } r \text{ of } \text{Exn } - \Rightarrow \text{True} \mid \text{Result } r \Rightarrow$ 
         $\text{succeeds } (\text{whileLoop } C' A (\text{ret } r s)) (st s)$ )
    and  $C = C$  and  $C' = C'$  and  $B = B$  and  $B' = A$  and  $I = x$  and  $I' = y$  and
 $s = s$  and  $s' = st s$ ])
  subgoal by (auto simp add: rel-XF-def)
  subgoal by (auto simp add: rel-XF-def cond-match pred-implly)
  subgoal using body-corres pred-inv
  apply (subst (asm) rel-XF-def)
  apply (clarsimp simp add: corresXF-refines-iff)
  apply (rule refines-strengthen[where  $R = \text{rel-XF } st \text{ ret } ex (\lambda \cdot \cdot \text{True})$  and
     $G = \lambda r s. \text{case } r \text{ of } \text{Exn } - \Rightarrow \text{True} \mid \text{Result } r \Rightarrow \text{succeeds } (\text{whileLoop } C' A r)$ 
    s])
  subgoal by auto
  apply assumption
  subgoal
  apply (subst (asm) (3) whileLoop-unroll)
  apply (auto simp add: succeeds-bind runs-to-partial-def-old split: xval-splits)
  done
  apply (auto simp: rel-XF-def rel-xval.simps)
  done
  subgoal
  by (auto simp add: rel-XF-def init-match pred-implly)
  subgoal
  by (auto simp add: rel-XF-def rel-xval.simps rel-exception-or-result.simps
    Exn-def default-option-def split: xval-splits )
  done
done

```

lemma *corresXF-name-pre:*

$$\llbracket \bigwedge s'. \text{corresXF } st \text{ ret } ex (\lambda s. P s \wedge s = s') A C \rrbracket \Longrightarrow$$

$$\text{corresXF } st \text{ ret } ex P A C$$
 by (clarsimp simp: corresXF-def)

lemma *corresXF-guarded-while-body:*

$$\text{corresXF } st \text{ ret } ex P A B \Longrightarrow$$

$$\text{corresXF } st \text{ ret } ex P$$

$$(\text{do}\{ r \leftarrow A; - \leftarrow \text{guard } (G r); \text{return } r \}) B$$
 apply (clarsimp simp add: corresXF-refines-iff)
 apply (clarsimp simp add: refines-def-old succeeds-bind reaches-bind)
 by (smt (verit) Exn-def case-exception-or-result-Exn case-exception-or-result-Result

default-option-def the-Exception-simp the-Exception-Result exception-or-result-cases

is-Exception-simps(1) is-Exception-simps(2) not-None-eq)

lemma *whileLoop-succeeds-terminates-guard-body:*
 assumes $B\text{-succeeds: } \bigwedge i s. \text{succeeds } (B i) s \Longrightarrow \text{succeeds } (B' i) s$
 assumes $B\text{-reaches: } \bigwedge i s r t. \text{reaches } (B' i) s r t \Longrightarrow \text{succeeds } (B i) s \Longrightarrow \text{reaches } (B i) s r t$

assumes *termi*: $\text{run } (\text{whileLoop } C \ B \ I) \ s \neq \top$
shows $\text{run } (\text{whileLoop } C \ B' \ I) \ s \neq \top$
using *termi*

proof (*induct rule: whileLoop-ne-top-induct*)
case *step* **show** *?case unfolding top-post-state-def*
apply (*rule whileLoop-ne-Failure*)
using *step B-succeeds B-reaches*
apply (*clarsimp simp add: runs-to-def-old*)
by (*metis top-post-state-def*)

qed

lemma *whileLoop-succeeds-guard-body*:

assumes *B-succeeds*: $\bigwedge i \ s. \text{succeeds } (B \ i) \ s \implies \text{succeeds } (B' \ i) \ s$
assumes *B-reaches*: $\bigwedge i \ s \ r \ t. \text{reaches } (B' \ i) \ s \ r \ t \implies \text{succeeds } (B \ i) \ s \implies \text{reaches } (B \ i) \ s \ r \ t$
assumes *termi*: $\text{succeeds } (\text{whileLoop } C \ B \ I) \ s$
shows $\text{succeeds } (\text{whileLoop } C \ B' \ I) \ s$
using *whileLoop-succeeds-terminates-guard-body [OF B-succeeds B-reaches] termi*
by (*auto simp: succeeds-def top-post-state-def*)

lemma *corresXF-guarded-while*:

assumes *body-corres*: $\bigwedge x \ y. \text{corresXF } st \ \text{ret } ex \ (\lambda s. P \ x \ s \wedge y = \text{ret } x \ s) \ (A \ y) \ (B \ x)$
and *cond-match*: $\bigwedge s \ r. \llbracket P \ r \ s; G \ (\text{ret } r \ s) \ (st \ s) \rrbracket \implies C \ r \ s = C' \ (\text{ret } r \ s) \ (st \ s)$
and *pred-inv*: $\bigwedge r \ s. P \ r \ s \implies C \ r \ s \implies \text{succeeds } (\text{whileLoop } C' \ A \ (\text{ret } r \ s)) \ (st \ s) \implies G \ (\text{ret } r \ s) \ (st \ s) \implies B \ r \cdot s \ ?\{\lambda r \ s. \text{case } r \ \text{of } Exn \ - \Rightarrow \text{True} \mid \text{Result } r \Rightarrow G \ (\text{ret } r \ s) \ (st \ s) \longrightarrow P \ r \ s \}$
and *pred-impl*: $\bigwedge s. \llbracket G \ y \ (st \ s); P' \ x \ s \rrbracket \implies P \ x \ s$
and *init-match*: $\bigwedge s. \llbracket G \ y \ (st \ s); P' \ x \ s \rrbracket \implies y = \text{ret } x \ s$
shows $\text{corresXF } st \ \text{ret } ex \ (P' \ x)$
 $(\text{do } \{$
 $\quad - \leftarrow \text{guard } (G \ y);$
 $\quad \text{whileLoop } C' \ (\lambda i. (\text{do } \{$
 $\quad \quad r \leftarrow A \ i;$
 $\quad \quad - \leftarrow \text{guard } (G \ r);$
 $\quad \quad \text{return } r$
 $\quad \quad \})) \ y$
 $\quad \})$
 $(\text{whileLoop } C \ B \ x)$

proof –

{fix *i s*
assume ***: $\text{succeeds } (\text{whileLoop } C' \ (\lambda i. (\text{do } \{ r \leftarrow A \ i; - \leftarrow \text{guard } (G \ r); \text{return } r \})) \ i) \ s$

```

have succeeds (whileLoop C' A i) s
  apply (rule whileLoop-succeeds-guard-body [OF - - *])
  subgoal by (auto simp add: succeeds-bind)

  subgoal apply (clarsimp simp add: reaches-bind succeeds-bind)
    by (smt (verit, best) dual-order.refl exception-or-result-split-asm le-boolD)
  done
} note new-body-fails-more = this

note new-body-corres = body-corres [THEN corresXF-guarded-while-body]

show ?thesis
  apply (rule corresXF-exec-abs-guard)
  apply (rule corresXF-name-pre)
  apply (rule corresXF-assume-pre)
  apply clarsimp
  subgoal for s'
    apply (rule corresXF-guard-imp)
    apply (rule corresXF-while [where
       $P = \lambda x s. P x s \wedge G (ret x s) (st s)$  and  $P' = \lambda x s. P' x s \wedge s = s'$ ])
    apply (rule corresXF-guard-imp)
    apply (rule new-body-corres)
    apply (clarsimp)
    apply (clarsimp)
    apply (rule cond-match, simp, simp)
  subgoal for r s
    using pred-inv [of r s, OF - - new-body-fails-more]
    apply clarsimp
    apply (subst (asm) whileLoop-unroll)
    apply (clarsimp simp add: cond-match)
    apply (clarsimp simp add: succeeds-bind)
    apply (clarsimp simp add: runs-to-partial-def-old split: xval-splits, intro
conjI)
    apply (metis (mono-tags, lifting) body-corres corresXF-exec-normal)
    using body-corres corresXF-exec-normal by fastforce
  subgoal
    using init-match by auto
  subgoal
    using init-match pred-imply by auto
  subgoal by auto
  done
done
qed

```

definition *ac-corres st check-termination AF Γ rx ex G* \equiv
 $\lambda A B. \forall s. (G s \wedge \text{succeeds } A (st s)) \longrightarrow$
 $(\forall t. \Gamma \vdash \langle B, \text{Normal } s \rangle \Rightarrow t \longrightarrow$
 (case t of
 Normal s' \Rightarrow reaches A (st s) (Result (rx s')) (st s')
 | Abrupt s' \Rightarrow reaches A (st s) (Exn (ex s')) (st s')
 | Fault e $\Rightarrow e \in AF$
 | - \Rightarrow False))
 \wedge (check-termination $\longrightarrow \Gamma \vdash B \downarrow \text{Normal } s$)

lemma *ccorresE-corresXF-merge:*
 \llbracket ccorresE st1 ct AF $\Gamma \top G1 M B$;
 corresXF st2 rx ex G2 A M;
 $\bigwedge s. st s = st2 (st1 s)$;
 $\bigwedge r s. rx' s = rx r (st1 s)$;
 $\bigwedge r s. ex' s = ex r (st1 s)$;
 $\bigwedge s. G s \longrightarrow (s \in G1 \wedge G2 (st1 s)) \rrbracket \implies$
ac-corres st ct AF Γ rx' ex' G A B
apply (unfold ac-corres-def)
apply clarsimp
apply (clarsimp simp: ccorresE-def)
apply (clarsimp simp: corresXF-def)
apply (erule allE, erule impE, force)
apply (erule allE, erule impE, force)
apply clarsimp
apply (erule allE, erule impE, fastforce)
subgoal for s t by (cases t; fastforce)
done

lemma *corresXF-corresXF-merge:*
 \llbracket corresXF st rx ex P A B; corresXF st' rx' ex' P' B C $\rrbracket \implies$
 corresXF (st o st') ($\lambda rv s. rx (rx' rv s) (st' s)$)
 ($\lambda rv s. ex (ex' rv s) (st' s)$) ($\lambda s. P' s \wedge P (st' s)$) A C
apply (clarsimp simp: corresXF-def split: xval-splits)
apply fastforce
done

lemma *ac-corres-guard-imp:*
 \llbracket ac-corres st ct AF G rx ex P A C; $\bigwedge s. P' s \implies P s \rrbracket \implies$ ac-corres st ct AF
 G rx ex P' A C
apply atomize
apply (clarsimp simp: ac-corres-def)
done

lemma *corresXF-modify-local*:

$\llbracket \bigwedge s. st\ s = st\ (M\ s); \bigwedge s. P\ s \implies ret\ ()\ (M\ s) = x \rrbracket$
 $\implies corresXF\ st\ ret\ ex\ P\ (return\ x)\ (modify\ M)$
by (*auto simp add: corresXF-def*)

lemma *corresXF-modify-global*:

$\llbracket \bigwedge s. P\ s \implies M\ (st\ s) = st\ (M'\ s) \rrbracket \implies$
 $corresXF\ st\ ret\ ex\ P\ (modify\ M)\ (modify\ M')$
by (*auto simp add: corresXF-def*)

lemma *corresXF-select-modify*:

$\llbracket \bigwedge s. P\ s \implies st\ s = st\ (M\ s); \bigwedge s. P\ s \implies ret\ ()\ (M\ s) \in x \rrbracket \implies$
 $corresXF\ st\ ret\ ex\ P\ (select\ x)\ (modify\ M)$
by (*auto simp add: corresXF-def*)

lemma *corresXF-select-select*:

$\llbracket \bigwedge s\ a. st\ s = st\ (M\ (a::('a \Rightarrow ('a::\{type\})))\ s);$
 $\bigwedge s\ x. \llbracket P\ s; x \in b \rrbracket \implies ret\ x\ s \in a \rrbracket \implies$
 $corresXF\ st\ ret\ ex\ P\ (select\ a)\ (select\ b)$
by (*auto simp add: corresXF-def*)

lemma *corresXF-modify-gets*:

$\llbracket \bigwedge s. P\ s \implies st\ s = st\ (M\ s); \bigwedge s. P\ s \implies ret\ ()\ (M\ s) = f\ (st\ (M\ s)) \rrbracket \implies$
 $corresXF\ st\ ret\ ex\ P\ (gets\ f)\ (modify\ M)$
by (*auto simp add: corresXF-def*)

lemma *corresXF-guard*:

$\llbracket \bigwedge s. P\ s \implies G'\ s = G\ (st\ s) \rrbracket \implies corresXF\ st\ ret\ ex\ P\ (guard\ G)\ (guard\ G')$
by (*auto simp add: corresXF-def*)

lemma *corresXF-fail*:

$corresXF\ st\ return\ xf\ exception\ xf\ P\ fail\ X$
by (*auto simp add: corresXF-def*)

lemma *corresXF-spec*:

$\llbracket \bigwedge s\ s'. ((s, s') \in A') = ((st\ s, st\ s') \in A); surj\ st \rrbracket$
 $\implies corresXF\ st\ ret\ ex\ P\ (state\ select\ A)\ (state\ select\ A')$
apply (*clarsimp simp add: corresXF-def*)
apply (*frule-tac y=undefined in surjD*)
apply (*clarsimp simp: image-def set-eq-UNIV*)
apply *metis*
done

lemma *corresXF-throw*:

$\llbracket \bigwedge s. P\ s \implies E\ B\ s = A \rrbracket \implies corresXF\ st\ V\ E\ P\ (throw\ A)\ (throw\ B)$
by (*auto simp add: corresXF-def Ern-def*)

lemma *corresXF-append-gets-abs*:
assumes *corres*: *corresXF st ret ex P L R*
and *consistent*: $\bigwedge s. P s \implies R \cdot s \ ?\{\lambda r s. \text{case } r \text{ of } \text{Exn } - \implies \top \mid \text{Result } v \implies M$
 $(\text{ret } v s) (st s) = \text{ret}' v s \}$
shows *corresXF st ret' ex P (L >>= (λr. gets (M r))) R*
using *corres consistent*
apply (*clarsimp simp add: corresXF-refines-iff runs-to-partial-def-old refines-def-old*
succeeds-bind reaches-bind rel-XF-def rel-xval.simps split: xval-splits)
using *Exn-def* **by** *force*

lemma *corresXF-skipE*:
corresXF st ret ex P skip skip
by (*auto simp add: corresXF-def Exn-def*)

lemma *corresXF-id*:
corresXF id (λr s. r) (λr s. r) P M M
by (*fastforce simp: corresXF-def split: xval-splits*)

lemma *corresXF-cong*:
 $\llbracket \bigwedge s. st s = st' s;$
 $\bigwedge s r. \text{ret-xf } r s = \text{ret-xf}' r s;$
 $\bigwedge s r. \text{ex-xf } r s = \text{ex-xf}' r s;$
 $\bigwedge s. P s = P' s;$
 $\bigwedge s s'. P' s' \implies \text{run } A s = \text{run } A' s;$
 $\bigwedge s. P' s \implies \text{run } C s = \text{run } C' s \rrbracket \implies$
 $\text{corresXF } st \text{ ret-xf } \text{ex-xf } P A C = \text{corresXF } st' \text{ ret-xf}' \text{ex-xf}' P' A' C'$
apply *atomize*
apply (*auto simp: corresXF-def reaches-def succeeds-def split: xval-splits*)
done

lemma *corresXF-exec-abs-select*:
 $\llbracket x \in Q; x \in Q \implies \text{corresXF } id \text{ rx } \text{ex } P (A x) A' \rrbracket \implies \text{corresXF } id \text{ rx } \text{ex } P$
 $(\text{select } Q >>= A) A'$
by (*fastforce simp add: corresXF-def succeeds-bind reaches-bind split: xval-splits*)

end

18.2 Hoare-Triples for L1 (internal use)

theory *L1Valid*
imports *L1Defs*
begin

definition
validE :: $(s \implies \text{bool}) \implies (e, a, s) \text{ exn-monad} \implies$
 $(a \implies s \implies \text{bool}) \implies$

$(\{e \Rightarrow s \Rightarrow \text{bool}\} \Rightarrow \text{bool})$
 $(\{-\} / - / (\{-\} / \{-\}))$
where
 $\{P\} f \{Q\}, \{E\} \equiv \forall s. P s \longrightarrow f \cdot s \text{ ?} \{ \lambda v s. \text{case } v \text{ of Result } r \Rightarrow Q r s \mid \text{Err } e \Rightarrow E e s \}$

lemma *hoareE-TrueI*: $\{P\} f \{\lambda -. \text{True}\}, \{\lambda r -. \text{True}\}$
by (*simp add: validE-def runs-to-partial-def-old split: xval-splits*)

lemma *combine-validE*: $\llbracket \{P\} x \{Q\}, \{E\} \rrbracket;$
 $\{P'\} x \{Q'\}, \{E'\} \rrbracket \Longrightarrow$
 $\{P \text{ and } P'\} x \{\lambda r. (Q r) \text{ and } (Q' r)\}, \{\lambda r. (E r) \text{ and } (E' r)\}$
by (*auto simp add: validE-def runs-to-partial-def-old split: xval-splits*)

lemma *L1-skip-wp*: $\{P ()\} L1\text{-skip } \{P\}, \{Q\}$
apply (*clarsimp simp: L1-skip-def validE-def*)
done

lemma *L1-modify-wp*: $\{\lambda s. P () (f s)\} L1\text{-modify } f \{P\}, \{Q\}$
apply (*clarsimp simp: L1-modify-def validE-def*)
apply (*runs-to-vcg*)
done

lemma *L1-spec-wp*: $\{\lambda s. \forall t. (s, t) \in f \longrightarrow P () t\} L1\text{-spec } f \{P\}, \{Q\}$
apply (*clarsimp simp add: L1-spec-def validE-def*)
apply (*runs-to-vcg*)
apply *auto*
done

lemma *L1-assume-wp*: $\{\lambda s. \forall t. ((), t) \in f s \longrightarrow P () t\} L1\text{-assume } f \{P\}, \{Q\}$
apply (*clarsimp simp add: L1-assume-def validE-def*)
apply (*runs-to-vcg*)
apply *auto*
done

lemma *L1-init-wp*: $\{\lambda s. \forall x. P () (f (\lambda-. x) s)\} L1\text{-init } f \{P\}, \{Q\}$
apply (*clarsimp simp add: L1-init-def validE-def*)
apply (*runs-to-vcg*)
apply *auto*
done

lemma *L1-skip-lp*: $\llbracket \bigwedge s. P s \Longrightarrow Q () s \rrbracket \Longrightarrow \{P\} L1\text{-skip } \{Q\}, \{E\}$
apply (*clarsimp simp: L1-skip-def*)
apply (*clarsimp simp: validE-def*)

done

lemma *L1-skip-lp-same-pre-post*: $\{P\} \text{ L1-skip } \{\lambda\cdot P\}, \{\lambda\cdot P\}$
by (*rule L1-skip-lp*)

lemma *L1-guard-lp*: $\llbracket \bigwedge s. P s \implies Q () s \rrbracket \implies \{P\} \text{ L1-guard } e \{Q\}, \{E\}$
apply (*clarsimp simp: L1-guard-def guard-def*)
apply (*clarsimp simp: validE-def*)
apply (*runs-to-vcg*)
apply *auto*
done

lemma *L1-guard-lp-same-pre-post*: $\{P\} \text{ L1-guard } e \{\lambda\cdot P\}, \{\lambda\cdot P\}$
by (*rule L1-guard-lp*)

lemma *L1-guarded-lp-same-pre-post*: $\{P\} c \{\lambda\cdot P\}, \{\lambda\cdot P\}$
 $\implies \{P\} \text{ L1-guarded } g c \{\lambda\cdot P\}, \{\lambda\cdot P\}$
unfolding *L1-defs L1-guarded-def validE-def*
by (*auto simp add: runs-to-partial-def-old succeeds-bind reaches-bind*)

lemma *L1-guarded-lp-gets*: $(\bigwedge p. \{P\} (c p) \{\lambda\cdot P\}, \{\lambda\cdot P\})$
 $\implies \{P\} \text{ L1-guarded } g (\text{gets dest } \gg (\lambda p. c p)) \{\lambda\cdot P\}, \{\lambda\cdot P\}$
unfolding *L1-defs L1-guarded-def validE-def*
by (*auto simp add: runs-to-partial-def-old succeeds-bind reaches-bind*)

lemma *L1-fail-lp*: $\{P\} \text{ L1-fail } \{Q\}, \{E\}$
apply (*clarsimp simp: L1-fail-def validE-def*)
done

lemma *L1-fail-lp-same-pre-post*: $\{P\} \text{ L1-fail } \{\lambda\cdot P\}, \{\lambda\cdot P\}$
by (*rule L1-fail-lp*)

lemma *L1-throw-lp*: $\llbracket \bigwedge s. P s \implies E () s \rrbracket \implies \{P\} \text{ L1-throw } \{Q\}, \{E\}$
apply (*clarsimp simp: L1-throw-def validE-def*)
done

lemma *L1-throw-lp-same-pre-post*: $\{P\} \text{ L1-throw } \{\lambda\cdot P\}, \{\lambda\cdot P\}$
by (*rule L1-throw-lp*)

lemma *L1-spec-lp*: $\llbracket \bigwedge s r. \llbracket (s, r) \in e; P s \rrbracket \implies Q () r \rrbracket \implies \{P\} \text{ L1-spec } e$
 $\{Q\}, \{E\}$
apply (*clarsimp simp: L1-spec-def validE-def*)
apply (*runs-to-vcg*)

apply *auto*
done

lemma *L1-spec-lp-same-pre-post*: $\llbracket \bigwedge s r. \llbracket (s, r) \in e; P s \rrbracket \implies P r \rrbracket$
 $\implies \{P\} \text{L1-spec } e \{ \lambda-. P \}, \{ \lambda-. P \}$
by (*rule L1-spec-lp*)

lemma *L1-modify-lp*: $\llbracket \bigwedge s. P s \implies Q () (f s) \rrbracket \implies \{P\} \text{L1-modify } f \{Q\}, \{E\}$
apply (*clarsimp simp: L1-modify-def validE-def*)
apply (*runs-to-vcg*)
apply *auto*
done

lemma *L1-modify-lp-same-pre-post*: $\llbracket \bigwedge s. P s \implies P (f s) \rrbracket \implies \{P\} \text{L1-modify } f$
 $\{ \lambda-. P \}, \{ \lambda-. P \}$
by (*rule L1-modify-lp*)

lemma *L1-call-lp*:
 $\llbracket \bigwedge s r. P s \implies Q () (\text{return-xf } r (\text{scope-teardown } s r));$
 $\bigwedge s r. P s \implies E () (\text{result-exn } (\text{scope-teardown } s r) r) \rrbracket \implies$
 $\{P\} \text{L1-call } \text{scope-setup } \text{dest-fn } \text{scope-teardown } \text{result-exn } \text{return-xf } \{Q\}, \{E\}$
apply (*clarsimp simp: L1-defs L1-call-def validE-def*)
apply (*runs-to-vcg*)
apply (*auto simp add: runs-to-partial-def-old reaches-bind*)
done

lemma *L1-call-lp-same-pre-post*:
 $\llbracket \bigwedge s r. P s \implies P (\text{return-xf } r (\text{scope-teardown } s r));$
 $\bigwedge s r. P s \implies P (\text{result-exn } (\text{scope-teardown } s r) r) \rrbracket \implies$
 $\{P\} \text{L1-call } \text{scope-setup } \text{dest-fn } \text{scope-teardown } \text{result-exn } \text{return-xf } \{ \lambda-. P \}, \{ \lambda-.$
 $P \}$
by (*rule L1-call-lp*)

lemma *L1-seq-lp*: \llbracket
 $\{P1\} A \{Q1\}, \{E1\};$
 $\{P2\} B \{Q2\}, \{E2\};$
 $\bigwedge s. P s \implies P1 s;$
 $\bigwedge s. Q1 () s \implies P2 s;$
 $\bigwedge s. Q2 () s \implies Q () s;$
 $\bigwedge s. E1 () s \implies E () s;$
 $\bigwedge s. E2 () s \implies E () s$
 $\rrbracket \implies \{P\} \text{L1-seq } A B \{Q\}, \{E\}$
apply (*clarsimp simp: L1-seq-def validE-def*)
apply (*runs-to-vcg*)
apply (*fastforce simp add: runs-to-partial-def-old reaches-bind split: vval-splits*)
done

lemma *L1-seq-lp-same-pre-post*: \llbracket

$\{\{P\}\} A \{\{\lambda-. P\}\}, \{\{\lambda-. P\}\};$
 $\{\{P\}\} B \{\{\lambda-. P\}\}, \{\{\lambda-. P\}\}$
 $\] \Longrightarrow \{\{P\}\} L1\text{-seq } A B \{\{\lambda-. P\}\}, \{\{\lambda-. P\}\}$
by (rule L1-seq-lp)

lemma L1-condition-lp:

$\[\{\{P1\}\} A \{\{Q1\}\}, \{\{E1\}\};$
 $\{\{P2\}\} B \{\{Q2\}\}, \{\{E2\}\};$
 $\wedge s. P s \Longrightarrow P1 s;$
 $\wedge s. P s \Longrightarrow P2 s;$
 $\wedge s. Q1 () s \Longrightarrow Q () s;$
 $\wedge s. Q2 () s \Longrightarrow Q () s;$
 $\wedge s. E1 () s \Longrightarrow E () s;$
 $\wedge s. E2 () s \Longrightarrow E () s \]$
 $\{\{P\}\} L1\text{-condition } c A B \{\{Q\}\}, \{\{E\}\}$
apply (clarsimp simp: L1-condition-def validE-def)
apply (runs-to-vcg)
apply (fastforce simp add: runs-to-partial-def-old split: xval-splits)+
done

lemma L1-condition-lp-same-pre-post:

$\[\{\{P\}\} A \{\{\lambda-. P\}\}, \{\{\lambda-. P\}\};$
 $\{\{P\}\} B \{\{\lambda-. P\}\}, \{\{\lambda-. P\}\}$
 $\] \Longrightarrow$
 $\{\{P\}\} L1\text{-condition } c A B \{\{\lambda-. P\}\}, \{\{\lambda-. P\}\}$
by (rule L1-condition-lp)

lemma L1-catch-lp:

$\[\{\{P1\}\} A \{\{Q1\}\}, \{\{E1\}\};$
 $\{\{P2\}\} B \{\{Q2\}\}, \{\{E2\}\};$
 $\wedge s. P s \Longrightarrow P1 s;$
 $\wedge s. E1 () s \Longrightarrow P2 s;$
 $\wedge s. Q1 () s \Longrightarrow Q () s;$
 $\wedge s. Q2 () s \Longrightarrow Q () s;$
 $\wedge s. E2 () s \Longrightarrow E () s \]$
 $\{\{P\}\} L1\text{-catch } A B \{\{Q\}\}, \{\{E\}\}$
apply (clarsimp simp: L1-catch-def validE-def)
apply (runs-to-vcg)
apply (fastforce simp add: runs-to-partial-def-old split: xval-splits)
done

lemma L1-catch-lp-same-pre-post:

$\[\{\{P\}\} A \{\{\lambda-. P\}\}, \{\{\lambda-. P\}\};$
 $\{\{P\}\} B \{\{\lambda-. P\}\}, \{\{\lambda-. P\}\}$
 $\] \Longrightarrow$
 $\{\{P\}\} L1\text{-catch } A B \{\{\lambda-. P\}\}, \{\{\lambda-. P\}\}$
by (rule L1-catch-lp)

lemma *L1-init-lp*: $\llbracket \bigwedge s. P\ s \implies \forall x. Q\ ()\ (f\ (\lambda-. x)\ s) \rrbracket \implies \{P\}\ L1\text{-init}\ f\ \{Q\}, \{E\}$
apply (*clarsimp simp add: L1-init-def validE-def*)
apply (*runs-to-vcg*)
apply *auto*
done

lemma *L1-init-lp-same-pre-post*: $\llbracket \bigwedge s. P\ s \implies \forall x. P\ (f\ (\lambda-. x)\ s) \rrbracket \implies \{P\}\ L1\text{-init}\ f\ \{\lambda-. P\}, \{\lambda-. P\}$
by (*rule L1-init-lp*)

lemma *validE-weaken*:
 $\llbracket \{P'\}\ A\ \{Q'\}, \{E'\}; \bigwedge s. P\ s \implies P'\ s; \bigwedge r\ s. Q'\ r\ s \implies Q\ r\ s; \bigwedge r\ s. E'\ r\ s \implies E\ r\ s \rrbracket \implies \{P'\}\ A\ \{Q'\}, \{E'\}$
apply (*fastforce simp add: validE-def runs-to-partial-def-old split: xval-splits*)
done

lemma *L1-while-lp*:
assumes *body-lp*: $\{P'\}\ B\ \{Q'\}, \{E'\}$
and *p-impl*: $\bigwedge s. P\ s \implies P'\ s$
and *q-impl*: $\bigwedge s. Q'\ ()\ s \implies Q\ ()\ s$
and *e-impl*: $\bigwedge s. E'\ ()\ s \implies E\ ()\ s$
and *inv*: $\bigwedge s. Q'\ ()\ s \implies P'\ s$
and *inv'*: $\bigwedge s. P'\ s \implies Q'\ ()\ s$
shows $\{P'\}\ L1\text{-while}\ c\ B\ \{Q'\}, \{E'\}$
apply (*rule validE-weaken [where P'=P' and Q'=Q' and E'=E']*)
apply (*clarsimp simp: L1-while-def validE-def*)

apply (*rule runs-to-partial-whileLoop-exn [where I= $\lambda r\ s. (case\ r\ of\ Exn\ e \Rightarrow E'\ ()\ s\ | - \Rightarrow P'\ s)$]*)
apply *simp*
apply (*simp add: inv'*)
apply (*simp*)
apply *simp*
subgoal using *body-lp*
by (*simp add: inv validE-def runs-to-partial-def-old split: xval-splits*)
apply (*simp add: p-impl*)
apply (*simp add: q-impl*)
apply (*simp add: e-impl*)
done

lemma *L1-while-lp-same-pre-post*:
assumes *body-lp*: $\{P'\}\ B\ \{\lambda-. P'\}, \{\lambda-. P'\}$
shows $\{P'\}\ L1\text{-while}\ c\ B\ \{\lambda-. P'\}, \{\lambda-. P'\}$
by (*rule L1-while-lp [OF body-lp]*)

lemma *on-exit-lp-same-pre-post*:

```

assumes cleanup:  $\bigwedge s t. (s,t) \in \text{cleanup} \implies P t = P s$ 
assumes c:  $\{P\} c \{ \lambda-. P \}, \{ \lambda-. P \}$ 
shows  $\{P\}$  on-exit c cleanup  $\{ \lambda-. P \}, \{ \lambda-. P \}$ 
apply (clarsimp simp add: validE-def on-exit'-def on-exit-def)
apply (runs-to-vcg)
using cleanup c
apply (fastforce simp add: validE-def runs-to-partial-def-old succeeds-bind
reaches-bind default-option-def Exn-def split: xval-splits)
done

```

lemma *seqE*:

```

assumes f-valid:  $\{A\} f \{B\}, \{E\}$ 
assumes g-valid:  $\bigwedge x. \{B x\} g x \{C\}, \{E\}$ 
shows  $\{A\}$  do  $\{ x \leftarrow f; g x \}$   $\{C\}, \{E\}$ 
using assms
apply (clarsimp simp add: validE-def)
apply (runs-to-vcg)
apply (fastforce simp add: runs-to-partial-def-old split: xval-splits)
done

```

named-theorems *with-fresh-stack-ptr-lp-same-pre-post*

context *stack-heap-state*

begin

lemma *with-fresh-stack-ptr-lp-same-pre-post*[*with-fresh-stack-ptr-lp-same-pre-post*]:

```

assumes c:  $\bigwedge p. \{P\} (c p) \{ \lambda-. P \}, \{ \lambda-. P \}$ 
assumes htd-indep:  $\bigwedge s g. P (htd-upd g s) = P s$ 
assumes hmem-indep:  $\bigwedge s f. P (hmem-upd f s) = P s$ 
shows  $\{P\}$  with-fresh-stack-ptr n g c  $\{ \lambda-. P \}, \{ \lambda-. P \}$ 
apply (clarsimp simp add: with-fresh-stack-ptr-def)
apply (rule seqE [where B= $\lambda-. P$ ])
subgoal
apply (clarsimp simp add: validE-def)
apply (runs-to-vcg)
by (clarsimp simp add: htd-indep hmem-indep)
subgoal
apply (rule on-exit-lp-same-pre-post [OF - c])
apply (auto simp add: htd-indep hmem-indep)
done
done
end

```

lemma *validE-weaken-dependent*:

```

assumes dep-spec:  $\bigwedge s. P s \implies \{P' s\} A \{Q' s\}, \{E' s\}$ 
assumes weaken-pre:  $\bigwedge s. P s \implies P' s s$ 
assumes weaken-norm:  $(\bigwedge s r t. P' s s \implies Q' s r t \implies Q r t)$ 
assumes weaken-exn:  $(\bigwedge s r t. P' s s \implies E' s r t \implies E r t)$ 
shows  $\{P\} A \{Q\}, \{E\}$ 
using assms

```

apply (*fastforce simp add: validE-def runs-to-partial-def-old split: xval-splits*)
done

lemma *validE-weaken-dependent-same:*

assumes *dep-spec:* $\bigwedge s. P\ s \implies \{\{P'\ s\}\ A\ \{\lambda-. P'\ s\}, \{\lambda-. P'\ s\}\}$

assumes *weaken-post:* $(\bigwedge s\ t. P'\ s\ t \implies P\ t)$

assumes *weaken-pre:* $\bigwedge s. P\ s \implies P'\ s\ s$

shows $\{\{P\}\ A\ \{\lambda-. P\}, \{\lambda-. P\}\}$

using *dep-spec weaken-pre weaken-post weaken-post*

by (*rule validE-weaken-dependent*)

end

Chapter 19

L2 phase: local variable abstraction with lambdas

```
theory L2Defs
imports CorresXF L1Defs L1Peephole L1Valid
begin

type-synonym ('s, 'a, 'e) L2-monad = ('e, 'a, 's) exn-monad

definition L2-unknown (ns :: nat list) ≡ select UNIV :: ('s, 'a, 'e) L2-monad
definition L2-seq (A :: ('s, 'a, 'e) L2-monad) (B :: 'a ⇒ ('s, 'b, 'e) L2-monad) ≡
(A >>= B) :: ('s, 'b, 'e) L2-monad
definition L2-modify m ≡ modify m :: ('s, unit, 'e) L2-monad
definition L2-gets f (names :: nat list) ≡ gets f :: ('s, 'a, 'e) L2-monad
definition L2-condition c (A :: ('s, 'a, 'e) L2-monad) (B :: ('s, 'a, 'e) L2-monad)
≡ condition c A B
definition L2-catch (A :: ('s, 'a, 'e) L2-monad) (B :: 'e ⇒ ('s, 'a, 'ee) L2-monad)
≡ (A <catch> B)
definition L2-try (A :: ('s, 'a, 'e + 'a) L2-monad) ≡ try A
definition L2-while c (A :: 'a ⇒ ('s, 'a, 'e) L2-monad) i (ns :: nat list) ≡ whileLoop
c A i :: ('s, 'a, 'e) L2-monad
definition L2-throw x (ns :: nat list) ≡ throw x :: ('s, 'a, 'e) L2-monad
definition L2-spec r ≡ (state-select r >>= (λ-. select UNIV)) :: ('s, 'a, 'e)
L2-monad
definition L2-assume r ≡ (assume-result-and-state r) :: ('s, 'a, 'e) L2-monad
definition L2-guard c ≡ (guard c) :: ('s, unit, 'e) L2-monad
definition L2-guarded P c ≡ L2-seq (L2-guard P) (λ-. c) — used to guard a
function-pointer call
definition L2-fail ≡ fail :: ('s, 'a, 'e) L2-monad
definition L2-call x emb (ns :: nat list) ≡ map-value (map-exn emb) x :: ('s, 'a,
'e) L2-monad

abbreviation L2-skip ≡ L2-gets (λ-. ()) []
definition L2-VARS f (names::nat list) ≡ f — Auxilliary construct to preserve
names, like in L2-unknown, L2-gets, ...
```

definition *gets-theE* :: ('s ⇒ 'a option) ⇒ ('e, 'a, 's) *exn-monad* **where**
gets-theE ≡ λx. *gets-the* x — Lifting into exception monad

definition *L2-folded-gets f names* ≡ *L2-gets f names* :: ('s, 'a, 'e) *L2-monad*

definition *L2-voidcall x emb ns* ≡ *L2-seq* (*L2-call* x emb ns) (λret. *L2-skip*) :: ('s, unit, 'e) *L2-monad*

definition *L2-modifycall x m emb ns* ≡ *L2-seq* (*L2-call* x emb ns) (λret. *L2-modify* (m ret)) :: ('s, unit, 'e) *L2-monad*

definition *L2-returncall x f emb ns* ≡ *L2-seq* (*L2-call* x emb ns) (λret. *L2-folded-gets* (f ret) []) :: ('s, 'a, 'e) *L2-monad*

definition *L2-seq-guard P A* ≡ *L2-seq* (*L2-guard* P) A

definition *L2-seq-gets c n A* ≡ *L2-seq* (*L2-gets* (λ-. c) n) A

definition *SANITIZE-NAMES prj ns ns'* = True

lemma *sanitize-namesI*: *SANITIZE-NAMES prj ns ns'*
by (*simp* add: *SANITIZE-NAMES-def*)

definition *DYN-CALL* :: prop ⇒ prop **where** *PROP DYN-CALL* (*PROP P*) ≡ *PROP P*

lemma *DYN-CALL-I*: *PROP P* ⇒ *PROP DYN-CALL* (*PROP P*)
by (*simp* add: *DYN-CALL-def*)

lemma *DYN-CALL-D*: *PROP DYN-CALL* (*PROP P*) ⇒ *PROP P*
by (*simp* add: *DYN-CALL-def*)

lemma *runs-to-partial-top[corres-top]*: ⊤ · s ?{ Q }
by (*simp* add: *runs-to-partial-def-old*)

lemma *admissible-runs-to-partial[corres-admissible]*: *ccpo.admissible Inf* (≥) (λf . f · s ?{ Q })

unfolding *runs-to-partial-def-old*

apply (*rule ccpo.admissibleI*)

apply (*clarsimp simp* add: *ccpo.admissible-def chain-def*
succeeds-def reaches-def split: prod.splits rval-splits)

subgoal

apply *transfer*

apply (*clarsimp simp* add: *Inf-post-state-def top-post-state-def image-def vim-age-def*

split: if-split-asm)

by (*metis outcomes.simps(2) post-state.distinct(1)*)

done

lemma *reaches-gets-theE* [*simp*]:

reaches (gets-theE M) s rv s' \longleftrightarrow ($\exists v'$. $rv = \text{Result } v' \wedge s' = s \wedge M s = \text{Some } v'$)

by (*auto simp add: gets-theE-def*)

lemma *seucceds-gets-theE [simp]:*

succeeds (gets-theE M) s = ($\exists v$. $M s = \text{Some } v$)

apply (*auto simp add: gets-theE-def*)

done

lemma *L2-folded-gets-bind:*

L2-seq (L2-folded-gets (λ -. x) ns) f = f x

unfolding *L2-seq-def L2-folded-gets-def L2-gets-def*

apply (*rule spec-monad-eqI*)

apply (*auto simp add: runs-to-iff*)

done

lemma *L2-call-throw-names: L2-call x emb ns = (x <catch> (λ r. L2-throw (emb r) ns))*

unfolding *L2-call-def L2-throw-def*

apply (*rule spec-monad-eqI*)

apply (*clarsimp simp add: runs-to-iff*)

apply (*auto simp add: runs-to-def-old map-exn-def split: xval-splits*)

done

lemmas *L2-remove-scaffolding-1 =*

L2-folded-gets-bind [THEN eq-reflection]

L2-returncall-def

L2-modifycall-def

L2-voidcall-def

lemmas *L2-remove-scaffolding-2 =*

L2-remove-scaffolding-1

L2-folded-gets-def

abbreviation (*input*) *L2-guarded-while G C B i n \equiv L2-seq (L2-guard (G i))*

(λ -. L2-while C (λ i. L2-seq (B i) (λ r. L2-seq (L2-guard (G r)) (λ -. L2-gets (λ -. r) n))) i n)

lemmas *L2-defs = L2-unknown-def L2-seq-def*

L2-modify-def L2-gets-def L2-condition-def L2-catch-def L2-while-def

L2-throw-def L2-spec-def L2-assume-def L2-guard-def L2-fail-def L2-folded-gets-def

L2-voidcall-def L2-modifycall-def L2-returncall-def

L2-try-def

lemma *L2-defs'*:

L2-unknown $n \equiv \text{unknown}$
L2-seq $A' B' \equiv A' \gg B'$
L2-modify $m \equiv \text{modify } m$
L2-gets $f n \equiv \text{gets } f$
L2-condition $c L R \equiv \text{condition } c L R$
L2-catch $A B \equiv (A \langle \text{catch} \rangle B)$
L2-try $A \equiv \text{try } A$
L2-while $c' B'' i n \equiv \text{whileLoop } c' B'' i$
L2-throw $x n \equiv \text{throw } x$
L2-spec $r \equiv (\text{state-select } r \gg (\lambda-. \text{select UNIV}))$
L2-assume $r' \equiv (\text{assume-result-and-state } r')$
L2-guard $c \equiv \text{guard } c$
L2-fail $\equiv \text{fail}$
by (*auto simp*: *L2-defs unknown-def*)

definition

L2corres $:: ('s \Rightarrow 't) \Rightarrow ('s \Rightarrow 'r) \Rightarrow ('s \Rightarrow 'e) \Rightarrow ('s \Rightarrow \text{bool})$
 $\Rightarrow ('e, 'r, 't) \text{ exn-monad} \Rightarrow (\text{unit}, \text{unit}, 's) \text{ exn-monad} \Rightarrow \text{bool}$

where

L2corres $st \text{ ret-xf } ex\text{-xf } P A C$
 $\equiv \text{corresXF } st (\lambda-. \text{ret-xf}) (\lambda-. \text{ex-xf}) P A C$

lemma *admissible-nondet-ord-L2corres* [*corres-admissible*]:

ccpo.admissible *Inf* (\geq) ($\lambda A. \text{L2corres } st \text{ ret-xf } ex\text{-xf } P A C$)

unfolding *L2corres-def*

apply (*rule admissible-nondet-ord-corresXF*)

done

lemma *L2corres-top* [*corres-top*]: *L2corres* $st \text{ ret-xf } ex\text{-xf } P \top C$

by (*auto simp add: L2corres-def corresXF-def*)

definition

L2-call-L1 $arg\text{-xf } gs \text{ ret-xf } l1body$
 $= \text{do } \{$
 $s \leftarrow \text{get-state};$
 $t \leftarrow \text{select } \{t. gs \ t = s \wedge arg\text{-xf } t\};$
 $\text{run-bind } l1body \ t (\lambda r' t'.$
 $(\text{case } r' \text{ of}$
 $\text{Exception } - \Rightarrow \text{fail}$
 $| \text{Result } - \Rightarrow \text{do } \{\text{set-state } (gs \ t'); \text{return } (\text{ret-xf } t')\})$
 $\})$

lemma *L2corres-L2-call-L1*:

L2corres $gs \text{ ret-xf } ex\text{-xf } arg\text{-xf}$

```

    (L2-call-L1 arg-xf gs ret-xf f) f
  apply (clarsimp simp: L2corres-def corresXF-refines-iff L2-call-L1-def L1-seq-def
    L1-guard-def
      split: xval-splits)
  apply (simp add: refines-def-old)
  apply (intro conjI impI)
  subgoal
    by (auto simp add: succeeds-bind succeeds-run-bind succeeds-exec-concrete-iff
      succeeds-catch)
  subgoal
    by (auto simp add: succeeds-bind succeeds-run-bind reaches-run-bind
      succeeds-exec-concrete-iff succeeds-catch reaches-bind default-option-def Exn-def
      rel-XF-def rel-xval.simps Exn-def default-option-def
      split: exception-or-result-splits)

    (smt (verit, ccfv-threshold) Exn-def Exn-neq-Result exception-or-result-cases
      imageI mem-Collect-eq old.unit.exhaust option.exhaust-sel)
  done

```

lemma *L2corres-L2-call-simpl:*
 $l1-f \equiv simpl-f \implies$
L2corres gs ret-xf ex-xf arg-xf
(L2-call-L1 arg-xf gs ret-xf simpl-f) l1-f
 by (simp add: L2corres-L2-call-L1)

lemma *L2corres-modify-global:*
 $\llbracket \bigwedge s. P s \implies M (st s) = st (M' s) \rrbracket \implies$
L2corres st ret ex P (L2-modify M) (L1-modify M')
unfolding *L2-defs L1-defs*
 by (auto simp add: L2corres-def corresXF-modify-global)

lemma *L2corres-modify-unknown:*
 $\llbracket \bigwedge s. P s \implies st s = st (M s) \rrbracket \implies$
L2corres st ret ex P (L2-unknown ns) (L1-modify M)
 apply (clarsimp simp: L2corres-def L2-defs L1-defs)
 apply (auto intro: corresXF-select-modify)
 done

lemma *L2corres-spec-unknown:*
 $\llbracket \bigwedge s a. st s = st (M (a::('a \Rightarrow ('a::\{type\}))) s) \rrbracket \implies$
L2corres st ret ex P (L2-unknown ns) (L1-init M)
 apply (auto simp add: L2corres-def corresXF-def L1-defs L2-defs
 reaches-bind reaches-succeeds succeeds-bind)
 done

lemma *L2corres-modify-gets*:

$\llbracket \bigwedge s. P\ s \implies st\ s = st\ (M\ s); \bigwedge s. P\ s \implies ret\ (M\ s) = f\ (st\ s) \rrbracket \implies$
 $L2corres\ st\ ret\ ex\ P\ (L2\text{-}gets\ (\lambda s. f\ s)\ n)\ (L1\text{-}modify\ M)$
by (*simp add: L2corres-def L2-defs L1-defs corresXF-modify-gets*)

lemma *L2corres-gets-skip*:

$L2corres\ st\ ret\ ex\ P\ L2\text{-}skip\ L1\text{-}skip$
by (*simp add: L2corres-def L2-defs L1-defs corresXF-def*)

lemma *L2corres-guard*:

$\llbracket \bigwedge s. P\ s \implies G'\ s = G\ (st\ s) \rrbracket \implies$
 $L2corres\ st\ return\text{-}xf\ exception\text{-}xf\ P\ (L2\text{-}guard\ G)\ (L1\text{-}guard\ G')$
by (*simp add: L2corres-def L2-defs L1-defs corresXF-guard*)

lemma *L2corres-fail*:

$L2corres\ st\ return\text{-}xf\ exception\text{-}xf\ P\ L2\text{-}fail\ X$
by (*clarsimp simp add: L2corres-def L1-defs L2-defs corresXF-def*)

lemma *L2corres-spec*:

$\llbracket \bigwedge s\ s'. ((s, s') \in A') = ((st\ s, st\ s') \in A); surj\ st \rrbracket$
 $\implies L2corres\ st\ return\text{-}xf\ exception\text{-}xf\ P\ (L2\text{-}spec\ A)\ (L1\text{-}spec\ A')$
apply (*clarsimp simp: L2corres-def L2-defs L1-spec-def corresXF-def succeeds-bind reaches-bind*)
by (*metis surjD*)

lemma *L2corres-assume*:

$\llbracket \bigwedge s\ s'. ((((), s') \in A'\ s) = ((((), st\ s') \in A\ (st\ s))) \rrbracket$
 $\implies L2corres\ st\ return\text{-}xf\ exception\text{-}xf\ P\ (L2\text{-}assume\ A)\ (L1\text{-}assume\ A')$
by (*clarsimp simp: L2corres-def L2-defs L1-assume-def corresXF-def*)

lemma *L2corres-seq*:

$\llbracket L2corres\ st\ return\text{-}xf\ exception\text{-}xf\ P\ A\ A';$
 $\bigwedge x. L2corres\ st\ return\text{-}xf'\ exception\text{-}xf\ (P'\ x)\ (B\ x)\ B';$
 $\{\!|R\}\ A'\ \{\!\lambda\text{-} s. P'\ (return\text{-}xf\ s)\ s\}, \{\!\lambda\text{-} \text{-}. True\}\};$
 $\bigwedge s. R\ s \implies P\ s \rrbracket \implies$
 $L2corres\ st\ return\text{-}xf'\ exception\text{-}xf\ R\ (L2\text{-}seq\ A\ B)\ (L1\text{-}seq\ A'\ B')$
apply (*unfold L2corres-def L2-seq-def L1-seq-def validE-def*)
apply (*rule corresXF-join*)
apply *assumption*
apply *assumption*
apply *simp*
apply *assumption*
done

lemma *L2corres-guard-imp*:

$\llbracket L2corres\ st\ ret\text{-}state\ ex\text{-}state\ Q\ M\ M'; pred\text{-}imp\ P\ Q \rrbracket \implies$

L2corres st ret-state ex-state P M M'
by (*simp add: L2corres-def pred-imp-def corresXF-guard-imp*)

lemma *L2corres-guarded''*:
assumes *bdy*: $\bigwedge s' s. g' s' \implies s = st s' \implies P s' \implies L2corres\ st\ ret\ ex\ P\ (c\ (dest\ s))\ (c'\ (dest'\ s'))$
assumes *g-g'*: $\bigwedge s'. P\ s' \implies g' s' = g\ (st\ s')$
assumes *dest*: $\bigwedge s' s. g' s' \implies s = st s' \implies P s' \implies dest'\ s' = dest\ (st\ s')$
shows *L2corres st ret ex P (L2-guarded g (L2-seq (L2-gets dest [])) c)* (*L1-guarded g' (gets dest' \ggg c')*)
using *assms*
by (*auto simp add: L2corres-def L2-defs L1-defs L2-guarded-def L1-guarded-def corresXF-def succeeds-bind reaches-bind split: xval-splits*)

lemma *L2corres-catch*:
 $\llbracket L2corres\ st\ V\ E\ P\ L\ L';$
 $\bigwedge x. L2corres\ st\ V\ E'\ (P'\ x)\ (R\ x)\ R';$
 $\{\!|Q|\!\} L'\ \{\!\lambda\ -. True\}, \{\!\lambda\ s. P'\ (E\ s)\ s\}; \bigwedge s. Q\ s \implies P\ s \rrbracket \implies$
 $L2corres\ st\ V\ E'\ Q\ (L2-catch\ L\ R)\ (L1-catch\ L'\ R')$
apply (*clarsimp simp: L2corres-def L2-catch-def L1-catch-def validE-def*)
apply (*rule corresXF-exception*)
apply (*assumption*)
apply (*assumption*)
apply *auto*
done

lemma *L2corres-throw*:
 $\llbracket \bigwedge s. P\ s \implies E\ s = A \rrbracket \implies L2corres\ st\ V\ E\ P\ (L2-throw\ A\ n)\ (L1-throw)$
by (*clarsimp simp: L2corres-def L2-throw-def L1-throw-def corresXF-throw*)

lemma *L2corres-conseq*:
assumes *corres*: *L2corres st return-xf exception-xf P C C'*
assumes *conseq*: $\bigwedge s. Q\ s \implies P\ s$
shows *L2corres st return-xf exception-xf Q C C'*
apply (*rule L2corres-guard-imp [OF corres]*)
using *conseq by (simp add: pred-imp-def)*

lemma *L2corres-cond*:
 $\llbracket L2corres\ st\ return-xf\ exception-xf\ P\ A\ A';$
 $L2corres\ st\ return-xf\ exception-xf\ P'\ B\ B';$
 $\bigwedge s. R\ s \implies P\ s;$
 $\bigwedge s. R\ s \implies P'\ s;$
 $\bigwedge s. R\ s \implies c'\ s = c\ (st\ s) \rrbracket \implies$
 $L2corres\ st\ return-xf\ exception-xf\ R\ (L2-condition\ c\ A\ B)\ (L1-condition\ c'\ A'\ B')$
apply (*unfold L2corres-def L2-condition-def L1-condition-def*)
apply (*rule corresXF-cond*)

apply (*auto intro: corresXF-guard-imp*)
done

lemma *L2corres-inject-return*:

$\llbracket L2corres\ st\ V\ E\ P\ L\ R; \{\!\{P'\}\!\} R \{\!\{\lambda\ -\ s.\ W\ (V\ s) = V'\ s\}\!\}, \{\!\{\lambda\ -\ .\ True\}\!\}; \bigwedge s.\ P'\ s \implies P\ s \rrbracket \implies$
 $L2corres\ st\ V'\ E\ P'\ (L2\text{-seq}\ L\ (\lambda x.\ L2\text{-gets}\ (\lambda\ -.\ W\ x)\ n))\ R$
apply (*clarsimp simp: L2corres-def validE-def*)
apply (*drule corresXF-guard-imp [where P=P'], simp*)
apply (*unfold L2-seq-def L2-gets-def*)
apply (*rule corresXF-guard-imp*)
apply (*erule corresXF-append-gets-abs*)
apply *simp*
apply *simp*
done

lemma *L2corres-inject-return'*:

$\llbracket L2corres\ st\ V\ E\ P\ L\ R; Gets = (\lambda x.\ L2\text{-gets}\ (\lambda\ -.\ W\ x)\ n); \{\!\{P'\}\!\} R \{\!\{\lambda\ -\ s.\ W\ (V\ s) = V'\ s\}\!\}, \{\!\{\lambda\ -\ .\ True\}\!\}; \bigwedge s.\ P'\ s \implies P\ s \rrbracket \implies$
 $L2corres\ st\ V'\ E\ P'\ (L2\text{-seq}\ L\ Gets)\ R$
by (*auto intro: L2corres-inject-return*)

lemma *L2corres-skip*:

$L2corres\ st\ return\text{-xf}\ exception\text{-xf}\ P\ L2\text{-skip}\ L1\text{-skip}$
by (*rule L2corres-gets-skip*)

lemma *L2corres-while*:

assumes *body-corres*: $\bigwedge x.\ L2corres\ st\ ret\ ex\ (P'\ x)\ (A\ x)\ B$
and *inv-holds*: $\{\!\{\lambda s.\ P\ (ret\ s)\ s\}\!\} B \{\!\{\lambda\ -\ s.\ P\ (ret\ s)\ s\}\!\}, \{\!\{\lambda\ -\ .\ True\}\!\}$
and *cond-match*: $\bigwedge s.\ P\ (ret\ s)\ s \implies c'\ s = c\ (ret\ s)\ (st\ s)$
and *pred-imply*: $\bigwedge s\ x.\ P\ x\ s \implies P'\ x\ s$
and *pred-extract*: $\bigwedge s.\ P\ x\ s \implies ret\ s = x$
and *pred-imply2*: $\bigwedge s.\ Q\ x\ s \implies P\ x\ s$
shows $L2corres\ st\ ret\ ex\ (Q\ x)\ (L2\text{-while}\ c\ A\ x\ n)\ (L1\text{-while}\ c'\ B)$
apply (*clarsimp simp: L2corres-def L2-while-def L1-while-def*)
apply (*rule corresXF-guard-imp*)
apply (*rule corresXF-while [*
where $P = \lambda r\ s.\ P\ (ret\ s)\ s$ **and** $C' = c$ **and** $C = \lambda\ -.\ c'$ **and** $A = A$ **and** $B = \lambda\ -.$
 B
and $ret = \lambda r\ s.\ ret\ s$ **and** $ex = \lambda r\ s.\ ex\ s$ **and** $st = st$ **and** $y = x$ **and** $x = ()$ **and**
 $P' = \lambda r\ s.\ Q\ x\ s]$)
apply (*rule corresXF-guard-imp*)
apply (*rule body-corres [unfolded L2corres-def]*)
apply (*clarsimp simp: pred-imply*)
apply (*clarsimp simp: cond-match*)
apply (*rule inv-holds [unfolded validE-def, rule-format]*)
apply *assumption*
apply (*metis pred-extract pred-imply2*)

apply (*metis pred-extract pred-implly2*)
apply *simp*
done

lemma *L2corres-returncall*:

\llbracket *L2corres st ret' ex' P' Z dest-fn*;
 $\bigwedge s. P s \implies P' (\text{scope-setup } s)$;
 $\bigwedge t s. st (\text{return-xf } t (\text{scope-teardown } s t)) = st t$;
 $\bigwedge t s. st (\text{result-exn } (\text{scope-teardown } s t) t) = st t$;
 $\bigwedge s. st (\text{scope-setup } s) = st s$;
 $\bigwedge t s. st (\text{scope-teardown } s t) = st t$;
 $\bigwedge t s. P s \implies \text{ret } (\text{return-xf } t (\text{scope-teardown } s t)) = F (\text{ret}' t) (st t)$;
 $\bigwedge t s. P s \implies (\text{ex } (\text{result-exn } (\text{scope-teardown } s t) t)) = \text{emb } (\text{ex}' t) \rrbracket \implies$
L2corres st ret ex P (L2-returncall Z F emb ns)

(*L1-call scope-setup dest-fn scope-teardown result-exn return-xf*)

unfolding *L1-call-def L1-seq-def L1-throw-def L2-returncall-def L2-call-def L2corres-def L2-defs*

apply (*rule corresXF-I*)

subgoal

apply (*clarsimp simp add: corresXF-def succeeds-bind succeeds-catch reaches-catch*

reaches-map-value reaches-bind default-option-def Exn-def split: xval-splits exception-or-result-splits)

by (*smt (z3) Exn-def Result-neq-Exn map-exn-simps(2) the-Result-simp*)

subgoal

apply (*clarsimp simp add: corresXF-def succeeds-bind succeeds-catch reaches-catch*

reaches-map-value reaches-bind default-option-def Exn-def split: xval-splits exception-or-result-splits)

by (*metis (mono-tags, opaque-lifting) Exception-eq-Exception Exn-def Exn-neq-Result map-exn-simps(1)*)

subgoal

apply (*auto simp add: corresXF-def succeeds-bind succeeds-catch reaches-catch*

reaches-map-value reaches-bind default-option-def Exn-def split: xval-splits exception-or-result-splits)

done

done

lemma *L2corres-dest-fn-simp*:

assumes *dest-fn: $\bigwedge s. P s \implies \text{dest-fn} = \text{dest-fn}'$*

assumes *corres: L2corres st ret ex P X*

(*L1-call scope-setup (measure-call dest-fn') scope-teardown result-exn return-xf*)

shows *L2corres st ret ex P X*

(*L1-call scope-setup (measure-call dest-fn) scope-teardown result-exn return-xf*)

using *corres dest-fn unfolding L2corres-def corresXF-def*

by *blast*

lemma *L2corres-l2-propagate-fixed-cong*:
 $(\bigwedge s. P\ s = P'\ s) \implies (\bigwedge s. P'\ s \implies A = A') \implies L2corres\ st\ ret\ ex\ P\ A\ C =$
 $L2corres\ st\ ret\ ex\ P'\ A'\ C$

unfolding *L2corres-def corresXF-def simp-implies-def*
by (*auto split: sum.splits prod.splits*)

lemma *L2corres-modifycall*:

$\llbracket L2corres\ st\ ret'\ ex'\ P'\ Z\ dest\ fn;$
 $\bigwedge s. P\ s \implies P'\ (scope\ setup\ s);$
 $\bigwedge s\ r. P\ r \implies F\ (ret'\ s)\ (st\ s) = st\ (return\ xf\ s\ (scope\ teardown\ r\ s));$
 $\bigwedge s. st\ (scope\ setup\ s) = st\ s;$
 $\bigwedge t\ s. st\ (scope\ teardown\ s\ t) = st\ t;$
 $\bigwedge t\ s. st\ (result\ exn\ (scope\ teardown\ s\ t)\ t) = st\ t;$
 $\bigwedge t\ s. P\ s \implies ex\ (result\ exn\ (scope\ teardown\ s\ t)\ t) = emb\ (ex'\ t)\rrbracket \implies$
 $L2corres\ st\ ret\ ex\ P\ (L2\ modifycall\ Z\ F\ emb\ ns)$
(L1-call scope-setup dest-fn scope-teardown result-exn return-xf)

apply (*clarsimp simp: L1-call-def L1-seq-def L1-throw-def L2-call-def L2-defs*
L2corres-def)

apply (*rule corresXF-I*)

subgoal

apply (*clarsimp simp add: corresXF-def succeeds-bind succeeds-catch reaches-catch*

reaches-map-value reaches-bind default-option-def Exn-def split: xval-splits
exception-or-result-splits)

by (*smt (z3) Exn-def Result-neq-Exn map-exn-simps(2) the-Result-simp*)

subgoal

apply (*clarsimp simp add: corresXF-def succeeds-bind succeeds-catch reaches-catch*

reaches-map-value reaches-bind default-option-def Exn-def split: xval-splits
exception-or-result-splits)

by (*metis (mono-tags, opaque-lifting) Exception-eq-Exception Exn-def Exn-neq-Result*
map-exn-simps(1))

subgoal

apply (*auto simp add: corresXF-def succeeds-bind succeeds-catch reaches-catch*
reaches-map-value reaches-bind default-option-def Exn-def split: xval-splits
exception-or-result-splits)

done

done

lemma *L2corres-voidcall*:

$\llbracket L2corres\ st\ ret'\ ex'\ P'\ Z\ dest\ fn;$
 $\bigwedge s. P\ s \implies P'\ (scope\ setup\ s);$
 $\bigwedge s\ r. st\ (return\ xf\ s\ (scope\ teardown\ r\ s)) = st\ s;$
 $\bigwedge s. st\ (scope\ setup\ s) = st\ s;$
 $\bigwedge t\ s. st\ (scope\ teardown\ s\ t) = st\ t;$
 $\bigwedge t\ s. st\ (result\ exn\ (scope\ teardown\ s\ t)\ t) = st\ t;$
 $\bigwedge t\ s. P\ s \implies ex\ (result\ exn\ (scope\ teardown\ s\ t)\ t) = emb\ (ex'\ t)\rrbracket \implies$

```

    L2corres st ret ex P (L2-voidcall Z emb ns)
    (L1-call scope-setup dest-fn scope-teardown result-exn return-xf)
apply (unfold L2-voidcall-def)
apply (rule subst[where t = L2-skip and s = L2-modify (λs. s)])
subgoal
  apply (simp add: L2-defs)
  apply (rule spec-monad-ext)
  apply simp
  done
apply (fold L2-modifycall-def L2corres-def)
apply (fastforce elim!: L2corres-modifycall)
done

```

lemma *L2corres-call*:

```

[[ L2corres st ret' ex' P' Z dest-fn;
  ∧ s. P s ⇒ P' (scope-setup s);
  ∧ s r. st (return-xf s (scope-teardown r s)) = st s;
  ∧ s r. ret (return-xf s (scope-teardown r s)) = ret' s;
  ∧ s. st (scope-setup s) = st s;
  ∧ t s. st (scope-teardown s t) = st t;
  ∧ t s. st (result-exn (scope-teardown s t) t) = st t;
  ∧ t s. P s ⇒ ex (result-exn (scope-teardown s t) t) = emb (ex' t) ]] ⇒
  L2corres st ret ex P (L2-call Z emb ns)
  (L1-call scope-setup dest-fn scope-teardown result-exn return-xf)
apply (clarsimp simp: L2corres-def L2-call-def L1-call-def L1-seq-def L1-throw-def
L2-defs)
apply (rule corresXF-I)
subgoal
  apply (clarsimp simp add: corresXF-def succeeds-bind succeeds-catch reaches-catch
    reaches-map-value reaches-bind default-option-def Exn-def split: xval-splits
    exception-or-result-splits)
  by (smt (z3) Exn-def Result-neq-Exn map-exn-simps(2) the-Result-simp)
subgoal
  apply (clarsimp simp add: corresXF-def succeeds-bind succeeds-catch reaches-catch
    reaches-map-value reaches-bind default-option-def Exn-def split: xval-splits
    exception-or-result-splits)
  by (metis (mono-tags, opaque-lifting) Exception-eq-Exception Exn-def Exn-neq-Result
    map-exn-simps(1))
subgoal
  apply (auto simp add: corresXF-def succeeds-bind succeeds-catch reaches-catch
    reaches-map-value reaches-bind default-option-def Exn-def split: xval-splits
    exception-or-result-splits)
  done
done

```

lemma (in *L1-functions*) *L2corres-dyn-call*:

L2corres st ret ex P X
(L1-guarded g (gets dest >>= (λp. L1-call scope-setup (P p) scope-teardown result-exn return-xf))) =>
L2corres st ret ex P X (L1-dyn-call g scope-setup dest scope-teardown result-exn return-xf)
by (*simp add: L1-dyn-call-def*)

lemma *L2-gets-bind*: $\llbracket \bigwedge s s'. V s = V s' \rrbracket \implies L2\text{-seq } (L2\text{-gets } V n) f = f (V \text{undefined})$
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*simp add: runs-to-iff*)
by (*auto simp add: runs-to-def-old metis+*)

lemma *L2corres-folded-gets*:
 $\llbracket \bigwedge a. L2\text{corres st ret ex } (P \text{ and } (\lambda s. a = c (st s))) (X a) Y \rrbracket \implies$
 $L2\text{corres st ret ex } P (L2\text{-seq } (L2\text{-folded-gets } c ns) X) Y$
by (*fastforce simp add: L2-defs L2corres-def corresXF-def succeeds-bind reaches-bind split: xval-splits*)

lemma *L2-call-cong* [*fundef-cong, cong*]:
 $f = f' \implies L2\text{-call } f = L2\text{-call } f'$
by *simp*

lemma *L2-call-liftE* [*simp*]:
 $L2\text{-call } (liftE x) \text{ emb } ns \equiv liftE x$
by (*clarsimp simp add: L2-call-def*)

lemma *L2-call-fail* [*simp*]: $L2\text{-call fail emb ns} = fail$
apply (*simp add: L2-call-def*)
done

lemma *L2-call-L2-gets* [*simp*]: $L2\text{-call } (L2\text{-gets } x n) \text{ emb } ns = L2\text{-gets } x n$
apply (*simp add: L2-defs L2-call-def*)
done

lemma *L2-split-fixup-1*:
 $(L2\text{-seq } A (\lambda x. \text{case } y \text{ of } (a, b) \Rightarrow B a b x)) =$
 $(\text{case } y \text{ of } (a, b) \Rightarrow L2\text{-seq } A (\lambda x. B a b x))$
by (*auto simp: split-def*)

lemma *L2-split-fixup-2*:
 $(L2\text{-seq } (\text{case } y \text{ of } (a, b) \Rightarrow B a b) A) =$
 $(\text{case } y \text{ of } (a, b) \Rightarrow L2\text{-seq } (B a b) A)$

by (*auto simp: split-def*)
lemma *L2-split-fixup-3*:
 $(\text{case } (x, y) \text{ of } (a, b) \Rightarrow P a b) = P x y$
by (*auto simp: split-def*)
lemma *L2-split-fixup-4*:
 $\text{case-prod } (\lambda a (b :: 'a \times 'b). P a) = \text{case-prod } (\lambda a. \text{case-prod } (\lambda(x :: 'a) (y :: 'b). P a))$
by (*auto simp: split-def*)
lemma *L2-split-fixup-f*:
 $(f (\text{case } y \text{ of } (a, b) \Rightarrow G a b) =$
 $(\text{case } y \text{ of } (a, b) \Rightarrow f (G a b)))$
by (*auto simp: split-def*)
lemma *L2-split-fixup-g*:
 $\text{case-prod } (\lambda a (b :: 'a \times 'b). P a b) = \text{case-prod } (\lambda a. \text{case-prod } (\lambda(x :: 'a) (y :: 'b). P a (x, y)))$
by (*auto simp: split-def*)

lemma *liftE-fixup*: $(\lambda x. \text{liftE } (\text{case } x \text{ of } (a, b) \Rightarrow f a b)) = (\lambda(a,b). \text{liftE } (f a b))$
by (*simp add: fun-eq-iff split: prod.splits*)

lemma *finally-fixup*: $(\lambda x. \text{finally } (\text{case } x \text{ of } (a, b) \Rightarrow f a b)) = (\lambda(a,b). \text{finally } (f a b))$
by (*simp add: fun-eq-iff split: prod.splits*)

lemma *try-fixup*: $(\lambda x. \text{try } (\text{case } x \text{ of } (a, b) \Rightarrow f a b)) = (\lambda(a,b). \text{try } (f a b))$
by (*simp add: fun-eq-iff split: prod.splits*)

lemmas *L2-split-fixups* =
L2-split-fixup-1
L2-split-fixup-2
L2-split-fixup-3
L2-split-fixup-4
liftE-fixup
finally-fixup
try-fixup

L2-split-fixup-f [**where** $f=L2\text{-guard}$]
L2-split-fixup-f [**where** $f=L2\text{-gets}$]
L2-split-fixup-f [**where** $f=L2\text{-modify}$]

L2-split-fixup-g [**where** $P=\lambda a b. L2\text{-gets } (P a b) n$ **for** $P n$]
L2-split-fixup-g [**where** $P=\lambda a b. L2\text{-guard } (P a b)$ **for** P]
L2-split-fixup-g [**where** $P=\lambda a b. L2\text{-modify } (P a b)$ **for** P]
L2-split-fixup-g [**where** $P=\lambda a b. L2\text{-spec } (P a b)$ **for** P]
L2-split-fixup-g [**where** $P=\lambda a b. L2\text{-assume } (P a b)$ **for** P]
L2-split-fixup-g [**where** $P=\lambda a b. L2\text{-throw } (P a b) n$ **for** $P n$]

L2-split-fixup-g [**where** $P=\lambda a b. L2\text{-seq } (L a b) (R a b)$ **for** $L R$]

L2-split-fixup-g [**where** $P = \lambda a b. L2\text{-while } (C a b) (B a b) (I a b) n$ **for** $C B I n$]
L2-split-fixup-g [**where** $P = \lambda a b. L2\text{-unknown } n$ **for** n]
L2-split-fixup-g [**where** $P = \lambda a b. L2\text{-catch } (L a b) (R a b)$ **for** $L R$]
L2-split-fixup-g [**where** $P = \lambda a b. L2\text{-condition } (C a b) (L a b) (R a b)$ **for** $C L R$]
L2-split-fixup-g [**where** $P = \lambda a b. L2\text{-call } (M a b)$ **for** M]
L2-split-fixup-g [**where** $P = \lambda a b. liftE (M a b)$ **for** M]
L2-split-fixup-g [**where** $P = \lambda a b. finally (M a b)$ **for** M]
L2-split-fixup-g [**where** $P = \lambda a b. try (M a b)$ **for** M]
lemmas *L2-split-fixups-congs* =
prod.case-cong

named-theorems *L2opt*

lemma *monotone-nondet-monad-L2-seq-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone* $R (\leq) X$
assumes *mono-Y*: $\bigwedge r. \text{monotone } R (\leq) (\lambda f. Y f r)$
shows *monotone* $R (\leq)$
 $(\lambda f. (L2\text{-seq } (X f) (Y f)))$
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-nondet-monad-L2-seq-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone* $R (\geq) X$
assumes *mono-Y*: $\bigwedge r. \text{monotone } R (\geq) (\lambda f. Y f r)$
shows *monotone* $R (\geq)$
 $(\lambda f. (L2\text{-seq } (X f) (Y f)))$
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-nondet-monad-L2-try-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone* $R (\leq) X$
shows *monotone* $R (\leq)$
 $(\lambda f. (L2\text{-try } (X f)))$
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
done

lemma *monotone-nondet-monad-L2-try-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone* $R (\geq) X$

shows *monotone* $R (\geq)$
 ($\lambda f. (L2\text{-try } (X f))$)
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
done

lemma *monotone-nondet-monad-L2-catch-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone* $R (\leq) X$
assumes *mono-Y*: $\bigwedge r. \text{monotone } R (\leq) (\lambda f. Y f r)$
shows *monotone* $R (\leq)$
 ($\lambda f. (L2\text{-catch } (X f) (Y f))$)
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-nondet-monad-L2-catch-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone* $R (\geq) X$
assumes *mono-Y*: $\bigwedge r. \text{monotone } R (\geq) (\lambda f. Y f r)$
shows *monotone* $R (\geq)$
 ($\lambda f. (L2\text{-catch } (X f) (Y f))$)
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-nondet-monad-L2-condition-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone* $R (\leq) X$
assumes *mono-Y*: *monotone* $R (\leq) Y$
shows *monotone* $R (\leq)$
 ($\lambda f. (L2\text{-condition } C (X f) (Y f))$)
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-nondet-monad-L2-condition-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone* $R (\geq) X$
assumes *mono-Y*: *monotone* $R (\geq) Y$
shows *monotone* $R (\geq)$
 ($\lambda f. (L2\text{-condition } C (X f) (Y f))$)
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)

done

lemma *monotone-nondet-monad-L2-guarded-le* [*partial-function-mono*]:
 assumes *mono-X*[*partial-function-mono*]: *monotone R* (\leq) *X*
 shows *monotone R* (\leq)
 ($\lambda f. L2\text{-guarded } C (X f)$)
 unfolding *L2-guarded-def*
 apply (*rule partial-function-mono*)+
 done

lemma *monotone-nondet-monad-L2-guarded-ge* [*partial-function-mono*]:
 assumes *mono-X*[*partial-function-mono*]: *monotone R* (\geq) *X*
 shows *monotone R* (\geq)
 ($\lambda f. L2\text{-guarded } C (X f)$)
 unfolding *L2-guarded-def*
 apply (*rule partial-function-mono*)+
 done

lemma *monotone-nondet-monad-L2-while-le* [*partial-function-mono*]:
 assumes *mono-B*: $\bigwedge r. \textit{monotone } R (\leq) (\lambda f. B f r)$
 shows *monotone R* (\leq) ($\lambda f. L2\text{-while } C (B f) I ns$)
 unfolding *L2-defs*
 apply (*rule partial-function-mono*)
 apply (*rule mono-B*)
 done

lemma *monotone-nondet-monad-L2-while-ge* [*partial-function-mono*]:
 assumes *mono-B*: $\bigwedge r. \textit{monotone } R (\geq) (\lambda f. B f r)$
 shows *monotone R* (\geq) ($\lambda f. L2\text{-while } C (B f) I ns$)
 unfolding *L2-defs*
 apply (*rule partial-function-mono*)
 apply (*rule mono-B*)
 done

lemma *monotone-nondet-monad-map-exn-le* [*partial-function-mono*]:
 assumes *X*[*partial-function-mono*]: *monotone R* (\leq) *X*
 shows *monotone R* (\leq) ($\lambda f. \textit{map-value (map-exn emb) (X f)}$)
 unfolding *map-exn-def*
 apply (*rule partial-function-mono*)+
 done

lemma *monotone-nondet-monad-map-exn-ge* [*partial-function-mono*]:
 assumes *X*[*partial-function-mono*]: *monotone R* (\geq) *X*
 shows *monotone R* (\geq) ($\lambda f. \textit{map-value (map-exn emb) (X f)}$)
 unfolding *map-exn-def*
 apply (*rule partial-function-mono*)+
 done

lemma *monotone-nondet-monad-L2-call-le* [*partial-function-mono*]:
assumes X [*partial-function-mono*]: *monotone* $R (\leq) X$
shows *monotone* $R (\leq)$
 $(\lambda f. L2\text{-call } (X f) \text{ emb } ns)$
unfolding *L2-call-def*
apply (*rule partial-function-mono*)+
done

lemma *monotone-nondet-monad-L2-call-ge* [*partial-function-mono*]:
assumes X [*partial-function-mono*]: *monotone* $R (\geq) X$
shows *monotone* $R (\geq)$
 $(\lambda f. L2\text{-call } (X f) \text{ emb } ns)$
unfolding *L2-call-def*
apply (*rule partial-function-mono*)+
done

19.1 Some Relators

lemma *exists-sum-unit-eq*: $(\exists l'::\text{unit}. \text{Inl } x = \text{Inl } l' \wedge \text{Inr } y = \text{Inr } l') = (x=()) \wedge y=())$
by *auto*

lemmas *unit-convs = exists-sum-unit-eq*

definition *rel-Nonlocal* = $(\lambda e v. \text{case } e \text{ of Nonlocal } x \Rightarrow v = x \mid - \Rightarrow \text{False})$

lemma *rel-Nonlocal-conv*: $(\text{rel-Nonlocal } e v) = (e = \text{Nonlocal } v)$
by (*cases e*) (*auto simp add: rel-Nonlocal-def*)

lemma *fun-of-rel-rel-Nonlocal*[*fun-of-rel-intros*]: *fun-of-rel rel-Nonlocal the-Nonlocal*
by (*auto simp add: fun-of-rel-def rel-Nonlocal-def split: c-exntype.splits*)

lemma *funp-rel-Nonlocal*[*funp-intros, corres-admissible*]: *funp rel-Nonlocal*
using *fun-of-rel-rel-Nonlocal*
by (*auto simp add: funp-def*)

lemma *rel-sum-rel-Nonlocal-case-InlI*: $e = \text{Nonlocal } v' \Longrightarrow \text{rel-sum rel-Nonlocal}$
 $(=) (\text{case } e \text{ of Nonlocal } v \Rightarrow \text{Inl } (\text{Nonlocal } v) \mid - \Rightarrow \text{Inr } x) (\text{Inl } v')$
by (*simp add: rel-Nonlocal-def*)

lemma *rel-xval-rel-Nonlocal-case-ExnI*: $e = \text{Nonlocal } v' \Longrightarrow \text{rel-xval rel-Nonlocal}$
 $(=) (\text{case } e \text{ of Nonlocal } v \Rightarrow \text{Exn } (\text{Nonlocal } v) \mid - \Rightarrow \text{Result } x) (\text{Exn } v')$
by (*simp add: rel-Nonlocal-def*)

lemma *rel-sum-rel-Nonlocal-case-InrI*: *is-local* $e \Longrightarrow x = v' \Longrightarrow \text{rel-sum rel-Nonlocal}$
 $(=) (\text{case } e \text{ of Nonlocal } v \Rightarrow \text{Inl } (\text{Nonlocal } v) \mid - \Rightarrow \text{Inr } x) (\text{Inr } v')$
apply (*cases e*)
apply (*auto simp add: rel-Nonlocal-def*)

done

lemma *rel-xval-rel-Nonlocal-case-ResultI*: $is\text{-local } e \implies x = v' \implies rel\text{-xval } rel\text{-Nonlocal}$
 $(=) (case\ e\ of\ Nonlocal\ v \Rightarrow Exn\ (Nonlocal\ v) \mid - \Rightarrow Result\ x) (Result\ v')$
apply (*cases e*)
apply (*auto simp add: rel-Nonlocal-def*)
done

lemma *rel-sum-rel-Nonlocal-map-sumI*: $v = (map\text{-sum } (\lambda x. x) f (case\ e\ of\ Non\text{-local } x \Rightarrow Inl\ x \mid - \Rightarrow Inr\ x)) \implies$
 $rel\text{-sum } rel\text{-Nonlocal } (=) (map\text{-sum } Nonlocal\ f (case\ e\ of\ Nonlocal\ x \Rightarrow Inl\ x \mid - \Rightarrow Inr\ x))\ v$
apply *simp*
apply (*cases e*)
apply (*auto simp add: rel-Nonlocal-def*)
done

lemma *rel-xval-rel-Nonlocal-map-xvalI*: $v = (map\text{-xval } (\lambda x. x) f (case\ e\ of\ Nonlocal\ x \Rightarrow Exn\ x \mid - \Rightarrow Result\ x)) \implies$
 $rel\text{-xval } rel\text{-Nonlocal } (=) (map\text{-xval } Nonlocal\ f (case\ e\ of\ Nonlocal\ x \Rightarrow Exn\ x \mid - \Rightarrow Result\ x))\ v$
apply *simp*
apply (*cases e*)
apply (*auto simp add: rel-Nonlocal-def*)
done

definition *rel-Inr* $v\ v' \equiv (v = Inr\ v')$

lemma *fun-of-rel-rel-Inr[fun-of-rel-intros]*:
 $fun\text{-of-rel } rel\text{-Inr } (\lambda x. case\ x\ of\ Inr\ v \Rightarrow v \mid Inl\ - \Rightarrow undefined)$
by (*auto simp add: rel-Inr-def fun-of-rel-def*)

lemma *funp-rel-Inr[funp-intros, corres-admissible]*: $funp\ rel\text{-Inr}$
using *fun-of-rel-rel-Inr* **by** (*auto simp add: funp-def*)

lemma *rel-InrI*: $v = Inr\ v' \implies rel\text{-Inr } v\ v'$
by (*simp add: rel-Inr-def*)

lemma *rel-Inr-apply*: $rel\text{-Inr } x\ y = (x = Inr\ y)$
by (*simp add: rel-Inr-def*)

lemma *rel-Inr-trivial [simp]*: $rel\text{-Inr } (Inr\ v)\ v$
by (*simp add: rel-Inr-def*)

definition *rel-liftE*:: $(e, 'a) xval \Rightarrow 'a\ val \Rightarrow bool$ **where** $rel\text{-liftE } v\ v' \equiv (v =$

Result (the-Res v')

lemma *fun-of-rel-rel-liftE*[*fun-of-rel-intros*]:
fun-of-rel rel-liftE ($\lambda x. \text{case } x \text{ of } \text{Result } v \Rightarrow \text{Result } v \mid \text{Err } - \Rightarrow \text{undefined}$)
by (*auto simp add: rel-liftE-def fun-of-rel-def*)

lemma *funp-rel-liftE*[*funp-intros, corres-admissible*]: *funp rel-liftE*
using *fun-of-rel-rel-liftE* **by** (*auto simp add: funp-def*)

lemma *rel-liftEI*: $v = \text{Result } (\text{the-Res } v') \implies \text{rel-liftE } v \ v'$
by (*simp add: rel-liftE-def*)

lemma *rel-liftE-apply*: $\text{rel-liftE } x \ y = (x = \text{Result } (\text{the-Res } y))$
by (*simp add: rel-liftE-def*)

lemma *rel-liftE-trivial* [*simp*]: $\text{rel-liftE } (\text{Result } v) (\text{Result } v)$
by (*simp add: rel-liftE-def*)

lemma *rel-liftE-rel-exception-or-result-conv*: $\text{rel-liftE} = \text{rel-exception-or-result } (\lambda \text{None} \Rightarrow \lambda -. \text{True} \mid \text{Some } - \Rightarrow \lambda -. \text{False}) (=)$
apply (*rule ext*)
apply (*auto simp add: rel-liftE-def rel-exception-or-result.simps*)
done

lemma *rel-liftE-Result-Result-iff*[*simp*]: $\text{rel-liftE } (\text{Result } v) (\text{Result } v') \longleftrightarrow v = v'$
by (*auto simp add: rel-liftE-def*)

definition $\text{rel-liftE}' \ v \ v' \equiv (v = \text{Inr } v')$

lemma *fun-of-rel-rel-liftE'*[*fun-of-rel-intros*]:
fun-of-rel rel-liftE' ($\lambda x. \text{case } x \text{ of } \text{Inr } v \Rightarrow v \mid \text{Inl } - \Rightarrow \text{undefined}$)
by (*auto simp add: rel-liftE'-def fun-of-rel-def*)

lemma *funp-rel-liftE'*[*funp-intros, corres-admissible*]: *funp rel-liftE'*
using *fun-of-rel-rel-liftE'* **by** (*auto simp add: funp-def*)

lemma *rel-liftE'I*: $v = \text{Inr } v' \implies \text{rel-liftE}' \ v \ v'$
by (*simp add: rel-liftE'-def*)

lemma *rel-liftE'-apply*: $\text{rel-liftE}' \ x \ y = (x = \text{Inr } y)$
by (*simp add: rel-liftE'-def*)

lemma *rel-liftE'-trivial* [*simp*]: $\text{rel-liftE}' (\text{Inr } v) \ v$
by (*simp add: rel-liftE'-def*)

lemma *rel-liftE'-Inr-iff*[*simp*]: $\text{rel-liftE}' (\text{Inr } v) \ v' \longleftrightarrow v = v'$
by (*auto simp add: rel-liftE'-def*)

lemma *rel-liftE'-rel-liftE-conv*: *rel-liftE' = rel-map to-xval OO rel-liftE OO rel-map the-Res*

apply (*rule ext*)+
apply (*auto simp add: relcompp.simps rel-map-def rel-liftE-def rel-liftE'-def*)
done

lemma *rel-liftE-rel-liftE'-conv*: *rel-liftE = rel-map from-xval OO rel-liftE' OO rel-map Result*

apply (*rule ext*)+
apply (*auto simp add: relcompp.simps rel-map-def rel-liftE-def rel-liftE'-def*)
done

lemma *rel-liftE'-Inr*: *rel-liftE' (Inr x) x*

by (*simp*)

definition *rel-throwE'* :: *('a ⇒ 'b ⇒ bool) ⇒ 'a + 'c ⇒ 'b ⇒ bool where*
rel-throwE' L a c ⟷ (case a of Inl a' ⇒ L a' c | Inr - ⇒ False)

lemma *rel-throwE'-iff*: *rel-throwE' L a c ⟷ (∃ l. a = Inl l ∧ L l c)*

by (*auto simp add: rel-throwE'-def split: sum.splits*)

lemma *rel-throwE'-Inl*:

L x y ⟹ rel-throwE' L (Inl x) y

unfolding *rel-throwE'-def*

by *auto*

lemma *rel-throwE'-Inl-iff[simp]*:

rel-throwE' L (Inl x) y ⟷ L x y

unfolding *rel-throwE'-def*

by *auto*

lemma *not-rel-throwE'-Inr[simp]*: *¬rel-throwE' R (Inr d) a*

by (*auto simp: rel-throwE'-def*)

definition *rel-throwE* :: *('a ⇒ 'b ⇒ bool) ⇒ ('a, 'c) xval ⇒ 'b ⇒ bool where*

rel-throwE L a c ⟷ (case a of Exn a' ⇒ L a' c | Result - ⇒ False)

lemma *rel-throwE-iff*: *rel-throwE L a c ⟷ (∃ e. a = Exn e ∧ L e c)*

by (*auto simp add: rel-throwE-def split: xval-splits*)

lemma *rel-throwE-Exn*:

L x y ⟹ rel-throwE L (Exn x) y

unfolding *rel-throwE-def*

by *auto*

lemma *rel-throwE-Exn-iff[simp]*:

rel-throwE $L (Exn\ x) y \longleftrightarrow L\ x\ y$
unfolding *rel-throwE-def*
by *auto*

lemma *not-rel-throwE-Result[simp]*: $\neg rel\text{-throwE}\ R\ (Result\ d)\ a$
by (*auto simp: rel-throwE-def*)

lemma *case-right-eq*: $(case\ v\ of\ Inl\ l \Rightarrow False\ |\ Inr\ r \Rightarrow P\ r) = (\exists r. P\ r \wedge v = Inr\ r)$

by (*auto split: sum.splits*)

lemma *case-left-eq*: $(case\ v\ of\ Inr\ r \Rightarrow False\ |\ Inl\ l \Rightarrow P\ l) = (\exists l. P\ l \wedge v = Inl\ l)$
by (*auto split: sum.splits*)

lemma *rel-sum-right*: $rel\text{-sum}\ L\ R\ (Inr\ r)\ v = (\exists r'. v = Inr\ r' \wedge R\ r\ r')$

by (*auto elim: rel-sum.cases intro: rel-sum.intros split: sum.splits*)

lemma *rel-sum-left*: $rel\text{-sum}\ L\ R\ (Inl\ l)\ v = (\exists l'. v = Inl\ l' \wedge L\ l\ l')$

by (*auto elim: rel-sum.cases intro: rel-sum.intros split: sum.splits*)

lemma *rel-sum-eq-apply*: $(rel\text{-sum}\ (=)\ (=)\ x\ y) = (x = y)$

by (*cases x (auto elim: rel-sum.cases intro: rel-sum.intros split: sum.splits)*)

lemma *gen-unit-eq*: $x = (y::unit)$

by *simp*

lemma *liftE-L2-while*: $L2\text{-while}\ c\ (\lambda r. liftE\ (B\ r))\ i\ n = liftE\ (whileLoop\ c\ B\ i)$

by (*clarsimp simp: L2-while-def liftE-whileLoop*)

lemma *liftE-L2-while-VARS*: $L2\text{-while}\ c\ (\lambda r. liftE\ (B\ r))\ i\ n = liftE\ (L2\text{-VARS}\ (whileLoop\ c\ B\ i)\ n)$

by (*simp add: liftE-L2-while L2-VARS-def*)

lemma *rel-XF-True-def*: $(rel\text{-XF}\ st\ ret\text{-xf}\ ex\text{-xf}\ (\lambda\ -. True)) =$

$(\lambda(v, t)\ (r, s). s = st\ t \wedge$

$(case\ v\ of\ Exn\ x \Rightarrow r = Exn\ (ex\text{-xf}\ x\ t) | Result\ x \Rightarrow r = Result\ (ret\text{-xf}\ x\ t)))$

apply (*rule ext*)**+**

apply (*auto simp add: rel-XF-def rel-xval.simps split: xval-splits*)

done

lemma *refines-corresXF*: $(\forall t. P\ t \longrightarrow refines\ C\ A\ t\ (st\ t)\ (\lambda(v, t)\ (r, s). s = st\ t \wedge$

$(case\ v\ of\ Exn\ x \Rightarrow r = Exn\ (ex\text{-xf}\ x\ t) | Result\ x \Rightarrow r = Result\ (ret\text{-xf}\ x\ t))))$

$\Longrightarrow corresXF\ st\ ret\text{-xf}\ ex\text{-xf}\ P\ A\ C$

by (*simp add: corresXF-refines-iff rel-XF-True-def*)

lemma *corresXF-refines*: $corresXF\ st\ ret\text{-xf}\ ex\text{-xf}\ P\ A\ C \Longrightarrow$

$(\forall t. P\ t \longrightarrow refines\ C\ A\ t\ (st\ t)\ (\lambda(v, t)\ (r, s). s = st\ t \wedge$

(*case v of Exn x ⇒ r = Exn (ex-xf x t) | Result x ⇒ r = Result (ret-xf x t)*))
by (*simp add: corresXF-refines-iff rel-XF-True-def*)

theorem *corresXF-refines-conv*:

⟨*corresXF st ret-xf ex-xf P A C* ↔
 (∀ *t. P t* → *refines C A t (st t) (λ(v, t) (r, s). s = st t ∧*
 (*case v of Exn x ⇒ r = Exn (ex-xf x t) | Result x ⇒ r = Result (ret-xf x t)*))⟩
using *corresXF-refines refines-corresXF ..*

lemma *sim-nondet-to-corresXF*: (∧ *s. refines f f' s s (=)*) ⇒

corresXF (λs. s) (λr -. r) (λr -. r) (λ-. True) f' f

unfolding *refines-def-old corresXF-def*

apply (*auto split: xval-splits*)

done

named-theorems *rel-spec-monad-rewrite-simps*

locale *sim-stack-heap-state* =

abstract: stack-heap-state htd_a htd-upd_a hmem_a hmem-upd_a S +

concrete: stack-heap-state htd_c htd-upd_c hmem_c hmem-upd_c S

for

htd_a:: 's_a ⇒ heap-typ-desc **and**

htd-upd_a:: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 's_a ⇒ 's_a **and**

hmem_a:: 's_a ⇒ heap-mem **and**

hmem-upd_a:: (heap-mem ⇒ heap-mem) ⇒ 's_a ⇒ 's_a **and**

htd_c:: 's_c ⇒ heap-typ-desc **and**

htd-upd_c:: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 's_c ⇒ 's_c **and**

hmem_c:: 's_c ⇒ heap-mem **and**

hmem-upd_c:: (heap-mem ⇒ heap-mem) ⇒ 's_c ⇒ 's_c **and**

S:: addr set

begin

definition *rel-stack-free where*

rel-stack-free s_c s_a ≡ stack-free (htd_c s_c) ⊆ stack-free (htd_a s_a)

lemma *refines-rel-stack-free-with-fresh-stack-ptr*:

assumes *s: rel-stack-free s_c s_a*

assumes *init-eq: ∧ s_c s_a. I_c s_c = I_a s_a*

assumes *f: ∧ s_c s_a p. rel-stack-free s_c s_a ⇒*

refines (f_c p) (f_a p) s_c s_a

(rel-prod (rel-xval L R) rel-stack-free)

shows *refines*

(concrete.with-fresh-stack-ptr n I_c f_c)

(abstract.with-fresh-stack-ptr n I_a f_a) s_c s_a

(rel-prod (rel-xval L R) rel-stack-free)

```

apply (simp add: concrete.with-fresh-stack-ptr-def abstract.with-fresh-stack-ptr-def)
apply (rule refines-bind-bind-exn [where Q=(rel-prod (rel-xval (λ-. False) (=))
rel-stack-free]))
  apply simp-all
subgoal
  apply (clarsimp simp add: refines-def-old rel-xval.simps)
  using s init-eq
  apply (simp add: rel-stack-free-def)
subgoal for p d vs bs
  apply (frule (1) stack-allocs-stack-free-mono)
  apply clarsimp
  subgoal for d'
    apply (rule exI[where x=hmem-upda (fold (λi. heap-update-padding (p
 $+_p$  int i) (vs ! i) (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs)))
 $[0..<length\ vs]$  (htd-upda (λ-. d') sa)]))
    apply clarsimp
    apply (rule conjI)
    apply (rule exI[where x=d'])
    apply clarsimp
    apply (rule exI[where x=vs])
    apply clarsimp
    apply blast
    using stack-free-stack-allocs
    apply blast
    done
  done
done
apply (rule refines-rel-prod-on-exit [where S' = rel-stack-free])
apply (rule f)
apply assumption
apply (simp add: rel-stack-free-def)
apply (metis abstract.htd-hmem-upd abstract.typing.get-upd concrete.htd-hmem-upd
  concrete.typing.get-upd stack-free-stack-releases-mono')
subgoal using stack-free-stack-releases-mono' by blast
done

```

definition *rel-stack-free-eq* **where**

$$rel_stack_free_eq\ s_c\ s_a \equiv stack_free\ (htd_c\ s_c) = stack_free\ (htd_a\ s_a)$$

lemma *refines-rel-stack-free-eq-with-fresh-stack-ptr:*

```

assumes s: rel-stack-free-eq sc sa
assumes init-eq: ∧ sc sa. Ic sc = Ia sa
assumes f: ∧ sc sa p. rel-stack-free-eq sc sa ⇒
  refines (fc p) (fa p) sc sa
  (rel-prod (rel-xval L R) rel-stack-free-eq)
shows refines
  (concrete.with-fresh-stack-ptr n Ic fc)
  (abstract.with-fresh-stack-ptr n Ia fa) sc sa

```

```

      (rel-prod (rel-xval L R) rel-stack-free-eq)
apply (simp add: concrete.with-fresh-stack-ptr-def abstract.with-fresh-stack-ptr-def)
apply (rule refines-bind-bind-exn [where Q=(rel-prod (rel-xval (λ- -. False) (=))
rel-stack-free-eq)])
  apply simp-all
  subgoal
    apply (clarsimp simp add: refines-def-old rel-xval.simps)
    using s init-eq
    apply (simp add: rel-stack-free-eq-def)
    subgoal for p d vs bs
      apply (frule (1) stack-allocs-stack-free-eq)
      apply clarsimp
      subgoal for d'
        apply (rule exI[where x=hmem-upda (fold (λi. heap-update-padding (p
+p int i) (vs ! i) (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs)))
[0..<length vs]) (htd-upda (λ-. d') sa))]
          apply clarsimp
          apply (rule conjI)
          apply (rule exI[where x=d'])
          apply clarsimp
          apply (rule exI[where x=vs])
          apply clarsimp
          apply blast
          using stack-free-stack-allocs
          apply blast
          done
        done
      done
    done
  apply (rule refines-rel-prod-on-exit [where S' = rel-stack-free-eq])
  apply (rule f)
  apply assumption
  apply clarsimp
  subgoal for v t t' sc sa bs
    apply (rule exI[where x= hmem-upda (heap-update-list (ptr-val v) bs) (htd-upda
(stack-releases n v) sa))]
      apply (clarsimp simp add: rel-stack-free-eq-def)
      by (metis dual-order.eq-iff stack-free-stack-releases-mono')
    subgoal by blast
  done

```

end

```

locale sim-stack-heap-raw-state =
  abstract: stack-heap-raw-state hrsa hrs-upda S +
  concrete: stack-heap-raw-state hrsc hrs-updc S
  for
    hrsa:: 'sa ⇒ heap-raw-state and
    hrs-upda:: (heap-raw-state ⇒ heap-raw-state) ⇒ 'sa ⇒ 'sa and

```

```

    hrs_c:: 's_c ⇒ heap-raw-state and
    hrs-upd_c:: (heap-raw-state ⇒ heap-raw-state) ⇒ 's_c ⇒ 's_c and
    S::addr set
begin
sublocale sim-stack-heap-state
  λs. hrs-htd (hrs_a s) λupd. hrs-upd_a (hrs-htd-update upd)
  λs. hrs-mem (hrs_a s) λupd. hrs-upd_a (hrs-mem-update upd)
  λs. hrs-htd (hrs_c s) λupd. hrs-upd_c (hrs-htd-update upd)
  λs. hrs-mem (hrs_c s) λupd. hrs-upd_c (hrs-mem-update upd)
  S
  by unfold-locales
end

named-theorems L2corres-with-fresh-stack-ptr

locale L2 = sim-stack-heap-state htd_a htd-upd_a hmem_a hmem-upd_a htd_c htd-upd_c
hmem_c hmem-upd_c S
  for
    htd_a:: 's_a ⇒ heap-typ-desc and
    htd-upd_a:: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 's_a ⇒ 's_a and
    hmem_a:: 's_a ⇒ heap-mem and
    hmem-upd_a:: (heap-mem ⇒ heap-mem) ⇒ 's_a ⇒ 's_a and

    htd_c:: 's_c ⇒ heap-typ-desc and
    htd-upd_c:: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 's_c ⇒ 's_c and
    hmem_c:: 's_c ⇒ heap-mem and
    hmem-upd_c:: (heap-mem ⇒ heap-mem) ⇒ 's_c ⇒ 's_c and
    S::addr set +
  fixes st:: 's_c ⇒ 's_a
  assumes htd-st:  $\bigwedge s. \text{htd}_a (st\ s) = \text{htd}_c\ s$ 
  assumes htd-upd-st:  $\bigwedge s\ f. (\text{htd-upd}_a\ f (st\ s)) = st\ (\text{htd-upd}_c\ f\ s)$ 
  assumes hmem-upd-st:  $\bigwedge s\ f. (\text{hmem-upd}_a\ f (st\ s)) = st\ (\text{hmem-upd}_c\ f\ s)$ 

begin
lemma rel-stack-free-eq-st: rel-stack-free-eq s (st s)
  by (simp add: rel-stack-free-eq-def htd-st)

definition rel-L2 where
  rel-L2 ex-xf ret-xf ≡ (λ (r_c, s_c) (r_a, s_a).
    rel-xval (λ- a. a = ex-xf s_c) (λ- a. a = ret-xf s_c) r_c r_a ∧
    (s_a = st s_c))

lemma rel-L2-def':
  (rel-L2 ex-xf ret-xf) = (λ(v, t) (r, s).
    s = st t ∧ (case v of Exn x ⇒ r = Exn (ex-xf t) | Result x ⇒ r =
    Result (ret-xf t)))
  apply (clarsimp simp add: rel-L2-def fun-eq-iff, intro iffI)
  subgoal using rel-xval.cases by fastforce

```

subgoal by(*auto split: xval-splits*)
done

lemma *refines-rel-L2-on-exit'*:

assumes *f*: *refines* (*f_c p*) (*f_a p*) *s_c* (*st s_c*) (*rel-L2 ex-xf ret-xf*)

assumes *ex-xf-htd-indep*: $\bigwedge f s. \text{ex-xf } (\text{htd-upd}_c f s) = \text{ex-xf } s$

assumes *ret-xf-htd-indep*: $\bigwedge f s. \text{ret-xf } (\text{htd-upd}_c f s) = \text{ret-xf } s$

shows *refines*

(*on-exit* (*f_c p*) $\{(s,t). t = \text{htd-upd}_c (\text{release } p) s\}$)

(*on-exit* (*f_a p*) $\{(s,t). t = \text{htd-upd}_a (\text{release } p) s\}$) *s_c* (*st s_c*)

(*rel-L2 ex-xf ret-xf*)

unfolding *on-exit-bind-exception-or-result-conv*

apply (*rule refines-bind-exception-or-result*)

apply (*rule refines-mono* [*OF - f*])

apply (*clarsimp*)

subgoal for *r s q t*

apply (*rule refines-bind-bind-exn*[**where**

$Q = (\lambda(r', s) (q', t). \text{is-Result } r' \wedge \text{is-Result } q' \wedge \text{rel-L2 ex-xf ret-xf } (r, s) (q, t))$])

subgoal

apply (*rule refines-assert-result-and-state*)

apply (*clarsimp simp add: rel-L2-def, intro conjI*)

subgoal

using *ex-xf-htd-indep ret-xf-htd-indep*

by *metis*

subgoal

using *htd-upd-st by force*

by *auto*

apply (*auto simp: is-Result-def*)

done

done

lemma *refines-rel-L2-on-exit*:

assumes *f*: *refines* (*f_c p*) (*f_a p*) *s_c* (*st s_c*) (*rel-L2 ex-xf ret-xf*)

assumes *ex-xf-htd-indep*: $\bigwedge g f s. \text{ex-xf } (\text{hmem-upd}_c g (\text{htd-upd}_c f s)) = \text{ex-xf } s$

assumes *ret-xf-htd-indep*: $\bigwedge g f s. \text{ret-xf } (\text{hmem-upd}_c g (\text{htd-upd}_c f s)) = \text{ret-xf } s$

assumes *nonempty-cleanup_a*: *cleanup_a* $\neq \{\}$

assumes *cleanup-htd*: $\bigwedge s s'. (s, s') \in \text{cleanup}_c \implies \exists g f. s' = (\text{hmem-upd}_c g (\text{htd-upd}_c f s))$

assumes *stuck-sim*: $\bigwedge s. \nexists s'. (s, s') \in \text{cleanup}_c \implies \nexists t'. (st s, t') \in \text{cleanup}_a$

assumes *cleanup-sim*: $\bigwedge s t. (s, t) \in \text{cleanup}_c \implies (st s, st t) \in \text{cleanup}_a$

shows *refines*

(*on-exit* (*f_c p*) *cleanup_c*)

(*on-exit* (*f_a p*) *cleanup_a*) *s_c* (*st s_c*)

(*rel-L2 ex-xf ret-xf*)

unfolding *on-exit-bind-exception-or-result-conv*

apply (*rule refines-bind-exception-or-result*)

apply (*rule refines-mono* [*OF - f*])

```

apply (clarsimp)
subgoal for r s q t
  apply (rule refines-bind-bind-exn [where
     $Q = \lambda(r', s) (q', t). \text{is-Result } r' \wedge \text{is-Result } q' \wedge \text{rel-L2 } \text{ex-xf } \text{ret-xf } (r, s) (q, t)$ 
  ]])
  subgoal
    apply (rule refines-state-select)
    apply (clarsimp simp add: rel-L2-def, intro conjI)
    subgoal
      using cleanup-htd cleanup-sim
      by blast
    subgoal
      using ex-xf-htd-indep ret-xf-htd-indep cleanup-htd
      by metis
    subgoal
      using stuck-sim by (auto simp: rel-L2-def)
    done
  apply (auto simp: is-Result-def)
done
done

```

lemma *refines-rel-L2-with-fresh-stack-ptr*:

```

assumes init-eq:  $I_c s_c = I_a (st s_c)$ 
assumes ex-xf-htd-indep:  $\bigwedge g f s. \text{ex-xf } (hmem-upd_c g (htd-upd_c f s)) = \text{ex-xf } s$ 
assumes ret-xf-htd-indep:  $\bigwedge g f s. \text{ret-xf } (hmem-upd_c g (htd-upd_c f s)) = \text{ret-xf } s$ 
assumes f:  $\bigwedge (p::'a::\text{mem-type ptr}) d vs bs. (p, d) \in \text{stack-allocs } n \mathcal{S} \text{TYPE}'a (htd_c s_c) \implies vs \in I_c s_c \implies \text{length } vs = n \implies \text{length } bs = n * \text{size-of TYPE}'a \implies$ 
  refines
    (fc p)
    (fa p)
    (hmem-updc (fold (λi. heap-update-padding (p +p int i) (vs ! i) (take (size-of TYPE}'a) (drop (i * size-of TYPE}'a) bs))) [0.. $\text{length } vs]$ ) ((htd-updc (λ-. d)) sc))
    (st (hmem-updc (fold (λi. heap-update-padding (p +p int i) (vs ! i) (take (size-of TYPE}'a) (drop (i * size-of TYPE}'a) bs))) [0.. $\text{length } vs]$ ) ((htd-updc (λ-. d)) sc))
    (rel-L2 ex-xf ret-xf)
shows refines
  (concrete.with-fresh-stack-ptr n Ic fc)
  (abstract.with-fresh-stack-ptr n Ia fa) sc (st sc)
  (rel-L2 ex-xf ret-xf)
apply (simp add: concrete.with-fresh-stack-ptr-def abstract.with-fresh-stack-ptr-def)
apply (rule refines-bind-bind-exn [where  $Q = \lambda(r_c, t_c) (r_a, t_a). (\exists d p vs bs. (p, d) \in \text{stack-allocs } n \mathcal{S} \text{TYPE}'a (htd_c s_c) \wedge r_c = \text{Result } p \wedge r_a = \text{Result } p \wedge vs \in I_c s_c \wedge \text{length } vs = n \wedge \text{length } bs = n * \text{size-of TYPE}'a \wedge t_c = hmem-upd_c (fold (λi. heap-update-padding (p +p int i) (vs ! i) (take$ 

```



```

(size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs))) [0..<length vs]) ((htd-updc
(λ-. d)) sc) ∧
  ta = st tc)))]
  apply simp-all
subgoal
  apply (clarsimp simp add: refines-def-old rel-xval.simps)
  by (metis (full-types) hmem-upd-st htd-st htd-upd-st init-eq )
subgoal for p
  apply clarsimp
  apply (rule refines-rel-L2-on-exit )
  apply (rule f)
  apply simp
  apply simp
  apply simp
  apply simp
  apply (rule ex-xf-htd-indep)
  apply (rule ret-xf-htd-indep)
  apply blast
subgoal by auto
subgoal by auto
subgoal using htd-upd-st hmem-upd-st by auto
done
done

```

lemma *L2corres-refines-rel-L2-conv*:
 $\langle L2corres\ st\ ret\text{-}xf\ ex\text{-}xf\ P\ A\ C \longleftrightarrow$
 $(\forall t. P\ t \longrightarrow refines\ C\ A\ t\ (st\ t)\ (rel\text{-}L2\ ex\text{-}xf\ ret\text{-}xf)) \rangle$
by (*simp add: L2corres-def corresXF-refines-conv rel-L2-def'*)

lemma *L2corres-refines-rel-L2*:
assumes *L2*: *L2corres st ret-xf ex-xf P A C*
assumes *P*: *P s_c*
shows $\langle refines\ C\ A\ s_c\ (st\ s_c)\ (rel\text{-}L2\ ex\text{-}xf\ ret\text{-}xf) \rangle$
using *L2 P* **by** (*simp add: L2corres-refines-rel-L2-conv*)

lemma *L2corres-with-fresh-stack-ptr*[*L2corres-with-fresh-stack-ptr*]:
assumes *l2*: $\bigwedge p. L2corres\ st\ ret\text{-}xf\ ex\text{-}xf\ P\ (l2\ p)\ (l1\ p)$
assumes *I*: $\bigwedge s. R\ s \implies I\text{-}l1\ s = I\text{-}l2\ (st\ s)$
assumes *P*: $\bigwedge s. R\ s \implies P\ s$
assumes *ex-xf-htd-indep*: $\bigwedge g\ f\ s. ex\text{-}xf\ (hmem\text{-}upd_c\ g\ (htd\text{-}upd_c\ f\ s)) = ex\text{-}xf\ s$
assumes *ret-xf-htd-indep*: $\bigwedge g\ f\ s. ret\text{-}xf\ (hmem\text{-}upd_c\ g\ (htd\text{-}upd_c\ f\ s)) = ret\text{-}xf\ s$
assumes *P-indep*: $\bigwedge f\ g\ s. P\ (hmem\text{-}upd_c\ g\ (htd\text{-}upd_c\ f\ s)) = P\ s$
shows *L2corres st ret-xf ex-xf R*
 $(abstract.with\text{-}fresh\text{-}stack\text{-}ptr\ n\ (I\text{-}l2)\ (L2\text{-}VARS\ l2\ nm))$
 $(concrete.with\text{-}fresh\text{-}stack\text{-}ptr\ n\ I\text{-}l1\ l1)$
apply (*clarsimp simp add: L2-VARS-def L2corres-refines-rel-L2-conv*)
apply (*rule refines-rel-L2-with-fresh-stack-ptr*)
apply (*rule I, assumption*)

```

    apply (rule ex-xf-htd-indep)
    apply (rule ret-xf-htd-indep)
    apply (rule L2corres-refines-rel-L2)
    apply (rule l2)
    apply (simp add: P-indep P)
  done

end

hide-const (open) L2

lemma refines-rel-prod-guard-on-exit:
  assumes f: refines fc fa sc sa (rel-prod R S')
  assumes cleanup:  $\bigwedge s_c s_a t_c. S' s_c s_a \implies \text{grd } s_a \implies (s_c, t_c) \in \text{cleanup}_c \implies \exists t_a. (s_a, t_a) \in \text{cleanup}_a \wedge S t_c t_a$ 
  assumes emp:  $\bigwedge s_c s_a. S' s_c s_a \implies \nexists t_c. (s_c, t_c) \in \text{cleanup}_c \implies \nexists t_a. (s_a, t_a) \in \text{cleanup}_a$ 
  shows refines (on-exit fc cleanupc) (guard-on-exit fa grd cleanupa) sc sa (rel-prod R S)
  unfolding on-exit-def on-exit'-def
  apply (rule refines-bind-exception-or-result[OF refines-weaken, OF f])
  apply (simp add: bind-assoc)
  apply (rule refines-bind-guard-right)
  apply (rule refines-bind'[OF refines-state-select])
  subgoal using cleanup by auto
  subgoal using emp by auto
  done

lemma refines-bind-no-throw-strong:
  assumes no-throw (λ-. True) f
  assumes no-throw (λ-. True) f'
  assumes f: refines f f' s s' Q
  assumes g:  $\bigwedge r t r' t'. \text{reaches } f s (\text{Result } r) t \implies \text{reaches } f' s' (\text{Result } r') t' \implies Q (\text{Result } r, t) (\text{Result } r', t') \implies \text{refines } (g r) (g' r') t t' R$ 
  shows refines (f >>= g) (f' >>= g') s s' R
  apply (rule refines-bind-bind-strong' [OF f])
  subgoal using assms
    apply (clarsimp simp add: no-throw-def runs-to-partial-def-old)
    by (metis the-Exception-simp the-Exception-Result reaches-succeeds)
  subgoal using assms
    apply (clarsimp simp add: no-throw-def runs-to-partial-def-old)
    by (metis the-Exception-simp the-Exception-Result reaches-succeeds)
  subgoal using assms
    apply (clarsimp simp add: no-throw-def runs-to-partial-def-old)
    by (metis the-Exception-simp the-Exception-Result reaches-succeeds)
  subgoal using g
    by auto
  done

```

lemma *refines-bind-no-throw*:
assumes *no-throw* $(\lambda-. \text{True}) f$
assumes *no-throw* $(\lambda-. \text{True}) f'$
assumes $f: \text{refines } f f' s s' Q$
assumes $g: \bigwedge r t r' t'.$
 $Q (\text{Result } r, t) (\text{Result } r', t') \implies \text{refines } (g r) (g' r') t t' R$
shows $\text{refines } (f \gg= g) (f' \gg= g') s s' R$
using *assms*
by (*rule refines-bind-no-throw-strong*)

lemma *no-throw-guard[simp]*: *no-throw* P (*guard* G)
by (*auto simp add: no-throw-def runs-to-partial-def-old*)

lemma *no-throw-assert-result-and-state[simp]*: *no-throw* P (*assert-result-and-state* f)
by (*auto simp add: no-throw-def runs-to-partial-def-old*)

lemma *no-throw-assume-result-and-state[simp]*: *no-throw* P (*assume-result-and-state* f)
by (*auto simp add: no-throw-def runs-to-partial-def-old*)

lemma *refines-guard-same*: $\text{refines } (\text{guard } P) (\text{guard } P) s s (\lambda(r, t) (r', t'). t=s \wedge t'=s)$
by (*simp add: refines-def-old*)

lemma *refines-state-select-same*:
 $\text{refines } (\text{state-select } R) (\text{state-select } R) s s (\lambda(r, t) (r', t'). t' = t \wedge (s, t) \in R)$
by (*simp add: refines-def-old*)

lemma *refines-assert-result-and-state-same*:
 $\text{refines } (\text{assert-result-and-state } R) (\text{assert-result-and-state } R) s s$
 $(\lambda(r, t) (r', t'). \exists v. (v, t) \in R s \wedge r = \text{Result } v \wedge r' = \text{Result } v \wedge t = t')$
by (*auto simp add: refines-def-old*)

lemma *refines-assume-result-and-state-same*:
 $\text{refines } (\text{assume-result-and-state } R) (\text{assume-result-and-state } R) s s$
 $(\lambda(r, t) (r', t'). \exists v. (v, t) \in R s \wedge r = \text{Result } v \wedge r' = \text{Result } v \wedge t = t')$
by (*auto simp add: refines-def-old*)

lemma *refines-assume-result-and-state-same'*:
assumes $R s = R' s$
shows $\text{refines } (\text{assume-result-and-state } R) (\text{assume-result-and-state } R') s s$
 $(\lambda(r, t) (r', t'). \exists v. (v, t) \in R s \wedge r = \text{Result } v \wedge r' = \text{Result } v \wedge t = t')$
using *assms*
by (*auto simp add: refines-def-old*)

lemma *refines-rel-xval-guard-on-exit*:
assumes $f: \text{refines } f_c f_a s s (\text{rel-prod } (\text{rel-xval } L R) (==))$
shows refines

```

      (guard-on-exit fc P cleanup)
      (guard-on-exit fa P cleanup) s s
      (rel-prod (rel-xval L R) (=))
unfolding guard-on-exit-bind-exception-or-result-conv
apply (rule refines-bind-exception-or-result')
apply (rule f)
apply clarsimp
apply (rule refines-bind-no-throw-strong [OF - - refines-guard-same])
apply simp
apply simp
apply clarsimp
apply (rule refines-bind-no-throw-strong [OF - - refines-state-select-same])
apply simp
apply simp
apply clarsimp
done

```

lemma *refines-rel-xval-on-exit:*

```

assumes f: refines fc fa s s (rel-prod (rel-xval L R) (=))
shows refines
      (on-exit fc cleanup)
      (on-exit fa cleanup) s s
      (rel-prod (rel-xval L R) (=))
unfolding on-exit-bind-exception-or-result-conv
apply (rule refines-bind-exception-or-result')
apply (rule f)
apply clarsimp
apply (rule refines-bind-no-throw-strong [OF - - refines-state-select-same])
apply simp
apply simp
apply clarsimp
done

```

lemma *refines-rel-xval-assume-on-exit:*

```

assumes f: refines fc fa s s (rel-prod (rel-xval L R) (=))
shows refines
      (assume-on-exit fc P cleanup)
      (assume-on-exit fa P cleanup) s s
      (rel-prod (rel-xval L R) (=))
unfolding assume-on-exit-bind-exception-or-result-conv
apply (rule refines-bind-exception-or-result')
apply (rule f)

apply (rule refines-bind-no-throw-strong [OF - - ])
apply simp
apply simp
apply clarsimp
apply (rule refines-assume-result-and-state-same)

```

```

apply clarsimp
apply (rule refines-bind-no-throw-strong [OF - - refines-state-select-same])
  apply simp
  apply simp
apply clarsimp
done

```

```

lemma refines-state-assume-pred:
  assumes P: P t
  assumes ref: refines f g s t R
  shows refines f (do {assume-result-and-state (λs. {(x, y). (λ() s'. s' = s ∧ P s) x y}); g }) s t R
  using P ref
  by (auto simp add: refines-def-old succeeds-bind reaches-bind)

```

```

lemma refines-rel-prod-assume-on-exit:
  assumes f: refines f_c f_a s_c s_a (rel-prod R S')
  assumes cleanup:  $\bigwedge s_c s_a t_c. S' s_c s_a \implies (s_c, t_c) \in \text{cleanup}_c \implies \exists t_a. (s_a, t_a) \in \text{cleanup}_a \wedge S t_c t_a$ 
  assumes emp:  $\bigwedge s_c s_a. S' s_c s_a \implies \nexists t_c. (s_c, t_c) \in \text{cleanup}_c \implies \nexists t_a. (s_a, t_a) \in \text{cleanup}_a$ 
  assumes conseq:  $\bigwedge s_c s_a. S' s_c s_a \implies P s_a$ 
  shows refines (on-exit f_c cleanup_c) (assume-on-exit f_a P cleanup_a) s_c s_a (rel-prod R S)
unfolding assume-on-exit-bind-exception-or-result-conv on-exit-bind-exception-or-result-conv
apply (rule refines-bind-exception-or-result')
  apply (rule f)
apply (rule refines-state-assume-pred)
subgoal using conseq by auto
subgoal
  apply (rule refines-bind-no-throw-strong [where Q = rel-prod (λ- -. True) S])
  apply simp
  apply simp
  apply (rule refines-state-select)
subgoal using cleanup by auto
subgoal using emp by auto
subgoal
  apply (rule refines-kill)
  apply auto
done
done

```

```

lemma refines-runs-to-partial-rel-prod-assume-on-exit:
  assumes f: refines f_c f_a s_c s_a (rel-prod R S')
  assumes runs-to: f_c · s_c ?[λr t. P t]
  assumes cleanup:  $\bigwedge s_c s_a t_c. S' s_c s_a \implies (s_c, t_c) \in \text{cleanup}_c \implies P s_c \implies \exists t_a. (s_a, t_a) \in \text{cleanup}_a \wedge S t_c t_a$ 
  assumes emp:  $\bigwedge s_c s_a. S' s_c s_a \implies P s_c \implies \nexists t_c. (s_c, t_c) \in \text{cleanup}_c \implies \nexists t_a.$ 

```

$(s_a, t_a) \in \text{cleanup}_a$
assumes *conseq*: $\bigwedge s_c s_a. S' s_c s_a \implies P s_c \implies P' s_a$
shows *refines* (*on-exit* f_c *cleanup*_c) (*assume-on-exit* f_a P' *cleanup*_a) $s_c s_a$ (*rel-prod* R S)
proof –
from *refines-runs-to-partial-fuse* [*OF* f *runs-to*]
have *refines* $f_c f_a s_c s_a$ $(\lambda(r, t) (r', t'). \text{rel-prod } R \ S' (r, t) (r', t') \wedge P t)$.
moreover have $(\lambda(r, t) (r', t'). \text{rel-prod } R \ S' (r, t) (r', t') \wedge P t) = \text{rel-prod } R$
 $(\lambda t t'. S' t t' \wedge P t)$
by (*auto simp add: rel-prod-conv*)
ultimately have *refines* $f_c f_a s_c s_a$ (*rel-prod* R $(\lambda t t'. S' t t' \wedge P t)$) **by** *simp*
then show *?thesis*
apply (*rule refines-rel-prod-assume-on-exit*)
subgoal using *cleanup* **by** *blast*
subgoal using *emp* **by** *blast*
subgoal using *conseq* **by** *auto*
done
qed

locale *L2-heap-raw-state* = *sim-stack-heap-raw-state* hrs_a hrs_upd_a hrs_c hrs_upd_c S

for

$hrs_a:: 's_a \Rightarrow \text{heap-raw-state}$ **and**
 $hrs_upd_a:: (\text{heap-raw-state} \Rightarrow \text{heap-raw-state}) \Rightarrow 's_a \Rightarrow 's_a$ **and**

$hrs_c:: 's_c \Rightarrow \text{heap-raw-state}$ **and**
 $hrs_upd_c:: (\text{heap-raw-state} \Rightarrow \text{heap-raw-state}) \Rightarrow 's_c \Rightarrow 's_c$ **and**
 $S:: \text{addr set} +$

fixes $st:: 's_c \Rightarrow 's_a$

assumes *hrs-htd-st*: $\bigwedge s. hrs_htd (hrs_a (st s)) = hrs_htd (hrs_c s)$

assumes *hrs-upd-st*: $\bigwedge s f. (hrs_upd_a f (st s)) = st (hrs_upd_c f s)$

begin

sublocale *L2*

$\lambda s. hrs_htd (hrs_a s) \lambda upd. hrs_upd_a (hrs_htd_update upd)$

$\lambda s. hrs_mem (hrs_a s) \lambda upd. hrs_upd_a (hrs_mem_update upd)$

$\lambda s. hrs_htd (hrs_c s) \lambda upd. hrs_upd_c (hrs_htd_update upd)$

$\lambda s. hrs_mem (hrs_c s) \lambda upd. hrs_upd_c (hrs_mem_update upd)$

S

apply (*unfold-locales*)

subgoal using *hrs-htd-st* **by** *simp*

subgoal using *hrs-upd-st* **by** *simp*

subgoal using *hrs-upd-st* **by** *simp*

done

end

locale *typ-heap-typing* = *pointer-lense* $r w$ + *heap-typing-state* *heap-typing* *heap-typing-upd*

for

$r :: 't \Rightarrow ('a::xmem\text{-}type) \text{ ptr} \Rightarrow 'a$ **and**
 $w :: 'a \text{ ptr} \Rightarrow ('a \Rightarrow 'a) \Rightarrow 't \Rightarrow 't$ **and**
 $heap\text{-}typing :: 't \Rightarrow heap\text{-}typ\text{-}desc$ **and**
 $heap\text{-}typing\text{-}upd :: (heap\text{-}typ\text{-}desc \Rightarrow heap\text{-}typ\text{-}desc) \Rightarrow 't \Rightarrow 't$ **and**
 $\mathcal{S} :: addr\ set$

begin

definition $stack\text{-}ptr\text{-}acquire :: nat \Rightarrow 'a\ list \Rightarrow 'a\ ptr \Rightarrow heap\text{-}typ\text{-}desc \Rightarrow 't \Rightarrow 't$
where $stack\text{-}ptr\text{-}acquire\ n\ vs\ p\ d\ s =$
 $(fold\ (\lambda i. w\ (p +_p\ int\ i)\ (\lambda \cdot. (vs\ !\ i)))\ [0..<n])\ (heap\text{-}typing\text{-}upd\ (\lambda \cdot. d)\ s)$

definition $stack\text{-}ptr\text{-}release :: nat \Rightarrow 'a\ ptr \Rightarrow 't \Rightarrow 't$
where $stack\text{-}ptr\text{-}release\ n\ p\ s =$
 $(heap\text{-}typing\text{-}upd\ (stack\text{-}releases\ n\ p)\ o\ (fold\ (\lambda i. w\ (p +_p\ int\ i)\ (\lambda \cdot. c\text{-}type\text{-}class.zero))\ [0..<n]))\ s$

definition $assume\text{-}stack\text{-}alloc :: nat \Rightarrow ('t \Rightarrow ('a::xmem\text{-}type)\ list\ set) \Rightarrow ('e::default,$
 $'a\ ptr, 't)\ spec\text{-}monad$ **where**
 $assume\text{-}stack\text{-}alloc\ n\ init \equiv assume\text{-}result\text{-}and\text{-}state\ (\lambda s. \{(p, t).\}$
 $\exists d\ vs. (p, d) \in stack\text{-}allocs\ n\ \mathcal{S}\ TYPE('a::xmem\text{-}type)\ (heap\text{-}typing\ s) \wedge$
 $vs \in init\ s \wedge$
 $length\ vs = n \wedge$
 $(\forall i \in \{0..<n\}. r\ s\ (p +_p\ int\ i) = ZERO('a::xmem\text{-}type)) \wedge$
 $t = stack\text{-}ptr\text{-}acquire\ n\ vs\ p\ d\ s\}$

definition $guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr :: nat \Rightarrow ('t \Rightarrow 'a\ list\ set) \Rightarrow ('a::xmem\text{-}type$
 $ptr \Rightarrow ('e::default, 'v, 't)\ spec\text{-}monad) \Rightarrow ('e, 'v, 't)\ spec\text{-}monad$ **where**
 $guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init\ c \equiv$
 $do\ \{$
 $\quad p \leftarrow assume\text{-}stack\text{-}alloc\ n\ init;$
 $\quad guard\text{-}on\text{-}exit\ (c\ p)$
 $\quad (\lambda s. \forall i < n. root\text{-}ptr\text{-}valid\ (heap\text{-}typing\ s)\ (p +_p\ int\ i))$
 $\quad \{(s, t). t = stack\text{-}ptr\text{-}release\ n\ p\ s\}$
 $\}$

definition $assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr :: nat \Rightarrow ('t \Rightarrow 'a\ list\ set) \Rightarrow ('a::xmem\text{-}type$
 $ptr \Rightarrow ('e::default, 'v, 't)\ spec\text{-}monad) \Rightarrow ('e, 'v, 't)\ spec\text{-}monad$ **where**
 $assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init\ c \equiv$
 $do\ \{$
 $\quad p \leftarrow assume\text{-}stack\text{-}alloc\ n\ init;$
 $\quad assume\text{-}on\text{-}exit\ (c\ p)$
 $\quad (\lambda s. \forall i < n. root\text{-}ptr\text{-}valid\ (heap\text{-}typing\ s)\ (p +_p\ int\ i))$
 $\quad \{(s, t). t = stack\text{-}ptr\text{-}release\ n\ p\ s\}$
 $\}$

definition $with\text{-}fresh\text{-}stack\text{-}ptr :: nat \Rightarrow ('t \Rightarrow 'a\ list\ set) \Rightarrow ('a::xmem\text{-}type\ ptr \Rightarrow$
 $('e::default, 'v, 't)\ spec\text{-}monad) \Rightarrow ('e, 'v, 't)\ spec\text{-}monad$ **where**
 $with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init\ c \equiv$
 $do\ \{$

```

  p ← assume-stack-alloc n init;
  on-exit (c p)
    {(s, t). t = stack-ptr-release n p s}
}

```

lemma *monotone-guard-with-fresh-stack-ptr-le*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p. \text{monotone } R (\leq) (\lambda f. c f p)$
shows *monotone* $R (\leq) (\lambda f. \text{guard-with-fresh-stack-ptr } n I (c f))$
unfolding *guard-with-fresh-stack-ptr-def*
by (*intro partial-function-mono*)

lemma *monotone-guard-with-fresh-stack-ptr-ge*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p. \text{monotone } R (\geq) (\lambda f. c f p)$
shows *monotone* $R (\geq) (\lambda f. \text{guard-with-fresh-stack-ptr } n I (c f))$
unfolding *guard-with-fresh-stack-ptr-def*
by (*intro partial-function-mono*)

lemma *monotone-assume-with-fresh-stack-ptr-le*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p. \text{monotone } R (\leq) (\lambda f. c f p)$
shows *monotone* $R (\leq) (\lambda f. \text{assume-with-fresh-stack-ptr } n I (c f))$
unfolding *assume-with-fresh-stack-ptr-def*
by (*intro partial-function-mono*)

lemma *monotone-assume-with-fresh-stack-ptr-ge*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p. \text{monotone } R (\geq) (\lambda f. c f p)$
shows *monotone* $R (\geq) (\lambda f. \text{assume-with-fresh-stack-ptr } n I (c f))$
unfolding *assume-with-fresh-stack-ptr-def*
by (*intro partial-function-mono*)

lemma *monotone-with-fresh-stack-ptr-le*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p. \text{monotone } R (\leq) (\lambda f. c f p)$
shows *monotone* $R (\leq) (\lambda f. \text{with-fresh-stack-ptr } n I (c f))$
unfolding *with-fresh-stack-ptr-def on-exit-def*
by (*intro partial-function-mono*)

lemma *monotone-with-fresh-stack-ptr-ge*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p. \text{monotone } R (\geq) (\lambda f. c f p)$
shows *monotone* $R (\geq) (\lambda f. \text{with-fresh-stack-ptr } n I (c f))$
unfolding *with-fresh-stack-ptr-def on-exit-def*
by (*intro partial-function-mono*)

lemma *refines-rel-xval-guard-with-fresh-stack-ptr*:
assumes *init-eq*: $\text{init}_c s = \text{init}_a s$
assumes *f*: $\bigwedge s p. \text{refines } (f_c p) (f_a p) s s (\text{rel-prod } (\text{rel-xval } L R) (=))$
shows
refines
(guard-with-fresh-stack-ptr $n \text{ init}_c (L2\text{-VARS } f_c nm)$
(guard-with-fresh-stack-ptr $n \text{ init}_a (L2\text{-VARS } f_a nm)) s s$
(rel-prod (rel-xval $L R) (=))$

unfolding *guard-with-fresh-stack-ptr-def L2-VARS-def on-exit'-def assume-stack-alloc-def*
apply (*rule refines-bind'*)
apply (*subst refines-assume-result-and-state-iff*)
apply (*subst init-eq*)
apply (*rule sim-set-refl*)
apply *clarsimp*
apply (*rule refines-bind-exception-or-result'[OF f]*)
apply (*rule refines-bind'*)
apply (*rule refines-bind'*)
apply (*rule refines-guard*)
apply *simp-all*
apply (*rule refines-assert-result-and-state*)
apply *simp-all*
done

lemma *refines-rel-xval-assume-with-fresh-stack-ptr:*

assumes *init-eq: init_c s = init_a s*
assumes *f: $\bigwedge s p. \text{refines } (f_c p) (f_a p) s s \text{ (rel-prod (rel-xval L R) (=))}$*
shows
refines
(assume-with-fresh-stack-ptr n init_c (L2-VARS f_c nm))
(assume-with-fresh-stack-ptr n init_a (L2-VARS f_a nm)) s s
(rel-prod (rel-xval L R) (=))

unfolding *assume-with-fresh-stack-ptr-def L2-VARS-def on-exit'-def assume-stack-alloc-def*

apply (*rule refines-bind'*)
apply (*subst refines-assume-result-and-state-iff*)
apply (*subst init-eq*)
apply (*rule sim-set-refl*)
apply *clarsimp*
apply (*rule refines-bind-exception-or-result'[OF f]*)
apply (*rule refines-bind'*)
apply (*rule refines-bind'*)
apply (*rule refines-assuming*)
apply *simp-all*
apply (*rule refines-assert-result-and-state*)
apply *simp-all*
done

lemma *refines-rel-xval-with-fresh-stack-ptr:*

assumes *init-eq: init_c s = init_a s*
assumes *f: $\bigwedge s p. \text{refines } (f_c p) (f_a p) s s \text{ (rel-prod (rel-xval L R) (=))}$*
shows
refines
(with-fresh-stack-ptr n init_c (L2-VARS f_c nm))
(with-fresh-stack-ptr n init_a (L2-VARS f_a nm)) s s
(rel-prod (rel-xval L R) (=))

unfolding *with-fresh-stack-ptr-def L2-VARS-def assume-stack-alloc-def*

apply (*rule refines-bind-no-throw [where Q = rel-prod (rel-xval (λ- . False))*)

```

(=) (=)]
  apply simp
  apply simp
  subgoal using init-eq by (auto simp add: refines-def-old rel-xval.simps)
  subgoal
    apply clarsimp
    apply (rule refines-rel-xval-on-exit)
    apply (rule f)
  done
done

```

```

lemma write-same-ZERO:
  r s p = ZERO('a)  $\implies$  w p ( $\lambda y. ZERO('a)$ ) s = s
  using write-same by simp

```

end

```

lemma refines-rel-prod-eq-on-exit:
  assumes f: refines fc fa s s (rel-prod Q (=))
  shows refines
    (on-exit fc cleanup)
    (on-exit fa cleanup) s s
    (rel-prod Q (=))
  unfolding on-exit-bind-exception-or-result-conv
  apply (rule refines-bind-exception-or-result')
  apply (rule f)
  apply clarsimp
  apply (rule refines-bind-no-throw )
  apply simp
  apply simp
  apply (rule refines-state-select-same)
  apply clarsimp
done

```

context stack-heap-state

begin

```

lemma refines-rel-prod-with-fresh-stack-ptr:
  assumes init-eq: initc s = inita s
  assumes f:  $\bigwedge$ s p. refines (fc p) (fa p) s s (rel-prod Q (=))
  shows
    refines
      (with-fresh-stack-ptr n initc (L2-VARS fc nm))
      (with-fresh-stack-ptr n inita (L2-VARS fa nm)) s s
      (rel-prod Q (=))
  unfolding with-fresh-stack-ptr-def L2-VARS-def
  apply (rule refines-bind-no-throw )
  apply simp
  apply simp

```

```

apply (rule refines-assume-result-and-state-same')
apply (simp only: init-eq)
apply clarsimp
apply (rule refines-rel-prod-eq-on-exit)
apply (rule f)
done
end

```

lemma *h-val-coerce-ptr-coerce-packed:*

```

fixes p::'c::packed-type ptr
assumes sz-eq: size-of TYPE('c) = size-of TYPE('a::packed-type)
shows h-val h (PTR-COERCE ('c → 'a) p) = COERCE ('c → 'a) (h-val h p)
by (simp add: h-val-def sz-eq from-bytes-def coerce-def to-bytes-p-def to-bytes-def

      packed-type-access-ti access-ti0-def field-access-update-same(1)
      size-of-fold sz-eq td-fafu-idem update-ti-t-def)

```

lemma *h-val-field-ptr-coerce-from-bytes-packed:*

```

fixes p::'c::packed-type ptr
assumes field-ti TYPE('a) f = Some t
assumes export-uinfo t = export-uinfo (typ-info-t TYPE('b::mem-type))
assumes [simp]: size-of TYPE('c) = size-of TYPE('a)
shows h-val h (PTR('b) &((PTR-COERCE('c::packed-type → 'a::packed-type)
p)→f)) =
      from-bytes (access-ti0 t (COERCE ('c → 'a) (h-val h p)))
apply (simp add:h-val-coerce-ptr-coerce-packed[symmetric])
apply (rule h-val-field-from-bytes' [OF assms(1) assms(2)])
done

```

lemma *packed-type-value-update-ti-explode:*

```

(v::'a::packed-type) = update-ti (typ-info-t TYPE('a)) (to-bytes-p v) v'
by (simp add: size-of-def update-ti-s-from-bytes update-ti-update-ti-t)

```

lemma *packed-type-to-bytes-to-bytes-p-conv:*

```

length bs = size-of TYPE('a::packed-type) ⇒ to-bytes v bs = to-bytes-p (v::'a)
apply (simp add: to-bytes-p-def)
by (simp add: packed-type-access-ti to-bytes-def)

```

lemma *packed-type-to-byte-p-coerce:*

```

assumes sz:size-of TYPE('c::packed-type) = size-of TYPE('a::packed-type)
shows to-bytes-p (COERCE('a → 'c) v) = to-bytes-p v
using sz
apply (simp add: to-bytes-p-def coerce-def)
by (metis (mono-tags, lifting) field-access-update-same(1) field-desc.simps(2)
field-desc-def
      from-bytes-def len length-replicate size-of-fold td-fafu-idem to-bytes-def
      update-ti-t-def wf-fd)

```

lemma *to-bytes-coerce-packed*:
 $size-of\ TYPE('a) = size-of\ TYPE('b) \implies length\ bs = size-of\ TYPE('a) \implies$
 $to-bytes\ (COERCE('a::packed-type \rightarrow 'b::packed-type)\ v)\ bs = to-bytes\ v\ bs$
by (*simp add: coerce-def field-access-update-same(1) from-bytes-def packed-type-access-ti*
 $size-of-def\ td-fafu-idem\ to-bytes-def\ update-ti-t-def$)

lemma *heap-update-ptr-coerse-packed*:
fixes $p::'c::packed-type\ ptr$
assumes $cgrd: c-guard\ p$
assumes [*simp*]: $size-of\ TYPE('c) = size-of\ TYPE('a::packed-type)$
shows $heap-update\ (PTR-COERCE('c \rightarrow 'a)\ p)\ v = heap-update\ p\ (COERCE('a$
 $\rightarrow 'c)\ v)$
apply (*rule ext*)
apply (*simp add: heap-update-def packed-type-to-bytes-to-bytes-p-conv packed-type-to-byte-p-coerce*)
done

lemma *c-guard-ptr-coerceI*:
 $size-td\ (typ-info-t\ TYPE('c)) = size-td\ (typ-info-t\ TYPE('a)) \implies$
 $align-td\ (typ-info-t\ TYPE('a)) \leq align-td\ (typ-info-t\ TYPE('c)) \implies$
 $c-guard\ p \implies$
 $c-guard\ (PTR-COERCE('c::c-type \rightarrow 'a::c-type)\ p)$
apply (*cases p*)
using *power-le-dvd*
by (*auto simp add: c-guard-def c-null-guard-def ptr-aligned-def align-of-def size-of-def*)

lemma *heap-update-field-ptr-coerce-from-bytes-packed*:
fixes $p::'c::\{xmem-type,\ packed-type\}\ ptr$
assumes $fl: field-lookup\ (typ-info-t\ TYPE('a::\{xmem-type,\ packed-type\}))\ f\ 0 =$
 $Some\ (t,\ n)$
assumes $match: export-uinfo\ t = typ-uinfo-t\ (TYPE('b::\{xmem-type\}))$
assumes $cgrd: c-guard\ (PTR-COERCE('c \rightarrow 'a)\ p)$
assumes $sz[simp]: size-of\ TYPE('c) = size-of\ TYPE('a)$
shows $heap-update\ (PTR('b)\ \&((PTR-COERCE('c \rightarrow 'a)\ p) \rightarrow f))\ v\ h =$
 $heap-update\ p\ (coerce-map\ (update-ti\ t\ (to-bytes-p\ v))\ (h-val\ h\ p))\ h$
apply (*subst heap-update-field-root-conv' [OF fl match cgrd]*)
apply (*simp add: heap-update-def h-val-coerce-ptr-coerce-packed*)
apply (*subst coerce-coerce-map-cancel-packed [OF sz[symmetric], symmetric]*)
apply (*simp add: coerce-map-def*)
apply (*subst to-bytes-coerce-packed*)
apply (*simp-all*)
done

named-theorems *L2-modify-heap-update-field-root-conv*

context *heap-state*

begin

lemma *heap-update-field-root-conv*:

fixes $p::'a::xmem\text{-}type$ ptr

assumes fl : *field-lookup* (*typ-info-t* $TYPE('a)$) $f\ 0 = Some$ (*adjust-ti* (*typ-info-t* $TYPE('b::xmem\text{-}type)$)) fld ($fld\text{-}update \circ (\lambda x \ . \ x)$), n

assumes $fg\text{-}cons$: $fg\text{-}cons\ fld$ ($fld\text{-}update \circ (\lambda x \ . \ x)$)

assumes $cgrd$: *c-guard* p

shows ($hmem\text{-}upd$ ($heap\text{-}update$ ($PTR('b)$ & $(p \rightarrow f)$) v)) =

($\lambda s.$ ($hmem\text{-}upd$ ($heap\text{-}update\ p$ ($fld\text{-}update$ ($\lambda \cdot.$ v) ($h\text{-}val$ ($hmem\ s$) p)))) s)

apply (*subst heap-update-field-root-conv-pointless'* [*OF fl fg-cons cgrd*])

apply (*rule ext*)

by (*metis (mono-tags, lifting) heap.upd-cong*)

lemma *heap-update-field-root-conv'*:

fixes $p::'a::xmem\text{-}type$ ptr

assumes fl : *field-lookup* (*typ-info-t* $TYPE('a)$) $f\ 0 = Some$ (t , n)

assumes $match$: *export-uinfo* $t = typ\text{-}uinfo\text{-}t\ TYPE('b::xmem\text{-}type)$

assumes $cgrd$: *c-guard* p

shows ($hmem\text{-}upd$ ($heap\text{-}update$ ($PTR('b)$ & $(p \rightarrow f)$) v)) =

($\lambda s.$ ($hmem\text{-}upd$ ($heap\text{-}update\ p$ (*update-ti* t (*to-bytes-p* v) ($h\text{-}val$ ($hmem\ s$) p)))) s)

using *heap-update-field-root-conv'* [*OF fl match cgrd, of v*]

using *heap.upd-cong* **by** *blast*

lemma *L2-modify-heap-update-field-root-conv*:

fixes $p::'a::xmem\text{-}type$ ptr

assumes fl : *field-lookup* (*typ-info-t* $TYPE('a)$) $f\ 0 = Some$ (*adjust-ti* (*typ-info-t* $TYPE('b::xmem\text{-}type)$)) fld ($fld\text{-}update \circ (\lambda x \ . \ x)$), n

assumes $fg\text{-}cons$: $fg\text{-}cons\ fld$ ($fld\text{-}update \circ (\lambda x \ . \ x)$)

assumes $cgrd$: *c-guard* p

shows *L2-modify* ($\lambda s.$ $hmem\text{-}upd$ ($heap\text{-}update$ ($PTR('b)$ & $(p \rightarrow f)$) ($v\ s$)) s) =

L2-modify ($\lambda s.$ ($hmem\text{-}upd$ ($heap\text{-}update\ p$ ($fld\text{-}update$ ($\lambda \cdot.$ $v\ s$) ($h\text{-}val$ ($hmem\ s$) p)))) s)

by (*simp add: heap-update-field-root-conv* [*OF fl fg-cons cgrd*])

lemma *L2-modify-heap-update-field-root-conv'* [*L2-modify-heap-update-field-root-conv*]:

fixes $p::'a::xmem\text{-}type$ ptr

assumes fl : *field-lookup* (*typ-info-t* $TYPE('a)$) $f\ 0 = Some$ (t , n)

assumes $match$: *export-uinfo* $t = typ\text{-}uinfo\text{-}t\ TYPE('b::xmem\text{-}type)$

assumes $cgrd$: *c-guard* p

shows *L2-modify* ($\lambda s.$ $hmem\text{-}upd$ ($heap\text{-}update$ ($PTR('b)$ & $(p \rightarrow f)$) ($v\ s$)) s) =

L2-modify ($\lambda s.$ ($hmem\text{-}upd$ ($heap\text{-}update\ p$ (*update-ti* t (*to-bytes-p* ($v\ s$)) ($h\text{-}val$ ($hmem\ s$) p)))) s)

by (*simp add: heap-update-field-root-conv'* [*OF fl match cgrd*])

lemma *L2-modify-heap-update-ptr-coerce-packed-conv'*:

fixes $p::'c::packed\text{-}type$ ptr

assumes $cgrd$: *c-guard* p

assumes sz : $size\text{-}td\ (typ\text{-}info\text{-}t\ TYPE('c)) = size\text{-}td\ (typ\text{-}info\text{-}t\ TYPE('a::packed\text{-}type))$

shows $L2\text{-}modify\ (\lambda s. hmem\text{-}upd\ (heap\text{-}update\ (PTR\text{-}COERCE('c \rightarrow 'a)\ p)\ (COERCE('c \rightarrow 'a)\ (v\ s))))\ s =$
 $L2\text{-}modify\ (\lambda s. (hmem\text{-}upd\ (heap\text{-}update\ p\ (v\ s))))\ s$

using $heap\text{-}update\text{-}ptr\text{-}coerse\text{-}packed\ [simplified\ size\text{-}of\text{-}def, OF\ cgrd\ sz]\ heap.\text{upd}\text{-}cong$
by $(metis\ assms(2)\ coerce\text{-}cancel\text{-}packed\ size\text{-}of\text{-}def)$

lemma $L2\text{-}modify\text{-}heap\text{-}update\text{-}ptr\text{-}coerce\text{-}packed\text{-}conv[L2\text{-}modify\text{-}heap\text{-}update\text{-}field\text{-}root\text{-}conv]$:

fixes $p::'c::packed\text{-}type\ ptr$
assumes $cgrd$: $c\text{-}guard\ p$
assumes sz : $size\text{-}td\ (typ\text{-}info\text{-}t\ TYPE('c)) = size\text{-}td\ (typ\text{-}info\text{-}t\ TYPE('a::packed\text{-}type))$

shows $L2\text{-}modify\ (\lambda s. hmem\text{-}upd\ (heap\text{-}update\ (PTR\text{-}COERCE('c \rightarrow 'a)\ p)\ (v\ s)))\ s =$
 $L2\text{-}modify\ (\lambda s. (hmem\text{-}upd\ (heap\text{-}update\ p\ (COERCE('a \rightarrow 'c)\ (v\ s))))\ s)$

using $heap\text{-}update\text{-}ptr\text{-}coerse\text{-}packed\ [simplified\ size\text{-}of\text{-}def, OF\ cgrd\ sz]\ heap.\text{upd}\text{-}cong$
by $(metis\ assms(2)\ coerce\text{-}cancel\text{-}packed\ size\text{-}of\text{-}def)$

lemma $L2\text{-}modify\text{-}heap\text{-}update\text{-}ptr\text{-}coerce\text{-}packed\text{-}field\text{-}root\text{-}conv [L2\text{-}modify\text{-}heap\text{-}update\text{-}field\text{-}root\text{-}conv]$:

fixes $p::'c::\{xmem\text{-}type, packed\text{-}type\}\ ptr$
assumes fl : $field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('a::\{xmem\text{-}type, packed\text{-}type\}))\ f\ 0 =$
 $Some\ (t, n)$

assumes $match$: $export\text{-}uinfo\ t = typ\text{-}uinfo\text{-}t\ TYPE('b::xmem\text{-}type)$
assumes $cgrd$: $c\text{-}guard\ (PTR\text{-}COERCE('c \rightarrow 'a)\ p)$
assumes sz : $size\text{-}td\ (typ\text{-}info\text{-}t\ TYPE('c)) = size\text{-}td\ (typ\text{-}info\text{-}t\ TYPE('a))$
shows $L2\text{-}modify\ (\lambda s. hmem\text{-}upd\ (heap\text{-}update\ (PTR('b)\ \&((PTR\text{-}COERCE('c \rightarrow 'a)\ p) \rightarrow f))\ (v\ s)))\ s =$
 $L2\text{-}modify\ (\lambda s. (hmem\text{-}upd\ (heap\text{-}update\ p\ (coerce\text{-}map\ (update\text{-}ti\ t\ (to\text{-}bytes\text{-}p\ (v\ s)))\ (h\text{-}val\ (hmem\ s)\ p))))\ s)$

using $heap\text{-}update\text{-}field\text{-}ptr\text{-}coerce\text{-}from\text{-}bytes\text{-}packed\ [simplified\ size\text{-}of\text{-}def, OF\ fl\ match\ cgrd\ sz]\ heap.\text{upd}\text{-}cong$
by $meson$
end

lemma $L2\text{-}try\text{-}state\text{-}asssumeE\text{-}bindE$:

$L2\text{-}try\ (assume\text{-}result\text{-}and\text{-}state\ P\ >>= f) = assume\text{-}result\text{-}and\text{-}state\ P\ >>=$
 $(\lambda x. L2\text{-}try\ (f\ x))$

apply $(rule\ spec\text{-}monad\text{-}eqI)$
apply $(auto\ simp\ add: runs\text{-}to\text{-}iff\ L2\text{-}defs)$
done

lemma $refines\text{-}L2\text{-}try\text{-}L2\text{-}seq\text{-}nondet$:

assumes $g\ [unfolded\ THIN\text{-}def, rule\text{-}format]$: $PROP\ THIN\ (\bigwedge v\ t. refines\ (L2\text{-}try\ (g\ v))\ (g' v)\ t\ t\ (rel\text{-}prod\ rel\text{-}liftE\ (=)))$
assumes $f\ [unfolded\ THIN\text{-}def]$: $PROP\ THIN\ (Trueprop\ (refines\ f\ f'\ s\ s\ (rel\text{-}prod\ rel\text{-}liftE\ (=))))$
shows $refines\ (L2\text{-}try\ (L2\text{-}seq\ f\ g))\ (f' >>= g')\ s\ s\ (rel\text{-}prod\ rel\text{-}liftE\ (=))$

```

unfolding L2-try-def L2-seq-def try-nested-bind-exception-or-result-conv
using g f
apply (clarsimp simp add: refines-def-old L2-defs succeeds-bind reaches-bind
reaches-try succeeds-bind-exception-or-result
rel-liftE-def unnest-exn-def reaches-bind-exception-or-result succeeds-bind-exception-or-result
split: xval-splits sum.splits)
apply (intro conjI allI impI)
apply (metis case-xval-simps(2))
apply clarsimp
subgoal for r s' r'
apply (cases r)
subgoal
apply (cases r')
subgoal
apply (clarsimp simp add: default-option-def Exn-def [symmetric])
by (meson Exn-neq-Result obj-sumE)

subgoal
apply (clarsimp simp add: default-option-def Exn-def [symmetric])
subgoal for s1 y r1
apply (cases r1)
subgoal
apply (clarsimp simp add: default-option-def Exn-def [symmetric])
by (metis Exn-neq-Result Result-eq-Result case-xval-simps(1) old.sum.simps(5)
sum-all-ex(2))
subgoal
by (auto simp add: default-option-def Exn-def [symmetric])
done
done
done
subgoal
apply (cases r')

subgoal
apply (clarsimp simp add: default-option-def Exn-def [symmetric])
by (meson Exn-neq-Result sum-all-ex(2))
subgoal
apply (clarsimp simp add: default-option-def Exn-def [symmetric])
subgoal for s1 r1
apply (cases r1)
subgoal
apply (clarsimp simp add: default-option-def Exn-def [symmetric])
by (metis Result-eq-Result case-xval-simps(1) old.sum.simps(6) sum-all-ex(2))
subgoal
by (metis case-xval-simps(2))
done
done
done
done

```

done

lemma *refines-L2-try-pure:*

assumes f : *refines* f (return f') s s (rel-prod rel-liftE (=))
shows *refines* (L2-try f) (return f') s s (rel-prod rel-liftE (=))
unfolding L2-defs L2-guarded-def **using** f
by (*fastforce simp add: refines-def-old reaches-try rel-liftE-def*)

lemma *refines-liftE-conv:* *refines* f f' s t (rel-prod rel-liftE (=)) \longleftrightarrow

refines f (liftE f') s t (=)
unfolding L2-defs L2-guarded-def
by (*auto simp add: refines-def-old reaches-liftE rel-liftE-def*)

lemma *refines-rel-liftE-L2-try-on-exit:*

assumes f : *refines* f_c f_a s s (rel-prod rel-liftE (=))
shows *refines* (L2-try (on-exit f_c cleanup)) (on-exit f_a cleanup) s s (rel-prod rel-liftE (=))
using f **unfolding** L2-defs try-def *refines-map-value-left-iff*
apply (*rule refines-rel-prod-eq-on-exit[THEN refines-weaken]*)
apply (*auto simp: rel-liftE-def*)
done

lemma *map-value-on-exit:*

map-value g (on-exit f cleanup) = *on-exit* (*map-value* g f) cleanup
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff on-exit-def on-exit'-def*)
done

lemma *L2-try-on-exit:* L2-try (on-exit f cleanup) = *on-exit* (L2-try f) cleanup

unfolding L2-try-def try-def
by (*simp add: map-value-on-exit*)

context *stack-heap-state*

begin

lemma *L2-try-with-fresh-stack-ptr:*

L2-try (with-fresh-stack-ptr n init f) = *with-fresh-stack-ptr* n init (λp . L2-try (f p))
by (*simp add: with-fresh-stack-ptr-def L2-try-state-assumeE-bindE L2-try-on-exit*)
end

lemma *map-value-guard-on-exit:*

map-value g (*guard-on-exit* f P cleanup) = *guard-on-exit* (*map-value* g f) P cleanup
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff on-exit'-def*)

done

lemma *L2-try-guard-on-exit*:

$L2\text{-try } (\text{guard-on-exit } f P \text{ cleanup}) = \text{guard-on-exit } (L2\text{-try } f) P \text{ cleanup}$

unfolding *L2-try-def try-def*

by (*simp add: map-value-guard-on-exit*)

lemma *map-value-assume-on-exit*:

$\text{map-value } g (\text{assume-on-exit } f P \text{ cleanup}) = \text{assume-on-exit } (\text{map-value } g f) P \text{ cleanup}$

apply (*rule spec-monad-eqI*)

apply (*auto simp add: runs-to-iff on-exit'-def*)

done

lemma *L2-try-assume-on-exit*:

$L2\text{-try } (\text{assume-on-exit } f P \text{ cleanup}) = \text{assume-on-exit } (L2\text{-try } f) P \text{ cleanup}$

unfolding *L2-try-def try-def*

by (*simp add: map-value-assume-on-exit*)

context *typ-heap-typing*

begin

lemma *L2-try-guard-with-fresh-stack-ptr*:

$L2\text{-try } (\text{guard-with-fresh-stack-ptr } n \text{ init } f) = \text{guard-with-fresh-stack-ptr } n \text{ init } (\lambda p. L2\text{-try } (f p))$

unfolding *guard-with-fresh-stack-ptr-def L2-try-state-assumeE-bindE assume-stack-alloc-def*

L2-try-guard-on-exit ..

lemma *L2-try-assume-with-fresh-stack-ptr*:

$L2\text{-try } (\text{assume-with-fresh-stack-ptr } n \text{ init } f) = \text{assume-with-fresh-stack-ptr } n \text{ init } (\lambda p. L2\text{-try } (f p))$

unfolding *assume-with-fresh-stack-ptr-def L2-try-state-assumeE-bindE assume-stack-alloc-def*

L2-try-assume-on-exit ..

lemma *L2-try-with-fresh-stack-ptr*:

$L2\text{-try } (\text{with-fresh-stack-ptr } n \text{ init } f) = \text{with-fresh-stack-ptr } n \text{ init } (\lambda p. L2\text{-try } (f p))$

by (*simp add: with-fresh-stack-ptr-def L2-try-state-assumeE-bindE assume-stack-alloc-def L2-try-on-exit*)

end

lemma *refines-L2-seq*:

assumes *f*: $\text{refines } f f' s s' Q$

assumes *ll*: $\bigwedge e e' t t'. Q (\text{Exn } e, t) (\text{Exn } e', t') \implies R (\text{Exn } e, t) (\text{Exn } e', t')$

assumes *lr*: $\bigwedge e v' t t'. Q (\text{Exn } e, t) (\text{Result } v', t') \implies \text{refines } (\text{throw } e) (g' v')$
t t' R

assumes *rl*: $\bigwedge v e' t t'. Q (\text{Result } v, t) (\text{Exn } e', t') \implies \text{refines } (g v) (\text{throw } e') t$
 $t' R$
assumes *rr*: $\bigwedge v v' t t'. Q (\text{Result } v, t) (\text{Result } v', t') \implies \text{refines } (g v) (g' v') t$
 $t' R$
shows $\text{refines } (L2\text{-seq } f g) (L2\text{-seq } f' g') s s' R$
unfolding *L2-seq-def* **using** *fl lr rl rr*
by (*rule refines-bind-bind-exn*)

lemma *rel-Nonlocal-Nonlocal*: $\text{rel-Nonlocal } (\text{Nonlocal } x) x$
by (*simp add: rel-Nonlocal-def*)

lemmas *rel-sum-Inl* = *rel-sum.intros(1)*
lemmas *rel-sum-Inr* = *rel-sum.intros(2)*

lemma *rel-sum-eq*: $\text{rel-sum } (=) (=) x x$
by (*auto simp add: rel-sum.simps*)

end

theory *LocalVarExtract*
imports *SimplConv L2Defs*
begin

lemma *Collect-prod-inter*:
 $\{(s, t). P s t\} \cap \{(s, t). Q s t\} = \{(s, t). P s t \wedge Q s t\}$
by (*fastforce intro: set-eqI*)

lemma *Collect-prod-union*:
 $\{(s, t). P s t\} \cup \{(s, t). Q s t\} = \{(s, t). P s t \vee Q s t\}$
by (*fastforce intro: set-eqI*)

end

theory *AutoCorresSimpset*
imports *SimplBucket*
begin

lemma *globals-surj*: *surj globals*
apply (*rule surjI* [**where** $f = \lambda x. \text{undefined}(\text{globals} := x)$])
apply *simp*
done

lemma *triv-ex-apply*: $\exists s1\ s0. f\ s0 = s1$
by (*iprover*)

lemmas *ptr-val-ptr-add-simps* =
ptr-add-word32
ptr-add-word32-signed
ptr-add-word64
ptr-add-word64-signed

ML ‹

```

val AUTOCORRES-SIMPSET =
  @{{context}} delsimps (
    (* interferes with heap-lift *)
    @{{thms fun-upd-apply}}
    (* affects boolean expressions *)
    @ @{{thms word-neq-0-conv neq0-conv}}
    (* interferes with struct-rewrite *)
    @ @{{thms ptr-coerce.simps ptr-add-0-id}}
    (* oversimplifies Spec sets prior to L2 stage
       (we will control this explicitly in L2Peephole) *)
    @ @{{thms CollectPairFalse}}
    @ @{{thms unsigned-of-nat unsigned-of-int}}
    @ @{{thms map-of-default.simps}}
    @ @{{thms field-lvalue-append}} @
    @{{thms ptr-val-ptr-add-simps}} )
  addsimps (
    (* Needed for L2corres-spec *)
    @{{thms globals-surj}}
  )
  addsimps (
    (* builds up by monad-equiv tactic, and record updates *)
    @{{thms triv-ex-apply}} (* FIXME Shouldn't this already be handled by HOL.ex-simps
and simproc defined-Ex.*)
  )
  addsimps (@{{thms ptr-NULL-conv}})
  delsimprocs
  [ @{{simproc case-prod-beta}}, @{{simproc case-prod-eta}} ]
  |> simpset-of

```

›

end

theory *ExceptionRewrite*
imports *L1Defs L1Peephole*
begin

definition *always-fail* $P A \equiv \forall s. P s \longrightarrow \text{run } A s = \text{Failure}$

lemma *always-fail-fail* [*simp*]: *always-fail* $P \text{ fail}$
by (*clarsimp simp: always-fail-def top-post-state-def*)

lemma *bindE-alwaysfail-lhs*: $\llbracket \text{always-fail } (\lambda-. \text{True}) L \rrbracket \Longrightarrow \text{always-fail } (\lambda-. \text{True})$
 $(L >>= R)$
apply (*clarsimp simp: always-fail-def run-bind*)
done

lemma *bindE-alwaysfail-rhs*: $\llbracket \text{always-progress } L; \text{no-throw } (\lambda-. \text{True}) L; \bigwedge x. \text{always-fail } (\lambda-. \text{True}) (R x) \rrbracket \Longrightarrow \text{always-fail } (\lambda-. \text{True}) (L >>= R)$
apply (*clarsimp simp: always-fail-def no-throw-def always-progress-def*)
apply (*clarsimp simp add: run-bind runs-to-partial.rep-eq*
bind-post-state-eq-top[unfolded top-post-state-def])
subgoal premises *prems* **for** s
using *prems(1,2)[rule-format, of s] prems(4)*
apply (*cases run L s*)
subgoal by (*auto simp: Ball-def split: prod.splits exception-or-result-splits*)
subgoal by *force*
done
done

lemma *handleE'-alwaysfail-lhs*: $\llbracket \text{always-fail } (\lambda-. \text{True}) L \rrbracket \Longrightarrow \text{always-fail } (\lambda-. \text{True}) (L <\text{catch}> R)$
by (*clarsimp simp: always-fail-def run-catch*)

lemma *handleE'-alwaysfail-rhs*: $\llbracket \text{always-progress } L; \text{no-return } (\lambda-. \text{True}) L; \bigwedge r. \text{always-fail } (\lambda-. \text{True}) (R r) \rrbracket \Longrightarrow \text{always-fail } (\lambda-. \text{True}) (L <\text{catch}> R)$
apply (*clarsimp simp: always-fail-def no-return-def always-progress-def*)
subgoal premises *prems* **for** s
proof (*cases run L s*)
case *Failure*
then show *?thesis* **by** (*auto simp add: run-catch*)
next
case (*Success x2*)
then show *?thesis* **using** *prems*
apply (*clarsimp simp add: run-catch runs-to-partial-def-old reaches-def*)
apply (*erule-tac x=s in allE*)
apply (*simp add: succeeds-def*)
apply (*clarsimp simp add: bot-post-state-def Exn-def default-option-def*)
using *image-iff* **by** *fastforce*
qed
done

lemma *alwaysfail-noreturn: always-fail P A \implies no-return P A*
by (*clarsimp simp: always-fail-def no-return-def runs-to-partial-def-old succeeds-def*
top-post-state-def)

lemma *alwaysfail-nothrow: always-fail P A \implies no-throw P A*
by (*clarsimp simp: always-fail-def no-throw-def runs-to-partial-def-old succeeds-def*
top-post-state-def)

lemma *no-return-Success-distrib-aux1:*
assumes $\forall (a ,b) \in X. \exists e. a = \text{Exception } e \wedge e \neq \text{default}$
shows $X = \text{apfst } (\text{Exception } o \text{ the-Exception}) \text{ ' } X \wedge (\forall a \in \text{fst ' } X. \text{the-Exception } a \neq \text{default})$
using *assms*
by (*force simp add: image-def apfst-def map-prod-def*)

lemma *no-return-Success-distrib-aux2:*
assumes $\forall (a ,b) \in X. \exists e. a = \text{Exception } e \wedge e \neq \text{default}$
shows $P (\bigsqcup x \in X. \text{case } x \text{ of } (\text{Exception } e, t) \Rightarrow f e t \mid (\text{Result } v, t) \Rightarrow g v t) \longleftrightarrow (P (\bigsqcup (e, t) \in \text{apfst } (\text{the-Exception}) \text{ ' } X. f e t))$
apply (*subst no-return-Success-distrib-aux1 [OF assms]*)
by (*smt (verit, ccfv-SIG) apfst-conv assms case-exception-or-result-Exception case-prodE case-prod-conv comp-apply image-cong image-image the-Exception-simp*)

lemma *no-return-Success-distrib-aux3:*
assumes $\forall (a ,b) \in X. \exists e. a = \text{Exception } e \wedge e \neq \text{default}$
shows $P (\bigsqcup x \in X. \text{case } x \text{ of } (\text{Exception } e, t) \Rightarrow f e t \mid (\text{Result } v, t) \Rightarrow g v t) \longleftrightarrow (X = \text{apfst } (\text{Exception } o \text{ the-Exception}) \text{ ' } X \wedge P (\bigsqcup (e, t) \in \text{apfst } (\text{the-Exception}) \text{ ' } X. f e t))$
using *no-return-Success-distrib-aux1 [OF assms] no-return-Success-distrib-aux2 [OF assms, where P=P]*
apply *simp*
done

lemma *no-return-bindE:*
no-return $(\lambda-. \text{True}) A \implies (A \gg= B) = A$
apply (*rule spec-monad-ext*)
subgoal for *s*
apply (*cases run A s*)
subgoal

```

    by (auto simp add: run-bind)
  subgoal for x2
    apply (cases x2 = {})
    apply (force simp: run-bind )
    apply (clarsimp simp add: run-bind no-return-def runs-to-partial-def-old
      reaches-def succeeds-def top-post-state-def bot-post-state-def)
    apply (erule-tac x=s in allE)
    apply (clarsimp, safe)
    apply (subst (asm) no-return-Success-distrib-aux2 [where P =  $\lambda X. X \neq$ 
      Success x2])
    subgoal
      apply auto
      done
    subgoal
      apply (force simp: image-image split-beta' pure-post-state-def Sup-Success)
      done
    done
  done
done
done

```

lemma *L1-skip-always-progress* [*simp, L1except, always-progress-intros*]:
always-progress L1-skip
unfolding *L1-defs* **by** (*simp add: always-progress-intros*)

lemma *L1-modify-always-progress* [*simp, L1except, always-progress-intros*]:
always-progress (L1-modify m)
unfolding *L1-defs* **by** (*simp add: always-progress-intros*)

lemma *L1-guard-always-progress* [*simp, L1except, always-progress-intros*]:
always-progress (L1-guard g)
unfolding *L1-defs* **by** (*simp add: always-progress-intros*)

lemma *L1-init-always-progress* [*simp, L1except, always-progress-intros*]:
always-progress (L1-init v)
unfolding *L1-defs* **by** (*simp add: always-progress-intros*)

lemma *L1-spec-always-progress* [*simp, L1except, always-progress-intros*]: *always-progress*
(L1-spec s)
unfolding *L1-defs* **by** (*simp add: always-progress-intros*)

lemma *L1-throw-always-progress* [*simp, L1except, always-progress-intros*]: *always-progress*
L1-throw
unfolding *L1-defs* **by** (*simp add: always-progress-intros*)

lemma *L1-fail-always-progress* [*simp,L1except, always-progress-intros*]: *always-progress L1-fail*

unfolding *L1-defs* **by** (*simp add: always-progress-intros*)

lemma *L1-condition-always-progress*[*always-progress-intros*]:

$\llbracket \text{always-progress } L; \text{always-progress } R \rrbracket \implies \text{always-progress } (L1\text{-condition } C L R)$

unfolding *L1-defs* **by** (*simp add: always-progress-intros*)

lemma *L1-seq-always-progress*[*always-progress-intros*]:

$\llbracket \text{always-progress } L; \text{always-progress } R \rrbracket \implies \text{always-progress } (L1\text{-seq } L R)$

unfolding *L1-defs* **by** (*simp add: always-progress-intros*)

lemma *L1-catch-always-progress*[*always-progress-intros*]:

$\llbracket \text{always-progress } L; \text{always-progress } R \rrbracket \implies \text{always-progress } (L1\text{-catch } L R)$

unfolding *L1-defs* **by** (*simp add: always-progress-intros*)

lemma *L1-while-always-progress*[*always-progress-intros*]:

$\text{always-progress } B \implies \text{always-progress } (L1\text{-while } C B)$

unfolding *L1-defs* **by** (*simp add: always-progress-intros*)

lemma *L1-call-always-progress*[*always-progress-intros*]: $\llbracket \text{always-progress } b \rrbracket \implies \text{always-progress } (L1\text{-call } a b c d e)$

unfolding *L1-call-def* **by** (*simp add: always-progress-intros*)

lemma *L1-skip-nothrow* [*simp,L1except*]: *no-throw* ($\lambda\cdot$. *True*) *L1-skip*

by (*clarsimp simp: L1-defs no-throw-def*)

lemma *L1-modify-nothrow* [*simp,L1except*]: *no-throw* ($\lambda\cdot$. *True*) (*L1-modify m*)

by (*clarsimp simp: L1-defs no-throw-def; runs-to-vcg*)

lemma *L1-guard-nothrow* [*simp,L1except*]: *no-throw* ($\lambda\cdot$. *True*) (*L1-guard g*)

by (*clarsimp simp: L1-defs no-throw-def; runs-to-vcg*)

lemma *L1-init-nothrow* [*simp,L1except*]: *no-throw* ($\lambda\cdot$. *True*) (*L1-init a*)

by (*clarsimp simp: L1-defs no-throw-def; runs-to-vcg*)

lemma *L1-spec-nothrow* [*simp,L1except*]: *no-throw* ($\lambda\cdot$. *True*) (*L1-spec a*)

by (*clarsimp simp: L1-defs no-throw-def; runs-to-vcg*)

lemma *L1-assume-nothrow* [*simp,L1except*]: *no-throw* ($\lambda\cdot$. *True*) (*L1-assume a*)

by (*clarsimp simp: L1-defs no-throw-def; runs-to-vcg*)

lemma *L1-fail-nothrow* [*simp,L1except*]: *no-throw* ($\lambda\cdot$. *True*) *L1-fail*

by (clarsimp simp: L1-defs no-throw-def; runs-to-vcg)

lemma L1-while-nothrow [L1except]: no-throw ($\lambda\cdot$. True) B \implies no-throw ($\lambda\cdot$. True) (L1-while C B)
 apply (clarsimp simp: L1-while-def)
 apply (clarsimp simp: no-throw-def)
 apply (rule runs-to-partial-whileLoop-exn [where I= λr -. case r of Exn e \implies False | - \implies True])
 apply simp
 apply simp
 apply simp
 apply (fastforce simp add: runs-to-partial-def-old)
 done

lemma L1-catch-nothrow-lhs: \llbracket no-throw ($\lambda\cdot$. True) L $\rrbracket \implies$ no-throw ($\lambda\cdot$. True) (L1-catch L R)
 apply (clarsimp simp: L1-defs no-return-def no-throw-def)
 apply (runs-to-vcg)
 apply (fastforce simp add: runs-to-partial-def-old)
 done

lemma L1-catch-nothrow-rhs: \llbracket no-throw ($\lambda\cdot$. True) R $\rrbracket \implies$ no-throw ($\lambda\cdot$. True) (L1-catch L R)
 apply (clarsimp simp: L1-defs no-return-def no-throw-def)
 apply (runs-to-vcg)
 apply (fastforce simp add: runs-to-partial-def-old)
 done

lemma L1-catch-nothrow-both [L1except]: no-throw ($\lambda\cdot$. True) L \vee no-throw ($\lambda\cdot$. True) R \implies no-throw ($\lambda\cdot$. True) (L1-catch L R)
 by (auto simp add: L1-catch-nothrow-lhs L1-catch-nothrow-rhs)

lemma bind-nothrow-simple: \llbracket no-throw ($\lambda\cdot$. True) L; ($\bigwedge x$. no-throw ($\lambda\cdot$. True) (R x)) $\rrbracket \implies$ no-throw ($\lambda\cdot$. True) (bind L R)
 apply (clarsimp simp: no-throw-def)
 apply (runs-to-vcg)
 apply (fastforce simp add: runs-to-partial-def-old)
 done

lemma L1-seq-nothrow [L1except]: \llbracket no-throw ($\lambda\cdot$. True) L; no-throw ($\lambda\cdot$. True) R $\rrbracket \implies$ no-throw ($\lambda\cdot$. True) (L1-seq L R)
 apply (clarsimp simp: L1-defs no-throw-def)
 apply (runs-to-vcg)
 apply (fastforce simp add: runs-to-partial-def-old)
 done

lemma L1-condition-nothrow [L1except]:
 \llbracket no-throw ($\lambda\cdot$. True) L; no-throw ($\lambda\cdot$. True) R $\rrbracket \implies$ no-throw ($\lambda\cdot$. True)


```

(L1-condition C L R)
  apply (clarsimp simp: L1-defs no-throw-def)
  apply (runs-to-vcg)
  apply (fastforce simp add: runs-to-partial-def-old)+
done

lemma L1-call-nothrow[L1except]: no-throw ( $\lambda$ -. True) y  $\implies$  no-throw ( $\lambda$ -. True)
(L1-call x y z q r)
  apply (clarsimp simp: L1-call-def L1-defs no-throw-def)
  apply (runs-to-vcg)
  by (auto simp add: runs-to-partial-def-old)
  (fastforce simp add: succeeds-bind reaches-bind)+

lemma no-throw-state-assumeE: no-throw ( $\lambda$ -. True) (assume-result-and-state p)
  apply (clarsimp simp: L1-defs no-throw-def)
  apply (runs-to-vcg)
done

lemma no-throw-on-exit:
  assumes c: no-throw ( $\lambda$ -. True) c
  shows no-throw ( $\lambda$ -. True) (on-exit c cleanup)
  using c
  apply (clarsimp simp: L1-defs no-throw-def on-exit'-def on-exit-def)
  apply (runs-to-vcg)
  apply (fastforce simp add: runs-to-partial-def-old succeeds-bind reaches-bind)
done

named-theorems no-throw-with-fresh-stack-ptr
lemma (in stack-heap-raw-state) no-throw-with-fresh-stack-ptr[no-throw-with-fresh-stack-ptr]:
  assumes c:  $\bigwedge p$ . no-throw ( $\lambda$ -. True) (c p)
  shows no-throw ( $\lambda$ -. True) (with-fresh-stack-ptr n init c)
  apply (simp add: with-fresh-stack-ptr-def)
  apply (rule bind-nothrow-simple)
  apply (rule no-throw-state-assumeE)
  apply (rule no-throw-on-exit)
  apply (rule c)
done

lemmas L1-nothrows =
  L1-seq-nothrow
  L1-skip-nothrow
  L1-modify-nothrow
  L1-condition-nothrow
  L1-catch-nothrow-both
  L1-while-nothrow
  L1-spec-nothrow
  L1-assume-nothrow
  L1-guard-nothrow

```

L1-init-nothrow
L1-call-nothrow
L1-fail-nothrow

lemma *L1-throw-noreturn* [*simp,L1except*]: *no-return* ($\lambda\cdot$. *True*) *L1-throw*
apply (*clarsimp simp: L1-defs no-return-exn-def*)
done

lemma *L1-fail-noreturn* [*simp,L1except*]: *no-return* ($\lambda\cdot$. *True*) *L1-fail*
apply (*clarsimp simp: L1-defs no-return-exn-def*)
done

lemma *L1-seq-noreturn-lhs*: *no-return* ($\lambda\cdot$. *True*) *L* \implies *no-return* ($\lambda\cdot$. *True*) (*L1-seq*
L R)
apply (*clarsimp simp: L1-defs no-return-exn-def*)
apply *runs-to-vcg*
apply (*fastforce simp add: runs-to-partial-def-old*)
done

lemma *L1-seq-noreturn-rhs*: \llbracket *no-return* ($\lambda\cdot$. *True*) *R* $\rrbracket \implies$ *no-return* ($\lambda\cdot$. *True*)
(*L1-seq L R*)
apply (*clarsimp simp: L1-defs no-return-exn-def*)
apply *runs-to-vcg*
apply (*fastforce simp add: runs-to-partial-def-old*)
done

lemma *L1-catch-noreturn*: \llbracket *no-return* ($\lambda\cdot$. *True*) *L*; *no-return* ($\lambda\cdot$. *True*) *R* $\rrbracket \implies$
no-return ($\lambda\cdot$. *True*) (*L1-catch L R*)
apply (*clarsimp simp: L1-defs no-return-exn-def*)
apply *runs-to-vcg*
apply (*fastforce simp add: runs-to-partial-def-old*)
done

lemma *L1-condition-noreturn*: \llbracket *no-return* ($\lambda\cdot$. *True*) *L*; *no-return* ($\lambda\cdot$. *True*) *R* \rrbracket
 \implies *no-return* ($\lambda\cdot$. *True*) (*L1-condition C L R*)
apply (*clarsimp simp: L1-defs no-return-exn-def*)
apply *runs-to-vcg*
apply (*fastforce simp add: runs-to-partial-def-old*)
done

lemma *bindE-noreturn-lhs*: \llbracket *no-return* ($\lambda\cdot$. *True*) *L* $\rrbracket \implies$ *no-return* ($\lambda\cdot$. *True*) (*L*
 \gg \gg *R*)
apply (*clarsimp simp: L1-defs no-return-def*)
apply *runs-to-vcg*
apply (*fastforce simp add: runs-to-partial-def-old*)

done

lemma *bindE-noreturn-rhs*: $\llbracket \bigwedge x. \text{no-return } (\lambda-. \text{True}) (R x) \rrbracket \implies \text{no-return } (\lambda-. \text{True}) (L \gg R)$
apply (*clarsimp simp: L1-defs no-return-def*)
apply *runs-to-vcg*
apply (*fastforce simp add: runs-to-partial-def-old*)
done

lemma *on-exit-noreturn*:
assumes *c*: *no-return* $(\lambda-. \text{True})$ *c*
shows *no-return* $(\lambda-. \text{True})$ (*on-exit c cleanup*)
using *c*
unfolding *on-exit-def on-exit'-def*
apply (*clarsimp simp add: no-return-def*)
apply *runs-to-vcg*
apply (*fastforce simp add: runs-to-partial-def-old succeeds-bind reaches-bind*)
done

named-theorems *no-return-with-fresh-stack-ptr*

lemma (**in** *stack-heap-raw-state*) *no-return-with-fresh-stack-ptr*[*no-return-with-fresh-stack-ptr*]:
assumes *c*: $\bigwedge p. \text{no-return } (\lambda-. \text{True}) (c p)$
shows *no-return* $(\lambda-. \text{True})$ (*with-fresh-stack-ptr n init c*)
apply (*simp add: with-fresh-stack-ptr-def*)
apply (*rule bindE-noreturn-rhs*)
apply (*rule on-exit-noreturn*)
apply (*rule c*)
done

lemma *L1-fail-alwaysfail* [*simp, L1except*]: *always-fail* $(\lambda-. \text{True})$ *L1-fail*
by (*clarsimp simp: L1-defs*)

lemma *L1-seq-alwaysfail-lhs*: $\llbracket \text{always-fail } (\lambda-. \text{True}) L \rrbracket \implies \text{always-fail } (\lambda-. \text{True}) (L1\text{-seq } L R)$
by (*auto intro!: bindE-alwaysfail-lhs simp: L1-defs*)

lemma *L1-seq-alwaysfail-rhs*: $\llbracket \text{always-progress } L; \text{no-throw } (\lambda-. \text{True}) L; \text{always-fail } (\lambda-. \text{True}) R \rrbracket \implies \text{always-fail } (\lambda-. \text{True}) (L1\text{-seq } L R)$
by (*auto intro!: bindE-alwaysfail-rhs simp: L1-defs*)

lemma *L1-catch-alwaysfail-lhs*: $\llbracket \text{always-fail } (\lambda-. \text{True}) L \rrbracket \implies \text{always-fail } (\lambda-. \text{True}) (L1\text{-catch } L R)$
by (*clarsimp simp add: L1-defs always-fail-def run-catch*)

lemma *L1-catch-alwaysfail-rhs*: $\llbracket \text{always-progress } L; \text{no-return } (\lambda-. \text{True}) L; \text{al-$

```

ways-fail ( $\lambda$ -. True) R ]
 $\implies$  always-fail ( $\lambda$ -. True) (L1-catch L R)
apply (clarsimp simp add: L1-defs always-fail-def always-progress-def no-return-def
run-catch
  bind-post-state-eq-top[unfolded top-post-state-def] prod-eq-iff runs-to-partial.rep-eq
  split: prod.splits xval-splits)
subgoal premises prems for s
using prems(1,2)[rule-format, of s] prems(4)
apply (cases run L s)
apply (auto simp: split-beta' Exn-def default-option-def)
done
done

```

```

lemma L1-condition-alwaysfail: [ always-fail ( $\lambda$ -. True) L; always-fail ( $\lambda$ -. True)
R ]  $\implies$  always-fail ( $\lambda$ -. True) (L1-condition C L R)
by (auto simp add: L1-defs always-fail-def run-condition)

```

```

lemma L1-catch-nothrow []:
no-throw ( $\lambda$ -. True) A  $\implies$  L1-catch A E = A
unfolding L1-defs
apply (rule spec-monad-eqI)
apply (clarsimp simp add: runs-to-iff)
apply (fastforce simp add: no-throw-def runs-to-partial-def-old runs-to-def-old)+
done

```

```

lemma L1-seq-noreturn [L1except]:
no-return ( $\lambda$ -. True) A  $\implies$  L1-seq A B = A
unfolding L1-defs
apply (rule spec-monad-eqI)
apply (clarsimp simp add: runs-to-iff)
apply (fastforce simp add: no-return-def runs-to-partial-def-old runs-to-def-old)+
done

```

```

lemma L1-catch-throw [L1except]:
L1-catch L1-throw E = E
unfolding L1-defs
apply (rule spec-monad-eqI)
apply (auto simp add: runs-to-iff)
done

```

```

lemma anything-to-L1-fail [L1except]:
always-fail ( $\lambda$ -. True) A  $\implies$  A = L1-fail
unfolding L1-defs
apply (rule spec-monad-ext)
apply (auto simp add: always-fail-def top-post-state-def)

```

done

lemmas *L1-is-local-simps* [*L1except*] = *is-local-simps*

lemma *L1-catch-return* [*L1except*]: ($\bigwedge s.$ *is-local* (*get-exn* (*upd-exn* *s*))) \implies

(*L1-seq*
 (*L1-modify* (*upd-exn*))
 (*L1-condition* ($\lambda s.$ *is-local* (*get-exn* *s*))
 L1-skip *L1-throw*))
 = *L1-modify* (*upd-exn*))

unfolding *L1-defs*

apply (*rule spec-monad-eqI*)

apply (*auto simp add: runs-to-iff*)

done

lemma *L1-catch-L1-seq-nothrow* [*L1except*]:

$\llbracket \text{no-throw } (\lambda-. \text{ True}) A \rrbracket \implies \text{L1-catch } (\text{L1-seq } A B) C = \text{L1-seq } A (\text{L1-catch } B C)$

unfolding *L1-defs*

apply (*rule spec-monad-eqI*)

apply (*clarsimp simp add: runs-to-iff*)

apply (*fastforce simp add: no-throw-def runs-to-partial-def-old runs-to-def-old*)+

done

lemma *L1-catch-simple-seq* [*L1except*]:

L1-catch (*L1-seq* *L1-skip* *B*) *E* = (*L1-seq* *L1-skip* (*L1-catch* *B* *E*))

L1-catch (*L1-seq* *L1-fail* *B*) *E* = (*L1-seq* *L1-fail* (*L1-catch* *B* *E*))

L1-catch (*L1-seq* (*L1-modify* *m*) *B*) *E* = (*L1-seq* (*L1-modify* *m*) (*L1-catch* *B* *E*))

L1-catch (*L1-seq* (*L1-spec* *s*) *B*) *E* = (*L1-seq* (*L1-spec* *s*) (*L1-catch* *B* *E*))

L1-catch (*L1-seq* (*L1-assume* *f*) *B*) *E* = (*L1-seq* (*L1-assume* *f*) (*L1-catch* *B* *E*))

L1-catch (*L1-seq* (*L1-guard* *g*) *B*) *E* = (*L1-seq* (*L1-guard* *g*) (*L1-catch* *B* *E*))

L1-catch (*L1-seq* (*L1-init* *i*) *B*) *E* = (*L1-seq* (*L1-init* *i*) (*L1-catch* *B* *E*))

apply (*subst L1-catch-seq-join* | *rule L1-nothrows* | *clarsimp simp: L1-defs* | *rule bind-nothrow-simple*)+

done

declare *L1-catch-simple-seq*(6) [*L1opt*]

lemma *L1-catch-L1-init-seq'*[*L1opt*, *L1except*]: *L1-catch* (*L1-seq* (*L1-seq* (*L1-init* *i*) *A*) *B*) *E* = (*L1-seq* (*L1-init* *i*) (*L1-catch* (*L1-seq* *A* *B*) *E*))

apply (*subst L1-catch-seq-join*)

apply *simp*

apply (*subst L1-seq-assoc*)

apply *simp*

done

lemma *L1-catch-call-simple-seq* [*L1except*]:
no-throw ($\lambda\cdot$. *True*) *b* \implies *L1-catch* (*L1-seq* (*L1-call* *a b c d e*) *B*) *E* = (*L1-seq*
(*L1-call* *a b c d e*) (*L1-catch* *B E*))
apply (*subst L1-catch-seq-join*)
apply (*rule L1-call-nothrow*)
apply (*assumption*)
apply *simp*
done

lemma *L1-catch-single* [*L1except*]:
L1-catch (*L1-skip*) *E* = *L1-skip*
L1-catch (*L1-fail*) *E* = *L1-fail*
L1-catch (*L1-modify m*) *E* = *L1-modify m*
L1-catch (*L1-spec s*) *E* = *L1-spec s*
L1-catch (*L1-assume f*) *E* = *L1-assume f*
L1-catch (*L1-guard g*) *E* = *L1-guard g*
L1-catch (*L1-init i*) *E* = *L1-init i*
apply (*subst L1-catch-nothrow | rule L1-nothrows | clarsimp simp: L1-defs | rule*
bind-nothrow-simple)
done

lemma *L1-catch-call-single* [*L1except*]: *no-throw* ($\lambda\cdot$. *True*) *b* \implies *L1-catch* (*L1-call*
a b c d e) *E* = *L1-call a b c d e*
apply (*subst L1-catch-nothrow*)
apply (*rule L1-call-nothrow*)
apply *assumption*
apply *simp*
done

lemma *L1-catch-single-while* [*L1except*]:
 \llbracket *no-throw* ($\lambda\cdot$. *True*) *B* $\rrbracket \implies$ *L1-catch* (*L1-while C B*) *E* = *L1-while C B*
apply (*rule L1-catch-nothrow*)
apply (*rule L1-while-nothrow*)
apply *assumption*
done

lemma *L1-catch-seq-while* [*L1except*]:
 \llbracket *no-throw* ($\lambda\cdot$. *True*) *B* $\rrbracket \implies$ *L1-catch* (*L1-seq* (*L1-while C B*) *X*) *E* = *L1-seq*
(*L1-while C B*) (*L1-catch X E*)
apply (*rule L1-catch-seq-join [symmetric]*)
apply (*rule L1-while-nothrow*)
apply *assumption*
done

lemma *L1-catch-single-cond* [*L1except*]:
L1-catch (*L1-condition C L R*) *E* = *L1-condition C* (*L1-catch L E*) (*L1-catch R*
E)

```

unfolding L1-defs
apply (rule spec-monad-eqI)
apply (auto simp add: runs-to-iff)
done

```

```

lemma L1-catch-cond-seq:
  L1-catch (L1-seq (L1-condition C L R) B) E = L1-condition C (L1-catch (L1-seq
L B) E) (L1-catch (L1-seq R B) E)
apply (subst L1-condition-distrib)
apply (rule L1-catch-single-cond)
done

```

```

lemma L1-catch-seq-cond-noreturn-ex:
   $\llbracket \text{no-return } (\lambda\cdot. \text{True}) E \rrbracket \implies (L1-catch (L1-seq (L1-condition c A B) C) E) =$ 
   $(L1-seq (L1-catch (L1-condition c A B) E) (L1-catch C E))$ 
unfolding L1-defs
apply (rule spec-monad-eqI)
apply (clarsimp simp add: runs-to-iff)
by (auto simp add: no-return-def runs-to-partial-def-old runs-to-def-old Exn-def)
  (metis Exception-eq-Result reaches-succeeds default-option-def) +

```

```

lemmas L1-catch-seq-cond-nothrow = L1-catch-L1-seq-nothrow [OF L1-condition-nothrow]

```

```

end

```

19.2 Nested Exceptions

```

theory L2ExceptionRewrite
imports
  L2Defs
  ExceptionRewrite
begin

```

```

synthesize-rules L2-rel-spec-monad

```

19.3 Transformations from single level exceptions to nested exceptions

```

lemma catch-yeild-map-value-conv:  $(f <catch> (\lambda e. \text{yield } (g e))) =$ 
   $(\text{map-value } (\lambda x. \text{case } x \text{ of } \text{Exn } e \Rightarrow g e \mid \text{Result } v \Rightarrow \text{Result } v) f)$ 
apply (rule spec-monad-eqI)
apply (auto simp add: runs-to-iff elim!: runs-to-weaken split: xval-splits)
done

```

lemma *rel-spec-monad-rel-xval-try-catch*:
assumes *bdy*: *rel-spec-monad* *Q* (*rel-xval* ($\lambda e e'$. (*rel-sum* *L* *R*) (*from-xval* (*f* *e*)) *e'*) *R*) *B* *B'*
shows *rel-spec-monad* *Q* (*rel-xval* *L* *R*) (*L2-catch* *B* (λe . *yield* (*f* *e*))) (*L2-try* *B'*)
apply (*clarsimp simp add: rel-spec-monad-iff-refines*)
apply (*intro conjI*)
subgoal for *s t*
using *bdy*
apply (*clarsimp simp add: rel-spec-monad-iff-refines*)
apply (*erule-tac x=s in allE*)
apply (*erule-tac x=t in allE*)
apply *safe*
unfolding *L2-defs try-def catch-yield-map-value-conv*
apply (*erule refines-map-value*)
apply (*auto simp add: rel-xval.simps rel-sum.simps unnest-exn-def rel-sum.simps*)
done
subgoal for *s t*
using *bdy*
apply (*clarsimp simp add: rel-spec-monad-iff-refines*)
apply (*erule-tac x=s in allE*)
apply (*erule-tac x=t in allE*)
apply *safe*
unfolding *L2-defs try-def catch-yield-map-value-conv*
apply (*erule refines-map-value*)
apply (*auto simp add: rel-xval.simps rel-sum.simps unnest-exn-def rel-sum.simps*)
done
done

lemma *rel-spec-monad-L2-seq-rel-xval-result-eq*:
assumes *f-f'*: *rel-spec-monad* *S* (*rel-xval* *L* (=)) *f* *f'*
assumes *Res-Res*: $\bigwedge v$. *rel-spec-monad* *S* (*rel-xval* *L* (=)) (*g* *v*) (*g'* *v*)
shows *rel-spec-monad* *S* (*rel-xval* *L* (=)) (*L2-seq* *f* *g*) (*L2-seq* *f'* *g'*)
unfolding *L2-defs using assms*
by (*rule rel-spec-monad-rel-xval-result-eq-bind*)

lemma *rel-spec-monad-rel-xval-L2-unknown*:
rel-spec-monad (=) (*rel-xval* *L* (=)) (*L2-unknown* *ns*) (*L2-unknown* *ns*)
unfolding *L2-unknown-def*
by (*auto simp add: rel-spec-monad-def rel-set-def*)

lemma *rel-spec-monad-rel-xval-L2-modify*:
rel-spec-monad (=) (*rel-xval* *L* (=)) (*L2-modify* *f*) (*L2-modify* *f*)
unfolding *L2-modify-def*
by (*auto simp add: rel-spec-monad-def rel-set-def*)

lemma *rel-spec-monad-rel-xval-L2-gets*:
rel-spec-monad (=) (*rel-xval* *L* (=)) (*L2-gets* *f* *ns*) (*L2-gets* *f* *ns*)
unfolding *L2-gets-def*
by (*auto simp add: rel-spec-monad-def rel-set-def*)

lemma *rel-spec-monad-eq-rel-xval-L2-condition:*
assumes $f: \text{rel-spec-monad } (=) (\text{rel-xval } L (=)) f f'$
and $g: \text{rel-spec-monad } (=) (\text{rel-xval } L (=)) g g'$
shows $\text{rel-spec-monad } (=) (\text{rel-xval } L (=)) (L2\text{-condition } P f g) (L2\text{-condition } P f' g')$
unfolding *L2-condition-def*
apply (*rule rel-spec-monad-condition [OF - f g]*)
apply (*auto simp add: rel-fun-def*)
done

lemma *rel-spec-monad-L2-throw-sanitize-names:*
assumes $xy: R (\text{Exn } x) (\text{Exn } y)$
assumes $ns': \text{SANITIZE-NAMES } y \text{ ns } ns'$
shows $\text{rel-spec-monad } (=) R (L2\text{-throw } x \text{ ns}) (L2\text{-throw } y \text{ ns}')$
unfolding *L2-throw-def*
using xy
by (*auto simp add: rel-spec-monad-def rel-set-def*)

lemma *rel-spec-monad-rel-xval-L2-spec:*
 $\text{rel-spec-monad } (=) (\text{rel-xval } L (=)) (L2\text{-spec } r) (L2\text{-spec } r)$
unfolding *L2-spec-def*
apply (*rule rel-spec-monad-rel-xval-bind [where P=(=)]*)
subgoal by (*auto simp add: rel-spec-monad-def rel-set-def run-assert-result-and-state*)
subgoal by (*auto simp add: rel-spec-monad-def rel-set-def*)
done

lemma *rel-spec-monad-rel-xval-L2-assume:*
 $\text{rel-spec-monad } (=) (\text{rel-xval } L (=)) (L2\text{-assume } r) (L2\text{-assume } r)$
unfolding *L2-assume-def*
by (*auto simp add: rel-spec-monad-def rel-set-def split: prod.splits*)

lemma *rel-spec-monad-rel-xval-L2-guard:*
 $\text{rel-spec-monad } (=) (\text{rel-xval } L (=)) (L2\text{-guard } c) (L2\text{-guard } c)$
unfolding *L2-guard-def*
by (*auto simp add: rel-spec-monad-def rel-set-def run-guard*)

lemma *rel-spec-monad-rel-xval-L2-guarded:*
assumes $c\text{-}c': \text{rel-spec-monad } (=) (\text{rel-xval } L (=)) c c'$
shows $\text{rel-spec-monad } (=) (\text{rel-xval } L (=)) (L2\text{-guarded } g c) (L2\text{-guarded } g c')$
unfolding *L2-guarded-def*
apply (*rule rel-spec-monad-L2-seq-rel-xval-result-eq*)
apply (*rule rel-spec-monad-rel-xval-L2-guard*)
apply (*rule c-c'*)
done

lemma *rel-spec-monad-L2-fail:*
 $\text{rel-spec-monad } Q R (L2\text{-fail}) (L2\text{-fail})$
unfolding *L2-fail-def*

by (rule rel-spec-monad-fail)

lemma *rel-spec-monad-rel-xval-L2-call*:

assumes *emb*: $\bigwedge e. L (emb\ e) (emb'\ e)$

assumes *ns'*: *SANITIZE-NAMES emb' ns ns'*

shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*L2-call f emb ns*) (*L2-call f emb' ns'*)

unfolding *L2-call-def*

using *emb*

by (*auto simp: rel-spec-monad-iff-rel-spec rel-spec-map-value-right-iff*

rel-spec-map-value-left-iff map-exn-def

split: xval-split

intro!: rel-spec-refl)

lemma *rel-fun-eq-refl*: *rel-fun* (=) (=) *f f*

by (rule *rel-funI*) *simp*

lemma *rel-spec-monad-result-exec-concrete*:

assumes *m-m'*: *rel-spec-monad* (=) *R m m'*

shows *rel-spec-monad* (=) *R* (*exec-concrete st m*) (*exec-concrete st m'*)

apply (*clarsimp simp add: rel-spec-monad-iff-refines, intro conjI*)

subgoal for *s*

using *m-m'*

by (*fastforce simp add: rel-spec-monad-iff-refines refines-def-old succeeds-exec-concrete-iff reaches-exec-concrete*)

subgoal for *s*

using *m-m'*

by (*fastforce simp add: rel-spec-monad-iff-refines refines-def-old succeeds-exec-concrete-iff reaches-exec-concrete*)

done

lemma *rel-spec-monad-result-exec-abstract*:

assumes *m-m'*: *rel-spec-monad* (=) *R m m'*

shows *rel-spec-monad* (=) *R* (*exec-abstract st m*) (*exec-abstract st m'*)

apply (*clarsimp simp add: rel-spec-monad-iff-refines, intro conjI*)

subgoal for *s*

using *m-m'*

by (*auto simp add: rel-spec-monad-iff-refines refines-def-old succeeds-exec-abstract-iff reaches-exec-abstract*)

subgoal for *s*

using *m-m'*

by (*auto simp add: rel-spec-monad-iff-refines refines-def-old succeeds-exec-abstract-iff reaches-exec-abstract*)

done

lemma *rel-spec-monad-rel-project-L2-unknown'*:

assumes *surj-prj*: *surj prj*

assumes *names*: *SANITIZE-NAMES prj ns ns'*

shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-unknown ns*) (*L2-unknown*

ns')
unfolding *L2-unknown-def*
using *surj-prj*
by (*auto simp add: rel-spec-monad-def rel-set-def rel-project-def*)

Note that *prj* is the identity function on unit here

lemma *rel-spec-monad-rel-project-L2-modify:*
rel-spec-monad (=) (*rel-xval L (rel-project prj)*) (*L2-modify f*) (*L2-modify f*)
unfolding *L2-modify-def*
by (*auto simp add: rel-spec-monad-def rel-set-def rel-project-def*)

lemma *rel-spec-monad-rel-project-L2-gets':*
assumes *g: $\bigwedge x. g\ x = prj\ (f\ x)$*
assumes *names: SANITIZE-NAMES prj ns ns'*
shows *rel-spec-monad* (=) (*rel-xval L (rel-project prj)*) (*L2-gets f ns*) (*L2-gets g ns'*)
unfolding *L2-gets-def g*
by (*auto simp add: rel-spec-monad-def rel-set-def rel-project-def*)

lemma *rel-spec-monad-rel-project-L2-condition:*
assumes *f: rel-spec-monad* (=) (*rel-xval L (rel-project prj)*) *f f'*
assumes *g: rel-spec-monad* (=) (*rel-xval L (rel-project prj)*) *g g'*
shows *rel-spec-monad* (=) (*rel-xval L (rel-project prj)*) (*L2-condition c f g*)
(*L2-condition c f' g'*)
unfolding *L2-condition-def*
apply (*rule rel-spec-monad-condition [OF - f g]*)
apply (*auto simp add: rel-fun-def*)
done

lemma *rel-spec-monad-rel-project-L2-throw:*
assumes *x-xs: L x y*
assumes *names: SANITIZE-NAMES (L, x) ns ns'*
shows *rel-spec-monad* (=) (*rel-xval L (rel-project prj)*) (*L2-throw x ns*) (*L2-throw y ns'*)
unfolding *L2-throw-def* **using** *x-xs*
by (*auto simp add: rel-spec-monad-def rel-set-def rel-project-def*)

lemma *rel-spec-monad-rel-project-L2-spec:*
assumes *prj-surj: surj prj*
shows *rel-spec-monad* (=) (*rel-xval L (rel-project prj)*) (*L2-spec r*) (*L2-spec r*)
unfolding *L2-spec-def*
apply (*rule rel-spec-monad-rel-xval-bind [where P=(=)]*)
subgoal by (*auto simp add: rel-spec-monad-def rel-set-def run-assert-result-and-state*)
subgoal using prj-surj by (*auto simp add: rel-spec-monad-def rel-set-def rel-project-def*)
done

lemma *rel-spec-monad-rel-project-L2-assume:*
shows *rel-spec-monad* (=) (*rel-xval L (=)*) (*L2-assume r*) (*L2-assume r*)
unfolding *L2-assume-def*

by (auto simp add: rel-spec-monad-def rel-set-def split: prod.splits)

lemma *rel-spec-monad-rel-project-L2-guard:*

rel-spec-monad (=) (*rel-xval* L (*rel-project* prj)) (*L2-guard* c) (*L2-guard* c)

unfolding *L2-guard-def*

by (auto simp add: rel-spec-monad-def rel-set-def run-guard rel-project-def)

lemma *rel-spec-monad-rel-project-L2-seq:*

assumes *m-n*: *rel-spec-monad* (=) (*rel-xval* L (*rel-project* prj')) *m n*

assumes *f-g*: $\bigwedge x. \text{rel-spec-monad } (=) (\text{rel-xval } L (\text{rel-project } prj)) (f\ x) (g\ (prj'\ x))$

shows *rel-spec-monad* (=) (*rel-xval* L (*rel-project* prj)) (*L2-seq* m f) (*L2-seq* n g)

unfolding *L2-seq-def*

apply (*rule* *rel-spec-monad-rel-xval-bind* [*OF* *m-n*])

using *f-g*

by (*simp* add: *rel-project-conv*)

lemma *rel-spec-monad-rel-project-L2-guarded:*

assumes *c-c'*: *rel-spec-monad* (=) (*rel-xval* L (*rel-project* prj)) *c c'*

shows *rel-spec-monad* (=) (*rel-xval* L (*rel-project* prj)) (*L2-guarded* g c) (*L2-guarded* g c')

unfolding *L2-guarded-def* *L2-seq-def*

apply (*rule* *rel-spec-monad-rel-xval-bind* [**where** *P*=(=)])

apply (*rule* *rel-spec-monad-rel-xval-L2-guard*)

apply (*rule* *c-c'*)

done

lemma *rel-spec-monad-rel-project-L2-try:*

assumes *fg*: *rel-spec-monad* (=) (*rel-xval* (*rel-sum* L (*rel-project* prj)) (*rel-project* prj)) *f g*

shows *rel-spec-monad* (=) (*rel-xval* L (*rel-project* prj)) (*L2-try* f) (*L2-try* g)

unfolding *L2-try-def*

apply (*clarsimp* *simp* add: *rel-spec-monad-iff-refines*, *intro* *conjI*)

subgoal for *s*

using *fg*

apply (*clarsimp* *simp* add: *rel-spec-monad-iff-refines* *refines-def-old* *succeeds-try* *reaches-try*)

apply (*erule-tac* *x=s* **in** *allE*)

apply *clarsimp*

subgoal for *s' r'*

apply (*erule-tac* *x=r'* **in** *allE*)

apply (*erule-tac* *x=s'* **in** *allE*)

apply *clarsimp*

subgoal for *x*

apply (*cases* *x*)

subgoal **apply** (*clarsimp* *simp* add: *Exn-def* [*symmetric*] *default-option-def*)

by (*smt* (*verit*, *best*) *Exn-neq-Result* *rel-sum.simps* *rel-xval.simps*

unnest-exn-simps(1) *unnest-exn-simps*(2))

```

    subgoal
      apply (clarsimp simp add: Exn-def [symmetric] default-option-def)
      by (smt (verit, best) Result-neq-Exn rel-xval.simps unnest-exn-simps(3))
    done
  done
done
subgoal for s
  using fg
  apply (clarsimp simp add: rel-spec-monad-iff-refines refines-def-old succeeds-try
reaches-try)
  apply (erule-tac x=s in allE)
  apply clarsimp
  subgoal for s' r'
    apply (erule-tac x=r' in allE)
    apply (erule-tac x=s' in allE)
    apply clarsimp
  subgoal for x
    apply (cases x)
  subgoal apply (clarsimp simp add: Exn-def [symmetric] default-option-def)
    by (smt (verit, best) Exn-neq-Result rel-sum.simps rel-xval.simps
unnest-exn-simps(1) unnest-exn-simps(2))
  subgoal
    apply (clarsimp simp add: Exn-def [symmetric] default-option-def)
    by (smt (verit, best) Result-neq-Exn rel-xval.simps unnest-exn-simps(3))
  done
done
done
done
done

```

Tailored projection for local handler functions that may emerge in IO phase to handle exit case. These handlers are composed from liftE functions followed by rethrowing the exception. We do not attempt to optimise projections for the handlers.

lemma *rel-spec-monad-rel-project-L2-catch*:

```

  assumes f: rel-spec-monad (=) (rel-xval (=) (rel-project prj)) f f'
  assumes g:  $\bigwedge v. (rel-spec-monad (=) (rel-xval L (\lambda -. False))) (g v) (g v)$ 
  shows rel-spec-monad (=) (rel-xval L (rel-project prj)) (L2-catch f g) (L2-catch
f' g)
  unfolding L2-catch-def
  apply (rule rel-spec-monad-rel-xval-catch [OF f])
  using g using rel-fun-def rel-spec-monad-mono
  by (smt (verit, del-Insts) Exn rel-xval.cases)

```

lemma *rel-spec-monad-rel-project-liftE*:

```

  assumes f: rel-spec-monad (=) (rel-map the-Res OO rel-project prj OO rel-map
Result) f f'
  shows rel-spec-monad (=) (rel-xval L (rel-project prj)) (liftE f) (liftE f')
  using assms

```

```

apply (clarsimp simp add: rel-spec-monad-iff-refines, intro conjI)
subgoal for s
  using f
  apply (clarsimp simp add: rel-spec-monad-iff-refines refines-def-old reaches-liftE
)
  apply (erule-tac x=s in allE)
  apply clarsimp
  subgoal for s' r'
    apply (erule-tac x=r' in allE)
    apply (erule-tac x=s' in allE)
    by (metis rel-map-def rel-xval.Result relcomppE the-Result-simp)
  done
subgoal for s
  using f
  apply (clarsimp simp add: rel-spec-monad-iff-refines refines-def-old reaches-liftE
)
  apply (erule-tac x=s in allE)
  apply clarsimp
  subgoal for s' r'
    apply (erule-tac x=r' in allE)
    apply (erule-tac x=s' in allE)
    by (metis rel-map-def rel-xval.Result relcomppE the-Result-simp)
  done
done

```

lemma *rel-project-Res-conv*:

```

  (rel-map the-Res OO rel-project prj OO rel-map Result) = (rel-project (Result o
prj o the-Res))
  apply (intro ext)
  apply (simp add: rel-map-def rel-project-def relcompp.simps)
  done

```

lemma *rel-spec-monad-rel-project-id*:

```

shows rel-spec-monad (=) (rel-project (λv. v)) f f
by (simp add: rel-project-id(2) rel-spec-monad-eq-conv)

```

lemma *rel-spec-monad-rel-project-unit*:

```

fixes f:: (unit, 's) res-monad
shows rel-spec-monad (=) (rel-project (λv. Result ())) f f
using rel-spec-monad-rel-project-id
by (metis Result-unit-eq rel-projectI rel-spec-monad-mono)

```

lemma *rel-spec-monad-rel-project-liftE-unit-id*:

```

shows rel-spec-monad (=) (rel-xval L (rel-project (prj::unit ⇒ unit))) (liftE f)
(liftE f)
apply (rule rel-spec-monad-rel-project-liftE)
apply (simp add: rel-project-Res-conv comp-def rel-spec-monad-rel-project-unit)

```

done

lemma *rel-project-unit-eq*: $(\text{rel-project } (\text{prj}::\text{unit} \Rightarrow \text{unit})) = (=)$
apply (*rule ext*)
apply (*auto simp add: rel-project-def*)
done

lemma *rel-spec-monad-L2-seq-rel-xval-same-split*:
assumes *mn*: $\text{rel-spec-monad } R (\text{rel-xval } L (=)) m n$
and *fg*: $\bigwedge x. (\text{rel-spec-monad } R (\text{rel-xval } L (=))) (f x) (g x)$
shows $\text{rel-spec-monad } R (\text{rel-xval } L (=)) (L2\text{-seq } m f) (L2\text{-seq } n g)$
using *assms*
by (*rule rel-spec-monad-L2-seq-rel-xval-result-eq*)

lemma *rel-spec-monad-L2-while-rel-xval-same-split*:
assumes *B*: $\bigwedge x. (\text{rel-spec-monad } (=) (\text{rel-xval } L (=))) (B x) (B' x)$
shows $\text{rel-spec-monad } (=) (\text{rel-xval } L (=)) (L2\text{-while } C' B I' ns) (L2\text{-while } C' B' I' ns)$
unfolding *L2-while-def*
apply (*rule rel-spec-monad-whileLoop-exn*)
apply (*auto simp add: B*)
done

lemma *rel-spec-monad-rel-project-L2-call-adapt-emb*:
assumes *L*: $\bigwedge x. L (\text{emb } x) (\text{emb}' x)$
assumes *prj*: $\bigwedge v. \text{prj } v = v$
shows $\text{rel-spec-monad } (=) (\text{rel-xval } L (\text{rel-project } \text{prj})) (L2\text{-call } x \text{ emb } ns) (L2\text{-call } x \text{ emb}' ns)$
unfolding *L2-call-def*
using *prj L*
apply (*auto simp: map-exn-def rel-project-def*)
rel-spec-monad-iff-rel-spec rel-spec-map-value-left-iff rel-spec-map-value-right-iff

split: xval-splits
intro!: rel-spec-refl
done

lemma *rel-spec-monad-liftE-id*: $\text{rel-spec-monad } (=) (\text{rel-xval } (\lambda - . \text{False}) (=)) (\text{liftE } f) (\text{liftE } f)$
by (*auto simp add: rel-spec-monad-iff-refines refines-def-old reaches-liftE rel-xval.simps*)

lemma *rel-spec-monad-L2-seq-exit-handler*:
assumes $\bigwedge v. \text{rel-spec-monad } (=) (\text{rel-xval } L (\lambda - . \text{False})) (g v) (g' v)$
shows $\text{rel-spec-monad } (=) (\text{rel-xval } L (\lambda - . \text{False})) (L2\text{-seq } (\text{liftE } f) g) (L2\text{-seq } (\text{liftE } f) g')$
unfolding *L2-defs*
apply (*rule rel-spec-monad-bind-strong-exn [OF rel-spec-monad-liftE-id]*)

apply (*auto simp add: assms*)
done

lemma *rel-spec-monad-rel-project-L2-while'*:
assumes $C-C'$: $\bigwedge v s. C v s = C' (prj v) s$
assumes I' : $I' = prj I$
assumes *names: SANITIZE-NAMES* $prj ns ns'$
assumes $B-B'$: $\bigwedge v. rel-spec-monad (=) (rel-xval L (rel-project prj)) (B v) (B' (prj v))$
shows $rel-spec-monad (=) (rel-xval L (rel-project prj)) (L2-while C B I ns) (L2-while C' B' I' ns')$
unfolding *L2-while-def*
apply (*rule rel-spec-monad-whileLoop-exn*)
subgoal using I' **by** (*auto simp add: rel-project-def*)
subgoal using $C-C'$ **by** (*auto simp add: rel-fun-def rel-project-def*)
subgoal using $B-B'$ **by** (*auto simp add: rel-project-def*)
done

lemma *rel-spec-monad-rel-xval-on-exit*:
assumes $c-c'$: $rel-spec-monad (=) (rel-xval L (=)) c c'$
shows $rel-spec-monad (=) (rel-xval L (=)) (on-exit c cleanup) (on-exit c' cleanup)$
unfolding *on-exit-bind-exception-or-result-conv* **thm** *rel-spec-monad-bind-exception-or-result-strong*
apply (*rule rel-spec-monad-bind-exception-or-result-strong [OF c-c']*)
apply (*rule rel-spec-monad-bind-strong-exn [where P=rel-xval ($\lambda - . False$) (=)]*)
subgoal
apply (*subst (1 2) liftE-state-select [symmetric]*)
apply (*rule rel-spec-monad-liftE-id*)
done
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto simp add: rel-spec-monad-yield*)
done

lemma (*in stack-heap-raw-state*) *rel-spec-monad-rel-xval-with-fresh-stack-ptr*:
assumes c : $\bigwedge p. rel-spec-monad (=) (rel-xval L (=)) (c p) (c' p)$
shows $rel-spec-monad (=) (rel-xval L (=)) (with-fresh-stack-ptr n init (L2-VARS c nm)) (with-fresh-stack-ptr n init (L2-VARS c' nm))$
unfolding *with-fresh-stack-ptr-def L2-VARS-def*
apply (*rule rel-spec-monad-bind-strong-exn [where P=rel-xval ($\lambda - . False$) (=)]*)
subgoal
apply (*subst (1 2) liftE-assume-result-and-state [symmetric]*)
apply (*rule rel-spec-monad-liftE-id*)
done
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*simp add: rel-spec-monad-rel-xval-on-exit c*)
done

lemma *rel-spec-monad-rel-project-on-exit*:
assumes $c-c'$: *rel-spec-monad* (=) (*rel-xval* L (*rel-project* prj)) c c'
shows *rel-spec-monad* (=) (*rel-xval* L (*rel-project* prj)) (*on-exit* c *cleanup*) (*on-exit* c' *cleanup*)
unfolding *on-exit-bind-exception-or-result-conv* **unfolding** *on-exit-def*
apply (*rule* *rel-spec-monad-bind-exception-or-result-strong* [*OF* $c-c'$])
apply (*rule* *rel-spec-monad-bind-strong-exn* [**where** $P=rel-xval$ ($\lambda-$ -. *False*) (=)])
subgoal
apply (*subst* (1 2) *liftE-state-select* [*symmetric*])
apply (*rule* *rel-spec-monad-liftE-id*)
done
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto simp add: rel-spec-monad-yield*)
done

lemma (*in stack-heap-raw-state*) *rel-spec-monad-rel-project-with-fresh-stack-ptr*:
assumes c : $\bigwedge p.$ *rel-spec-monad* (=) (*rel-xval* L (*rel-project* prj)) (c p) (c' p)
shows *rel-spec-monad* (=) (*rel-xval* L (*rel-project* prj)) (*with-fresh-stack-ptr* n *init* ($L2-VARS$ c nm)) (*with-fresh-stack-ptr* n *init* ($L2-VARS$ c' nm))
unfolding *with-fresh-stack-ptr-def* $L2-VARS-def$
apply (*rule* *rel-spec-monad-bind-strong-exn* [**where** $P=rel-xval$ ($\lambda-$ -. *False*) (=)])
subgoal
apply (*subst* (1 2) *liftE-assume-result-and-state* [*symmetric*])
apply (*rule* *rel-spec-monad-liftE-id*)
done
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*simp add: rel-spec-monad-rel-project-on-exit* c)
done

lemma *rel-spec-monad-L2-VARS*:
assumes $f-f'$: *rel-spec-monad* P Q f f'
shows *rel-spec-monad* P Q ($L2-VARS$ f ns) ($L2-VARS$ f' ns)
unfolding $L2-VARS-def$
by (*rule* $f-f'$)

lemma *cond-return1*: ($\lambda a.$
 $L2-condition$ ($\lambda s.$ P a) ($L2-gets$ ($\lambda s.$ f a) ns)
 $(L2-throw$ (g a) xs)) =
 $(\lambda a.$ *yield* (*if* P a *then* *Result* (f a) *else* (*Exn* (g a))))
unfolding $L2-defs$
apply (*rule* *ext*)
apply (*rule* *spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *cond-return2*: $(\lambda(a, b).$
 L2-condition $(\lambda s. P a b) (L2\text{-gets } (\lambda s. f a b) ns)$
 $(L2\text{-throw } (g a b) xs)) =$
 $(\lambda(a, b). \text{yield } (if P a b \text{ then Result } (f a b) \text{ else } (Exn (g a b))))$
unfolding *L2-defs*
apply *(rule ext)*
apply *(rule spec-monad-eqI)*
apply *(auto simp add: runs-to-iff)*
done

lemma *rel-spec-monad-rel-xvalI*:
rel-spec-monad R (rel-xval (=) (=)) f g \implies rel-spec-monad R (=) f g
by *(simp add: rel-xval-eq)*

lemma *is-local-split*: $((is\text{-local } a \longrightarrow e = \text{Result } b) \wedge (\neg is\text{-local } a \longrightarrow e = \text{Exn } (the\text{-Nonlocal } a)))$
 $=$
 $(case a \text{ of Nonlocal } x \Rightarrow e = \text{Exn } x \mid - \Rightarrow e = \text{Result } b)$
by *(cases a) auto*

lemma *is-local-splitI* :
 $(case a \text{ of Nonlocal } x \Rightarrow e = \text{Exn } x \mid - \Rightarrow e = \text{Result } b) \implies$
 $((is\text{-local } a \longrightarrow e = \text{Result } b) \wedge (\neg is\text{-local } a \longrightarrow e = \text{Exn } (the\text{-Nonlocal } a)))$
apply *(simp add: is-local-split)*
done

lemma *if-is-local-cases*: $(if is\text{-local } e \text{ then } f \text{ else } g) = (case e \text{ of Nonlocal } x \Rightarrow g \mid - \Rightarrow f)$
by *(cases e) auto*

lemma *if-Break-cases*: $(if e = \text{Break} \text{ then } f \text{ else } g) = (case e \text{ of Break} \Rightarrow f \mid - \Rightarrow g)$
by *(cases e) auto*

lemma *if-Continue-cases*: $(if e = \text{Continue} \text{ then } f \text{ else } g) = (case e \text{ of Continue} \Rightarrow f \mid - \Rightarrow g)$
by *(cases e) auto*

lemma *if-Return-cases*: $(if e = \text{Return} \text{ then } f \text{ else } g) = (case e \text{ of Return} \Rightarrow f \mid - \Rightarrow g)$
by *(cases e) auto*

lemmas *if-c-exntype-cases* =
if-is-local-cases if-Break-cases if-Continue-cases if-Return-cases

lemmas *case-sum-c-exntype-swap* = *c-exntype.case-distrib*

lemma *ex-c-exntype-cases-distrib*: $(\exists a. P a \wedge$
 $(\text{case } e \text{ of Break} \Rightarrow \text{brk } a \mid \text{Continue} \Rightarrow \text{cnt } a \mid \text{Return} \Rightarrow \text{ret } a \mid \text{Goto}$
 $l \Rightarrow \text{goto } l a$
 $\mid \text{Nonlocal } gx \Rightarrow \text{nonlocal } gx a)) =$
 $(\text{case } e \text{ of Break} \Rightarrow \exists a. P a \wedge \text{brk } a \mid \text{Continue} \Rightarrow \exists a. P a \wedge \text{cnt } a \mid \text{Return} \Rightarrow$
 $\exists a. P a \wedge \text{ret } a$
 $\mid \text{Goto } l \Rightarrow \exists a. P a \wedge \text{goto } l a$
 $\mid \text{Nonlocal } gx \Rightarrow \exists a. P a \wedge \text{nonlocal } gx a)$

by (*auto split: c-exntype.splits*)

19.4 Transformations for procedure local exceptions

Introduce constant for relations to avoid higher order patterns in term net for rules

19.4.1 Transformations for *try* and *finally*

19.5 Transformations on global exceptions

definition

lift-exit-status :: $(\text{'e}, \text{'a}, \text{'s}) \text{exn-monad} \Rightarrow (\text{'e } c\text{-exntype}, \text{'a}, \text{'s}) \text{exn-monad}$

where

lift-exit-status *f* $\equiv \text{map-value } (\text{map-exn } \text{Nonlocal}) \text{ } f$

19.5.1 Removing unused tuple components

lemma *rel-project-eqI*: $\text{rel-spec-monad } (=) (\text{rel-xval } (=) (\text{rel-project } (\text{ETA-TUPLED } (\lambda v. v)))) \text{ } f g \Longrightarrow f = g$

by (*simp add: rel-project-id rel-xval-eq rel-spec-monad-eq-conv ETA-TUPLED-def*)

19.5.2 Setup basic rules

lemma *rel-xval-case-Nonlocal-sameI*:

assumes *L*: $\bigwedge v. e = \text{Nonlocal } v \Longrightarrow L (\text{Nonlocal } v) (\text{Nonlocal } v)$

assumes *R*: $\text{is-local } e \Longrightarrow R v v'$

shows $\text{rel-xval } L R (\text{case } e \text{ of Nonlocal } x \Rightarrow \text{Exn } (\text{Nonlocal } x) \mid - \Rightarrow \text{Result } v)$
 $(\text{case } e \text{ of Nonlocal } x \Rightarrow \text{Exn } (\text{Nonlocal } x) \mid - \Rightarrow \text{Result } v')$

using *L R*

```

    by (auto split: c-exntype.splits)

lemma rel-sum-map-sum-InlI: L (l x) v  $\implies$  rel-sum L R (map-sum l r (Inl x))
(Inl v)
  by (auto)

lemma rel-sum-map-sum-InrI: R (r x) v  $\implies$  rel-sum L R (map-sum l r (Inr x))
(Inr v)
  by (auto)

lemma rel-map-xval-xval-ExnI: L (l x) v  $\implies$  rel-xval L R (map-xval l r (Exn x))
(Exn v)
  by (auto)

lemma rel-map-xval-xval-ResultI: R (r x) v  $\implies$  rel-xval L R (map-xval l r (Result
x)) (Result v)
  by (auto)

ML <
structure Rel-Spec-Monad-Synthesize-Rules =
struct

fun gen-number-bounds len [] = []
  | gen-number-bounds len (x::xs) = ((len, []), x):: gen-number-bounds (len - 1)
xs;

fun number-bounds xs = gen-number-bounds (length xs - 1) xs

exception Incompatible-Use
(* Note that we only perform inner procedural analysis. Hence an global exception
(i.e.
* all constructors are Inl and there is no final Inr) are irrelevant for the analysis
*)
datatype result = Inl of result | Inr of int (* position in tuple, from left to right,
starting with 1 *)
datatype uses = Unused | Used | Propagated of result list

local
fun inc-bound n (i, aliases) = (i + n, map (fn i => i + n) aliases)
fun inc-bounds n = map (fn (b, x) => (inc-bound n b, x))
fun add-alias i (b, aliases) = (b, i::aliases)

fun gen-number-pos n [] = []
  | gen-number-pos n (x::xs) = (n, x)::gen-number-pos (n + 1) xs
val number-pos = gen-number-pos 1

fun eq-bound (b, aliases) i = member (op =) (b::aliases) i
fun same-bound (b1, aliases1) (b2, aliases2) = not (null (inter (op =) (b1::aliases1)
(b2::aliases2)))

```

```

fun propagation (Propagated -) = true
  | propagation - = false

fun merge-result (Inr x) (Inr y) = if x = y then [Inr x] else [Inr x, Inr y]
  | merge-result (Inl x) (Inl y) = map Inl (merge-result x y)
  | merge-result x y = [x, y]

fun pop-result (Inl x) = [x]
  | pop-result - = []

fun merge-uses Unused x = x
  | merge-uses x Unused = x
  | merge-uses Used x = Used
  | merge-uses x Used = Used
  | merge-uses (Propagated rs) (Propagated rs^) =
    Propagated (fold (fn res => fn rsx => flat (map (merge-result res) rsx)) rs'
  rs)

fun returned (Propagated rs) = get-first (fn Inr x => SOME x | - => NONE) rs
  | returned - = NONE

fun remove-return (Propagated rs) =
  let val rs' = filter-out (fn Inr x => true | - => false) rs
  in if null rs' then Unused else Propagated rs' end
  | remove-return x = x

fun return-to-used (Propagated rs) =
  if exists (fn Inr x => true | - => false) rs then Used else Propagated rs
  | return-to-used x = x

fun handled (Propagated rs) = Propagated (flat (map pop-result rs))
  | handled x = x

fun analyse-result bounds constr pos propagate t =
  case t of
  (Bound i) => map (fn (b, -) =>
    if eq-bound b i
    then if propagate then Propagated [constr pos] else Used
    else Unused) bounds
  | (t1 $ t2) =>
    let
      val rs1 = analyse-result bounds constr pos false t1
      val rs2 = analyse-result bounds constr pos false t2
    in map2 merge-uses rs1 rs2 end
  | - => map (K Unused) bounds

fun analyse-used bounds = analyse-result bounds Inr 0 false

```

```

fun merge-results uss = foldl1 (uncurry (map2 merge-uses)) uss

fun merge-results-default d uss = if null uss then d else merge-results uss

fun propagate-returns off = map2 (fn (b, x) => fn r =>
  (case returned r of SOME p => inc-bound off b |> add-alias (off - p) | - =>
  inc-bound off b, x))

fun merge-while-cond-bdy new-pos cond-uses bdy-uses = bdy-uses |> map (fn r =>
  case returned r of
    SOME p => merge-uses (nth cond-uses (new-pos p)) r
  | NONE => r)

datatype exn-constr = L | R
fun dest-throw-value (Const (@{const-name Inl}, -) $ x) =
  let val (constrs, bdy) = dest-throw-value x in (L :: constrs, bdy) end
| dest-throw-value (Const (@{const-name Inr}, -) $ x) =
  let val (constrs, bdy) = dest-throw-value x in (R :: constrs, bdy) end
| dest-throw-value x = ([], x)

fun local-exn cs = (List.last cs = R)
fun mk-exn [R] x = Inr x
  | mk-exn (c::cs) x = ((case c of L => Inl | - => raise Incompatible-Use) o mk-exn
  cs) x

fun extend-bounds xs base =
  let
    fun ext x base = if not (member (fn (x, y) => same-bound (fst x) (fst y)) base
  x) then base @ [x] else base
  in
    fold ext xs base
  end

exception Done of uses list

fun fold-done done f xs a =
  fold (fn x => fn a =>
    let
      val res = f x a
    in
      if done res then raise Done res else res
    end) xs a

fun fold-sup inf f xs =
  fold-done (forall (fn u => u = Used)) f xs inf

fun fold-merge inf bounds f = fold-sup (inf bounds) (fn x => fn a => merge-results
[a, f bounds x])

```

```

in
(*)
analyse-uses bounds t: returns the list of usages corresponding to the list of input
bounds.
each input bound is represented as ((de bruijn index, [aliases]), (name, type))
The position in the list corresponds to the position in the tuple of the binding.
Special care has to be taken to handle L2-while as this may shuffle the tuple and
introduce
various aliases in the initialisation expression, the condition and the body.
*)
fun analyse-uses bounds t =
  let
    val sup = map (K Used) bounds
    val inf = map (K Unused) bounds
    val done = forall (fn u => u = Used)
    fun fastpath uses = if done uses then raise Done uses else ()
    val fold-merge = fold-merge inf
    fun analyse bounds t =
      — We do not handle L2-catch only L2-try as this analysis is supposed to
      happen after replacing L2-catch by L2-try. After phase IO the freshly emerging
      L2-catch are only very local handler functions which we do not attempt to optimise
      case t of
        Const (@{const-name L2-gets}, -) $ Abs (-, -, bdy) $ - =>
          let
            val bounds = inc-bounds 1 bounds
            val vars = HOLogic.strip-tuple bdy |> number-pos
            val uss = map (fn (pos, t) => analyse-result bounds Inr pos true t) vars
          in
            merge-results-default (inf bounds) (* i.e. () is returned *) uss
          end
        | Const (@{const-name L2-throw}, -) $ t1 $ - =>
          let
            val (constrs, bdy) = dest-throw-value t1
            val vars = HOLogic.strip-tuple bdy |> number-pos
            val uss = map (fn (pos, t) => analyse-result bounds (mk-ern (L::constrs)))
          in
            merge-results-default (inf bounds) (* e.g. if Inr () is thrown *) uss
          end
        | Const (@{const-name L2-seq}, -) $ t1 $ t2 =>
          let
            val uses1 = analyse bounds t1
            val - = fastpath uses1
            val (off, bdy) = (Synthesize-Rules.strip-abs-prod t2) |>> length
            val bounds' = propagate-returns off bounds uses1
            val uses2 = analyse bounds' bdy
          in
            merge-results [map remove-return uses1, uses2]
          end
  end

```

```

    end
  | Const (@{const-name L2-try}, -) $ t1 =>
    let
      val uses1 = analyse bounds t1
      val handled-uses = map handled uses1
    in
      handled-uses
    end
  | Const (@{const-name L2-while}, -) $ c $ bdy $ vars $ - =>
    let
      val n = length bounds
      val vars = HOLogic.strip-tuple vars |> number-pos
      val off = length vars
      val uses-vars = map (fn (pos, t) => analyse-result bounds Inr pos true
t) vars
      |> merge-results-default (inf bounds)
      val (cond-bounds, cond-bdy) = Synthesize-Rules.strip-abs-prod c |>>
number-bounds
      (* extend bounds to cover at least the arity of cond and body, strip in
the end *)
      val bounds' = propagate-returns off bounds uses-vars |> extend-bounds
cond-bounds
      val uses-cond = analyse bounds' cond-bdy
      val - = @assert (forall (not o propagation) uses-cond)
      fun new-pos p =
        let val i = nth cond-bounds (p - 1) |> #1 |> #1
            in find-index (fn (b, -) => eq-bound b i) bounds' end
      val - = fastpath uses-cond
      val (-, bdy) = Synthesize-Rules.strip-abs-prod bdy
      val uses-bdy = analyse bounds' bdy
      val cond-bdy = merge-while-cond-bdy new-pos uses-cond uses-bdy
      val res = merge-results [uses-vars, take n uses-cond, take n cond-bdy]
    in
      res
    end
  | Const (@{const-name L2-unknown}, -) $ - => inf bounds
  | Const (@{const-name L2-call}, -) $ f $ emb $ - =>
    fold-merge bounds analyse [f, emb]
  | t1 $ t2 =>
    fold-merge bounds analyse [t1, t2]
  | Abs (-, -, bdy) => analyse-uses (inc-bounds 1 bounds) bdy
  | leaf => analyse-used bounds leaf;
in
  analyse bounds t
  handle Done - => sup
end

fun mk-tuple [] = (HOLogic.unit, HOLogic.unitT)
  | mk-tuple [(i, -), (-, T)] = (Bound i, T)

```



```

| mk-tuple (((i, -), (-, T)):: xs) =
  let
    val (t, T') = mk-tuple xs
  in (HOLogic.pair-const T T' $ Bound i $ t, HOLogic.mk-prodT (T, T')) end

fun result-from-rel prj-rel res =
  case (res, prj-rel) of
    (Inr p, @{\term-pat rel-sum - (rel-project ?prj)}) =>
      ((nth (Synthesize-Rules.arity-from-projection prj) (p - 1))
       handle Subscript => raise TERM (prj-from-rel: cannot infer projection ^
@{\make-string} (p, prj), [prj, prj-rel]))
  | (Inr -, @{\term-pat rel-sum - (=)}) => true
  | (Inr -, @{\term-pat (=)}) => true
  | (Inl l, @{\term-pat rel-sum ?L1 -}) => result-from-rel L1 l
  | - => raise TERM (prj-from-rel: result ' ^ @{\make-string} res ^ ' cannot be
extracted, [prj-rel])

fun prj-of-uses prj-rel us = us |> map (fn u =>
  case u of
    Used => true
  | Unused => false
  | Propagated rs => exists (result-from-rel prj-rel) rs) (* Value propagated until
end but still used as a result *)

local
val merge-prj = map2 (fn b1 => fn b2 => b1 orelse b2)
in
  (*
  A while loop in itself might have dependencies of the bound variables between the
  body and
  the condition. These dependencies should not be optimised away even if the value
  is no longer used
  after the while loop.
  *)
  fun constrain-projection prj-rel prj t =
    case t of
      Const (@{\const-name L2-seq},-) $ t1 $ t2 => constrain-projection prj-rel prj
t2
  | Const (@{\const-name L2-call},-) $ - $ - $ - => map (K true) prj
  (* As we don't recurse into body, preserve result type of call *)
  | Const (@{\const-name L2-condition},-) $ - $ t1 $ t2 =>
    let
      val prj1 = constrain-projection prj-rel prj t1
      val prj2 = constrain-projection prj-rel prj t2
      val merge = merge-prj prj1 prj2
    in
      merge
    end
end

```

```

    end
  | Const (@{const-name L2-while}, wT) $ c $ bdy $ vars $ names =>
    let
      val vars' = HLogic.strip-tupleT (fastype-of vars) |> number-bounds
        |> map (fn ((i, aliases), T) => ((i, aliases), (x ^ string-of-int i, T)))
      val (bounds, -) = mk-tuple vars'
      val w = Const (@{const-name L2-while}, wT) $ c $ bdy $ bounds $ names
      val uses = analyse-uses vars' w
      val uses = if null uses (* vars is unit *) then map (K Unused) prj else uses
      val relevant-uses = map remove-return uses
      val uses-needed = prj-of-uses prj-rel relevant-uses
      val merge = merge-prj prj uses-needed
    in
      merge
    end
  | - => prj
end
end

fun analyse-uses' ctxt prj-rel t =
  let
    val (bounds, bdy) = Synthesize-Rules.strip-abs-prod t
    val uses = analyse-uses (number-bounds bounds) bdy
    val - = Utils.verbose-msg 2 ctxt (fn - => analysed uses: ^ @{make-string}
    (map fst bounds ^^ uses))
  in
    prj-of-uses prj-rel uses
  end

fun mk-case-prod (bdy, bT) [] = (bdy, bT)
  | mk-case-prod (bdy, bT) [(x, T)] = (Abs (x, T, bdy), bT)
  | mk-case-prod (bdy, bT) [(x, xT), (y, yT)] = (HLogic.case-prod-const (xT, yT,
bT) $ (Abs (x, xT, Abs (y, yT, bdy))), HLogic.mk-prodT (xT, yT))
  | mk-case-prod (bdy, bT) ((x, xT)::xs) =
    let
      val (bdy', bT') = mk-case-prod (bdy, bT) xs
    in
      (HLogic.case-prod-const (xT, bT', bT) $ (Abs (x, xT, bdy')), HLogic.mk-prodT
(xT, bT')) end

fun gen-mk-projection Ts prj-enc =
  let
    val n = length prj-enc
    val tagged-bounds = prj-enc |> gen-number-bounds (n - 1)
      |> map (fn ((i, aliases), t) => ((i, aliases), t, (Tuple-Tools.mk-el-name (n -
i), Tuple-Tools.mk-elT' Ts (n - i))))
    val proj-bounds = map-filter (fn (i, t, xT) => if t then SOME (i, xT) else
NONE) tagged-bounds
    val vars = map #3 tagged-bounds
    val (bdy, bT) = mk-tuple proj-bounds
  end

```

```

    val prj = mk-case-prod (bdy, bT) vars
  in
    fst prj
  end

fun mk-projection prj-enc = gen-mk-projection (map Tuple-Tools.mk-elT (1 upto
length prj-enc)) prj-enc

val - = Tuple-Tools.assert-cterm (mk-projection [true,false,false,true])
  @ {cterm λ(x1, x2, x3, x4). (x1, x4)}

val - = Tuple-Tools.assert-cterm (mk-projection [false,false])
  @ {cterm λ(x1, x2). ()}

fun split-project-rule ctxt prj-var orig-names new-names rule =
  let val rule = Thm.trim-context rule
      in fn prj-arity => if null prj-arity then rule else
        let
          val orig-arity = length prj-arity
          val new-arity = prj-arity |> filter I |> length
          val name-arities = map (rpair orig-arity) orig-names @ map (rpair new-arity)
            new-names
          val prj = mk-projection prj-arity
          val ctxt' = Variable.declare-term prj ctxt
          val prj = Thm.cterm-of ctxt' prj
          val [rule'] = rule
            |> Drule.infer-instantiate ctxt' [(prj-var, prj)]
            |> single |> Proof-Context.export ctxt' ctxt
        in
          Tuple-Tools.gen-split-rule ctxt name-arities rule'
        end
      end
  end

— Tailored for case L2-seq, will it ever be used for other cases?
fun infer-projection-arity benv ctxt pattern [arity-var, constrain-var, prjL-var, prjR-var]
  concl =
  case Synthesize-Rules.match-rule-vars benv ctxt pattern [arity-var, constrain-var,
prjL-var, prjR-var] concl of
  [arity-stmt, constrain-stmt, prjL, prjR] =>
    let
      val prj-rel = infer-instantiate <L = prjL and prj = prjR in term <rel-sum
L (rel-project prj)>> ctxt
      val tagged-used = analyse-uses' ctxt prj-rel arity-stmt
      val - = Utils.verbose-msg 2 ctxt (fn - => derived projection (raw): ^
@ {make-string} tagged-used)
      val constrained-used = constrain-projection prj-rel tagged-used constrain-stmt
      val - = Utils.verbose-msg 2 ctxt (fn - => constrained projection: ^
@ {make-string} constrained-used)
      in constrained-used end

```

| - => []

```
fun mk-rel-spec-monad-pattern ctxt - (@{term-pat Trueprop (rel-spec-monad ?Q ?R
?f ?g)}) =
  let
    val mi = fold (curry Int.max) (map maxidx-of-term [Q, R, f, g]) 0
    val gT = Logic.incr-tvar (mi + 1) (fastype-of g) (* FIXME: do I need incr-tvar?
*)
    val pat = infer-instantiate ⟨Q = Q and R = R and f = f and g = ⟨Var ((-g,
mi + 1), gT)⟩
      in prop ⟨rel-spec-monad Q R f g⟩ ctxt
    in
      pat
    end
  | mk-rel-spec-monad-pattern - - - = raise Match
```

```
fun add-rel-spec-monad-split-rule rules-name = Synthesize-Rules.gen-add-split-rule
rules-name {only-schematic-goal = false} mk-rel-spec-monad-pattern
fun add-rel-spec-monad-split-rules rules-name = fold-map (fn (name, priority, names,
thm) => add-rel-spec-monad-split-rule rules-name name priority names thm)
```

```
val add-rel-spec-monad-infer-project-split-rule =
  Synthesize-Rules.gen-add-infer-project-split-rule mk-rel-spec-monad-pattern infer-projection-arity
```

```
fun add-rel-spec-monad-infer-project-split-rules rules-name rs context =
  context |> fold-map (fn (name, priority, names, new-names, prj-name, con-
strain-name, prjL-name, prjR-name, thm) =>
    let
      fun get-var name = Term.add-vars (Thm.prop-of thm) [] |> filter (fn ((n, -),-)
=> n = name)
        |> distinct (op =) |> the-single |> fst
      val prj-var = get-var prj-name
      val split = split-project-rule (Context.proof-of context) prj-var names new-names
    in
      add-rel-spec-monad-infer-project-split-rule split rules-name {only-schematic-goal
= false} name priority names [constrain-name, prjL-name, prjR-name] thm
    end) rs
```

```
fun add-rel-spec-monad-project-split-rule split rules-name name priority prj-name
thm context =
  Synthesize-Rules.gen-add-project-split-rule mk-rel-spec-monad-pattern split
  rules-name {only-schematic-goal = false} name priority prj-name thm context
```

```
fun add-rel-spec-monad-project-split-rules rules-name rs context =
  context |> fold-map (fn (name, priority, names, new-names, prj-name, thm) =>
    let
```

```

    val - = assert (not (null names)) add-rel-spec-monad-project-split-rules: ex-
    pecting at least one variable name
    fun get-var name = Term.add-vars (Thm.prop-of thm) [] |> filter (fn ((n, -),-)
    => n = name)
    |> distinct (op =) |> the-single |> fst
    val prj-var = get-var prj-name
    val split = split-project-rule (Context.proof-of context) prj-var names new-names
  thm
  in
    add-rel-spec-monad-project-split-rule split rules-name name priority prj-name
  thm
  end) rs

```

```

fun add-rel-spec-monad-rule rules-name = Synthesize-Rules.gen-add-rule rules-name
{only-schematic-goal = false} NONE mk-rel-spec-monad-pattern
fun add-rel-spec-monad-rules rules-name = fold (fn (name, priority, thm) =>
add-rel-spec-monad-rule rules-name name priority thm)

```

(* solve trivial leftover vars like ?f x y of type unit *)

```

fun smash-unit-vars ctxt = SUBGOAL (fn (t, i) =>

```

```

  let

```

```

    fun make-inst (v, T) =

```

```

      let

```

```

        val (argTs, @{typ unit}) = strip-type T

```

```

        fun abs T bdy = Abs (, T, bdy)

```

```

      in ((v, T), Thm.cterm-of ctxt (fold-rev abs argTs @{term ()})) end

```

```

    val unit-vars = Term.add-vars t []

```

```

    |> filter (fn (-, T) => body-type T = @{typ unit})

```

```

    val insts = map (make-inst) unit-vars

```

```

  in

```

```

    if null insts then no-tac

```

```

    else PRIMITIVE (Thm.instantiate (TVars.empty, Vars.make insts))

```

```

  end)

```

```

local

```

```

  val ss = simpset-of (put-simpset HOL-basic-ss @{context})

```

```

  addsimps @{thms HOL.simp-thms

```

```

    surj-def rel-project-conv rel-Nonlocal-conv Product-Type.prod.case

```

```

  c-exntype.case

```

```

    rel-liftE-apply rel-sum-eq-apply

```

```

    if-True if-False

```

```

    Product-Type.prod.inject c-exntype.inject List.list.inject String.char.inject

```

```

  c-exntype.distinct

```

```

    sum.map-ident unit-convs}

```

```

  |> Simplifier.add-cong @{thm if-cong})

```

```

in

```

```

fun clarsimp-solve-tac ctxt i =
  let
    val more-simps = Named-Theorems.get ctxt @ {named-theorems rel-spec-monad-rewrite-simps}
  in
    CHANGED (TRY (smash-unit-vars ctxt i)
      THEN clarsimp-tac (put-simpset ss ctxt addsimps more-simps) i
      THEN (REPEAT (resolve-tac ctxt @ {thms TrueI refl gen-unit-eq conjI} i)))
  end
end

fun get-prj-sum @ {term-pat rel-sum ?L ?R} @ {term-pat Inl ?x} = get-prj-sum L x
  | get-prj-sum @ {term-pat rel-sum ?L ?R} @ {term-pat Inr ?x} = get-prj-sum R x
  | get-prj-sum @ {term-pat rel-project ?prj} - = prj
  | get-prj-sum t - = t

fun get-prj @ {term-pat (?R, ?x)} = get-prj-sum R x
  | get-prj t = t

fun sanitize-names-tac ctxt = SUBGOAL (fn (t, i) =>
  if not (member (op =) (Term.add-const-names t [])) @ {const-name SANITIZE-NAMES})
then no-tac else
  let
    fun mk-abs [] t = t
      | mk-abs (T::Ts) t = Abs (Name.uu-, T, mk-abs Ts t)

    val concl = Utils.concl-of-subgoal-open t
    val @ {term-pat <Trueprop (SANITIZE-NAMES ?prj ?ns ?ns')>} = concl
    val (ns', -) = strip-comb ns'
  in
    if is-Var ns' then
      let
        val Var (ns', T) = ns'
        val names = Utils.decode-isa-list ns
        val prj' = get-prj prj
        val bs = Synthesize-Rules.arity-from-projection prj'

        val names' = (if length bs <= 1
          then filter-out (member (op =) [const <global-exn-var-clocal>]) names
          else if length bs = length names then filter (fn (b, -) => b) (bs ~
names) |> map snd
          else (warning (sanitize-names-tac: unexpected number of names in:
^ Syntax.string-of-term ctxt concl ^\n ^
          (bs, names): ^@ {make-string} (bs, names)
          ; names))
          |> Utils.encode-isa-list @ {typ nat} |> mk-abs (binder-types T) |>
Thm.cterm-of ctxt
        in
          PRIMITIVE (Thm.instantiate (TVars.empty, Vars.make [((ns', T), names')]))
        THEN

```

```

    resolve-tac ctxt @{thms sanitize-namesI} i
  end
else
  resolve-tac ctxt @{thms sanitize-namesI} i
end
handle Bind => no-tac
)

(* Avoid superfluous dependency of variable on constant terms, e.g. '?x ()'. Otherwise resolution results in
* non eta-contracted terms which make Net.match-term fail later on.
*)
fun norm-synthesis-var ctxt = SUBGOAL (fn (t,i) => fn st =>
  let
    val concl = Utils.concl-of-subgoal-open t
    val @{term-pat Trueprop (rel-spec-monad ?P ?Q ?f ?g)} = concl
    val (Var ((g, i), gT), args as (-:-)) = strip-comb g
    val (argTs, resT) = strip-type gT
    val n = length args
    val (appliedTs, unappliedTs) = chop n argTs
    val tagged-args = (args ~~ appliedTs) |> map (fn (arg, T) => (null (loose-bnos arg), (arg, T)))
    val (args', appliedTs') = tagged-args |> filter-out fst |> map snd |> split-list
    val - = if length args' = n then raise Bind else () (* avoid trivial identity instantiations *)
    val g' = Var ((g, Thm.maxidx-of st + 1), appliedTs' @ unappliedTs ----> resT)
    fun mk-inst [] g' b = g'
      | mk-inst ((tag, (t, T))::args) g' b =
        let
          val g' = if tag then g' else (g'$Bound b)
          in Abs (, T, mk-inst args g' (b - 1)) end
        val insts = [(((g,i), gT), Thm.ctrm-of ctxt (mk-inst tagged-args g' (length tagged-args - 1)))]
        in
          if null insts then no-tac st
          else PRIMITIVE (Thm.instantiate (TVars.empty, Vars.make insts)) st
        end
    handle Bind => no-tac st
  )

fun norm-resolve-split-thm rules-name ctxt =
  let
    val simp-ctxt = Simplifier.put-simpset HOL-basic-ss ctxt
    addsimps @{thms
      HOL.simp-thms if-True if-False
      c-exntype.case-distrib [where h=from-xval]
      Product-Type.prod.case-distrib[where h=from-xval]
      Product-Type.prod.case c-exntype.case from-xval-simps}
  end

```

```

    > Simplifier.add-cong @{thm if-weak-cong}
    > Simplifier.add-cong @{thm c-exntype.case-cong}
  in
    TRY' (norm-synthesis-var ctxt) THEN' Synthesize-Rules.resolve-split-thm rules-name
  ctxt
    THEN-ALL-NEW (Simplifier.simp-tac simp-ctxt)
  end
end
>

```

declare [[*verbose=3*]]

setup <

Context.theory-map (

Rel-Spec-Monad-Synthesize-Rules.add-rel-spec-monad-rules @{*synthesize-rules-name*
L2-rel-spec-monad} [

```

  (@{binding L2-unknown – rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-unknown}),
  (@{binding L2-modify – rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-modify}),
  (@{binding L2-gets – rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-gets}),
  (@{binding L2-condition – rel-xval}, 10, @{thm rel-spec-monad-eq-rel-xval-L2-condition}),
  (@{binding L2-throw – rel-xval}, 10, @{thm rel-spec-monad-L2-throw-sanitize-names}),
  (@{binding L2-spec – rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-spec}),
  (@{binding L2-assume – rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-assume}),
  (@{binding L2-guard – rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-guard}),
  (@{binding L2-guarded – rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-guarded}),
  (@{binding L2-fail}, 10, @{thm rel-spec-monad-L2-fail}),
  (@{binding L2-call – rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-call}),

```

```

  (@{binding exec-concrete}, 10, @{thm rel-spec-monad-result-exec-concrete}),
  (@{binding exec-abstract}, 10, @{thm rel-spec-monad-result-exec-abstract}),

```

```

  (@{binding L2-unknown – rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-unknown}),
  (@{binding L2-modify – rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-modify}),
  (@{binding L2-gets – rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-gets}),
  (@{binding L2-condition – rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-condition}),
  (@{binding L2-throw – rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-throw}),
  (@{binding L2-spec – rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-spec}),
  (@{binding L2-assume – rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-assume}),
  (@{binding L2-guard – rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-guard}),
  (@{binding L2-guarded – rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-guarded}),
  (@{binding L2-try – rel-project}, 20, @{thm rel-spec-monad-rel-project-L2-try}),
  (@{binding L2-catch – rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-catch}),
  (@{binding liftE unit – rel-project}, 10, @{thm rel-spec-monad-rel-project-liftE-unit-id})
]

```

```

#> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE

```

```

  (K (K (K @{pattern Trueprop (rel-sum ?L ?R (Inl ?l) ?V)}))) @{binding

```



```

rel-sum-Inl} 10 @{thm rel-sum.intros(1)}
#> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
(K (K (K @{pattern Trueprop (rel-sum ?L ?R (Inr ?r) ?V)}))) @{binding
rel-sum-Inr} 10 @{thm rel-sum.intros(2)}
#> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false}
NONE (K (K (K @{pattern Trueprop (rel-sum ?L ?R (map-sum ?l ?r (Inl
?x) ?v)}))) @{binding rel-sum-map-sum-InlI} 10 @{thm rel-sum-map-sum-InlI}
#> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
(K (K (K @{pattern Trueprop (rel-sum ?L ?R (map-sum ?l ?r (Inr ?x)
?v)}))) @{binding rel-sum-map-sum-InrI} 10 @{thm rel-sum-map-sum-InrI}

#> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
(K (K (K @{pattern Trueprop (rel-xval ?L ?R (Exn ?l) ?V)}))) @{binding
rel-xval-Exn} 10 @{thm rel-xval.intros(1)}
#> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
(K (K (K @{pattern Trueprop (rel-xval ?L ?R (Result ?r) ?V)}))) @{binding
rel-xval-Result} 10 @{thm rel-xval.intros(2)}
#> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false}
NONE (K (K (K @{pattern Trueprop (rel-xval ?L ?R (map-xval ?l ?r (Exn
?x) ?v)}))) @{binding rel-map-xval-xval-ExnI} 10 @{thm rel-map-xval-xval-ExnI}
#> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
(K (K (K @{pattern Trueprop (rel-xval ?L ?R (map-xval ?l ?r (Result ?x)
?v)}))) @{binding rel-map-xval-xval-ResultI} 10 @{thm rel-map-xval-xval-ResultI}

#> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
(K (K (K @{pattern Trueprop (rel-xval rel-Nonlocal ?R (case ?e of Nonlocal
v => Exn (Nonlocal v) | - => Result ?x) (Exn ?v'))})))
@{binding rel-xval-rel-Nonlocal-case-ExnI} 10 @{thm rel-xval-rel-Nonlocal-case-ExnI}
#> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
(K (K (K @{pattern Trueprop (rel-xval rel-Nonlocal ?R (case ?e of Nonlocal
v => Exn (Nonlocal v) | - => Result ?x) (Result ?v'))})))
@{binding rel-xval-rel-Nonlocal-case-ResultI} 10 @{thm rel-xval-rel-Nonlocal-case-ResultI}
#> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
(K (K (K @{pattern Trueprop (rel-xval rel-Nonlocal (=) (map-xval Nonlocal
?f (case ?e of Nonlocal x => Exn x | - => Result ?x) ?v)})))
@{binding rel-xval-rel-Nonlocal-map-xvalI} 10 @{thm rel-xval-rel-Nonlocal-map-xvalI}
#> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
(K (K (K @{pattern Trueprop (rel-xval ?L ?R (case ?e of Nonlocal x => Exn

```

```

(Nonlocal x) | - => Result ?v)
  (case ?e of Nonlocal x => Exn (Nonlocal x) | - => Result ?v'))))
  @{binding rel-xval-case-Nonlocal-sameI} 10 @{thm rel-xval-case-Nonlocal-sameI}
  #> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
  {only-schematic-goal = false} NONE
  (K (K (K @{pattern Trueprop (rel-project ?prj ?x ?y)}))) @{binding
rel-project} 10 @{thm rel-projectI}

#> Rel-Spec-Monad-Synthesize-Rules.add-rel-spec-monad-split-rules @{synthesize-rules-name
L2-rel-spec-monad} [
  (@{binding L2-seq - rel-xval}, 10, [f, g], @{thm rel-spec-monad-L2-seq-rel-xval-same-split}),
  (@{binding L2-while - rel-xval}, 10, [B, B'], @{thm rel-spec-monad-L2-while-rel-xval-same-split}),
  (@{binding L2-call - rel-project}, 10, [emb, emb'], @{thm rel-spec-monad-rel-project-L2-call-adapt-emb}),
  (@{binding L2-seq - rel-project (exit handler)}, 20, [g, g'], @{thm rel-spec-monad-L2-seq-exit-handler})
]
##>> Synthesize-Rules.gen-add-split-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false}
  (K (K (K @{pattern Trueprop (rel-spec-monad ?R (rel-xval ?L (=)) (L2-catch
?m ?f) ?g)})))
  @{binding L2-catch-try - rel-xval} 10 [f] @{thm rel-spec-monad-rel-xval-try-catch}
##>> Synthesize-Rules.gen-add-split-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false}
  (K (K (K @{pattern Trueprop (rel-fun ?A ?B ?f ?g)})))
  @{binding rel-funI} 10 [f, g] @{thm rel-funI}
##>> Rel-Spec-Monad-Synthesize-Rules.add-rel-spec-monad-infer-project-split-rules
@{synthesize-rules-name L2-rel-spec-monad} [
  (@{binding L2-seq - rel-project}, 10, [f], [g], [prj', m, L, prj], @{thm rel-spec-monad-rel-project-L2-seq})
]
##>> Rel-Spec-Monad-Synthesize-Rules.add-rel-spec-monad-project-split-rules @{synthesize-rules-name
L2-rel-spec-monad} [
  (@{binding L2-while - rel-project}, 10, [B, C, I], [B', C', I'], prj, @{thm rel-spec-monad-rel-project-L2-while'})
]
#> snd)
,

```

context *stack-heap-raw-state*

begin

declaration <

fn phi =>

Rel-Spec-Monad-Synthesize-Rules.add-rel-spec-monad-rules @{synthesize-rules-name
L2-rel-spec-monad} [

(@{binding with-fresh-stack-ptr - rel-sum}, 10, Morphism.thm phi @{thm
rel-spec-monad-rel-xval-with-fresh-stack-ptr}),

(@{binding with-fresh-stack-ptr - rel-project}, 10, Morphism.thm phi @{thm
rel-spec-monad-rel-project-with-fresh-stack-ptr}))]

>

end

declare [[*verbose=0*]]

lemma *map-sum-right*: $(\text{map-sum } l \ r \ v = \text{Inr } x) = (\exists v'. v = \text{Inr } v' \wedge x = r \ v')$
by *(cases v) auto*

lemma *map-sum-left*: $(\text{map-sum } l \ r \ v = \text{Inl } x) = (\exists v'. v = \text{Inl } v' \wedge x = l \ v')$
by *(cases v) auto*

lemma *case-Nonlocal-Inr*: $((\text{case } e \text{ of Nonlocal } x \Rightarrow \text{Inl } x \mid - \Rightarrow \text{Inr } v) = \text{Inr } y) =$
 $(\text{is-local } e \wedge (v = y))$
by *(cases e) auto*

lemma *case-Nonlocal-Inl*: $((\text{case } e \text{ of Nonlocal } x \Rightarrow \text{Inl } x \mid - \Rightarrow \text{Inr } v) = \text{Inl } y) =$
 $(e = \text{Nonlocal } y)$
by *(cases e) auto*

method-setup *resolve-split* = \langle
Scan.succeed $(\text{fn } \text{ctxt} \Rightarrow$
let
val *simp-ctxt* = $(\text{ctxt} \mid > \text{Simplifier.clear-simpset}) \text{ addsimps } @\{\text{thms}$
from-xval-simps
if-distrib $[\text{where } f = \text{from-xval}]\}$
in
SIMPLE-METHOD'
 $(\text{Rel-Spec-Monad-Synthesize-Rules.norm-resolve-split-thm } @\{\text{synthesize-rules-name}$
*L2-rel-spec-monad}\} \text{ ctxt}
THEN-ALL-NEW $(\text{simp-tac } \text{simp-ctxt})$
*end)
*resolve with rel-spec-monad rules of right arity***

method-setup *sanitize-names* = \langle
Scan.succeed $(\text{fn } \text{ctxt} \Rightarrow \text{SIMPLE-METHOD}' (\text{Rel-Spec-Monad-Synthesize-Rules.sanitize-names-tac}$
*ctxt)))
*sanitize name annotations for L2-throw**

method-setup *verbose-msg* =
 $\langle \text{Scan.lift } (\text{Parse.embedded } >>$
 $(\text{fn } \text{msg} \Rightarrow \text{fn } \text{ctxt} \Rightarrow \text{SIMPLE-METHOD} (\text{fn } \text{st} \Rightarrow (\text{Utils.verbose-msg } 1 \ \text{ctxt}$
 $(\text{fn } - \Rightarrow \text{msg}); \text{all-tac } \text{st})))) \rangle$
print a message (if flag autocorres-verbose is turned on)

method-setup *clarsimp-solve* = \langle
Scan.succeed $(\text{fn } \text{ctxt} \Rightarrow \text{SIMPLE-METHOD}' (\text{Rel-Spec-Monad-Synthesize-Rules.clarsimp-solve-tac}$
*ctxt)))
*clarsimp and then solve, instantiating variables**

method-setup *trace* =
 $\langle \text{Scan.lift } ((\text{Parse.embedded-input } -- \text{Parse.embedded}) >> (\text{fn } (\text{method-src}, \text{msg})$
 $\Rightarrow \text{fn } \text{ctxt} \Rightarrow$
let

```

    val (m, tok) = Method.read-closure-input ctxt method-src
    val msg-m = Method.Basic (fn ctxt => SIMPLE-METHOD (fn st =>
(Utills.verbose-msg 1 ctxt (fn - => msg); all-tac st)))
    val trace-meth = Method.Combinator (Method.no-combinator-info, Method.Then,
[m, msg-m])
    in
    Utills.timeap-method 3 ctxt (fn - => msg) (Method.evaluate trace-meth ctxt)
end))
trace a method application (if flag autocorres-verbose is turned on)

```

```

method rel-spec-monad-L2-step uses more-intro-thms =
((rule-tac more-intro-thms, verbose-msg applied: rule-tac) |
trace resolve-split applied: resolve-split |
trace assumption applied: assumption |
trace sanitize-names applied: sanitize-names |
trace clarsimp-solve applied: clarsimp-solve
)

```

```

method rel-spec-monad-L2-rewrite =
(use in ⟨rel-spec-monad-L2-step more-intro-thms: method-facts⟩)+

```

```

ML ⟨
structure L2-Exception-Rewrite =
struct

```

```

fun rhs-conv conv eq-thm =
Conv.fconv-rule (Conv.arg-conv conv) eq-thm

```

```

val rel-spec-monad-L2-rewrite-tac = UMM-Tools.tactic-from-src @⟨context⟩ ⟨rel-spec-monad-L2-rewrite⟩

```

```

fun abstract-try-catch ctxt t =
let
val goal = infer-instantiate ⟨t = t in prop (schematic) ⟨t = a⟩⟩ ctxt
val thm = Goal.prove ctxt [] [] goal (fn {prems, context,...} =>
simp-tac ((Simplifier.add-cong @⟨thm c-exntype.case-cong⟩ (put-simpset
HOL-basic-ss context))
addsimps @⟨thms cond-return1 cond-return2 if-c-exntype-cases rel-spec-monad-eq-conv
[symmetric]⟩) 1 THEN
resolve-tac context @⟨thms rel-spec-monad-rel-xval⟩ 1 THEN
rel-spec-monad-L2-rewrite-tac context [] THEN
print-unsolved-tac abstract-try-catch: unfinished goal ctxt
)
in
SOME thm
end
handle ERROR - => raise TERM (abstract-try-catch failed, [t])

```

```

fun abstract-try-catch-conv ctxt ct =
  case abstract-try-catch ctxt (Thm.term-of ct) of
    SOME eq => mk-meta-eq (Simplifier.simplify (Simplifier.clear-simpset ctxt
  addsimps @{\thms from-xval-simps}) eq)
  | NONE => (warning (abstract-try-catch-conv: failed to convert to nested excep-
tions. ^
  @{\make-string} ct); Conv.all-conv ct)

fun rel-spec-monad-conv trace-unfinished eq-intro ctxt lhs =
  let
    val ([lhs], ctxt') = Variable.import-terms false [lhs] ctxt
    val goal = infer-instantiate <lhs = lhs' in prop (schematic) <lhs = A>> ctxt'
    val thm = Goal.prove ctxt' [] [] goal (fn {context, ...} =>
      resolve-tac context [eq-intro] 1 THEN
      simp-tac (Simplifier.clear-simpset context addsimprocs [@{simproc ETA-TUPLED}]))
  1 THEN
    rel-spec-monad-L2-rewrite-tac context [] THEN
    trace-unfinished context)
  val [eq] = Variable.export ctxt' ctxt [mk-meta-eq thm]
  val eq = Thm.instantiate (Thm.match (Thm.lhs-of eq, Thm.cterm-of ctxt lhs))
eq
  in
    SOME eq
  end
  handle ERROR x => NONE

fun project-used-components-conv ctxt ct =
  let
    val trace = print-unsolved-tac project-used-components-conv: unfinished goal
  in
    case rel-spec-monad-conv trace @{\thm rel-project-eqI} ctxt (Thm.term-of ct) of
      SOME eq => eq |> rhs-conv (Simplifier.rewrite (put-simpset HOL-basic-ss
  ctxt addsimps
    (Utils.get-rules ctxt @{\named-theorems L2opt}))) — get rid of now unused
  result values, e.g. L2-unknown
    | NONE => (warning (project-used-components failed on: ^ @{\make-string}
  ct); Conv.all-conv ct)
  end

end
>

end

```

19.6 Peep-hole optimisations applied to L2

```

theory L2Peephole
imports L2Defs Tuple-Tools

```

begin

definition $STOP :: 'a::\{\} \Rightarrow 'a$
where $STOP P \equiv P$

lemma $STOP\text{-cong}$: $STOP P \equiv STOP P$
by $simp$

lemma $do\text{-}STOP$: $P \equiv STOP P$
by $(simp\ add:\ STOP\text{-def})$

lemmas $id\text{-def}$ [$L2opt$]

lemma $L2\text{-seq-skip}$ [$L2opt$]:
 $L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ ())\ n)\ X = (X\ ())$
 $L2\text{-seq}\ Y\ (\lambda\cdot.\ (L2\text{-gets}\ (\lambda\cdot.\ ())\ n)) = Y$
apply $(clarsimp\ simp:\ L2\text{-seq-def}\ L2\text{-gets-def}\ gets\text{-return})$
done

lemma $L2\text{-seq-L2-gets}$ [$L2opt$]: $L2\text{-seq}\ X\ (\lambda x.\ L2\text{-gets}\ (\lambda\cdot.\ x)\ n) = X$
apply $(clarsimp\ simp:\ L2\text{-defs}\ gets\text{-return})$
done

lemma $L2\text{-seq-L2-gets-unit}$ [$L2opt$]: $L2\text{-seq}\ (L2\text{-gets}\ g\ ns)\ (\lambda x::\ unit.\ f\ x) = f\ ()$
apply $(clarsimp\ simp:\ L2\text{-defs})$
by $(rule\ spec\ monad\ eqI)\ (auto\ simp\ add:\ runs\text{-to}\text{-iff})$

lemma $L2\text{-seq-L2-gets-const}$: $L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ X = X\ c$
apply $(clarsimp\ simp:\ L2\text{-defs}\ liftE\text{-def}\ gets\text{-return})$
done

lemma $const\text{-propagation-cong}$: $X\ c = X'\ c \implies (L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ X) =$
 $(L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ X')$
by $(clarsimp\ simp:\ L2\text{-seq-L2-gets-const})$

lemma $L2\text{-seq-const}'$:
assumes $bdy\text{-eq}:\ f\ c \equiv g\ c$
shows $L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ f \equiv L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ g$
apply $(rule\ eq\ reflection)$
unfolding $L2\text{-defs}$
apply $(rule\ spec\ monad\ eqI)$
apply $(auto\ simp\ add:\ bdy\text{-eq}\ runs\text{-to}\text{-iff})$
done

lemma $L2\text{-seq-const}$:
assumes $bdy\text{-eq}:\ f' \equiv g'$

assumes $f\text{-app}: f\ c \equiv f'$
assumes $g\text{-app}: g\ c \equiv g'$
shows $L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ f \equiv L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ g$
proof –
from $bdy\text{-eq}\ f\text{-app}\ g\text{-app}$ **have** $f\ c \equiv g\ c$
by $simp$
then show $PROP\ ?thesis$
by $(rule\ L2\text{-seq}\text{-const}')$
qed

lemma $L2\text{-seq}\text{-const}\text{-stop}$:
assumes $bdy\text{-eq}: f' \equiv g'$
assumes $f\text{-app}: f\ c \equiv f'$
assumes $g\text{-app}: g\ c \equiv g'$
shows $L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ f \equiv STOP\ (L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ g)$
unfolding $STOP\text{-def}$ **using** $bdy\text{-eq}\ f\text{-app}\ g\text{-app}$
by $(rule\ L2\text{-seq}\text{-const})$

lemma $L2\text{-seq}\text{-const}\text{-stop}'$:
assumes $bdy\text{-eq}: f\ c \equiv g'$
assumes $g\text{-app}: g\ c \equiv g'$
shows $L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ f \equiv STOP\ (L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ g)$
apply $(rule\ L2\text{-seq}\text{-const}\text{-stop})$
apply $(rule\ bdy\text{-eq})$
apply $simp$
apply $(rule\ g\text{-app})$
done

lemma $L2\text{-seq}\text{-const}\text{-stop}''$:
assumes $bdy\text{-eq}: f\ c \equiv g\ c$
shows $L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ f \equiv STOP\ (L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ g)$
apply $(rule\ L2\text{-seq}\text{-const}\text{-stop}'')$
apply $(rule\ bdy\text{-eq})$
apply $simp$
done

lemma $L2\text{-seq}\text{-const}\text{-stop}'''$:
assumes $bdy\text{-eq}: f\ c \equiv g$
shows $L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ f \equiv STOP\ (L2\text{-seq}\ (L2\text{-gets}\ (\lambda\cdot.\ c)\ n)\ (\lambda\cdot.\ g))$
apply $(rule\ L2\text{-seq}\text{-const}\text{-stop}''')$
apply $(rule\ bdy\text{-eq})$
apply $simp$
done

lemma $L2\text{-marked}\text{-seq}\text{-gets}\text{-cong}$:
 $c=c' \implies L2\text{-seq}\text{-gets}\ c\ n\ A \equiv L2\text{-seq}\text{-gets}\ c'\ n\ A$
by $simp$

lemma $L2\text{-marked}\text{-seq}\text{-gets}\text{-stop}$:

assumes *bdy-eq*: $f\ c \equiv g$
shows *L2-seq-gets* $c\ n\ f \equiv STOP\ (L2-seq-gets\ c\ n\ (\lambda\cdot\ g))$
unfolding *L2-seq-gets-def* **using** *bdy-eq*
by (*rule L2-seq-const-stop''*)

lemma *L2-guarded-block-cong*: $L2-guarded\ g\ c = L2-guarded\ g\ c$
by *simp*

lemma *L2-guarded-cong-stop'*:
assumes *guard-eq*: $\bigwedge s. g\ s \equiv g'\ s$
assumes *bdy-eq*: $\bigwedge s. g'\ s \implies run\ c\ s \equiv run\ c'\ s$
shows $L2-guarded\ g\ c \equiv STOP\ (L2-guarded\ g'\ c')$
unfolding *L2-guarded-def L2-defs STOP-def*
apply (*rule eq-reflection*)
apply (*rule spec-monad-ext*)
using *guard-eq bdy-eq*
apply (*auto simp add: run-guard run-bind*)
done

lemma *L2-seq-guard-cong-stop0*:
assumes *guard-eq*: $\bigwedge s. g\ s \equiv g'\ s$
assumes *bdy-eq*: $\bigwedge s. g'\ s \implies run\ c\ s \equiv run\ c'\ s$
shows $L2-seq-guard\ g\ (\lambda\cdot\ c) \equiv STOP\ (L2-seq-guard\ g'\ (\lambda\cdot\ c'))$
unfolding *L2-seq-guard-def*
using *assms*
by (*rule L2-guarded-cong-stop'[simplified L2-guarded-def]*)

lemma *L2-guarded-cong-stop*:
assumes *guard-eq*: $g \equiv g'$
assumes *bdy-eq*: $\bigwedge s. g'\ s \implies run\ c\ s \equiv run\ c'\ s$
shows $L2-guarded\ g\ c \equiv STOP\ (L2-guarded\ g'\ c')$
unfolding *L2-guarded-def L2-defs STOP-def*
apply (*rule eq-reflection*)
apply (*rule spec-monad-ext*)
using *guard-eq bdy-eq*
apply (*auto simp add: run-guard run-bind*)
done

lemma *L2-seq-assoc* :
 $L2-seq\ (L2-seq\ A\ (\lambda x. B\ x))\ C = L2-seq\ A\ (\lambda x. L2-seq\ (B\ x)\ C)$
apply (*clarsimp simp: L2-seq-def bind-assoc*)
done

lemma *L2-seq-assoc' [L2opt]*:
 $L2-seq\ (L2-seq\ A\ B)\ C = L2-seq\ A\ (\lambda x. L2-seq\ (B\ x)\ C)$
apply (*clarsimp simp: L2-seq-def bind-assoc*)

done

lemma *L2-trim-after-throw* [L2opt]:
L2-seq (L2-throw x n) Z = (L2-throw x n)
apply (clarsimp simp: L2-seq-def L2-throw-def)
done

lemma *L2-guard-true* [L2opt]: L2-guard ($\lambda\cdot$. True) = L2-gets ($\lambda\cdot$. ()) []
unfolding L2-defs
apply (rule spec-monad-ext)
apply (simp add: run-guard)
done

This rule can be too expensive in simplification as it might invoke the arithmetic solver

lemma *L2-guard-solve-true*: $\llbracket \bigwedge s. P s \rrbracket \implies L2-guard P = L2-gets (\lambda\cdot. ()) []$
unfolding L2-defs
apply (rule spec-monad-ext)
apply (simp add: run-guard)
done

lemma *L2-guard-false* [L2opt]: L2-guard ($\lambda\cdot$. False) = L2-fail
unfolding L2-defs
apply (rule spec-monad-ext)
apply (simp add: run-guard)
done

This rule can be too expensive in simplification as it might invoke the arithmetic solver

lemma *L2-guard-solve-false*: $\llbracket \bigwedge s. \neg P s \rrbracket \implies L2-guard P = L2-fail$
unfolding L2-defs
apply (rule spec-monad-ext)
apply (simp add: run-guard)
done

lemma *L2-spec-empty* [L2opt]:

L2-spec {} = L2-fail
 $\llbracket \bigwedge s t. \neg C s t \rrbracket \implies L2-spec \{(s, t). C s t\} = L2-fail$
unfolding L2-defs
apply (rule spec-monad-ext; simp add: run-bind run-assert-result-and-state)+
done

lemma *L2-unknown-bind-unbound*[L2opt]:
L2-seq (L2-unknown ns) ($\lambda x. f$) = f
unfolding L2-defs
apply (rule spec-monad-ext)
apply (simp add: run-bind)
done

lemma *L2-seq-L2-unknown-unit*[*L2opt*]: $L2\text{-seq } (L2\text{-unknown } ns) (\lambda x:: \text{unit}. f x) = f ()$
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

The following rule can cause some exponential blowup, especially when rewriting is not successful. Note that f is duplicated in the premise. The more restricted $L2\text{-seq } (L2\text{-unknown } ?ns) (\lambda x. ?f) = ?f$ should also do the job, in case of properly initialised variables. Maybe we should remove it entirely from "L2opt".

lemma *L2-unknown-bind* []:
 $(\bigwedge a b. f a = f b) \implies (L2\text{-seq } (L2\text{-unknown } ns) f) = f \text{ undefined}$
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add: runs-to-iff*)
apply (*clarsimp simp add: runs-to-def-old*)
apply *metis*
done

lemma *L2-seq-guard-cong*:
 $\llbracket P = P'; \bigwedge s. P' s \implies \text{run } X s = \text{run } X' s \rrbracket \implies L2\text{-seq } (L2\text{-guard } P) (\lambda-. X) = L2\text{-seq } (L2\text{-guard } P') (\lambda-. X')$
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind run-guard*)
done

lemma *L2-seq-guard-cong'*:
 $\llbracket P \equiv P'; \bigwedge s. P' s \implies \text{run } X s \equiv \text{run } X' s \rrbracket \implies L2\text{-seq } (L2\text{-guard } P) (\lambda-. X) \equiv L2\text{-seq } (L2\text{-guard } P') (\lambda-. X')$
apply (*rule eq-reflection*)
apply (*rule L2-seq-guard-cong*)
by *auto*

lemma *L2-seq-guard-cong-stop*:
 $\llbracket P = P'; \bigwedge s. P' s \implies \text{run } X s = \text{run } X' s \rrbracket \implies L2\text{-seq } (L2\text{-guard } P) (\lambda-. X) = \text{STOP } (L2\text{-seq } (L2\text{-guard } P') (\lambda-. X'))$
unfolding *STOP-def*
by (*rule L2-seq-guard-cong, auto*)

lemma *L2-seq-guard-cong-stop'*:
 $\llbracket P \equiv P'; \bigwedge s. P' s \implies \text{run } X s \equiv \text{run } X' s \rrbracket \implies L2\text{-seq } (L2\text{-guard } P) (\lambda-. X) \equiv \text{STOP } (L2\text{-seq } (L2\text{-guard } P') (\lambda-. X'))$

apply (*rule eq-reflection*)
apply (*rule L2-seq-guard-cong-stop*)
by *auto*

lemma *L2-seq-guard-cong-stop''*:

$\llbracket \bigwedge s. P s \implies \text{run } X s \equiv \text{run } X' s \rrbracket \implies L2\text{-seq } (L2\text{-guard } P) (\lambda\cdot. X) \equiv STOP$
 $(L2\text{-seq } (L2\text{-guard } P) (\lambda\cdot. X'))$
apply (*rule eq-reflection*)
apply (*rule L2-seq-guard-cong-stop*)
by *auto*

lemma *L2-seq-guard-cong-stop'''*:

$\llbracket \bigwedge s. P s \implies \text{run } (X ()) s \equiv \text{run } X' s \rrbracket \implies L2\text{-seq } (L2\text{-guard } P) X \equiv STOP$
 $(L2\text{-seq } (L2\text{-guard } P) (\lambda\cdot. X'))$
unfolding *STOP-def L2-defs*
apply (*rule eq-reflection*)
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind run-guard*)
done

lemma *L2-marked-seq-guard-cong*:

$\llbracket P = P'; \bigwedge s. P' s \implies \text{run } (X ()) s = \text{run } X' s \rrbracket \implies L2\text{-seq-guard } P X =$
 $L2\text{-seq-guard } P' (\lambda\cdot. X')$
unfolding *L2-seq-guard-def L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind run-guard*)
done

lemma *L2-gaurded-keep-guard-cong*:

$(\bigwedge s. g s \implies \text{run } c s = \text{run } c' s) \implies L2\text{-guarded } g c = L2\text{-guarded } g c'$
unfolding *L2-guarded-def L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind run-guard*)
done

lemma *gets-guard-move-before [L2opt]*:

$L2\text{-seq } (L2\text{-gets } f ns) (\lambda r. L2\text{-seq } (L2\text{-guard } P) (\lambda\cdot. X r)) =$
 $L2\text{-seq } (L2\text{-guard } P) (\lambda\cdot. L2\text{-seq } (L2\text{-gets } f ns) X)$
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *L2-seq-guard-cong''*:

$\llbracket \bigwedge s. P s = P' s; \bigwedge s. P' s \implies \text{run } X s = \text{run } X' s \rrbracket \implies L2\text{-seq } (L2\text{-guard } P)$
 $(\lambda\cdot. X) = L2\text{-seq } (L2\text{-guard } P') (\lambda\cdot. X')$
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind run-guard*)

done

lemma *guard-triv* [L2opt]: $L2\text{-seq } (L2\text{-guard } (\lambda s. \text{True})) (\lambda \cdot. X) = X$
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind run-guard*)
done

lemma *L2-condition-cong*:
 $\llbracket C = C'; \bigwedge s. C' s \implies \text{run } A s = \text{run } A' s; \bigwedge s. \neg C' s \implies \text{run } B s = \text{run } B' s \rrbracket \implies L2\text{-condition } C A B = L2\text{-condition } C' A' B'$
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add: run-condition*)
done

lemma *L2-condition-cong-stop*:
 $\llbracket C = C'; \bigwedge s. C' s \implies \text{run } A s = \text{run } A' s; \bigwedge s. \neg C' s \implies \text{run } B s = \text{run } B' s \rrbracket \implies L2\text{-condition } C A B = \text{STOP } (L2\text{-condition } C' A' B')$
unfolding *STOP-def*
by (*rule L2-condition-cong*)

lemma *L2-condition-cong'*:
 $\llbracket \bigwedge s. C s = C' s; \bigwedge s. C' s \implies \text{run } A s = \text{run } A' s; \bigwedge s. \neg C' s \implies \text{run } B s = \text{run } B' s \rrbracket \implies L2\text{-condition } C A B = L2\text{-condition } C' A' B'$
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add: run-condition*)
done

lemma *L2-condition-true* [L2opt]: $\llbracket \bigwedge s. C s \rrbracket \implies L2\text{-condition } C A B = A$
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add: run-condition*)
done

lemma *L2-condition-false* [L2opt]: $\llbracket \bigwedge s. \neg C s \rrbracket \implies L2\text{-condition } C A B = B$
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add: run-condition*)
done

lemma *L2-condition-true'* [simp]: $L2\text{-condition } (\lambda s. \text{True}) A B = A$
unfolding *L2-defs*
by *simp*

lemma *L2-condition-false'* [simp]: $L2\text{-condition } (\lambda s. \text{False}) A B = B$
unfolding *L2-defs*
by *simp*

lemma *L2-condition-same* [L2opt]: *L2-condition C A A = A*
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add: run-condition*)
done

lemma *L2-fail-seq* [L2opt]: *L2-seq L2-fail X = L2-fail*
unfolding *L2-defs*
by *simp*

lemma *bind-fail-propagates*: $\llbracket \text{no-throw } (\lambda\cdot. \text{True}) A; \text{always-progress } A \rrbracket \implies A$
 $\gg = (\lambda\cdot. \text{fail}) = \text{fail}$
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add: runs-to-iff*)
apply (*clarsimp simp add: no-throw-def runs-to-def-old runs-to-partial-def-old*)
by (*meson always-progressD*)

lemma *state-select-select-fail*:
 $\text{state-select } S \gg = (\lambda\cdot. \text{select UNIV}) \gg = (\lambda\cdot. \text{fail}) = \text{fail}$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *L2-fail-propagates* [L2opt]:
 $L2\text{-seq } (L2\text{-gets } V n) (\lambda\cdot. L2\text{-fail}) = L2\text{-fail}$
 $L2\text{-seq } (L2\text{-modify } M) (\lambda\cdot. L2\text{-fail}) = L2\text{-fail}$
 $L2\text{-seq } (L2\text{-spec } S) (\lambda\cdot. L2\text{-fail}) = L2\text{-fail}$
 $L2\text{-seq } (L2\text{-guard } G) (\lambda\cdot. L2\text{-fail}) = L2\text{-fail}$
 $L2\text{-seq } (L2\text{-unknown } ns) (\lambda\cdot. L2\text{-fail}) = L2\text{-fail}$
 $L2\text{-seq } L2\text{-fail } (\lambda\cdot. L2\text{-fail}) = L2\text{-fail}$
unfolding *L2-defs*
apply (*simp-all add: bind-fail-propagates always-progress-intros*
state-select-select-fail)
done

lemma *L2-condition-distrib*:
 $L2\text{-seq } (L2\text{-condition } C L R) X = L2\text{-condition } C (L2\text{-seq } L X) (L2\text{-seq } R X)$
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemmas *L2-fail-propagate-condition* [L2opt] = *L2-condition-distrib* [**where** $X = \lambda\cdot. L2\text{-fail}$]

lemma *L2-seq-condition-skip-throw* [L2opt]: *L2-seq*
(L2-condition c (L2-gets (λ-. ())) ns)
(L2-throw x ms)
(λr. y) =
(L2-condition c y
(L2-throw x ms))
by (*simp add: L2-condition-distrib L2-seq-skip L2-trim-after-throw*)

lemma *L2-fail-propagate-catch* [L2opt]:
(L2-seq (L2-catch L R) (λ-. L2-fail)) = (L2-catch (L2-seq L (λ-. L2-fail)) (λe.
L2-seq (R e) (λ-. L2-fail)))
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add: runs-to-iff default-option-def Exn-def, safe*)
apply (*auto simp add: runs-to-def-old*)
done

lemma *L2-condition-fail-lhs* [L2opt]:
L2-condition C L2-fail A = L2-seq (L2-guard (λs. ¬ C s)) (λ-. A)
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff default-option-def Exn-def*)
done

lemma *L2-condition-fail-rhs* [L2opt]:
L2-condition C A L2-fail = L2-seq (L2-guard (λs. C s)) (λ-. A)
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff default-option-def Exn-def*)
done

lemma *L2-catch-fail* [L2opt]: *L2-catch L2-fail A = L2-fail*
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff default-option-def Exn-def*)
done

lemma *L2-try-fail* [L2opt]: *L2-try L2-fail = L2-fail*
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff default-option-def Exn-def*)
done

lemma *L2-while-fail* [L2opt]: *L2-while C (λ-. L2-fail) i n = (L2-seq (L2-guard*
(λs. ¬ C i s)) (λ-. L2-gets (λs. i) n))
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*subst whileLoop-unroll*)
apply (*auto simp add: run-condition run-bind run-guard*)

done

lemma *unit-bind*: $(\lambda x. f (x::unit)) = (\lambda-. f ())$
apply (*rule ext*)
subgoal for *x* **by** (*cases x, auto*)
done

lemma *unit-bind'*: $f \equiv (\lambda-. f ())$
by (*simp add: unit-bind*)

lemma *L2-while-cong*:
assumes *c-eq*: $\bigwedge r s. c r s = c' r s$
assumes *bdy-eq*: $\bigwedge r s. c' r s \implies \text{run } (A r) s = \text{run } (A' r) s$
shows $L2\text{-while } c A = L2\text{-while } c' A'$
apply (*rule ext*)
unfolding *L2-while-def*
using *whileLoop-cong c-eq bdy-eq*
apply *metis*
done

lemma *L2-while-simp-cong*:
assumes *c-eq*: $\bigwedge r s. c r s = c' r s$
assumes *bdy-eq[simplified simp-implies-def]*: $\bigwedge r s. c' r s = \text{simp} \implies \text{run } (A r) s = \text{run } (A' r) s$
shows $L2\text{-while } c A = L2\text{-while } c' A'$
apply (*rule L2-while-cong*)
apply (*simp add: c-eq*)
apply (*simp add: bdy-eq*)
done

lemma *L2-while-cong'*:
assumes *c-eq*: $c = c'$
assumes *bdy-eq*: $\bigwedge r s. c' r s \implies \text{run } (A r) s = \text{run } (A' r) s$
shows $L2\text{-while } c A = L2\text{-while } c' A'$
apply (*rule L2-while-cong*)
apply (*simp add: c-eq*)
apply (*simp add: bdy-eq*)
done

lemma *L2-while-simp-cong'*:
assumes *c-eq*: $c = c'$
assumes *bdy-eq[simplified simp-implies-def]*: $\bigwedge r s. c' r s = \text{simp} \implies \text{run } (A r) s = \text{run } (A' r) s$
shows $L2\text{-while } c A = L2\text{-while } c' A'$
apply (*rule L2-while-cong*)
apply (*simp add: c-eq*)

apply (*simp add: bdy-eq*)
done

lemma *L2-while-cong-split*:

assumes *c-eq*: $c = c'$
assumes *bdy-eq*: *PROP SPLIT* ($\bigwedge r s. c' r s \implies \text{run } (A r) s = \text{run } (A' r) s$)
shows $L2\text{-while } c A = L2\text{-while } c' A'$
apply (*rule L2-while-simp-cong' [OF c-eq]*)
using *bdy-eq*
by (*simp add: SPLIT-def simp-implies-def*)

lemma *L2-while-cong-simp-split*:

assumes *c-eq*: $c = c'$
assumes *bdy-eq*: *PROP SPLIT* ($\bigwedge r s. c' r s = \text{simp} \implies \text{run } (A r) s = \text{run } (A' r) s$)
shows $L2\text{-while } c A = L2\text{-while } c' A'$
apply (*rule L2-while-simp-cong' [OF c-eq]*)
using *bdy-eq*
by (*simp add: SPLIT-def simp-implies-def*)

lemma *L2-while-cong-split-stop*:

assumes *c-eq*: $c = c'$
assumes *bdy-eq*: *PROP SPLIT* ($\bigwedge r s. c' r s \implies \text{run } (A r) s = \text{run } (A' r) s$)
shows $L2\text{-while } c A = \text{STOP } (L2\text{-while } c' A')$
apply (*simp add: STOP-def*)
apply (*rule L2-while-simp-cong' [OF c-eq]*)
using *bdy-eq*
by (*simp add: SPLIT-def simp-implies-def*)

lemma *L2-while-cong-simp-split-stop*:

assumes *c-eq*: $c = c'$
assumes *bdy-eq*: *PROP SPLIT* ($\bigwedge r s. c' r s = \text{simp} \implies \text{run } (A r) s = \text{run } (A' r) s$)
shows $L2\text{-while } c A = \text{STOP } (L2\text{-while } c' A')$
apply (*simp add: STOP-def*)
apply (*rule L2-while-simp-cong' [OF c-eq]*)
using *bdy-eq*
by (*simp add: SPLIT-def simp-implies-def*)

lemma *L2-while-cong''*:

assumes *c-eq*: $c = c'$
assumes *bdy-eq*: $\bigwedge r s. c' r s \implies \text{run } (A r) s = \text{run } (A' r) s$
assumes *i-eq*: $i = i'$
shows $L2\text{-while } c A i ns = L2\text{-while } c' A' i' ns$
apply (*simp add: i-eq*)
using *c-eq bdy-eq L2-while-cong*
by *metis*

lemma *L2-returncall-trivial* [*L2opt*]:
 $\llbracket \bigwedge s v. f v s = v \rrbracket \implies L2\text{-returncall } x f = L2\text{-call } x$
unfolding *L2-defs L2-call-def L2-returncall-def*
apply (*rule ext*)
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add: runs-to-iff*)
by (*auto simp add: runs-to-def-old map-exn-def Exn-def default-option-def split:xval-splits*)
)

lemma *L2-gets-unused*:
 $\llbracket \bigwedge x y s. \text{run } (B x) s = \text{run } (B y) s \rrbracket \implies L2\text{-seq } (L2\text{-gets } A n) B = (B \text{ undefined})$
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind*)
done

lemma *L2-gets-unbound*[*L2opt*]:
 $L2\text{-seq } (L2\text{-gets } A n) (\lambda x. f) = f$
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind*)
done

lemma *L2-gets-bind*:
 $L2\text{-seq } (L2\text{-gets } (\lambda-. x :: 'var\text{-type}) n) f = f x$
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind*)
done

lemma *L2-gets-bind-stop-cong*:
 $L2\text{-seq } (L2\text{-gets } (\lambda-. x) n) f = L2\text{-seq } (L2\text{-gets } (\lambda-. x) n) f$
by *simp*

lemma *L2-seq-stop-cong*:
 $L2\text{-seq } x y = L2\text{-seq } x y$
by *simp*

lemma *L2-marked-seq-gets-apply*:
 $L2\text{-seq-gets } c n A \equiv A c$
unfolding *L2-seq-gets-def*
apply (*rule eq-reflection*)
by (*rule L2-gets-bind*)

lemma *split-seq-assoc* [L2opt]:
 $(\lambda x. L2\text{-seq} (\text{case } x \text{ of } (a, b) \Rightarrow B a b) (G x)) = (\lambda x. \text{case } x \text{ of } (a, b) \Rightarrow (L2\text{-seq} (B a b) (G x)))$
by (*rule ext*) *clarsimp*

lemma *whileLoop-succeeds-terminates-infinite'*:
assumes *run* (*whileLoop* ($\lambda\cdot. C$) ($\lambda x. \text{gets} (\lambda s. B s x)$) *i*) $s \neq \top$
shows $C s \Longrightarrow \text{False}$
using *assms*
by (*induct rule: whileLoop-ne-top-induct*) (*auto simp: runs-to-iff*)

lemma *run-whileLoop-infinite'*: *run* (*whileLoop* ($\lambda i. C$)
 $(\lambda x. \text{gets} (\lambda s. B s x))$
i)
 $s =$
 $\text{run} (\text{guard} (\lambda s. \neg C s) \gg (\lambda\cdot. \text{gets} (\lambda\cdot. i))) s$

proof (*cases C s*)
case *True*
show *?thesis*
apply (*rule antisym*)
subgoal
apply (*subst whileLoop-unroll*)
apply (*simp add: run-guard True run-bind*)
done
subgoal
proof –
have $\neg \text{run} (\text{whileLoop} (\lambda\cdot. C) (\lambda x. \text{gets} (\lambda s. B s x)) i) s \neq \text{top}$
using *True whileLoop-succeeds-terminates-infinite'[of C B i s]* **by** *auto*
hence $\neg \text{succeeds} (\text{whileLoop} (\lambda\cdot. C) (\lambda x. \text{gets} (\lambda s. B s x)) i) s$
by (*simp add: succeeds-def*)
then show *?thesis*
by (*simp add: run-guard run-bind succeeds-def True top-post-state-def*)
qed
done

next
case *False*
then show *?thesis* **apply** (*subst whileLoop-unroll*) **by** (*simp add: run-bind run-guard*)
qed

lemma *whileLoop-infinite'*:
 $\text{whileLoop} (\lambda i. C)$
 $(\lambda x. \text{gets} (\lambda s. B s x))$
i
 $=$
 $\text{guard} (\lambda s. \neg C s) \gg (\lambda\cdot. \text{gets} (\lambda\cdot. i))$
apply (*rule spec-monad-ext*)
apply (*rule run-whileLoop-infinite'*)

done

lemma *L2-while-infinite* [L2opt]:

$L2\text{-while } (\lambda i s. C s) (\lambda x. L2\text{-gets } (\lambda s. B s x) n') i n$
 $= (L2\text{-seq } (L2\text{-guard } (\lambda s. \neg C s)) (\lambda-. L2\text{-gets } (\lambda-. i) n))$

unfolding *L2-defs*

by (*rule whileLoop-infinite'*)

lemmas *L2-gets-bind-c-exntype* [L2opt] = *L2-gets-bind* [**where** 'var-type='gx c-exntype]

lemmas *L2-gets-bind-unit* [L2opt] = *L2-gets-bind* [**where** 'var-type=unit]

declare *L2-voidcall-def* [L2opt]

declare *L2-modifycall-def* [L2opt]

declare *ucast-id* [L2opt]

declare *scast-id* [L2opt]

declare *singleton-iff* [L2opt]

lemma *L2-gets-L2-seq-if-P-1-0* [L2opt]:

$L2\text{-seq } (L2\text{-gets } (\lambda s. \text{if } P s \text{ then } 1 \text{ else } 0) n) (\lambda x. Q x)$
 $= (L2\text{-seq } (L2\text{-gets } P n) (\lambda x. Q (\text{if } x \text{ then } 1 \text{ else } 0)))$

unfolding *L2-defs*

apply (*rule spec-monad-eqI*)

apply (*auto simp add: runs-to-iff*)

done

lemma *L2-guard-join-simple* [L2opt]:

$L2\text{-seq } (L2\text{-guard } A) (\lambda-. L2\text{-guard } B) = L2\text{-guard } (\lambda s. A s \wedge B s)$

unfolding *L2-defs*

apply (*rule spec-monad-eqI*)

apply (*auto simp add: runs-to-iff*)

done

lemma *L2-guard-join-nested* [L2opt]:

$L2\text{-seq } (L2\text{-guard } A) (\lambda-. L2\text{-seq } (L2\text{-guard } B) (\lambda-. C))$
 $= L2\text{-seq } (L2\text{-guard } (\lambda s. A s \wedge B s)) (\lambda-. C)$

unfolding *L2-defs*

apply (*rule spec-monad-eqI*)

apply (*auto simp add: runs-to-iff*)

done

end

Chapter 20

IO phase: In/Out Parameters

20.1 Heap Typing for Split Heap

```
theory TypHeapSimple
  imports
    AutoCorres-Base
begin
```

20.2 Valid root footprint

lemma *c-null-guard-size-of-limit*:

c-null-guard ($p::'a::c\text{-type } ptr$) \implies *size-td* (*typ-uinfo-t* (*TYPE*('a))) $<$ $2 \wedge$ *len-of* *TYPE*(*addr-bitsize*)

unfolding *c-null-guard-def* *size-of-def*

by (*metis* (*no-types*, *opaque-lifting*) *add commute add-diff-cancel-left' add-lessD1*)

cancel-comm-monoid-add-class.diff-cancel first-in-intvl nat-less-le nat-neq-iff not-add-less1 typ-uinfo-size unat-lt2p unsigned-eq-0-iff zero-not-in-intvl-no-overflow)

lemma *root-ptr-valid-legacy-def*: *root-ptr-valid* d $p \longleftrightarrow$

valid-root-footprint d (*ptr-val* ($p::'a$ *ptr*)) (*typ-uinfo-t* *TYPE*('a)) \wedge

valid-simple-footprint d (*ptr-val* ($p::'a::c\text{-type } ptr$)) (*typ-uinfo-t* *TYPE*('a)) \wedge

d , *c-guard* $\models_t p$

using *c-null-guard-size-of-limit* [*of p*]

by (*auto simp add: root-ptr-valid-def c-guard-def addr-card-len-of-conv*

h-t-valid-def valid-root-footprint-valid-footprint intro!: valid-root-footprint-valid-simple-footprint)

definition

simple-lift $::$ *heap-raw-state* \implies ($'a::c\text{-type}$) *ptr* \implies $'a$ *option*

where

simple-lift s $p =$ (
 if (*root-ptr-valid* (*hrs-htd* s) p) *then*

```

      (Some (h-val (hrs-mem s) p))
    else
      None)

```

lemma *simple-lift-root-ptr-valid:*

```

simple-lift s p = Some x  $\implies$  root-ptr-valid (hrs-htd s) p
apply (clarsimp simp: simple-lift-def split: if-split-asm)
done

```

lemma *simple-lift-h-t-valid:*

```

simple-lift s p = Some x  $\implies$  (hrs-htd s), c-guard  $\models_t$  p
apply (clarsimp simp: simple-lift-def root-ptr-valid-legacy-def split: if-split-asm)
done

```

lemma *simple-lift-valid-root-footprint:*

```

simple-lift s (p::'a::c-type ptr) = Some x  $\implies$  valid-root-footprint (hrs-htd s)
(ptr-val p) (typ-uinfo-t (TYPE('a)))
apply (clarsimp simp: simple-lift-def root-ptr-valid-def split: if-split-asm)
done

```

lemma *simple-lift-c-guard:*

```

assumes lift: simple-lift s p = Some x
shows c-guard p
using simple-lift-h-t-valid [OF lift]
by (simp add: h-t-valid-def)

```

lemma *root-ptr-valid-heap-update-other:*

```

assumes val-p: root-ptr-valid d (p::'a::mem-type ptr)
and val-q: root-ptr-valid d (q::'b::c-type ptr)
and neq: ptr-val p  $\neq$  ptr-val q
shows h-val (heap-update p v h) q = h-val h q
apply (clarsimp simp: h-val-def heap-update-def)
apply (subst heap-list-update-disjoint-same)
apply simp
apply (rule root-ptr-valid-neq-disjoint [OF val-p val-q neq])
apply simp
done

```

lemma *root-ptr-valid-heap-update-other-tyt:*

```

assumes val-p: root-ptr-valid d (p::'a::mem-type ptr)
and val-q: root-ptr-valid d (q::'b::c-type ptr)
and neq: typ-uinfo-t TYPE('a)  $\neq$  typ-uinfo-t TYPE('b)
shows h-val (heap-update p v h) q = h-val h q
apply (clarsimp simp: h-val-def heap-update-def)

```

```

apply (subst heap-list-update-disjoint-same)
apply simp
apply (rule root-ptr-valid-type-neq-disjoint [OF val-p val-q neq])
apply simp
done

```

```

lemma simple-lift-heap-update:
  [[ root-ptr-valid (hrs-htd h) p ]]  $\implies$ 
    simple-lift (hrs-mem-update (heap-update p v) h)
      = (simple-lift h)(p := Some (v::'a::mem-type))
apply (rule ext)
apply (clarsimp simp: simple-lift-def hrs-mem-update h-val-heap-update)
apply (fastforce simp: root-ptr-valid-heap-update-other)
done

```

```

lemma simple-lift-heap-update-other:
  [[ root-ptr-valid (hrs-htd d) (p::'b::mem-type ptr);
    typ-uinfo-t TYPE('a)  $\neq$  typ-uinfo-t TYPE('b) ]]  $\implies$ 
    simple-lift (hrs-mem-update (heap-update p v) d) = ((simple-lift d)::'a::c-type
  typ-heap)
apply (rule ext)+
apply (clarsimp simp: simple-lift-def h-val-heap-update hrs-mem-update)
apply (auto intro: root-ptr-valid-heap-update-other-tyt)
done

```

```

lemma h-val-simple-lift:
  simple-lift h p = Some v  $\implies$  h-val (hrs-mem h) p = v
apply (clarsimp simp: simple-lift-def split: if-split-asm)
done

```

```

lemma the-simple-lift-h-val-conv:
  root-ptr-valid (hrs-htd h) p  $\implies$  the (simple-lift h) p = h-val (hrs-mem h) p
apply (clarsimp simp: simple-lift-def split: if-split-asm)
done

```

```

lemma slift-clift-Some-same:
assumes slift: simple-lift s p = Some x
assumes clift: clift s p = Some y
shows x = y
  using h-val-simple-lift [OF slift] h-val-clift' [OF clift]
  by simp

```

```

lemma simple-lift-Some-clift-Some:
assumes slift: simple-lift s p = Some x
shows clift s p = Some x

```

using *simple-lift-h-t-valid* [*OF slift*] *h-t-valid-clift-Some-iff* *slift-clift-Some-same*
 [*OF slift*]
by (*cases s*) *auto*

lemma *h-val-field-simple-lift*:

\llbracket *simple-lift* *h* (*pa* :: 'a *ptr*) = *Some* (*v*::'a::*mem-type*);
field-ti *TYPE*('a) *f* = *Some* *t*;
export-uinfo (*the* (*field-ti* *TYPE*('a) *f*)) = *export-uinfo* (*typ-info-t* *TYPE*('b ::
mem-type)) $\rrbracket \implies$
h-val (*hrs-mem* *h*) (*Ptr* &(*pa*→*f*) :: 'b :: *mem-type* *ptr*) = *from-bytes* (*access-ti*₀
(*the* (*field-ti* *TYPE*('a) *f*)) *v*)
apply (*clarsimp simp: simple-lift-def split: if-split-asm*)
apply (*clarsimp simp: h-val-field-from-bytes*)
done

lemma *simple-lift-heap-update'*:

simple-lift *h* *p* = *Some* *v'* \implies
simple-lift (*hrs-mem-update* (*heap-update* (*p*::('a::{*mem-type*}) *ptr*) *v*) *h*)
= (*simple-lift* *h*)(*p* := *Some* *v*)
apply (*rule simple-lift-heap-update*)
apply (*erule simple-lift-root-ptr-valid*)
done

lemma *simple-lift-hrs-mem-update-None* [*simp*]:

(*simple-lift* (*hrs-mem-update* *a* *hp*) *x* = *None*) = (*simple-lift* *hp* *x* = *None*)
apply (*clarsimp simp: simple-lift-def*)
done

lemma *simple-lift-hrs-mem-update-Some*: ($\exists z$. *simple-lift* (*hrs-mem-update* *upd* *h*)
x = *Some* *z*)

\longleftrightarrow ($\exists z$. *simple-lift* *h* *x* = *Some* *z*)

apply (*cases simple-lift h x*)
apply *simp*
apply (*cases simple-lift (hrs-mem-update upd h) x*)
apply (*auto*)
done

lemma *clift-hrs-mem-update-None* [*simp*]:

(*clift* (*hrs-mem-update* *a* *hp*) *x* = *None*) = (*clift* *hp* *x* = *None*)
using *h-t-valid-clift-Some-iff*
apply (*cases hp*)
apply (*clarsimp simp add: hrs-mem-update*)
apply (*metis hrs-htd-def hrs-htd-mem-update lift-t-if option.distinct(1) prod.collapse*)
done

lemma *clift-hrs-mem-update-Some*:

($\exists z$. *clift* (*hrs-mem-update* *a* *hp*) *x* = *Some* *z*) = ($\exists z$. *clift* *hp* *x* = *Some* *z*)


```

apply (cases clift hp x)
apply simp
apply (cases clift (hrs-mem-update a hp) x)
apply (auto)
done

```

lemma *simple-lift-data-eq*:

```

[[ h-val (hrs-mem h) p = h-val (hrs-mem h') p';
  root-ptr-valid (hrs-htd h) p = root-ptr-valid (hrs-htd h') p' ]] ==>
  simple-lift h p = simple-lift h' p'
apply (clarsimp simp: simple-lift-def)
done

```

lemma *h-val-heap-update-disjoint*:

```

[[ {ptr-val p ..+ size-of TYPE('a::c-type)}
  ∩ {ptr-val q ..+ size-of TYPE('b::mem-type)} = {} ]] ==>
  h-val (heap-update (q :: 'b ptr) r h) (p :: 'a ptr) = h-val h p
apply (clarsimp simp: h-val-def)
apply (clarsimp simp: heap-update-def)
apply (subst heap-list-update-disjoint-same)
apply clarsimp
apply blast
apply clarsimp
done

```

lemma *update-ti-t-valid-size*:

```

size-of TYPE('b) = size-td t ==>
  update-ti-t t (to-bytes-p (val::'b::mem-type)) obj = update-ti t (to-bytes-p val) obj
apply (clarsimp simp: update-ti-t-def to-bytes-p-def)
done

```

lemma *h-val-field-from-bytes'*:

```

[[ field-ti TYPE('a::{mem-type}) f = Some t;
  export-uinfo t = export-uinfo (typ-info-t TYPE('b::{mem-type})) ]] ==>
  h-val h (Ptr &(pa→f) :: 'b ptr) = from-bytes (access-ti0 t (h-val h pa))
apply (insert h-val-field-from-bytes[where f=f and pa=pa and t=t and h=(h,x)
  and 'a='a and 'b='b for x])
apply (clarsimp simp: hrs-mem-def)
done

```

lemma *h-val-field-from-root*:

```

fixes p::'a:: mem-type ptr
assumes fl:
  field-lookup (typ-info-t TYPE('a::{mem-type})) f 0 =
    Some (adjust-ti (typ-info-t TYPE('b::mem-type)) fld fld-update, n)
assumes fg-cons: fg-cons fld (fld-update)
shows h-val h (PTR('b) &(p→f)) = fld (h-val h p)

```

```

proof –
  from fl
  have fl: field-ti TYPE('a) f = Some (adjust-ti (typ-info-t TYPE('b::mem-type))
fld fld-update)
    using field-lookup-field-ti by blast
  have exp: export-uinfo (adjust-ti (typ-info-t TYPE('b)) fld fld-update) =
    export-uinfo (typ-info-t TYPE('b))
    using fg-cons
    by simp
  from h-val-field-from-bytes' [OF fl exp]
  show ?thesis
    by simp
qed

```

```

lemma simple-lift-super-field-update-lookup:
  fixes dummy :: 'b :: mem-type
  assumes field-lookup (typ-info-t TYPE('b::mem-type)) f 0 = Some (s,n)
    and typ-uinfo-t TYPE('a) = export-uinfo s
    and simple-lift h p = Some v'
  shows (super-field-update-t (Ptr (&(p→f))) (v::'a::mem-type) ((simple-lift h)::'b
ptr ⇒ 'b option)) =
    ((simple-lift h)(p ↦ field-update (field-desc s) (to-bytes-p v) v'))

```

```

proof –
  from assms have [simp]: unat (of-nat n :: addr) = n
    apply (subst unat-of-nat)
    apply (subst mod-less)
    apply (drule td-set-field-lookupD)+
    apply (drule td-set-offset-size)+
    apply (subst len-of-addr-card)
    apply (subst (asm) size-of-def [symmetric, where t=TYPE('b)])+
    apply (subgoal-tac size-of TYPE('b) < addr-card)
    apply arith
    apply simp
    apply simp
  done
  from assms show ?thesis
    supply unsigned-of-nat [simp del]
    apply (clarsimp simp: super-field-update-t-def)
    apply (rule ext)
    apply (clarsimp simp: field-lvalue-def split: option.splits)
    apply (safe, simp-all)
    apply (frule-tac v=v and v'=v' in update-field-update)
    apply (clarsimp simp: field-of-t-def field-of-def typ-uinfo-t-def)
    apply (frule-tac m=0 in field-names-SomeD2)
    apply simp
    apply clarsimp
    apply (simp add: field-typ-def field-typ-untyped-def)
    apply (frule field-lookup-export-uinfo-Some)
    apply (frule-tac s=k in field-lookup-export-uinfo-Some)

```

```

apply simp
apply (drule (1) field-lookup-inject)
  apply (subst typ-ufinfo-t-def [symmetric, where t=TYPE('b)])
  apply simp
apply simp
apply (drule field-of-t-mem)+
apply (cases h)
apply (clarsimp simp: simple-lift-def split: if-split-asm)
apply (drule (1) root-ptr-valid-neq-disjoint)
  apply simp
apply fast
apply (clarsimp simp: field-of-t-def field-of-def)
apply (subst (asm) td-set-field-lookup)
  apply simp
apply simp
apply (frule field-lookup-export-ufinfo-Some)
apply (simp add: typ-ufinfo-t-def)
apply (clarsimp simp: field-of-t-def field-of-def)
apply (subst (asm) td-set-field-lookup)
  apply simp
apply simp
apply (frule field-lookup-export-ufinfo-Some)
apply (simp add: typ-ufinfo-t-def)
done

```

qed

lemma *field-offset-addr-card*:

```

 $\exists x. \text{field-lookup } (\text{typ-info-t } \text{TYPE}('a::\text{mem-type})) f 0 = \text{Some } x$ 
 $\implies \text{field-offset } \text{TYPE}('a) f < \text{addr-card}$ 
apply (clarsimp simp: field-offset-def field-offset-untyped-def typ-ufinfo-t-def)
apply (subst field-lookup-export-ufinfo-Some)
  apply assumption
apply (frule td-set-field-lookupD)
apply (drule td-set-offset-size)
apply (insert max-size [where ?'a='a])
apply (clarsimp simp: size-of-def)
done

```

lemma *unat-of-nat-field-offset*:

```

 $\exists x. \text{field-lookup } (\text{typ-info-t } \text{TYPE}('a::\text{mem-type})) f 0 = \text{Some } x \implies$ 
 $\text{unat } (\text{of-nat } (\text{field-offset } \text{TYPE}('a) f)) \text{ :: addr} = \text{field-offset } \text{TYPE}('a) f$ 
apply (subst word-unat.Abs-inverse)
apply (clarsimp simp: unats-def)
apply (insert field-offset-addr-card [where f=f and ?'a='a][1])
apply (fastforce simp: addr-card)
apply simp
done

```

lemma *field-of-t-field-lookup*:

```

assumes a: field-lookup (typ-info-t TYPE('a::mem-type)) f 0 = Some (s, n)
assumes b: export-uinfo s = typ-uinfo-t TYPE('b::mem-type)
assumes n: n = field-offset TYPE('a) f
shows field-of-t (Ptr &(ptr→f) :: ('b ptr)) (ptr :: 'a ptr)
supply unsigned-of-nat [simp del]
apply (clarsimp simp del: field-lookup-offset-eq
        simp: field-of-t-def field-of-def)
apply (subst td-set-field-lookup)
apply (rule wf-desc-ty-tag)
apply (rule exI [where x=f])
using a[simplified n] b
apply (clarsimp simp: typ-uinfo-t-def)
apply (subst field-lookup-export-uinfo-Some)
apply assumption
apply (clarsimp simp del: field-lookup-offset-eq
        simp: field-lvalue-def unat-of-nat-field-offset)
done

lemma simple-lift-field-update':
fixes val :: 'b :: mem-type and ptr :: 'a :: mem-type ptr
assumes fl: field-lookup (typ-info-t TYPE('a)) f 0 =
        Some (adjust-ti (typ-info-t TYPE('b)) xf xfu, n)
and xf-xfu: fg-cons xf xfu
and cl: simple-lift hp ptr = Some z
shows (simple-lift (hrs-mem-update (heap-update (Ptr &(ptr→f)) val) hp)) =
        (simple-lift hp)(ptr ↦ xfu val z)
        (is ?LHS = ?RHS)
proof (rule ext)
fix p

have eui: typ-uinfo-t TYPE('b) =
        export-uinfo (adjust-ti (typ-info-t TYPE('b)) xf xfu)
using xf-xfu
apply (subst export-tag-adjust-ti2 [OF - wf-lf wf-desc])
apply (simp add: fg-cons-def )
apply (rule meta-eq-to-obj-eq [OF typ-uinfo-t-def])
done

have n-is-field-offset: n = field-offset TYPE('a) f
apply (insert field-lookup-offset-eq [OF fl])
apply (clarsimp)
done

have equal-case: ?LHS ptr = ?RHS ptr
supply unsigned-of-nat [simp del]
apply (insert cl)
apply (clarsimp simp: simple-lift-def split: if-split-asm)
apply (clarsimp simp: hrs-mem-update)
apply (subst h-val-super-update-bs)

```

```

apply (rule field-of-t-field-lookup [OF fl])
apply (clarsimp simp: eui)
apply (clarsimp simp: n-is-field-offset)
apply clarsimp
apply (unfold from-bytes-def)
apply (subst fi-fu-consistentD [where f=f and s=adjust-ti (typ-info-t TYPE('b))
xf xfu])
apply (clarsimp simp: fl)
apply (clarsimp simp: n-is-field-offset field-lvalue-def)
apply (metis unat-of-nat-field-offset fl)
apply clarsimp
apply (clarsimp simp: size-of-def)
apply (clarsimp simp: size-of-def)
apply clarsimp
apply (subst update-ti-s-from-bytes)
apply clarsimp
apply (subst update-ti-t-adjust-ti)
apply (rule xf-xfu)
apply (subst update-ti-s-from-bytes)
apply clarsimp
apply clarsimp
apply (clarsimp simp: h-val-def)
done

```

```

show ?LHS p = ?RHS p
apply (cases p = ptr)
apply (erule ssubst)
apply (rule equal-case)
apply (insert cl)
apply (clarsimp simp: simple-lift-def hrs-mem-update split: if-split-asm)
apply (rule h-val-heap-update-disjoint)
apply (insert field-tag-sub [OF fl, where p=ptr])
apply (clarsimp simp: size-of-def)
apply (clarsimp simp: root-ptr-valid-legacy-def)
apply (frule (1) valid-simple-footprint-neq-disjoint, fastforce)
apply clarsimp
apply blast
done

```

qed

lemma simple-lift-field-update:

```

fixes val :: 'b :: mem-type and ptr :: 'a :: mem-type ptr
assumes fl: field-ti TYPE('a) f =
          Some (adjust-ti (typ-info-t TYPE('b)) xf (xfu o (λx -. x)))
and xf-xfu: fg-cons xf (xfu o (λx -. x))
and cl: simple-lift hp ptr = Some z
shows (simple-lift (hrs-mem-update (heap-update (Ptr &(ptr→f)) val) hp)) =
        (simple-lift hp)(ptr ↦ xfu (λ-. val) z)
(is ?LHS = ?RHS)

```

```

apply (insert fl [unfolded field-ti-def])
apply (clarsimp split: option.splits)
apply (subst simple-lift-field-update' [where xf=xf and xfu=xfu o (λx -. x) and
z=z])
  apply (clarsimp simp: o-def split: option.splits)
  apply (rule refl)
  apply (rule xf-xfu)
  apply (rule cl)
apply clarsimp
done

```

```

lemma simple-heap-diff-types-impl-diff-ptrs:
  [| root-ptr-valid h (p::('a::c-type) ptr);
    root-ptr-valid h (q::('b::c-type) ptr);
    typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('b) |] ⇒
  ptr-val p ≠ ptr-val q
apply (clarsimp simp: root-ptr-valid-legacy-def)
apply (clarsimp simp: valid-simple-footprint-def)
done

```

```

lemma h-val-update-regions-disjoint:
  [| { ptr-val p ..+ size-of TYPE('a) } ∩ { ptr-val x ..+ size-of TYPE('b) } = { } |]
  ⇒
  h-val (heap-update p (v::('a::mem-type)) h) x = h-val h (x::('b::c-type) ptr)
apply (clarsimp simp: heap-update-def)
apply (clarsimp simp: h-val-def)
apply (subst heap-list-update-disjoint-same)
apply clarsimp
apply clarsimp
done

```

```

lemma h-val-heap-update-padding-disjoint:
  fixes p::'a::mem-type ptr
  fixes q::'b::c-type ptr
  shows [| ptr-span p ∩ ptr-span q = { }; length bs = size-of TYPE('a) |] ⇒
  h-val (heap-update-padding p v bs h) q = h-val h q
apply (clarsimp simp: heap-update-padding-def)
apply (clarsimp simp: h-val-def)
apply (subst heap-list-update-disjoint-same)
apply clarsimp
apply clarsimp
done

```

```

lemma simple-lift-field-update-t:
  fixes val :: 'b :: mem-type and ptr :: 'a :: mem-type ptr
  assumes fl: field-ti TYPE('a) f = Some t
  and diff: typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('c :: c-type)
  and eu: export-uinfo t = export-uinfo (typ-info-t TYPE('b))

```

```

and      cl: simple-lift hp ptr = Some z
shows ((simple-lift (hrs-mem-update (heap-update (Ptr &(ptr→f)) val) hp)) :: 'c
ptr ⇒ 'c option) =
          simple-lift hp
apply (rule ext)
subgoal for x
  apply (cases simple-lift hp x)
  apply clarsimp
  apply (cases ptr-val x = ptr-val ptr)
  apply clarsimp
  apply (clarsimp simp: simple-lift-def hrs-mem-update split: if-split-asm)
  apply (cut-tac simple-lift-root-ptr-valid [OF cl])
  apply (drule (1) simple-heap-diff-types-impl-diff-ptrs [OF - - diff])
  apply simp
  apply (clarsimp simp: simple-lift-def hrs-mem-update split: if-split-asm)
  apply (rule field-ti-field-lookupE [OF fl])
  apply (frule-tac p=ptr in field-tag-sub)
  apply (clarsimp simp: h-val-def heap-update-def)
  apply (subst heap-list-update-disjoint-same)
  apply clarsimp
  apply (cut-tac simple-lift-root-ptr-valid [OF cl])
  apply (drule (2) root-ptr-valid-neq-disjoint)
  apply (clarsimp simp: export-size-of [unfolded typ-uinfo-t-def, OF eu])
  apply blast
apply simp
done
done

```

```

lemma simple-lift-heap-update-other':
  [| simple-lift h (p::'b::mem-type ptr) = Some v';
    typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('b) |] ⇒
    simple-lift (hrs-mem-update (heap-update p v) h) = ((simple-lift h)::'a::c-type
typ-heap)
apply (rule simple-lift-heap-update-other)
apply (erule simple-lift-root-ptr-valid)
apply simp
done

```

```

lemma simple-lift-heap-update-bytes-in-other:
  [| simple-lift h (p::'b::mem-type ptr) = Some v';
    typ-uinfo-t TYPE('b) ≠ typ-uinfo-t TYPE('c);
    { ptr-val q ..+ size-of TYPE('a) } ⊆ { ptr-val p ..+ size-of TYPE('b) } |] ⇒
    simple-lift (hrs-mem-update (heap-update (q::'a::mem-type ptr) v) h) = ((simple-lift
h)::'c::mem-type typ-heap)
apply (rule ext)

```

```

apply (clarsimp simp: simple-lift-def split: if-split-asm)
apply (drule (1) root-ptr-valid-type-neq-disjoint, simp)
apply (clarsimp simp: hrs-mem-update)
apply (rule h-val-heap-update-disjoint)
apply blast
done

```

lemma *typ-name-neq*:

```

  typ-name (export-uinfo (typ-info-t TYPE('a::c-type)))
    ≠ typ-name (export-uinfo (typ-info-t TYPE('b::c-type)))
  ⇒ typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('b)
apply (metis typ-uinfo-t-def)
done

```

lemma *of-nat-mod-div-decomp*:

```

  of-nat k
    = of-nat (k div size-of TYPE('b)) * of-nat (size-of TYPE('b::mem-type)) +
      of-nat (k mod size-of TYPE('b))
by (metis mod-div-decomp of-nat-add of-nat-mult)

```

lemma *c-guard-array-c-guard*:

```

  [ [  $\wedge x. x < \text{CARD}('a) \implies \text{c-guard} (\text{ptr-coerce } p +_p \text{ int } x :: 'b \text{ ptr})$  ] ] ⇒ c-guard
  ( p :: ('b :: mem-type, 'a :: finite) array ptr )
apply atomize
apply (clarsimp simp: c-guard-def)
apply (rule conjI)
  apply (drule-tac x=0 in spec)
  apply (clarsimp simp: ptr-aligned-def align-of-def align-td-array)
apply (simp add: c-null-guard-def)
apply (clarsimp simp: intvl-def)
apply (drule-tac x=k div size-of TYPE('b) in spec)
apply (erule impE)
  apply (metis (full-types) less-nat-zero-code mult-is-0 neq0-conv td-gal-lt)
apply clarsimp
apply (drule-tac x=k mod size-of TYPE('b) in spec)
apply (clarsimp simp: CTypesDefs.ptr-add-def)
apply (subst (asm) add.assoc)
apply (subst (asm) of-nat-mod-div-decomp [symmetric])
apply clarsimp
done

```

lemma *heap-list-update-list'*:

```

  [ [  $n + x \leq \text{length } v; \text{length } v < \text{addr-card}; q = (p + \text{of-nat } x)$  ] ] ⇒
    heap-list (heap-update-list p v h) n q = take n (drop x v)
by (metis heap-list-update-list)

```

lemma *outside-intvl-range*:


```

  p ∉ {a ..+ b} ⇒ p < a ∨ p ≥ a + of-nat b
apply (clarsimp simp: intvl-def not-le not-less)
apply (drule-tac x=unat (p-a) in spec)
apply clarsimp
apply (metis add-diff-cancel2 le-less-linear le-unat-uoι
  mpl-lem not-add-less2 unat-mono word-less-minus-mono-left)
done

```

lemma *root-ptr-valid-intersect-array*:

```

[[ ∀ j < n. root-ptr-valid htd (p +p int j);
  root-ptr-valid htd (q :: ('a :: c-type) ptr) ]]
  ⇒ (∃ m < n. q = (p +p int m))
  ∨ ({ptr-val p ..+ size-of TYPE ('a) * n} ∩ {ptr-val q ..+ size-of TYPE ('a ::
c-type)}) = {}
apply (induct n)
apply clarsimp
subgoal for n
  apply atomize
  apply simp
  apply (cases n = 0)
  apply clarsimp
  apply (metis root-ptr-valid-neq-disjoint ptr-val-inj)
  apply (erule disjE)
  apply (metis less-Suc-eq)
  apply (cases q = p +p int n)
  apply force
  apply (frule-tac x=n in spec)
  apply (erule impE, simp)
  apply (drule (1) root-ptr-valid-neq-disjoint)
  apply simp
  apply (simp add: CTypesDefs.ptr-add-def)
  apply (rule disjI2)
  apply (cut-tac a= of-nat n * of-nat (size-of TYPE ('a))
    and p=ptr-val p and n=n * size-of TYPE ('a) + size-of TYPE ('a) in
intvl-split)
  apply clarsimp
  apply (clarsimp simp: field-simps Int-Un-distrib2)
  apply (metis IntI emptyE intvl-empty intvl-inter intvl-self neq0-conv)
done
done

```

lemmas *simple-lift-simps* =

```

  typ-name-neq
  simple-lift-c-guard
  h-val-simple-lift
  simple-lift-heap-update
  simple-lift-heap-update-other

```

c-guard-field
h-val-field-simple-lift
simple-lift-field-update
simple-lift-field-update-t
c-guard-array-field

lemmas *typ-simple-heap-simps = simple-lift-simps*

lemma *valid-footprint-overlap-cases:*

assumes *valid-p: valid-footprint d (ptr-val (p::'a::mem-type ptr)) (typ-uinfo-t TYPE('a))*

assumes *valid-q: valid-footprint d (ptr-val (q::'b::mem-type ptr)) (typ-uinfo-t TYPE('b))*

assumes *overlap: ptr-span p \cap ptr-span q \neq {}*

shows *TYPE('a) \leq_τ TYPE('b) \vee TYPE('b) \leq_τ TYPE('a)*

proof (*cases typ-uinfo-t (TYPE('a)) = typ-uinfo-t (TYPE('b))*)

case *True*

thus *?thesis by (simp add: sub-typ-def)*

next

case *False*

note *not-eq = False*

show *?thesis*

proof (*cases TYPE('a) \perp_τ TYPE('b)*)

case *False*

thus *?thesis by (simp add: tag-disj-typ-def tag-disj-def sub-typ-def)*

next

case *True*

note *disj-types = True*

with *disj-types not-eq have ne-le-b-a: \neg typ-uinfo-t (TYPE('b)) < typ-uinfo-t (TYPE('a))*

apply (*simp add: tag-disj-typ-def tag-disj-def*)

by (*meson le-less*)

have *not-field-of-p-q: \neg field-of (ptr-val p - ptr-val q) (typ-uinfo-t (TYPE('a))) (typ-uinfo-t (TYPE('b)))*

apply (*rule ccontr, simp*)

apply (*drule field-of-sub*)

using *disj-types*

apply (*simp add: tag-disj-typ-def tag-disj-def*)

done

from *valid-footprint-neq-disjoint [OF valid-p valid-q ne-le-b-a not-field-of-p-q]*

have *ptr-span p \cap ptr-span q = {}*

by (*simp add: size-of-def*)

with *overlap show ?thesis by auto*

qed
qed

lemma *valid-root-footprint-valid-footprint-overlap-case:*
assumes *valid-p: valid-root-footprint d (ptr-val (p::'a:: mem-type ptr)) (typ-uinfo-t (TYPE('a)))*
assumes *valid-q: valid-footprint d (ptr-val (q::'b::mem-type ptr)) (typ-uinfo-t (TYPE('b)))*
assumes *overlap: ptr-span p \cap ptr-span q \neq {}*
shows *TYPE('b) \leq_{τ} TYPE('a)*
proof –
from *overlap [simplified size-of-def, folded typ-uinfo-t-def]*
have *{ptr-val p.. size-td (typ-uinfo-t TYPE('a))} \cap {ptr-val q.. size-td (typ-uinfo-t TYPE('b))} \neq {}*
by *simp*
from *valid-root-footprint-overlap-sub-typ [OF valid-p valid-q this]*
have *typ-uinfo-t TYPE('b) \leq typ-uinfo-t TYPE('a).*
thus *?thesis*
by *(simp add: sub-typ-def)*
qed

lemma *valid-root-footprint-overlap-contained:*
assumes *valid-root-x: valid-root-footprint d x t*
assumes *valid-y: valid-footprint d y s*
assumes *overlap: {x.. size-td t} \cap {y.. size-td s} \neq {}*
shows *y \in {x.. size-td t}*
using *valid-root-x overlap valid-y*
apply *(clarsimp simp add: valid-root-footprint-def valid-footprint-def Let-def typ-tag-le-def)*
by *(metis (no-types, opaque-lifting) intvl-inter intvl-inter-le le-less-trans less-irrefl overlap valid-footprint-sub2 valid-root-footprint-overlap-sub-typ valid-root-footprint-valid-footprint valid-root-x valid-y zero-le)*

lemma *valid-footprint-field-of-contained:*
assumes *valid-x: valid-footprint d x t*
assumes *field: field-of off s t*
shows *{x + off.. size-td s} \subseteq {x.. size-td t}*
proof –
from *field have (s, unat off) \in td-set t 0*
by *(simp add: field-of-def)*
from *td-set-offset-size [OF this] have size-td s + unat off \leq size-td t.*
thus *?thesis*
by *(simp add: intvl-sub-offset)*
qed

lemma *valid-root-footprint-overlap-field-of:*

assumes *valid-root-x*: *valid-root-footprint* $d\ x\ t$
assumes *valid-y*: *valid-footprint* $d\ y\ s$
assumes $y: y \in \{x \text{ ..+ size-td } t\}$
shows *field-of* $(y - x)\ s\ t$
proof –
from y **have** $\{x \text{ ..+ size-td } t\} \cap \{y \text{ ..+ size-td } s\} \neq \{\}$
by (*meson disjoint-iff intvl-self valid-footprint-def valid-y*)
from *valid-root-footprint-overlap-sub-typ* [*OF valid-root-x valid-y this*]
have $s \leq t$.
with y **show** *?thesis*
by (*meson le-less-trans less-irrefl valid-footprint-sub valid-root-footprint-valid-footprint valid-root-x valid-y*)
qed

lemma *valid-root-footprint-overlap-contained'*:
assumes *valid-root-x*: *valid-root-footprint* $d\ x\ t$
assumes *valid-y*: *valid-footprint* $d\ y\ s$
assumes *overlap*: $\{x \text{ ..+ size-td } t\} \cap \{y \text{ ..+ size-td } s\} \neq \{\}$
shows $\{y \text{ ..+ size-td } s\} \subseteq \{x \text{ ..+ size-td } t\}$
proof –
from *valid-root-footprint-overlap-contained* [*OF valid-root-x valid-y overlap*]
have $y \in \{x \text{ ..+ size-td } t\}$.
from *valid-root-footprint-overlap-field-of* [*OF valid-root-x valid-y this*]
have *field-of* $(y - x)\ s\ t$.
from *valid-footprint-field-of-contained* [*OF valid-root-footprint-valid-footprint* [*OF valid-root-x*] *this*]
have $\{x + (y - x) \text{ ..+ size-td } s\} \subseteq \{x \text{ ..+ size-td } t\}$.
then show *?thesis*
by *auto*
qed

lemma *valid-footprint-sub-cases*:
assumes *valid-p*: *valid-footprint* $d\ p\ s$
assumes *valid-q*: *valid-footprint* $d\ q\ t$
assumes *sub*: $\neg t < s$
shows $\{p \text{ ..+ size-td } s\} \cap \{q \text{ ..+ size-td } t\} = \{\} \vee \text{field-of } (p - q)\ (s)\ (t)$
using *valid-footprint-neq-disjoint valid-p valid-q sub* **by** *blast*

lemma *valid-root-footprint-dist-type-cases*:
assumes *valid-p*: *valid-root-footprint* $d\ (\text{ptr-val } (p::'a:: \text{mem-type ptr}))\ (\text{typ-uinfo-t } (\text{TYPE}'a))$
assumes *valid-q*: *valid-footprint* $d\ (\text{ptr-val } (q::'b:: \text{mem-type ptr}))\ (\text{typ-uinfo-t } (\text{TYPE}'b))$
assumes *dist-type*: $\text{typ-uinfo-t } (\text{TYPE}'a) \neq \text{typ-uinfo-t } (\text{TYPE}'b)$
assumes *nested-case*: $\bigwedge f. f \in \text{set } (\text{field-names-u } (\text{typ-uinfo-t } (\text{TYPE}'a)))\ (\text{typ-uinfo-t } (\text{TYPE}'b)) \implies$
 $\text{field-lookup } (\text{typ-uinfo-t } (\text{TYPE}'a))\ f\ 0 = \text{Some } (\text{typ-uinfo-t } (\text{TYPE}'b), \text{unat } (\text{ptr-val } q - \text{ptr-val } p)) \implies$

$field-of-t (PTR('b) \&(p \rightarrow f)) p \implies$
 $ptr-val q = \&(p \rightarrow f) \implies$
 $ptr-span q \subseteq ptr-span p \implies P$
assumes *disj-case*: $ptr-span p \cap ptr-span q = \{\} \implies P$
shows P
proof (*cases* $ptr-span p \cap ptr-span q = \{\}$)
case *True* **thus** *?thesis* **by** (*rule disj-case*)
next
case *False*
from *valid-root-footprint-valid-footprint-overlap-case* [*OF valid-p valid-q False*]
have *le-b-a*: $typ-uinfo-t TYPE('b) \leq typ-uinfo-t TYPE('a)$ **by** (*simp add: sub-typ-def*)

from *sub-typ-field-names-u-nonempty* [*OF this*]
obtain f **where** $f \in set (field-names-u (typ-uinfo-t (TYPE('a))) (typ-uinfo-t (TYPE('b))))$
by *fastforce*

from *False* **have** $\{ptr-val p..+size-td (typ-uinfo-t TYPE('a))\} \cap \{ptr-val q..+size-td (typ-uinfo-t TYPE('b))\} \neq \{\}$
by (*simp add: size-of-def*)
from *valid-root-footprint-overlap-contained'* [*OF valid-p valid-q this*]
have $\{ptr-val q..+size-td (typ-uinfo-t TYPE('b))\} \subseteq \{ptr-val p..+size-td (typ-uinfo-t TYPE('a))\}$
hence $ptr-val q \in \{ptr-val p..+size-td (typ-uinfo-t TYPE('a))\}$
by (*simp add: size-of-tag subsetD*)
from *valid-root-footprint-overlap-field-of* [*OF valid-p valid-q this*]
have $field-of (ptr-val q - ptr-val p) (typ-uinfo-t TYPE('b)) (typ-uinfo-t TYPE('a))$

from *field-of-lookup-info* [*OF this, of p*]
obtain f **where**
 $f \in set (field-names-u (typ-uinfo-t (TYPE('a))) (typ-uinfo-t (TYPE('b))))$
 $field-lookup (typ-uinfo-t TYPE('a)) f 0 =$
 $Some (typ-uinfo-t TYPE('b), unat (ptr-val q - ptr-val p))$
 $field-of-t (PTR('b) \&(p \rightarrow f)) p$
 $ptr-val q = \&(p \rightarrow f)$
 $ptr-span q \subseteq ptr-span p$
by (*auto simp add: size-of-def*)
from *nested-case* [*OF this*]
show *?thesis* .
qed

lemma *cvalid-field-pres*:

assumes *lookup*: $field-lookup (typ-uinfo-t TYPE('a::mem-type)) f 0 = Some (typ-uinfo-t TYPE('b::mem-type), n)$
assumes *valid*: $d, c-guard \models_t (p::'a:: mem-type ptr)$
shows $d, c-guard \models_t PTR('b) \&(p \rightarrow f)$
using *field-lookup-export-uinfo-Some-rev* [*OF lookup [simplified typ-uinfo-t-def]*]
apply (*clarsimp*)

```

apply (rule h-t-valid-mono [rule-format, OF - - c-guard-mono valid])
apply (auto simp add: typ-uinfo-t-def)
done

```

lemma *cvalid-field-pres'*:

```

assumes ptr-val-eq: ptr-val q =  $\&(p \rightarrow f)$ 
assumes lookup: field-lookup (typ-uinfo-t TYPE('a::mem-type)) f 0 = Some
(typ-uinfo-t TYPE('b::mem-type), n)
assumes valid: d, c-guard  $\models_t$  (p::'a:: mem-type ptr)
shows d, c-guard  $\models_t$  (q::'b ptr)
proof -
from ptr-val-eq have q = PTR('b) ( $\&(p \rightarrow f)$ )
by (cases q) (auto simp add: ptr-val-def)
with cvalid-field-pres [OF lookup valid]
show ?thesis
by simp
qed

```

lemma *cvalid-field-pres''*:

```

assumes ptr-val-eq: ptr-val q =  $\&(p \rightarrow f)$ 
assumes lookup: field-lookup (typ-uinfo-t TYPE('a::mem-type)) f 0 = Some (s,
n)
assumes match: export-uinfo s = typ-uinfo-t TYPE('b::mem-type)
assumes valid: d, c-guard  $\models_t$  (p::'a:: mem-type ptr)
shows d, c-guard  $\models_t$  (q::'b ptr)
proof -
from cvalid-field-pres' [OF ptr-val-eq - valid] lookup match
show ?thesis
by (simp add: UMM.field-lookup-typ-uinfo-t-Some)
qed

```

lemma *cvalid-field-pres'''*:

```

assumes lookup: field-lookup (typ-uinfo-t TYPE('a::mem-type)) f 0 = Some (t,
n)
assumes match: export-uinfo t = typ-uinfo-t TYPE('b::mem-type)
assumes valid: d, c-guard  $\models_t$  (p::'a:: mem-type ptr)
shows d, c-guard  $\models_t$  PTR('b)  $\&(p \rightarrow f)$ 
using lookup match valid
using c-guard-mono h-t-valid-mono by blast

```

lemma *the-clift-eq-h-val-eq*:

```

assumes h-val-eq: hrs-htd hp  $\models_t$  p  $\implies$  h-val (hrs-mem (hrs-mem-update a hp))
p = h-val (hrs-mem hp) p
shows the (clift (hrs-mem-update a hp) p) = the (clift hp p)
proof (cases clift hp p)
case None
with clift-hrs-mem-update-None show ?thesis by metis
next
case (Some v)

```

with *clift-hrs-mem-update-None* **obtain** v' **where** *clift* (*hrs-mem-update* a hp) p
 $=$ *Some* v'
by (*cases* *clift* (*hrs-mem-update* a hp) p) *auto*
with *Some* *h-val-clift'* *h-val-eq* [*OF* *h-t-valid-clift* [*OF* *Some*]]
show *?thesis* **by** *metis*
qed

lemma *field-lvalue-same-conv*: $\&(p::'a:: c\text{-type } ptr \rightarrow f) = \&(q::'a:: c\text{-type } ptr \rightarrow f)$
 $\implies p = q$
by (*simp* *add*: *field-lvalue-def*)

lemma *ptr-val-field-convD*: $ptr\text{-val } p = \&(q \rightarrow f) \implies p = Ptr \ \&(q \rightarrow f)$
by (*cases* p) *auto*

lemma *ptr-val-field-conv*: $ptr\text{-val } p = \&(q \rightarrow f) \longleftrightarrow p = Ptr \ \&(q \rightarrow f)$
by (*cases* p) *auto*

lemma *ptr-val-array-index-convD*:
 $ptr\text{-val } p = ptr\text{-val } (array\text{-ptr-index } q \text{ False } j) \implies p = array\text{-ptr-index } q \text{ False } j$
by (*cases* p) *auto*

lemma *ptr-val-conv'*: $ptr\text{-coerce } p = Ptr \ q \longleftrightarrow ptr\text{-val } p = q$
by (*cases* p) *simp*

lemma *ptr-val-conv*: $p = q \longleftrightarrow ptr\text{-val } p = ptr\text{-val } q$
by *auto*

lemma *ptr-val-neq-conv*: $p \neq q \longleftrightarrow ptr\text{-val } p \neq ptr\text{-val } q$
by *auto*

lemma *the-simple-lift-strong-eqI*:
fixes $p::'a::mem\text{-type } ptr$
fixes $q::'b::mem\text{-type } ptr$

assumes $eq: \bigwedge x1 \ x2. root\text{-ptr-valid } (hrs\text{-htd } s) \ q \implies$
 $simple\text{-lift } s \ q = \text{Some } x1 \implies$
 $(simple\text{-lift } (hrs\text{-mem-update } (heap\text{-update } p \ v) \ s) \ q) = \text{Some } x2 \implies$
 $x1 = x2$

shows *the* (*simple-lift*
 $(hrs\text{-mem-update } (heap\text{-update } p \ v) \ s) \ q) =$
 $the \ (simple\text{-lift } s \ q)$

proof (*cases* *root-ptr-valid* (*hrs-htd* s) q)

case *True*

with eq **show** *?thesis* **by** (*fastforce* *simp* *add*: *simple-lift-def*)

next

case *False*

hence $simple\text{-lift } s \ q = \text{None}$

by (*simp* *add*: *simple-lift-def*)

with *simple-lift-hrs-mem-update-None*

show *?thesis* **by** *metis*
qed

lemma *the-simple-lift-hval-eqI*:

fixes *p::'a::mem-type ptr*

fixes *q::'b::mem-type ptr*

assumes *eq: root-ptr-valid (hrs-htd s) q* \implies
 $(h\text{-val } ((heap\text{-update } p \ v) \ (hrs\text{-mem } s)) \ q) = h\text{-val } (hrs\text{-mem } s) \ q$

shows *the (simple-lift*
 $(hrs\text{-mem-update } (heap\text{-update } p \ v) \ s) \ q) =$
 $the \ (simple\text{-lift } s \ q)$

apply *(rule the-simple-lift-strong-eqI)*

apply *(drule h-val-simple-lift)*

apply *(drule h-val-simple-lift)*

apply *(auto simp add: hrs-mem-update eq)*

done

lemma *h-t-valid-hrs-mem-update-pres: hrs-htd s,g* $\models_t q \implies$
 $hrs\text{-htd } ((hrs\text{-mem-update } (heap\text{-update } p \ v) \ s)), g \models_t q$
by *(cases s) (auto simp add: hrs-htd-def hrs-mem-update-def)*

lemma *field-the-clift-hval-eqI*:

fixes *p::'a::mem-type ptr*

fixes *q::'b::mem-type ptr*

assumes *eq: hrs-htd s, c-guard* $\models_t q \implies$
 $f \ (h\text{-val } ((heap\text{-update } p \ v) \ (hrs\text{-mem } s)) \ q) = f \ (h\text{-val } (hrs\text{-mem } s) \ q)$

shows *f (the (clift*
 $(hrs\text{-mem-update } (heap\text{-update } p \ v) \ s) \ q)) =$
 $f \ (the \ (clift \ s \ q))$

proof *(cases (clift s q))*

case *None*

then show *?thesis using clift-hrs-mem-update-None* **by** *metis*

next

case *(Some x)*

from *h-t-valid-clift [OF this]* **have** *valid: hrs-htd s* $\models_t q$.

hence $hrs\text{-htd } ((hrs\text{-mem-update } (heap\text{-update } p \ v) \ s)) \models_t q$

by *(rule h-t-valid-hrs-mem-update-pres)*

with *h-t-valid-clift-Some-iff*

obtain *y* **where** *y: clift (hrs-mem-update (heap-update p v) s) q = Some y*

by *blast*

from *h-val-clift' [OF Some] h-val-clift' [OF this] eq [OF valid]*

show *?thesis* **by** *(auto simp add: Some y hrs-mem-update)*

qed

lemma *the-clift-hval-eqI*:

fixes *p::'a::mem-type ptr*

fixes *q::'b::mem-type ptr*

assumes *eq*: *hrs-htd s*, *c-guard* $\models_t q \implies$
 $(h\text{-val } ((heap\text{-update } p \ v) \ (hrs\text{-mem } s)) \ q) = (h\text{-val } (hrs\text{-mem } s) \ q)$
shows *the* (*clift*
 $(hrs\text{-mem-update } (heap\text{-update } p \ v) \ s) \ q) =$
 $(the \ (clift \ s \ q))$
apply (*rule* *field-the-clift-hval-eqI*)
by (*rule* *eq*)

lemma *valid-root-footprint-same-type-cases*:

assumes *valid-p*: *valid-root-footprint* *d* (*ptr-val* (*p*::'*a*:: *mem-type ptr*)) (*typ-uinfo-t*
 $(TYPE('a))$)
assumes *valid-q*: *valid-footprint* *d* (*ptr-val* (*q*::'*a*::*mem-type ptr*)) (*typ-uinfo-t*
 $(TYPE('a))$)

assumes *eq-case*: $p = q \implies P$
assumes *disj-case*: $ptr\text{-span } p \cap ptr\text{-span } q = \{\} \implies P$
shows *P*

proof (*cases* $ptr\text{-span } p \cap ptr\text{-span } q = \{\}$)

case *True*

then show *?thesis* **by** (*simp* *add*: *disj-case*)

next

case *False*

from *valid-root-footprint-overlap-contained* [*OF* *valid-p* *valid-q*] *False*

have *ptr-val* *q* $\in ptr\text{-span } p$

by (*simp* *add*: *typ-uinfo-t-def* *size-of-def*)

hence $p = q$

by (*metis* *ptr-val-conv* *size-of-tag* *valid-footprint-neq-nmem* *valid-p* *valid-q* *valid-root-footprint-valid-footprint*)

thus *?thesis* **by** (*simp* *add*: *eq-case*)

qed

lemma *valid-footprint-overlap-contained*:

assumes *valid-root-x*: *valid-footprint* *d* *x* *t*

assumes *valid-y*: *valid-footprint* *d* *y* *s*

assumes *overlap*: $\{x \ ..+ \ size\text{-td } t\} \cap \{y \ ..+ \ size\text{-td } s\} \neq \{\}$

shows $y \in \{x \ ..+ \ size\text{-td } t\} \vee x \in \{y \ ..+ \ size\text{-td } s\}$

using *valid-root-x* *overlap* *valid-y*

apply (*clarsimp* *simp* *add*: *valid-footprint-def* *Let-def* *typ-tag-le-def*)

by (*meson* *intvl-inter* *overlap*)

lemma *valid-footprint-same-type-cases*:

assumes *valid-p*: *valid-footprint* *d* (*ptr-val* (*p*::'*a*:: *mem-type ptr*)) (*typ-uinfo-t*
 $(TYPE('a))$)

assumes *valid-q*: *valid-footprint* *d* (*ptr-val* (*q*::'*a*::*mem-type ptr*)) (*typ-uinfo-t*
 $(TYPE('a))$)

assumes *eq-case*: $p = q \implies P$

assumes *disj-case*: $ptr\text{-span } p \cap ptr\text{-span } q = \{\} \implies P$

shows *P*

proof (*cases* $ptr\text{-span } p \cap ptr\text{-span } q = \{\}$)

```

case True
then show ?thesis by (simp add: disj-case)
next
case False
from valid-footprint-overlap-contained [OF valid-p valid-q] False
have ptr-val q ∈ ptr-span p ∨ ptr-val p ∈ ptr-span q
  by (simp add: typ-uinfo-t-def size-of-def)
hence  $p = q$ 
  by (metis ptr-val-conv size-of-tag valid-footprint-neq-nmem valid-p valid-q)
thus ?thesis by (simp add: eq-case)
qed

theorem subfield-deref-append:
  fixes  $q::'a::mem\text{-}type\ ptr$ 
  fixes  $p::'b::mem\text{-}type\ ptr$ 
  assumes base: ptr-val p = &(q→g)
  assumes  $g: g \in set\ (field\text{-}names\text{-}u\ (typ\text{-}uinfo\text{-}t\ (TYPE('a)))\ (typ\text{-}uinfo\text{-}t\ (TYPE('b))))$ 
  assumes  $f: f \in set\ (all\text{-}field\text{-}names\ (typ\text{-}uinfo\text{-}t\ (TYPE('b))))$ 
  shows  $\&(p\rightarrow f) = \&(q\rightarrow(g@f))$ 
proof –
  have wf-a: wf-desc (typ-info-t (TYPE('a))) by simp
  have wf-b: wf-desc (typ-info-t (TYPE('b))) by simp
  from  $g$  have  $g \in set\ (field\text{-}names\ (typ\text{-}info\text{-}t\ (TYPE('a)))\ (typ\text{-}uinfo\text{-}t\ (TYPE('b))))$ 
    by (simp add: typ-uinfo-t-def field-names-u-field-names-export-uinfo-conv(1))
  from field-names-Some2(1) [rule-format, OF wf-a this] obtain  $n\ s$  where
    fl-g: field-lookup (typ-info-t TYPE('a)) g 0 = Some (s, n) and
    s: export-uinfo s = typ-uinfo-t TYPE('b)
    by blast

  from fl-g have off-g: field-offset TYPE('a) g = n
    by (simp)

  from all-field-names-exists-field-names-u(1) [OF f] obtain  $t$ 
    where  $f \in set\ (field\text{-}names\text{-}u\ (typ\text{-}uinfo\text{-}t\ TYPE('b))\ t)$  by blast
  hence  $f \in set\ (field\text{-}names\ (typ\text{-}info\text{-}t\ TYPE('b))\ t)$ 
    by (simp add: field-names-u-field-names-export-uinfo-conv(1) typ-uinfo-t-def)
  from field-names-Some2(1) [rule-format, OF wf-b this] obtain  $t'\ m$  where
    fl-f: field-lookup (typ-info-t TYPE('b)) f 0 = Some (t', m) and
    t': export-uinfo t' = t
    by blast

  from fl-f have off-f: field-offset TYPE('b) f = m
    by simp

  from field-lookup-prefix-Some''(1) [rule-format, OF fl-g wf-a, of f]
  have fl-g-f: field-lookup (typ-info-t TYPE('a)) (g @ f) 0 = field-lookup s f n .

  from field-lookup-export-uinfo-Some [OF fl-f]  $t'$ 
  have field-lookup (typ-uinfo-t TYPE('b)) f 0 = Some (t, m)

```

by (*simp add: typ-uinfo-t-def*)
from *field-lookup-offset-shift'* [*OF this, of n*]
have *field-lookup (typ-uinfo-t TYPE('b)) f n = Some (t, m + n)*
by *simp*

with *s* **have** *field-lookup (export-uinfo s) f n = Some (t, m + n)*
by *simp*
from *field-lookup-export-uinfo-Some-rev* [*OF this*] **obtain** *t''* **where**
fl-s-f: field-lookup s f n = Some (t'', m + n) **and**
t'': t = export-uinfo t''
by *blast*
from *fl-g-f fl-s-f* **have** *off-g-f: field-offset TYPE('a) (g@f) = m + n*
by (*simp*)

show *?thesis*
using *base*
by (*simp add: field-lvalue-def off-g off-f off-g-f*)
qed

lemmas *root-ptr-valid-same-type-cases = valid-root-footprint-same-type-cases* [*OF root-ptr-valid-valid-root-footprint h-t-valid-valid-footprint*]
lemmas *ptr-valid-same-type-cases = valid-footprint-same-type-cases* [*OF h-t-valid-valid-footprint h-t-valid-valid-footprint*]

theorem *root-ptr-valid-heap-update-other'*:
assumes *val-p: root-ptr-valid d (p::'a::mem-type ptr)*
assumes *val-q: d ⊨_t (q::'b::mem-type ptr)*
assumes *not-subtype: ¬ TYPE('b) ≤_τ TYPE('a)*
shows *h-val (heap-update p v h) q = h-val h q*
apply (*clarsimp simp: h-val-def heap-update-def*)
apply (*subst heap-list-update-disjoint-same*)
apply *simp*
apply (*rule root-ptr-valid-not-subtype-disjoint* [*OF val-p val-q not-subtype*])
apply *simp*
done

theorem *root-ptr-valid-heap-update-field-other*:
assumes *val-p: root-ptr-valid d (p::'a::mem-type ptr)*
assumes *val-q: d ⊨_t (q::'b::mem-type ptr)*
assumes *fl: field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (t, n)*
assumes *match: export-uinfo t = typ-uinfo-t TYPE('c::mem-type)*
assumes *not-subtype-b-c: ¬ TYPE('b) ≤_τ TYPE('c)*
assumes *not-subtype-c-b: ¬ TYPE('c) ≤_τ TYPE('b)*
shows *h-val (heap-update (PTR('c) &(p→f)) v h) q = h-val h q*

```

proof (cases ptr-span (PTR('c) &(p→f)) ∩ ptr-span q = {} )
  case True
  then show ?thesis by (simp add: h-val-update-regions-disjoint)
next
  case False
  from valid-footprint-overlap-cases [OF - - False] val-p val-q field-lookup-typ-uinfo-t-Some
  [OF fl]
    not-subtype-b-c not-subtype-c-b match
  have False
  by (smt (verit) cvalid-field-pres h-t-valid-valid-footprint root-ptr-valid-legacy-def)

  thus ?thesis
  by simp
qed

```

```

theorem root-ptr-valid-heap-update-field-other':
  assumes val-p: root-ptr-valid d (p::'a::mem-type ptr)
  assumes val-q: d ⊨t (q::'b::mem-type ptr)
  assumes fl: field-lookup (typ-info-t TYPE('b)) f 0 = Some (t, n)
  assumes match: export-uinfo t = typ-uinfo-t TYPE('c::mem-type)
  assumes not-subtype-b-a: ¬ TYPE('b) ≤τ TYPE('a)
  shows h-val (heap-update p v h) (PTR ('c) &(q→f)) = h-val h (PTR ('c)
  &(q→f))
proof (cases ptr-span p ∩ ptr-span q = {} )
  case True
  from fl val-q match have subset: ptr-span (PTR('c) &(q→f)) ⊆ ptr-span q
  by (metis export-uinfo-size field-tag-sub ptr-val.ptr-val-def size-of-tag)
  show ?thesis
  apply (rule h-val-update-regions-disjoint)
  using True subset
  by auto
next
  case False
  from valid-root-footprint-valid-footprint-overlap-case [OF - - False] val-p val-q
  have TYPE('b) ≤τ TYPE('a)
  by (meson False cvalid-field-pres'' fl match ptr-val.ptr-val-def root-ptr-valid-not-subtype-disjoint)
  with not-subtype-b-a have False by simp
  thus ?thesis
  by simp
qed

```

```

lemmas root-ptr-valid-dist-type-cases = valid-root-footprint-dist-type-cases [OF root-ptr-valid-valid-root-footprint
  h-t-valid-valid-footprint,
  consumes 3, case-names Field Disjoint-Spans]

```

```

theorem ptr-valid-heap-update-field-disj:

```

```

assumes val-p:  $d \models_t (p::'a::mem\text{-}type\ ptr)$ 
assumes val-q:  $d \models_t (q::'b::mem\text{-}type\ ptr)$ 
assumes subtype:  $TYPE('b) \leq_\tau TYPE('a)$ 
assumes disj [rule-format] :
   $\forall f. f \in set (field\text{-}names\text{-}u (typ\text{-}uinfo\text{-}t (TYPE('a))) (typ\text{-}uinfo\text{-}t (TYPE('b))))$ 
 $\longrightarrow q \neq Ptr (\&(p \rightarrow f))$ 
shows h-val (heap-update p v h) q = h-val h q
proof (cases ptr-span p  $\cap$  ptr-span q =  $\{\}$ )
  case True
    then show ?thesis
      by (simp add: h-val-update-regions-disjoint)
  next
    case False
      from subtype have  $typ\text{-}uinfo\text{-}t\ TYPE('b) \leq typ\text{-}uinfo\text{-}t\ TYPE('a)$  by (simp add: sub-typ-def)

      from this False valid-footprint-sub-cases [OF h-t-valid-valid-footprint [OF val-q]
h-t-valid-valid-footprint [OF val-p]]
      have field-of:  $field\text{-}of (ptr\text{-}val\ q - ptr\text{-}val\ p) (typ\text{-}uinfo\text{-}t\ TYPE('b)) (typ\text{-}uinfo\text{-}t\ TYPE('a))$ 
      apply (simp add: size-of-def)
      by (metis Int-commute le-less-trans less-irrefl)
      then obtain f where
        fl:  $field\text{-}lookup (typ\text{-}uinfo\text{-}t (TYPE('a)))\ f\ 0 = Some (typ\text{-}uinfo\text{-}t\ TYPE('b), unat (ptr\text{-}val\ q - ptr\text{-}val\ p))$ 
        using field-of-lookup-info by blast

      then obtain s where
        fl':  $field\text{-}lookup (typ\text{-}info\text{-}t (TYPE('a)))\ f\ 0 = Some (s, unat (ptr\text{-}val\ q - ptr\text{-}val\ p))$  and
        s:  $export\text{-}uinfo\ s = typ\text{-}uinfo\text{-}t\ TYPE('b)$ 
        using field-lookup-uinfo-Some-rev by blast

      from fl have f:  $f \in set (field\text{-}names\text{-}u (typ\text{-}uinfo\text{-}t\ TYPE('a)) (typ\text{-}uinfo\text{-}t\ TYPE('b)))$ 
      by (metis field-names-Some3(1) field-names-u-field-names-export-uinfo-conv(1) fl' s typ-uinfo-t-def)

      from fl'
      have ptr-val-q:  $ptr\text{-}val\ q = \&(p \rightarrow f)$ 
      by (simp add: field-lvalue-def)

      from disj [OF f] ptr-val-q
      show ?thesis
      by (auto simp add: ptr-val-field-convD)
qed

```

lemma *is-padding-tag-access-ti-eq*: $is\text{-}padding\text{-}tag\ s \implies access\text{-}ti\ s\ x\ bs = access\text{-}ti$

s y bs

by (*clarsimp simp add: is-padding-tag-def padding-tag-def from-bytes-def access-ti₀-def padding-desc-def*)

lemma *is-padding-tag-access-ti₀: is-padding-tag s \implies access-ti₀ s x = replicate (size-td s) 0*

by (*clarsimp simp add: is-padding-tag-def padding-tag-def from-bytes-def access-ti₀-def padding-desc-def*)

lemma *is-padding-tag-from-bytes-eq: is-padding-tag s \implies from-bytes (access-ti₀ s x) = from-bytes (access-ti₀ s y)*

by (*simp add: from-bytes-def is-padding-tag-access-ti₀*)

theorem *ptr-valid-heap-update-field-disj'*:

assumes *val-p: d \models_t (p::'a::xmem-type ptr)*

assumes *val-q: d \models_t (q::'b::xmem-type ptr)*

assumes *subtype: TYPE('b) \leq_τ TYPE('a)*

assumes *disj [rule-format] :*

$\forall f. f \in \text{set} (\text{field-names-no-padding} (\text{typ-info-t} (\text{TYPE}('a))) (\text{export-uinfo} (\text{typ-info-t} (\text{TYPE}('b))))) \longrightarrow q \neq \text{Ptr} (\&(p \rightarrow f))$

shows *h-val (heap-update p v h) q = h-val h q*

proof (*cases ptr-span p \cap ptr-span q = {}*)

case *True*

then show *?thesis*

by (*simp add: h-val-update-regions-disjoint*)

next

case *False*

from *subtype* **have** *typ-uinfo-t TYPE('b) \leq typ-uinfo-t TYPE('a)* **by** (*simp add: sub-typ-def*)

from *this False valid-footprint-sub-cases [OF h-t-valid-valid-footprint [OF val-q] h-t-valid-valid-footprint [OF val-p]]*

have *field-of: field-of (ptr-val q - ptr-val p) (typ-uinfo-t TYPE('b)) (typ-uinfo-t TYPE('a))*

apply (*simp add: size-of-def*)

by (*metis Int-commute le-less-trans less-irrefl*)

then obtain *f* **where**

fl: field-lookup (typ-uinfo-t (TYPE('a))) f 0 = Some (typ-uinfo-t TYPE('b), unat (ptr-val q - ptr-val p))

using *field-of-lookup-info* **by** *blast*

then obtain *s* **where**

fl': field-lookup (typ-uinfo-t (TYPE('a))) f 0 = Some (s, unat (ptr-val q - ptr-val p)) **and**

s: export-uinfo s = typ-uinfo-t TYPE('b)

using *field-lookup-uinfo-Some-rev* **by** *blast*

from *fl'* **have** *fl'': field-ti (TYPE('a)) f = Some s*

```

    by (simp add: field-ti-def)

from fl' have f: f ∈ set (field-names (typ-info-t TYPE('a)) (typ-uinfo-t TYPE('b)))

    by (metis field-names-Some3(1) fl' s)

from fl'
have ptr-val-q: ptr-val q = &(p→f)
  by (simp add: field-lvalue-def)
then have q: q = PTR ('b) &(p→f)
  by (cases q) auto

show ?thesis
proof (cases qualified-padding-field-name f)
  case True
  from field-lookup-qualified-padding-field-name(1) [OF fl' True]
  have padding-tag-s: is-padding-tag s
    by (simp add: wf-padding)
  note h-val = h-val-field-from-bytes' [where pa=p, OF fl'' s [simplified typ-uinfo-t-def]]
  show ?thesis apply (simp add: q)
  apply (simp add: h-val)
  apply (simp add: is-padding-tag-from-bytes-eq [OF padding-tag-s])
  done
next
  case False
  with f fl' s have f ∈ set (field-names-no-padding (typ-info-t (TYPE('a)))
    (export-uinfo (typ-info-t (TYPE('b)))))
  by (simp add: set-field-names-no-padding-all-field-names-no-padding-conv
    all-field-names-no-padding-def
    fold-typ-uinfo-t set-field-names-all-field-names-conv)
  with disj [OF this] ptr-val-q show ?thesis by (auto simp add: ptr-val-field-convD)
qed

```

qed

lemma *is-padding-tag-update-ti-id*: *is-padding-tag s* \implies *update-ti s bs v = v*
 by (auto simp add: is-padding-tag-def padding-tag-def padding-desc-def)

theorem *ptr-valid-heap-update-field-disj''*:

```

assumes val-p: d ⊨t (p::'a::xmem-type ptr)
assumes val-q: d ⊨t (q::'b::xmem-type ptr)
assumes subtype: TYPE('b) ≤τ TYPE('a)
assumes fl-g: field-lookup (typ-info-t TYPE('a)) g 0 = Some (t, n)
assumes match: export-uinfo t = typ-uinfo-t TYPE('c::xmem-type)
assumes disj [rule-format] :
  ∀ f. f ∈ set (field-names-no-padding (typ-info-t (TYPE('a))) (export-uinfo (typ-info-t
    (TYPE('b)))))
  ⟶ ptr-span (PTR('b) &(p→f)) ∩ ptr-span (PTR('c) &(p→g)) = {}
shows h-val (heap-update q v h) (PTR('c) &(p→g)) = h-val h (PTR('c) &(p→g))

```

```

proof (cases ptr-span q  $\cap$  ptr-span (PTR('c) &(p→g)) = {})
  case True
  then show ?thesis by (simp add: h-val-update-regions-disjoint)
next
  case False
  from subtype have st: typ-uinfo-t TYPE('b)  $\leq$  typ-uinfo-t TYPE('a) by (simp
add: sub-typ-def)
  from fl-g match have ptr-span (PTR('c) &(p→g))  $\subseteq$  ptr-span p
  by (metis export-uinfo-size field-tag-sub ptr-val.ptr-val-def size-of-tag)
  with False have ptr-span q  $\cap$  ptr-span p  $\neq$  {}
  by auto

  from this st valid-footprint-sub-cases [OF h-t-valid-valid-footprint [OF val-q]
h-t-valid-valid-footprint [OF val-p]]
  have field-of: field-of (ptr-val q - ptr-val p) (typ-uinfo-t TYPE('b)) (typ-uinfo-t
TYPE('a))
  by (simp add: size-of-def)

  then obtain f where
    fl: field-lookup (typ-uinfo-t (TYPE('a))) f 0 = Some (typ-uinfo-t TYPE('b),
unat (ptr-val q - ptr-val p))
    using field-of-lookup-info by blast

  then obtain s where
    fl': field-lookup (typ-uinfo-t (TYPE('a))) f 0 = Some (s, unat (ptr-val q -
ptr-val p)) and
    s: export-uinfo s = typ-uinfo-t TYPE('b)
    using field-lookup-uinfo-Some-rev by blast

  from fl' have fl'': field-ti (TYPE('a)) f = Some s
  by (simp add: field-ti-def)

  from fl' have f: f  $\in$  set (field-names (typ-uinfo-t TYPE('a)) (typ-uinfo-t TYPE('b)))

  by (metis field-names-Some3(1) fl' s)

  from val-p have cgrd-p: c-guard p
  by (simp add: h-t-valid-c-guard)
  from fl'
  have ptr-val-q: ptr-val q = &(p→f)
  by (simp add: field-lvalue-def)
  then have q: q = PTR ('b) &(p→f)
  by (cases q) auto

  show ?thesis
  proof (cases qualified-padding-field-name f)
  case True
  from field-lookup-qualified-padding-field-name(1) [OF fl' True]
  have padding-tag-s: is-padding-tag s

```



```

    by (simp add: wf-padding)
  note h-upd = heap-update-field-root-conv''' [OF fl'' cgrd-p s]

  show ?thesis
  apply (simp add: q)
  by (simp add: h-upd is-padding-tag-update-ti-id [OF padding-tag-s] xmem-type-class.heap-update-id)
next
case False
  with f fl' s have f ∈ set (field-names-no-padding (typ-info-t (TYPE('a)))
    (export-uinfo (typ-info-t (TYPE('b')))))
  by (simp add: set-field-names-no-padding-all-field-names-no-padding-conv
    all-field-names-no-padding-def
    fold-typ-uinfo-t set-field-names-all-field-names-conv)
  from disj [OF this] have ptr-span (PTR('b) &(p→f)) ∩ ptr-span (PTR('c)
    &(p→g)) = {} .
  then show ?thesis
  by (simp add: q h-val-update-regions-disjoint)
qed
qed

theorem ptr-valid-heap-update-field-access-ti-disj:
  assumes val-p: d ⊨t (p::'a::xmem-type ptr)
  assumes val-q: d ⊨t (q::'b::xmem-type ptr)
  assumes subtype: TYPE('b) ≤τ TYPE('a)
  assumes disj [rule-format] :
    ∀f. f ∈ set (field-names-u (typ-uinfo-t (TYPE('a))) (typ-uinfo-t (TYPE('b'))))
  →
    (access-ti (fst (the (field-lookup (typ-info-t TYPE('a)) f 0)))
      v
      (heap-list h (size-of TYPE('b)) (ptr-val q))) =
    (access-ti (fst (the (field-lookup (typ-info-t TYPE('a)) f 0)))
      (h-val h p)
      (heap-list h (size-of TYPE('b)) (ptr-val q)))
  shows h-val (heap-update p v h) q = h-val h q
proof (cases ptr-span p ∩ ptr-span q = {})
case True
  then show ?thesis
  by (simp add: h-val-update-regions-disjoint)
next
case False

  from subtype have typ-uinfo-t TYPE('b) ≤ typ-uinfo-t TYPE('a) by (simp add:
    sub-typ-def)

  from this False valid-footprint-sub-cases [OF h-t-valid-valid-footprint [OF val-q]
    h-t-valid-valid-footprint [OF val-p]]
  have field-of: field-of (ptr-val q - ptr-val p) (typ-uinfo-t TYPE('b)) (typ-uinfo-t
    TYPE('a))
  apply (simp add: size-of-def)

```

by (*metis Int-commute le-less-trans less-irrefl*)
then obtain f where
 fl : *field-lookup* (*typ-uinfo-t* (*TYPE('a)*)) f $0 = \text{Some}$ (*typ-uinfo-t* *TYPE('b)*,
unat (*ptr-val* $q - \text{ptr-val } p$))
using *field-of-lookup-info* **by** *blast*

then obtain s where
 fl' : *field-lookup* (*typ-uinfo-t* (*TYPE('a)*)) f $0 = \text{Some}$ (s , *unat* (*ptr-val* $q -$
ptr-val p)) **and**
 s : *export-uinfo* $s = \text{typ-uinfo-t}$ *TYPE('b)*
using *field-lookup-uinfo-Some-rev* **by** *blast*

from fl **have** f : $f \in \text{set}$ (*field-names-u* (*typ-uinfo-t* *TYPE('a)*) (*typ-uinfo-t*
TYPE('b)))
by (*metis field-names-Some3(1) field-names-u-field-names-export-uinfo-conv(1)*)
 fl' s *typ-uinfo-t-def*)

from fl'
have *ptr-val-q*: *ptr-val* $q = \&(p \rightarrow f)$
by (*simp add: field-lvalue-def*)

from *field-of*
have *contained*: *ptr-span* $q \subseteq \text{ptr-span } p$
by (*metis (no-types, opaque-lifting)*)
add-diff-cancel-left' export-uinfo-size field-lvalue-def field-of-lookup-info ptr-val-q
size-of-def typ-uinfo-t-def)

from *disj [OF f] fl' have from-bytes-eq*:
(*access-ti* s v (*heap-list* h (*size-of* *TYPE('b)*) (*ptr-val* q))) =
(*access-ti* s (*h-val* h p) (*heap-list* h (*size-of* *TYPE('b)*) (*ptr-val* q))) **by** *simp*

from *ptr-val-q* **have** q : *ptr-val* $q = \text{ptr-val } p + \text{word-of-nat}$ (*unat* (*ptr-val* $q -$
ptr-val p))
by (*simp add: field-lvalue-def field-offset-def field-offset-untyped-def fl*)

from $fl' s$
have *sz-bound*: *size-of* *TYPE('b)* + *unat* (*ptr-val* $q - \text{ptr-val } p$)
 $\leq \text{length}$ (*to-bytes* v (*heap-list* h (*size-of* *TYPE('a)*) (*ptr-val* p)))
by (*metis export-uinfo-size field-lookup-offset-size heap-list-length len size-of-def*
typ-uinfo-size)

have *lheap-list*: *length* (*heap-list* h (*size-of* *TYPE('a)*) (*ptr-val* p)) = *size-of*
(*TYPE('a)*)
by *simp*
from s
have *sz-s*: *size-td* $s = \text{size-of}$ *TYPE('b)*
by (*simp add: export-size-of*)

```

from val-p
have p-no-overflow:  $\text{unat } (\text{ptr-val } p) + \text{size-of } \text{TYPE}('a) \leq \text{addr-card}$ 
  by (meson c-guard-no-wrap' h-t-valid-def)

from contained q
have q-bound:  $\text{unat } (\text{ptr-val } q - \text{ptr-val } p) + \text{size-of } \text{TYPE}('b) \leq \text{size-of } \text{TYPE}('a)$ 
  using sz-bound by auto

from heap-list-take-drop' [OF p-no-overflow q-bound]
have take-drop-eq:
  take (size-of  $\text{TYPE}('b)$ )
    (drop ( $\text{unat } (\text{ptr-val } q - \text{ptr-val } p)$ )
      (heap-list h (size-of  $\text{TYPE}('a)$ ) (ptr-val p))) =
    heap-list h (size-of  $\text{TYPE}('b)$ ) (ptr-val q)
  by (simp add: q [symmetric])

note acc-eq = mem-type-field-lookup-access-ti-take-drop [OF fl' lheap-list, sim-
plified sz-s, of v, simplified take-drop-eq]

note acc-eq' = mem-type-field-lookup-access-ti-take-drop [OF fl' lheap-list, sim-
plified sz-s, of h-val h p, simplified take-drop-eq]

have acc-eq'':
  (access-ti (typ-info-t  $\text{TYPE}('a)$ ) (h-val h p)
    (heap-list h (size-of  $\text{TYPE}('a)$ ) (ptr-val p))) = (heap-list h (size-of
 $\text{TYPE}('a)$ ) (ptr-val p))
  apply (simp add: h-val-def)
  apply (simp add: to-bytes-def [symmetric])
  by (simp add: to-bytes-from-bytes-id)

show ?thesis
  apply (simp add: heap-update-def h-val-def)
  apply (subst (1 2) q)
  apply (subst heap-list-update-list [OF sz-bound])
  apply simp
  apply (simp add: to-bytes-def)
  apply (subst acc-eq [symmetric])
  apply (subst from-bytes-eq)
  apply (subst acc-eq')
  apply (subst acc-eq'')
  apply (simp add: take-drop-eq)
  done
qed

lemma access-ti-field-names-no-padding-to-field-names-u:
  assumes val-p:  $d \models_t (p::'a::\text{xmem-type ptr})$ 
  assumes val-q:  $d \models_t (q::'b::\text{xmem-type ptr})$ 
  assumes subtype:  $\text{TYPE}('b) \leq_\tau \text{TYPE}('a)$ 

```

```

assumes disj [rule-format]:
   $\forall f. f \in \text{set} (\text{field-names-no-padding } (\text{typ-info-t } (\text{TYPE}('a))) (\text{export-uinfo } (\text{typ-info-t } (\text{TYPE}('b))))) \longrightarrow$ 
    (access-ti (fst (the (field-lookup (typ-info-t TYPE('a)) f 0)))
      v
      (heap-list h (size-of TYPE('b)) (ptr-val q)) =
      (access-ti (fst (the (field-lookup (typ-info-t TYPE('a)) f 0)))
        (h-val h p)
        (heap-list h (size-of TYPE('b)) (ptr-val q)))
assumes f-in:  $f \in \text{set} (\text{field-names-u } (\text{typ-uinfo-t } (\text{TYPE}('a))) (\text{typ-uinfo-t } (\text{TYPE}('b))))$ 
shows (access-ti (fst (the (field-lookup (typ-info-t TYPE('a)) f 0)))
  v
  (heap-list h (size-of TYPE('b)) (ptr-val q)) =
  (access-ti (fst (the (field-lookup (typ-info-t TYPE('a)) f 0)))
    (h-val h p)
    (heap-list h (size-of TYPE('b)) (ptr-val q)))
proof –
from f-in
obtain n s where
  fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n) and
  match: export-uinfo s = export-uinfo (typ-info-t (TYPE('b)))
by (smt (verit) field-names-Some2(1) fold-typ-uinfo-t set-field-names-all-field-names-conv
  set-field-names-u-all-field-names-conv wf-desc)

show ?thesis
proof (cases qualified-padding-field-name f)
  case True
    from field-lookup-qualified-padding-field-name(1) [OF fl True]
    have padding-tag-s: is-padding-tag s
      by (simp add: wf-padding)
    show ?thesis by (simp add: fl is-padding-tag-access-ti-eq [OF padding-tag-s])
  next
    case False
      with fl match field-lookup-all-field-names(1) [OF fl]
      have  $f \in \text{set} (\text{field-names-no-padding } (\text{typ-info-t } (\text{TYPE}('a))) (\text{export-uinfo } (\text{typ-info-t } (\text{TYPE}('b)))))$ 
      by (simp add: set-field-names-no-padding-all-field-names-no-padding-conv
      all-field-names-no-padding-def)

      from disj [rule-format, OF this]
      show ?thesis .
qed
qed

```

```

theorem ptr-valid-heap-update-field-access-ti-disj':
assumes val-p:  $d \models_t (p::'a::\text{xmem-type } \text{ptr})$ 
assumes val-q:  $d \models_t (q::'b::\text{xmem-type } \text{ptr})$ 

```

assumes *subtype*: $TYPE('b) \leq_{\tau} TYPE('a)$
assumes *disj* :
 $\forall f. f \in \text{set} (\text{field-names-no-padding} (\text{typ-info-t} (TYPE('a))) (\text{export-uinfo} (\text{typ-info-t} (TYPE('b)))))) \longrightarrow$
 $(\text{access-ti} (\text{fst} (\text{the} (\text{field-lookup} (\text{typ-info-t} TYPE('a)) f 0)))$
 $\quad v$
 $(\text{heap-list } h (\text{size-of } TYPE('b)) (\text{ptr-val } q))) =$
 $(\text{access-ti} (\text{fst} (\text{the} (\text{field-lookup} (\text{typ-info-t} TYPE('a)) f 0)))$
 $(\text{h-val } h p)$
 $(\text{heap-list } h (\text{size-of } TYPE('b)) (\text{ptr-val } q)))$
shows $h\text{-val} (\text{heap-update } p v h) q = h\text{-val } h q$
apply (*rule ptr-valid-heap-update-field-access-ti-disj* [*OF val-p val-q subtype*])
apply *clarify*
apply (*rule access-ti-field-names-no-padding-to-field-names-u* [*OF val-p val-q subtype disj*])
apply *assumption*
done

theorem *ptr-valid-heap-update-subtype-field-access-ti-disj*:

assumes *val-p*: $d \models_t (p::'a::\text{xmem-type } ptr)$
assumes *val-q*: $d \models_t (q::'b::\text{xmem-type } ptr)$
assumes *subtype-b-a*: $TYPE('b) \leq_{\tau} TYPE('a)$
assumes *fl-g*: $\text{field-lookup} (\text{typ-uinfo-t } TYPE('b)) g 0 = \text{Some} (\text{typ-uinfo-t } TYPE('c::\text{xmem-type}), n)$
assumes *disj* [*rule-format*] :
 $\forall f. f \in \text{set} (\text{field-names-u} (\text{typ-uinfo-t} (TYPE('a))) (\text{typ-uinfo-t} (TYPE('b))))$
 \longrightarrow
 $(\text{access-ti} (\text{fst} (\text{the} (\text{field-lookup} (\text{typ-info-t} TYPE('a)) f 0)))$
 $\quad v$
 $(\text{heap-list } h (\text{size-of } TYPE('b)) (\text{ptr-val } q))) =$
 $(\text{access-ti} (\text{fst} (\text{the} (\text{field-lookup} (\text{typ-info-t} TYPE('a)) f 0)))$
 $(\text{h-val } h p)$
 $(\text{heap-list } h (\text{size-of } TYPE('b)) (\text{ptr-val } q)))$
shows $h\text{-val} (\text{heap-update } p v h) (PTR('c) (\&(q \rightarrow g))) = h\text{-val } h (PTR('c) (\&(q \rightarrow g)))$
proof (*cases ptr-span p \cap ptr-span q = {}*)
case *True*
from *fl-g* **have** $\text{ptr-span} (PTR('c) (\&(q \rightarrow g))) \subseteq \text{ptr-span } q$
by (*metis export-uinfo-size field-lookup-uinfo-Some-rev field-tag-sub ptr-val.ptr-val-def size-of-tag*)
with *True* **have** $\text{ptr-span } p \cap \text{ptr-span} (PTR('c) (\&(q \rightarrow g))) = \{\}$ **by** *blast*
then show *?thesis*
by (*simp add: h-val-update-regions-disjoint*)
next
case *False*
from *subtype-b-a* **have** $\text{typ-uinfo-t } TYPE('b) \leq \text{typ-uinfo-t } TYPE('a)$ **by** (*simp add: sub-typ-def*)

from *this False valid-footprint-sub-cases* [*OF h-t-valid-valid-footprint* [*OF val-q*]

h-t-valid-valid-footprint [*OF val-p*]
have *field-of*: *field-of* (*ptr-val* *q* – *ptr-val* *p*) (*typ-uinfo-t* *TYPE('b)*) (*typ-uinfo-t* *TYPE('a)*)
apply (*simp add: size-of-def*)
by (*metis Int-commute le-less-trans less-irrefl*)
then obtain *f* **where**
fl: *field-lookup* (*typ-uinfo-t* (*TYPE('a)*)) *f* 0 = *Some* (*typ-uinfo-t* *TYPE('b)*,
unat (*ptr-val* *q* – *ptr-val* *p*))
using *field-of-lookup-info* **by** *blast*

then obtain *s* **where**
fl': *field-lookup* (*typ-uinfo-t* (*TYPE('a)*)) *f* 0 = *Some* (*s*, *unat* (*ptr-val* *q* –
ptr-val *p*)) **and**
s: *export-uinfo* *s* = *typ-uinfo-t* *TYPE('b)*
using *field-lookup-uinfo-Some-rev* **by** *blast*

from *fl* **have** *f*: *f* ∈ *set* (*field-names-u* (*typ-uinfo-t* *TYPE('a)*) (*typ-uinfo-t* *TYPE('b)*))
by (*metis field-names-Some3(1) field-names-u-field-names-export-uinfo-conv(1)*
fl' s typ-uinfo-t-def)

from *fl'*
have *ptr-val-q*: *ptr-val* *q* = &(p→f)
by (*simp add: field-lvalue-def*)

from *field-of*
have *contained*: *ptr-span* *q* ⊆ *ptr-span* *p*
by (*metis (no-types, opaque-lifting)*
add-diff-cancel-left' export-uinfo-size field-lvalue-def field-of-lookup-info ptr-val-q
size-of-def typ-uinfo-t-def)

from *fl-g* **have** *contained'*: *ptr-span* (*PTR('c)* (&(q→g))) ⊆ *ptr-span* *q*
by (*metis export-uinfo-size field-lookup-uinfo-Some-rev field-tag-sub ptr-val.ptr-val-def*
size-of-tag)

from *disj* [*OF f*] *fl'* **have** *from-bytes-eq*:
(*access-ti* *s* *v* (*heap-list* *h* (*size-of* *TYPE('b)*) (*ptr-val* *q*))) =
(*access-ti* *s* (*h-val* *h* *p*) (*heap-list* *h* (*size-of* *TYPE('b)*) (*ptr-val* *q*))) **by** *simp*

from *ptr-val-q* **have** *q*: *ptr-val* *q* = *ptr-val* *p* + *word-of-nat* (*unat* (*ptr-val* *q* –
ptr-val *p*))
by (*simp add: field-lvalue-def field-offset-def field-offset-untyped-def fl*)

from *fl-g* **have** *q-g*: &(q→g) = *ptr-val* *q* + *word-of-nat* *n*
by (*simp add: field-lvalue-def field-offset-def field-offset-untyped-def*)
with *q* **have** *q-g'*: &(q→g) = *ptr-val* *p* + *word-of-nat* (*unat* (*ptr-val* *q* – *ptr-val*
p) + *n*)
by *simp*

```

from fl' s
have sz-bound: size-of TYPE('b) + unat (ptr-val q - ptr-val p)
   $\leq$  length (to-bytes v (heap-list h (size-of TYPE('a)) (ptr-val p)))
  by (metis export-uinfo-size field-lookup-offset-size heap-list-length len size-of-def
typ-uinfo-size)

from fl-g have subtype-c-b: TYPE('c::xmem-type)  $\leq_{\tau}$  TYPE('b)
  by (meson sub-typ-def td-set-field-lookupD typ-tag-le-def)

from contained' subtype-c-b fl-g
have sz-c-n: size-of TYPE('c) + n  $\leq$  size-of TYPE('b)
  by (metis size-of-def td-set-field-lookupD td-set-offset-size typ-uinfo-size)

from sz-c-n sz-bound
have sz-bound': size-of TYPE('c) + (unat (ptr-val q - ptr-val p) + n)
   $\leq$  length (to-bytes v (heap-list h (size-of TYPE('a)) (ptr-val p)))
  by simp
have lheap-list: length (heap-list h (size-of TYPE('a)) (ptr-val p)) = size-of
(TYPE('a))
  by simp
from s
have sz-s: size-td s = size-of TYPE('b)
  by (simp add: export-size-of )

from val-p
have p-no-overflow: unat (ptr-val p) + size-of TYPE('a)  $\leq$  addr-card
  by (meson c-guard-no-wrap' h-t-valid-def)

from contained q
have q-bound: unat (ptr-val q - ptr-val p) + size-of TYPE('b)  $\leq$  size-of TYPE('a)
  using sz-bound by auto

from fl-g s obtain s' where
  fl-s-g: field-lookup s g 0 = Some (s', n) and
  s': export-uinfo s' = typ-uinfo-t TYPE('c)
  by (metis CTypes.field-lookup-export-uinfo-Some-rev)

from s'
have sz-s': size-td s' = size-of TYPE('c)
  by (simp add: export-size-of )

from sz-c-n q-bound
have q-bound': unat (ptr-val q - ptr-val p) + n + size-of TYPE('c)  $\leq$  size-of
TYPE('a)
  by simp
have take-drop-eq:
  (take (size-of TYPE('c))
  (drop (unat (ptr-val q - ptr-val p) + n)
  (heap-list h (size-of TYPE('a)) (ptr-val p)))) =

```

```

heap-list h (size-of TYPE('c)) (&(q→g))
  apply (subst q-g')
  apply (rule heap-list-take-drop' [OF p-no-overflow q-bound'])
done

from fl-s-g field-lookup-prefix-Some''(1) [rule-format, OF fl' wf-desc, where g=g
]
have fl-f-g: field-lookup (typ-info-t TYPE('a)) (f @ g) 0 = Some (s', unat (ptr-val
q - ptr-val p) + n)
  by (simp add: field-lookup-offset)

  note acc-eq = mem-type-field-lookup-access-ti-take-drop [OF fl-f-g lheap-list,
simplified sz-s', of v, simplified take-drop-eq]
  note acc-eq' = mem-type-field-lookup-access-ti-take-drop [OF fl-f-g lheap-list, sim-
plified sz-s', of h-val h p, simplified take-drop-eq]

from s fl' have wf-fd-s: wf-fd s by (meson wf-fd wf-fd-field-lookupD)
from s fl' have wf-desc-s: wf-desc s by (meson field-lookup-wf-desc-pres(1)
wf-desc)
from s fl' have wf-sz-s: wf-size-desc s by (meson field-lookup-wf-size-desc-pres(1)
wf-size-desc)

from s have lheap-list': length (heap-list h (size-of TYPE('b)) (ptr-val q)) =
size-td s
  using sz-s by simp

from from-bytes-eq
  field-lookup-access-ti-take-drop [OF fl-s-g wf-fd-s wf-desc-s wf-sz-s lheap-list', of
v]
  field-lookup-access-ti-take-drop [OF fl-s-g wf-fd-s wf-desc-s wf-sz-s lheap-list', of
h-val h p]
have from-bytes-eq':
  (access-ti s' v (heap-list h (size-of TYPE('c)) (&(q→g)))) =
  (access-ti s' (h-val h p) (heap-list h (size-of TYPE('c)) (&(q→g))))
by (metis add-leD2 add-le-imp-le-diff drop-heap-list-le q-g sz-c-n sz-s' take-heap-list-le)

have acc-eq'':
  (access-ti (typ-info-t TYPE('a)) (h-val h p)
    (heap-list h (size-of TYPE('a)) (ptr-val p))) = (heap-list h (size-of
TYPE('a)) (ptr-val p))
  apply (simp add: h-val-def)
  apply (simp add: to-bytes-def [symmetric])
  by (simp add: to-bytes-from-bytes-id)

show ?thesis
  apply (simp add: heap-update-def h-val-def)
  apply (subst (1 2) q-g')

```


apply (*subst heap-list-update-list* [*OF sz-bound'*])
apply *simp*
apply (*simp add: to-bytes-def*)
apply (*subst acc-eq* [*symmetric*])

apply (*subst from-bytes-eq'*)
apply (*subst acc-eq'*)
apply (*subst acc-eq''*)
apply (*simp add: take-drop-eq q-g'*)
done

qed

corollary *ptr-valid-heap-update-subtype-field-access-ti-disj'*:

assumes *val-p*: $d \models_t (p::'a::xmem\text{-}type\ ptr)$
assumes *val-q*: $d \models_t (q::'b::xmem\text{-}type\ ptr)$
assumes *subtype-b-a*: $TYPE('b) \leq_\tau TYPE('a)$
assumes *fl-g*: *field-lookup* (*typ-info-t* $TYPE('b)$) *g* $0 = Some (s, n)$
assumes *match*: *export-uinfo* *s* = *typ-uinfo-t* $TYPE('c::xmem\text{-}type)$
assumes *disj* :
 $\forall f. f \in set (field\text{-}names\text{-}no\text{-}padding (typ\text{-}info\text{-}t (TYPE('a))) (export\text{-}uinfo (typ\text{-}info\text{-}t (TYPE('b)))))) \longrightarrow$
 $(access\text{-}ti (fst (the (field\text{-}lookup (typ\text{-}info\text{-}t TYPE('a)) f 0)))$
 $\quad v$
 $(heap\text{-}list\ h (size\text{-}of\ TYPE('b)) (ptr\text{-}val\ q))) =$
 $(access\text{-}ti (fst (the (field\text{-}lookup (typ\text{-}info\text{-}t TYPE('a)) f 0)))$
 $(h\text{-}val\ h\ p)$
 $(heap\text{-}list\ h (size\text{-}of\ TYPE('b)) (ptr\text{-}val\ q)))$
shows $h\text{-}val (heap\text{-}update\ p\ v\ h) (PTR('c) (\&(q \rightarrow g))) = h\text{-}val\ h (PTR('c) (\&(q \rightarrow g)))$
apply (*rule ptr-valid-heap-update-subtype-field-access-ti-disj* [*OF val-p val-q subtype-b-a*])
apply (*simp add: match* [*symmetric*] *field-lookup-typ-uinfo-t-Some* [*OF fl-g*])
apply *clarify*
apply (*rule access-ti-field-names-no-padding-to-field-names-u* [*OF val-p val-q subtype-b-a disj*])
apply *assumption*
done

lemma *array-ptr-index-field-lvalue-conv*:

fixes *p*: (*a*::*c-type* [*b*::*finite*]) *ptr*
assumes *i-bound*: $i < CARD('b)$
shows (*array-ptr-index* *p* *False* *i*) = ($PTR('a) \&(p \rightarrow [replicate\ i\ CHR\ '1'])$)
proof –
from *field-lookup-array* [*OF i-bound, of 0, simplified*]
have *field-lookup* (*typ-info-t* $TYPE('a['b])$) [*replicate i CHR '1'*] $0 =$
 $Some (adjust\text{-}ti (typ\text{-}info\text{-}t\ TYPE('a)) (\lambda x. x.[i]) (\lambda x f. Arrays.update\ f\ i$
 $x),$
 $i * size\text{-}of\ TYPE('a)) .$

from *field-lookup-offset-eq* [OF this]
have *field-offset* $TYPE('a['b])$ [replicate i CHR "1"] = $i * \text{size-of } TYPE('a)$.
then show ?thesis
by (*simp add: field-lvalue-def array-ptr-index-def ptr-add-def*)
qed

lemma *field-lvalue-array-index*:
fixes $p::('a::c\text{-type}['b::finite])\ ptr$
shows $i < CARD('b) \implies \&(p \rightarrow [\text{replicate } i \text{ CHR "1"}]) =$
 $\text{ptr-val } (PTR\text{-COERCE}('a['b] \rightarrow 'a) p +_p \text{int } i)$
using *array-ptr-index-field-lvalue-conv* [where $i=i$ and $'b='b$ and $'a='a$ and
 $p=p$]
by (*simp add: array-ptr-index-def*)

lemma *field-lvalue-array-index'*:
fixes $p::('a::c\text{-type}['b::finite])\ ptr$
shows $i < CARD('b) \implies$
 $PTR('a) \&(p \rightarrow [\text{replicate } i \text{ CHR "1"}]) =$
 $(PTR\text{-COERCE}('a['b] \rightarrow 'a) p +_p \text{int } i)$
by (*simp add: field-lvalue-array-index*)

thm *field-lvalue-append*

corollary *ptr-valid-heap-update-subtype-array-access-ti-disj*:

assumes $val\text{-}p: d \models_t (p::'a::xmem\text{-type } ptr)$
assumes $val\text{-}q: d \models_t (q::'b::array\text{-outer-max-size}['c::array\text{-max-count}])\ ptr$
assumes *subtype-b-a*: $TYPE('b['c]) \leq_\tau TYPE('a)$
assumes *i-bound*: $i < CARD('c)$
assumes *disj* :
 $\forall f. f \in \text{set } (field\text{-names-no-padding } (typ\text{-info-t } (TYPE('a))) (export\text{-uinfo } (typ\text{-info-t } (TYPE('b['c]))))) \longrightarrow$
 $(access\text{-ti } (fst (the (field\text{-lookup } (typ\text{-info-t } TYPE('a)) f 0)))$
 $\quad v$
 $(heap\text{-list } h (size\text{-of } TYPE('b['c])) (ptr\text{-val } q))) =$
 $(access\text{-ti } (fst (the (field\text{-lookup } (typ\text{-info-t } TYPE('a)) f 0)))$
 $(h\text{-val } h p)$
 $(heap\text{-list } h (size\text{-of } TYPE('b['c])) (ptr\text{-val } q)))$

shows $h\text{-val } (heap\text{-update } p v h) (array\text{-ptr-index } q \text{ False } i) = h\text{-val } h (array\text{-ptr-index } q \text{ False } i)$

proof –

note $fl = \text{field-lookup-array}$ [OF *i-bound*, of 0]

have *export-uinfo*

$(adjust\text{-ti } (typ\text{-info-t } TYPE('b)) (\lambda x. x.[i]) (\lambda x f. Arrays.update (f::'b['c]) i x)) = typ\text{-uinfo-t } TYPE('b)$

using *i-bound*

by (*simp*)

from *ptr-valid-heap-update-subtype-field-access-ti-disj'* [OF *val-p val-q subtype-b-a fl this disj*]

```

have h-val (heap-update p v h) (PTR('b) &(q→[replicate i CHR "1"])) =
  h-val h (PTR('b) &(q→[replicate i CHR "1"])).
then show ?thesis
  by (simp add: array-ptr-index-field-lvalue-conv i-bound)
qed

theorem ptr-valid-heap-update-subtype-field-access-ti-indep:
  assumes val-p: d ⊨t (p::'a::xmem-type ptr)
  assumes val-q: d ⊨t (q::'a::xmem-type ptr)
  assumes fl-f: field-lookup (typ-info-t TYPE('a)) f 0 = Some (t, n)
  assumes match: export-uinfo t = typ-uinfo-t (TYPE('b::xmem-type))
  assumes disj:
    (access-ti t v (heap-list h (size-of TYPE('b)) (ptr-val q))) =
    (access-ti t (h-val h p) (heap-list h (size-of TYPE('b)) (ptr-val q)))
  shows h-val (heap-update p v h) (PTR('b) (&(q→f))) = h-val h (PTR('b) (&(q→f)))
  proof (cases ptr-span p ∩ ptr-span q = {})
    case True
      from fl-f have ptr-span (PTR('b) (&(q→f))) ⊆ ptr-span q
        by (metis export-uinfo-size field-tag-sub match ptr-val.ptr-val-def size-of-def
          typ-uinfo-t-def)
      with True have ptr-span p ∩ ptr-span (PTR('b) (&(q→f))) = {} by blast
      then show ?thesis
        by (simp add: h-val-update-regions-disjoint)
    next
      case False
        from val-p val-q False have pq: p = q
          by (meson ptr-valid-same-type-cases)

        from fl-f have fl-f': field-ti TYPE('a) f = Some t
          using field-lookup-field-ti by blast

        from h-val-field-from-bytes' [OF fl-f' match [simplified typ-uinfo-t-def], of h p ]
        have eq1: h-val h (PTR('b) &(p→f)) = from-bytes (access-ti0 t (h-val h p)) .

        from h-val-field-from-bytes' [OF fl-f' match [simplified typ-uinfo-t-def], of heap-update
          p v h p]
        have eq2: h-val (heap-update p v h) (PTR('b) &(p→f)) = from-bytes (access-ti0
          t (h-val (heap-update p v h) p)).

        thm field-lookup-access-ti-to-bytes-field-conv [OF fl-f match ]

        have eq-upto1: eq-upto-padding (export-uinfo t)
          (access-ti t v (replicate (size-td t) 0))
          ((access-ti t v (replicate (size-td t) 0)))
          by (metis access-ti0 eq-upto-padding-refl export-uinfo-size fd-cons fd-cons-length-p
            fd-consistentD fl-f)

        have eq-pad-zeros: eq-padding (export-uinfo t) (replicate (size-td t) 0) (replicate
          (size-td t) 0)

```

```

    by simp

  from field-lookup-access-ti-eq-upto-padding [OF fl-f match, of (replicate (size-td
t) 0)]
  have eq-upto-padding (export-uinfo t)
    (access-ti t v (replicate (size-td t) 0))
    (access-ti t v (heap-list h (size-of TYPE('b)) (ptr-val q)))
  by simp

  moreover

  have length (heap-list h (size-of TYPE('b)) (ptr-val q)) = size-td t
  by (simp add: export-size-of match)
  from field-lookup-access-ti-eq-upto-padding [OF fl-f match this]
  have eq-upto-padding (export-uinfo t)
    (access-ti t (h-val h p) (heap-list h (size-of TYPE('b)) (ptr-val q)))
    (access-ti t (h-val h p) (replicate (size-td t) 0))
  by simp

  ultimately
  have eq-upto: eq-upto-padding (export-uinfo t)
    (access-ti t v (replicate (size-td t) 0))
    (access-ti t (h-val h p) (replicate (size-td t) 0))
  using disj
  by (simp add: eq-upto-padding-def)

  from field-lookup-access-ti-eq-padding-value[OF fl-f match, of (replicate (size-td
t) 0)]
  have eq-padding (export-uinfo t)
    (access-ti t v (replicate (size-td t) 0))
    (access-ti t (h-val h p) (replicate (size-td t) 0))
  by simp

  from this eq-upto
  have acc-eq: access-ti t v (replicate (size-td t) 0) =
    access-ti t (h-val h p) (replicate (size-td t) 0)
  by (rule eq-padding-eq-upto-padding-eq)

  show ?thesis
  apply (simp add: pq [symmetric] eq1 eq2)
  apply (simp add: access-ti0-def h-val-heap-update)

  apply (simp add: from-bytes-def)
  apply (simp add: acc-eq)
  done
qed

lemmas clift-field' = clift-field [simplified fold-typ-uinfo-t]

```

```

lemma field-lookup-cons-Some:
  fixes t::('a, 'b) typ-desc
  and st::('a, 'b) typ-struct
  and ts::('a, 'b) typ-tuple list
  and x::('a, 'b) typ-tuple
  shows
    wf-desc t  $\implies$  field-lookup t (f#fs) n = Some (s, m)  $\implies$ 
       $\exists w k. \text{field-lookup } t [f] n = \text{Some } (w, k) \wedge \text{field-lookup } w \text{ fs } k = \text{Some } (s, m)$ 
    wf-desc-struct st  $\implies$  field-lookup-struct st (f # fs) n = Some (s, m)  $\implies$ 
       $\exists w k. \text{field-lookup-struct } st [f] n = \text{Some } (w, k) \wedge \text{field-lookup } w \text{ fs } k = \text{Some } (s, m)$ 
    wf-desc-list ts  $\implies$  field-lookup-list ts (f#fs) n = Some (s, m)  $\implies$ 
       $\exists w k. \text{field-lookup-list } ts [f] n = \text{Some } (w, k) \wedge \text{field-lookup } w \text{ fs } k = \text{Some } (s, m)$ 
    wf-desc-tuple x  $\implies$  field-lookup-tuple x (f # fs) n = Some (s, m)  $\implies$ 
       $f = \text{dt-snd } x \wedge \text{field-lookup } (\text{dt-fst } x) \text{ fs } n = \text{Some } (s, m)$ 
  proof (induct t and st and ts and x arbitrary: n s m f fs and n s m f fs and n s m f fs and n s m f fs)
    case (TypDesc nat typ-struct list)
      then show ?case by auto
    next
      case (TypScalar nat1 nat2 a)
        then show ?case by auto
    next
      case (TypAggregate list)
        then show ?case by auto

  next
    case Nil-typ-desc
      then show ?case by auto
  next
    case (Cons-typ-desc t ts)
      thus ?case apply clarsimp
      by (metis Cons-typ-desc.prem1 append-Cons append-Nil field-lookup-list-None field-lookup-list-Some field-lookup-prefix-Some''(3) fl4 not-Some-eq-tuple)

  next
    case (DTuple-typ-desc typ-desc list b)
      then show ?case by (auto split: if-split-asm)
  qed

lemma field-offset-append:
  assumes f: field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (typ-uinfo-t TYPE('b), n)
  assumes g: field-lookup (typ-uinfo-t TYPE('b)) g 0 = Some x
  shows field-offset TYPE('a::mem-type) (f @ g) =

```

$field_offset (TYPE('a::mem-type)) f + field_offset (TYPE('b::mem-type)) g$
proof –
from $field_lookup_prefix_Some''(1)$ [rule-format , OF f]
have $field_lookup (typ_uinfo_t TYPE('a)) (f @ g) 0 = field_lookup (typ_uinfo_t TYPE('b)) g n$
by *simp*
with $f g$ $field_lookup_offset_non_zero$ **show** *?thesis*
apply (*simp add: field_offset_def field_offset_untyped_def*)
by (*smt (verit, ccfv-threshold) eq_snd_iff field_lookup_offset_option.sel*)
qed

lemma $field_lookup_struct_offset_shift$:
 $field_lookup_struct st f m = Some (s, k) \implies field_lookup_struct st f n = Some (s, k + n - m)$
by (*metis (no-types, lifting) add.commute field_lookup_offset'(2) field_lookup_offset-le(2)*)
 $le_add_diff_inverse ordered_cancel_comm_monoid_diff_class.diff_add_assoc2$

lemma $field_lookup_list_offset_shift$:
 $field_lookup_list ts f m = Some (s, k) \implies field_lookup_list ts f n = Some (s, k + n - m)$
by (*metis Nat.add_diff_assoc2 add.commute field_lookup_list_offset field_lookup_list_offset2 field_lookup_offset-le(3)*)

lemma $field_lookup_tuple_offset_shift$:
 $field_lookup_tuple x f m = Some (s, k) \implies field_lookup_tuple x f n = Some (s, k + n - m)$
by (*metis (no-types, lifting) add.commute field_lookup_offset'(4) field_lookup_offset-le(4) ordered_cancel_comm_monoid_diff_class.add_diff_assoc2 ordered_cancel_comm_monoid_diff_class.add_diff_inverse*)

lemma $h_val_field_update$:
fixes $p::'a::mem_type$ ptr
and $q::'a::mem_type$ ptr
and $v::'b::mem_type$
assumes $fl: field_ti TYPE('a) f = Some t$
assumes $match: export_uinfo t = typ_uinfo_t TYPE('b)$
and $valid_p: hrs_htd h \models_t p$
and $valid_q: hrs_htd h \models_t q$
shows $(h_val (heap_update (Ptr \&(p \rightarrow f)) v (hrs_mem h)) q) =$
 $(if p = q then field_update (field_desc t) (to_bytes_p v) (h_val (hrs_mem h) p) else$
 $(h_val (hrs_mem h) q))$
proof –
from $clift_Some_eq_valid valid_p$ **obtain** z **where** $z: clift h p = Some z$ **by** *metis*
from $clift_field_update$ [OF fl match [simplified typ_uinfo_t_def] *this*]
show *?thesis*
apply *simp*

by (smt (verit) fun-upd-apply h-val-clift' hrs-htd-def hrs-mem-update lift-t-if prod.collapse valid-q z)

qed

theorem *root-ptr-valid-field-disj-type-h-val-heap-update:*

assumes *valid-p*: root-ptr-valid (hrs-htd h) (p::'b::mem-type ptr)
 assumes *valid-q*: root-ptr-valid (hrs-htd h) (q::'a::mem-type ptr)
 assumes *neq*: typ-uinfo-t TYPE('a) \neq typ-uinfo-t TYPE('b)
 assumes *f*: field-lookup (typ-info-t (TYPE('a))) f 0 = Some (s, n)
 assumes *match*: export-uinfo s = typ-uinfo-t (TYPE('c'))
 shows *h-val* (heap-update (PTR('c::mem-type) &(q→f)) v (hrs-mem h)) p =
 h-val (hrs-mem h) p

proof –

from *root-ptr-valid-type-neq-disjoint* [OF *valid-q valid-p neq*]

have *disj*: ptr-span q \cap ptr-span p = {} .

from *export-uinfo-size* [of *s*] have *size-td* s = *size-td* (typ-uinfo-t TYPE('c)) by
 (*simp add: match*)

moreover have *size-td* (typ-info-t TYPE('c)) = *size-td* (typ-uinfo-t (TYPE('c)))
 by (*simp add: typ-uinfo-t-def*)

ultimately have *sz*: *size-td* s = *size-of* (TYPE('c'))

unfolding *size-of-def*

by *metis*

from *field-tag-sub* [OF *f*] have {&(q→f)..+*size-td* s} \subseteq ptr-span q .

with *disj sz*

have ptr-span (PTR('c::mem-type) &(q→f)) \cap ptr-span p = {}

apply (*simp*)

by *blast*

from *h-val-update-regions-disjoint* [OF *this*]

show *?thesis* .

qed

lemma *ptr-empty-qualified-field-name-conv:*

fixes *p*::'a::c-type ptr

shows *p* = PTR ('a) &(p→[])

by (*simp add: field-lvalue-def*)

theorem *root-ptr-valid-field-disj-h-val-heap-update-eq:*

assumes *valid-p*: root-ptr-valid d (p::'a::mem-type ptr)

assumes *valid-q*: root-ptr-valid d (q::'b::mem-type ptr)

assumes *f*: field-lookup (typ-info-t (TYPE('a))) f 0 = Some (t, n)

assumes *g*: field-lookup (typ-info-t (TYPE('b'))) g 0 = Some (s, m)

assumes *match-t*: export-uinfo t = typ-uinfo-t (TYPE('c'))

```

assumes match-s: export-uinfo s = typ-uinfo-t (TYPE('d))
assumes disj: typ-uinfo-t (TYPE('a)) = (typ-uinfo-t (TYPE('b)))  $\implies$ 
      ptr-val p = ptr-val q  $\implies$ 
      ptr-span (PTR('c::mem-type) &(p→f))  $\cap$  ptr-span (PTR('d::mem-type)
&(q→g)) = {}
shows h-val (heap-update (PTR('c::mem-type) &(p→f)) v h) (PTR('d::mem-type)
&(q→g)) =
      h-val h (PTR('d::mem-type) &(q→g))
proof –
  from f match-t
  have sub-f-p: ptr-span (PTR('c::mem-type) &(p→f))  $\subseteq$  ptr-span p
    using export-size-of field-tag-sub by fastforce
  from g match-s
  have sub-g-q: ptr-span (PTR('d::mem-type) &(q→g))  $\subseteq$  ptr-span q
    using export-size-of field-tag-sub by fastforce
  show ?thesis
  proof (cases typ-uinfo-t (TYPE('a)) = typ-uinfo-t (TYPE('b)))
    case True
      note typ-eq = this
      show ?thesis
      proof (cases ptr-val p = ptr-val q)
        case True
          from disj [OF typ-eq True]
          have ptr-span (PTR('c::mem-type) &(p→f))  $\cap$  ptr-span (PTR('d::mem-type)
&(q→g)) = {} by blast
          then show ?thesis
            by (simp add: h-val-update-regions-disjoint)
        next
          case False
          from root-ptr-valid-neq-disjoint [OF valid-p valid-q False]
          have ptr-span p  $\cap$  ptr-span q = {}.
          with sub-f-p sub-g-q
          have ptr-span (PTR('c::mem-type) &(p→f))  $\cap$  ptr-span (PTR('d::mem-type)
&(q→g)) = {} by blast
          then show ?thesis
            by (simp add: h-val-update-regions-disjoint)
        qed
      next
        case False
        from root-ptr-valid-type-neq-disjoint [OF valid-p valid-q False]
        have ptr-span p  $\cap$  ptr-span q = {}.
        with sub-f-p sub-g-q
        have ptr-span (PTR('c::mem-type) &(p→f))  $\cap$  ptr-span (PTR('d::mem-type)
&(q→g)) = {} by blast
        then show ?thesis
          by (simp add: h-val-update-regions-disjoint)
        qed
      qed

```


definition *PTR-MATCH* $p\ q = (p = q)$

lemma *PTR-MATCH-field*:

PTR-MATCH (*PTR*('a::c-type) &(p→f)) (*PTR*('a) &(p→f))
by (*simp add: PTR-MATCH-def*)

lemma *PTR-MATCH-dummy*:

PTR-MATCH (p::'a::c-type ptr) (*PTR*('a) &(p→[]))
by (*simp add: PTR-MATCH-def*)

theorem *root-ptr-valid-field-disj-h-val-heap-update-eq'*:

assumes p' : *PTR-MATCH* p' (*PTR*('c) &(p→f))
assumes q' : *PTR-MATCH* q' (*PTR*('d) &(q→g))
assumes *valid-p*: *root-ptr-valid* d (p::'a::mem-type ptr)
assumes *valid-q*: *root-ptr-valid* d (q::'b::mem-type ptr)
assumes *disj*: *typ-uinfo-t* (*TYPE*('a)) = (*typ-uinfo-t* (*TYPE*('b))) \implies
 ptr-val $p = \text{ptr-val } q \implies$
 ptr-span (*PTR*('c) &(p→f)) \cap *ptr-span* (*PTR*('d) &(q→g)) = {}
assumes *match-s*: *export-uinfo* $s = \text{typ-uinfo-t } (\text{TYPE}('d::\text{mem-type}))$
assumes *match-t*: *export-uinfo* $t = \text{typ-uinfo-t } (\text{TYPE}('c::\text{mem-type}))$
assumes g : *field-lookup* (*typ-info-t* (*TYPE*('b))) $g\ 0 = \text{Some } (s, n)$
assumes f : *field-lookup* (*typ-info-t* (*TYPE*('a))) $f\ 0 = \text{Some } (t, m)$
shows *h-val* (*heap-update* $p'\ v\ h$) $q' = \text{h-val } h\ q'$
using *root-ptr-valid-field-disj-h-val-heap-update-eq* [*OF valid-p valid-q f g match-t match-s disj*] $p'\ q'$
by (*simp add: PTR-MATCH-def*)

lemma *field-lookup-single-field-disj'*:

fixes $t::('a, 'b)$ *typ-desc*
and $st::('a, 'b)$ *typ-struct*
and $ts::('a, 'b)$ *typ-tuple list*
and $x::('a, 'b)$ *typ-tuple*
shows
field-lookup $t\ [f]\ n = \text{Some } (s, n + m) \implies$
 field-lookup $t\ [g]\ n = \text{Some } (s', n + k) \implies f \neq g \implies$
 $\{m \dots m + \text{size-td } s\} \cap \{k \dots k + \text{size-td } s'\} = \{\}$
field-lookup-struct $st\ [f]\ n = \text{Some } (s, n + m) \implies$
 field-lookup-struct $st\ [g]\ n = \text{Some } (s', n + k) \implies f \neq g \implies$
 $\{m \dots m + \text{size-td } s\} \cap \{k \dots k + \text{size-td } s'\} = \{\}$
field-lookup-list $ts\ [f]\ n = \text{Some } (s, n + m) \implies$
 field-lookup-list $ts\ [g]\ n = \text{Some } (s', n + k) \implies f \neq g \implies$
 $\{m \dots m + \text{size-td } s\} \cap \{k \dots k + \text{size-td } s'\} = \{\}$
field-lookup-tuple $x\ [f]\ n = \text{Some } (s, n + m) \implies$
 field-lookup-tuple $x\ [g]\ n = \text{Some } (s', n + k) \implies f \neq g \implies$
 $\{m \dots m + \text{size-td } s\} \cap \{k \dots k + \text{size-td } s'\} = \{\}$

```

proof (induct t and st and ts and x arbitrary: n s s' m f g k and n s s' m f g k
and n s s' m f g k and n s s' m f g k)
  case (TypDesc nat typ-struct list)
then show ?case by auto
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
obtain d nm y where x: x = DTuple d nm y by (cases x) auto
show ?case
proof (cases nm=f)
  case True
  from True Cons-typ-desc.prem obtain
    fl-g: field-lookup-list fs [g] (n + size-td s) = Some (s', n + k) and
    neq: f ≠ g and
    d-s: d = s and m: m = 0
  by (auto simp add: x split: if-split-asm)
from field-lookup-offset-le(3) [OF fl-g] have n + size-td s ≤ n + k .
hence size-td s ≤ k by simp
then show ?thesis by (simp add: m)
next
  case False
note not-nm-f = this
show ?thesis
proof (cases nm = g)
  case True
  from True Cons-typ-desc.prem obtain
    fl-f: field-lookup-list fs [f] (n + size-td s') = Some (s, n + m) and
    neq: f ≠ g
    d = s' and
    k: k = 0
  by (auto simp add: x split: if-split-asm)
from field-lookup-offset-le(3) [OF fl-f] have n + size-td s' ≤ n + m .
hence size-td s' ≤ m by simp
then show ?thesis by (simp add: k)
next
  case False
  from False not-nm-f Cons-typ-desc.prem obtain
    fl-f: field-lookup-list fs [f] (n + size-td d) = Some (s, n + m) and
    fl-g: field-lookup-list fs [g] (n + size-td d) = Some (s', n + k) and
    neq: f ≠ g

```

```

    by (auto simp add: x split: if-split-asm)
  from field-lookup-list-offset-shift [OF fl-f, of n, simplified]
  have fl-f': field-lookup-list fs [f] n = Some (s, m + n - size-td d) .
  from field-lookup-offset-le(3)[OF fl-f]
  have bounds1: n + size-td d ≤ n + m.
  then have eq1: m + n - size-td d = n + (m - size-td d)
    by simp

  from field-lookup-list-offset-shift [OF fl-g, of n, simplified]
  have fl-g': field-lookup-list fs [g] n = Some (s', k + n - size-td d) .
  from field-lookup-offset-le(3)[OF fl-g]
  have bounds2: n + size-td d ≤ n + k.
  then have eq2: k + n - size-td d = n + (k - size-td d)
    by simp
  from Cons-typ-desc.hyps(2) [OF fl-f' [simplified eq1] fl-g' [simplified eq2]
neq ]
  have {m - size-td d..<m - size-td d + size-td s} ∩
    {k - size-td d..<k - size-td d + size-td s'} = {} .
  then show ?thesis using bounds1 bounds2 by auto
qed
qed

next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by (auto split: option.splits if-split-asm)
qed

lemma field-lookup-single-field-disj:
  assumes fl-f: field-lookup t [f] 0 = Some (s, m)
  assumes fl-g: field-lookup t [g] 0 = Some (s', k)
  assumes neq: f ≠ g
  shows {m ..<m + size-td s} ∩ {k ..<k + size-td s'} = {}
  using field-lookup-single-field-disj'(1) [where n=0, simplified, OF fl-f fl-g neq ]
  by simp

lemma field-lookup-non-prefix-disj:
  assumes wf: wf-desc t
  assumes f: field-lookup t f 0 = Some (tf, n)
  assumes g: field-lookup t g 0 = Some (tg, m)
  assumes g-f: ¬ prefix g f
  assumes f-g: ¬ prefix f g
  shows {n ..<n + size-td tf} ∩ {m ..<m + size-td tg} = {}
  using wf f g g-f f-g
  proof (induct f arbitrary: t tf n g tg m )
    case Nil
    then show ?case by simp
  next

```

case (*Cons f1 fs*)
from *Cons.premis* **obtain**
wf: *wf-desc t* **and**
fl-f: *field-lookup t (f1 # fs) 0 = Some (tf, n)* **and**
fl-g: *field-lookup t g 0 = Some (tg, m)* **and**
not-prefix-g-f: \neg *prefix g (f1 # fs)* **and**
not-prefix-f-g: \neg *prefix (f1 # fs) g* **by** *simp*

from *field-lookup-append-Some [OF wf, where f=[f1] and g=fs, simplified, OF fl-f]*
obtain *w k* **where** *fl-f1*: *field-lookup t [f1] 0 = Some (w, k)* **and** *fl-fs*:
field-lookup w fs k = Some (tf, n)
by *blast*

from *field-lookup-wf-desc-pres(1) [OF wf fl-f1]* **have** *wf-w*: *wf-desc w*.
show *?case*
proof (*cases g*)
case *Nil*
with *Cons.premis* **show** *?thesis* **by** *simp*
next
case (*Cons g1 gs*)
from *field-lookup-append-Some [OF wf, where f=[g1] and g=gs, simplified, OF fl-g [simplified Cons]]*
obtain *v l* **where** *fl-g1*: *field-lookup t [g1] 0 = Some (v, l)* **and** *fl-gs*:
field-lookup v gs l = Some (tg, m)
by *blast*
show *?thesis*
proof (*cases f1 = g1*)
case *True*
with *fl-f1 fl-g1 fl-gs* **obtain**
v-w: *v=w* **and** *k-l*: *k=l* **and**
fl-g1': *field-lookup t [g1] 0 = Some (w, k)* **and** *fl-gs'*: *field-lookup w gs k*
= *Some (tg, m)*
by *auto*
from *not-prefix-g-f Cons True* **have** *not-prefix-gs-fs*: \neg *prefix gs fs* **by** *auto*
from *not-prefix-f-g Cons True* **have** *not-prefix-fs-gs*: \neg *prefix fs gs* **by** *auto*

from *field-lookup-offset-le(1) [OF fl-fs]* **have** *le-k-n*: $k \leq n$.
from *field-lookup-offset-shift' [OF fl-fs, of 0, simplified]*
have *fl-fs'*: *field-lookup w fs 0 = Some (tf, n - k)*.

from *field-lookup-offset-le(1) [OF fl-gs']* **have** *le-k-m*: $k \leq m$.
from *field-lookup-offset-shift' [OF fl-gs', of 0, simplified]*
have *fl-gs'*: *field-lookup w gs 0 = Some (tg, m - k)*.

from *Cons.hyps [OF wf-w fl-fs' fl-gs' not-prefix-gs-fs not-prefix-fs-gs]*
have $\{n - k..<n - k + \text{size-td } tf\} \cap \{m - k..<m - k + \text{size-td } tg\} = \{\}$

```

    with le-k-n le-k-m show ?thesis
      by auto
  next
  case False
  from field-lookup-single-field-disj [OF fl-f1 fl-g1 False]
  have disj: {k.. $k + \text{size-td } w$ }  $\cap$  {l.. $l + \text{size-td } v$ } = {} .
  from field-lookup-offset-le(1) [OF fl-fs] have k-n:  $k \leq n$  .
  from field-lookup-offset-le(1) [OF fl-gs] have l-m:  $l \leq m$  .
  from field-lookup-offset-shift' [OF fl-fs, of 0, simplified]
  have fl-fs': field-lookup w fs 0 = Some (tf,  $n - k$ ) .
  from field-lookup-offset-shift' [OF fl-gs, of 0, simplified]
  have fl-gs': field-lookup v gs 0 = Some (tg,  $m - l$ ) .
  from field-lookup-offset-size' [OF fl-fs^] have sz1:  $\text{size-td } tf + (n - k) \leq$ 
size-td w .
  from field-lookup-offset-size' [OF fl-gs^] have sz2:  $\text{size-td } tg + (m - l) \leq$ 
size-td v .

  show ?thesis
    using disj k-n l-m sz1 sz2
    by auto
  qed
qed
qed

```

theorem root-ptr-valid-non-prefix-disj:

```

  assumes valid-p: root-ptr-valid d (p::'a::mem-type ptr)
  assumes f: field-lookup (typ-info-t (TYPE('a::mem-type))) f 0 = Some (t, n)
  assumes g: field-lookup (typ-info-t (TYPE('a::mem-type))) g 0 = Some (s, m)
  assumes match-t: export-uinfo t = typ-uinfo-t (TYPE('c))
  assumes match-s: export-uinfo s = typ-uinfo-t (TYPE('d))
  assumes non-prefix-f-g:  $\neg$  prefix f g
  assumes non-prefix-g-f:  $\neg$  prefix g f
  shows ptr-span (PTR('c::mem-type)  $\&(p \rightarrow f)$ )  $\cap$  ptr-span (PTR('d::mem-type)
 $\&(p \rightarrow g)$ ) = {}

```

proof –

```

  from field-lookup-non-prefix-disj [OF wf-desc f g non-prefix-g-f non-prefix-f-g]
  have disj: {n.. $n + \text{size-td } t$ }  $\cap$  {m.. $m + \text{size-td } s$ } = {} .

```

```

  from match-t have sz-t:  $\text{size-td } t = \text{size-of } \text{TYPE}('c)$ 
  using export-size-of by force
  from match-s have sz-s:  $\text{size-td } s = \text{size-of } \text{TYPE}('d)$ 
  using export-size-of by force
  from f have off-f: field-offset TYPE('a) f = n
  using field-lookup-offset-eq by blast
  from g have off-g: field-offset TYPE('a) g = m
  using field-lookup-offset-eq by blast
  have le-addr-card:  $\text{size-td } (\text{typ-info-t } \text{TYPE}('a)) < \text{addr-card}$ 

```

```

  by (metis max-size size-of-def)
  from field-lookup-offset-size' [OF f] have t-bound: size-td t + n ≤ size-td (typ-info-t
  TYPE('a)).
  with le-addr-card have t-bound': size-td t + n < addr-card
  by simp
  from field-lookup-offset-size' [OF g] have s-bound: size-td s + m ≤ size-td
  (typ-info-t TYPE('a)).
  with le-addr-card have s-bound': size-td s + m < addr-card
  by simp

show ?thesis
  using disj t-bound' s-bound'
  apply (simp add: sz-t sz-s field-lvalue-def off-f off-g intvl-def)
  apply (safe; clarsimp simp add: unat-of-nat-eq)
  subgoal premises prems for k ka
  proof -
    from prems have n + k = m + ka
    by (metis (no-types, opaque-lifting) add commute len-of-addr-card
    less-trans nat-add-left-cancel-less of-nat-add sz-t t-bound' unat-of-nat-eq)
    with prems show False by simp
  qed
  subgoal premises prems for k ka
  proof -
    from prems have n + k = m + ka
    by (metis (no-types, opaque-lifting) add commute len-of-addr-card
    less-trans nat-add-left-cancel-less of-nat-add s-bound' sz-s unat-of-nat-eq)
    with prems show ?thesis by simp
  qed
done
qed

```

theorem *root-ptr-valid-non-prefix-h-val-heap-update-eq'*:

```

  assumes p': PTR-MATCH p' (PTR('c) &(p→f))
  assumes q': PTR-MATCH q' (PTR('d) &(q→g))
  assumes valid-p: root-ptr-valid d (p::'a::mem-type ptr)
  assumes valid-q: root-ptr-valid d (q::'b::mem-type ptr)
  assumes disj: typ-uinfo-t (TYPE('a)) = (typ-uinfo-t (TYPE('b))) ⇒
    ptr-val p = ptr-val q ⇒
    ¬ prefix f g ∧ ¬ prefix g f
  assumes match-s: export-uinfo s = typ-uinfo-t (TYPE('d::mem-type))
  assumes match-t: export-uinfo t = typ-uinfo-t (TYPE('c::mem-type))
  assumes g: field-lookup (typ-info-t (TYPE('b))) g 0 = Some (s, n)
  assumes f: field-lookup (typ-info-t (TYPE('a'))) f 0 = Some (t, m)
  shows h-val (heap-update p' v h) q' = h-val h q'

proof -
  {
    assume typ-eq: typ-uinfo-t TYPE('a) = typ-uinfo-t TYPE('b)
    assume ptr-val-eq: ptr-val p = ptr-val q
  }

```

have $ptr\text{-span } (PTR('c) \ \&(p \rightarrow f)) \cap ptr\text{-span } (PTR('d) \ \&(q \rightarrow g)) = \{\}$
proof –
from $disj \ [OF \ typ\text{-eq} \ ptr\text{-val}\text{-eq}]$ **obtain**
 $non\text{-prefix}\text{-f}\text{-g}: \neg \text{prefix } f \ g$ **and** $non\text{-prefix}\text{-g}\text{-f}: \neg \text{prefix } g \ f$ **by** $blast$

from $typ\text{-eq} \ ptr\text{-val}\text{-eq} \ g \ match\text{-s} \ ptr\text{-val}\text{-eq}$
have $span\text{-eq}: (PTR('d) \ \&(q \rightarrow g)) = (PTR('d) \ \&(p \rightarrow g))$
by $(simp \ add: \ field\text{-lvalue}\text{-def} \ field\text{-offset}\text{-def})$

from $field\text{-lookup}\text{-typ}\text{-uinfo}\text{-t}\text{-Some} \ [OF \ g] \ typ\text{-eq}$
have $fl\text{-u}: field\text{-lookup} \ (typ\text{-uinfo}\text{-t} \ TYPE('a)) \ g \ 0 = Some \ (export\text{-uinfo} \ s, \ n)$
by $simp$
from $field\text{-lookup}\text{-export}\text{-uinfo}\text{-Some}\text{-rev} \ [OF \ fl\text{-u} \ [simplified \ typ\text{-uinfo}\text{-t}\text{-def}]]$
 $match\text{-s}$
obtain s' **where** $fl\text{-g}': field\text{-lookup} \ (typ\text{-info}\text{-t} \ TYPE('a)) \ g \ 0 = Some \ (s', \ n)$
and
 $match\text{-s}': export\text{-uinfo} \ s' = typ\text{-uinfo}\text{-t} \ TYPE('d)$
by $auto$

from $root\text{-ptr}\text{-valid}\text{-non}\text{-prefix}\text{-disj} \ [OF \ valid\text{-p} \ f \ fl\text{-g}' \ match\text{-t} \ match\text{-s}' \ non\text{-prefix}\text{-f}\text{-g}$
 $non\text{-prefix}\text{-g}\text{-f}]$
show $?thesis$ **by** $(simp \ add: \ span\text{-eq})$
qed
} note $disj = this$
from $root\text{-ptr}\text{-valid}\text{-field}\text{-disj}\text{-h}\text{-val}\text{-heap}\text{-update}\text{-eq}' \ [OF \ p' \ q' \ valid\text{-p} \ valid\text{-q} \ disj$
 $match\text{-s} \ match\text{-t} \ g \ f]$
show $?thesis$ **by** $simp$
qed

lemma $field\text{-tag}\text{-sub}'$:

assumes $fl: field\text{-lookup} \ (typ\text{-uinfo}\text{-t} \ TYPE('a::mem\text{-type})) \ f \ 0 = Some \ (t, \ n)$
shows $\{\&(p::'a \ ptr \rightarrow f)..\text{+size}\text{-td} \ t\} \subseteq ptr\text{-span} \ p$
proof –
from $field\text{-lookup}\text{-uinfo}\text{-Some}\text{-rev} \ [OF \ fl]$ **obtain** s **where**
 $fl': field\text{-lookup} \ (typ\text{-info}\text{-t} \ TYPE('a)) \ f \ 0 = Some \ (s, \ n)$ **and** $s\text{-t}: export\text{-uinfo}$
 $s = t$
by $auto$
from $field\text{-tag}\text{-sub} \ [OF \ fl']$
have $\{\&(p \rightarrow f)..\text{+size}\text{-td} \ s\} \subseteq ptr\text{-span} \ p$.
with $export\text{-uinfo}\text{-size} \ s\text{-t}$
show $\{\&(p \rightarrow f)..\text{+size}\text{-td} \ t\} \subseteq ptr\text{-span} \ p$ **by** $auto$
qed

lemma $all\text{-field}\text{-names}\text{-field}\text{-lookup}$:

assumes $f \in set \ (all\text{-field}\text{-names} \ (typ\text{-uinfo}\text{-t} \ TYPE('a::mem\text{-type})))$
shows $\exists t \ n. field\text{-lookup} \ (typ\text{-uinfo}\text{-t} \ TYPE('a::mem\text{-type})) \ f \ 0 = Some \ (t, \ n)$
using $all\text{-field}\text{-names}\text{-union}\text{-field}\text{-names}\text{-u}\text{-conv} \ field\text{-names}\text{-u}\text{-field}\text{-names}\text{-export}\text{-uinfo}\text{-conv}(1)$

```

field-names-SomeD assms
  by (metis all-field-names-exists-field-names-u(1) field-lookup-export-uinfo-Some
not-Some-eq-tuple typ-uinfo-t-def)

lemma field-lvalue-same-type-conv:
  assumes valid-p: d,g ⊨t (p::'a::mem-type ptr)
  assumes valid-q: d,g ⊨t (q::'a::mem-type ptr)
  assumes f1: f1 ∈ set (all-field-names (typ-uinfo-t TYPE('a)))
  assumes f2: f2 ∈ set (all-field-names (typ-uinfo-t TYPE('a)))
  shows (&(p→f1) = &(q→f2)) ↔ (p = q ∧ field-offset (TYPE('a)) f1 =
field-offset TYPE('a) f2)
proof (cases rule: ptr-valid-same-type-cases [OF valid-p valid-q, case-names Same
Disj])
  case Same
  with f1 f2 show ?thesis
    apply (simp add: field-lvalue-def)
    apply (metis all-field-names-exists-field-names-u(1) field-names-SomeD2
field-names-u-field-names-export-uinfo-conv(1) typ-uinfo-t-def unat-of-nat-field-offset
wf-desc)
  done
next
  case Disj
  from all-field-names-field-lookup [OF f1]
  obtain t1 n1 where fl1: field-lookup (typ-uinfo-t TYPE('a)) f1 0 = Some (t1,
n1)
  by blast
  from all-field-names-field-lookup [OF f2]
  obtain t2 n2 where fl2: field-lookup (typ-uinfo-t TYPE('a)) f2 0 = Some (t2,
n2)
  by blast

  from field-tag-sub' [OF fl1, of p] field-tag-sub' [OF fl2, of q] Disj
  have {&(p→f1)..+size-td t1} ∩ {&(q→f2)..+size-td t2} = {}
  by blast
  moreover
  from field-lookup-wf-size-desc-gt [OF fl1] have 0 < size-td t1 by simp
  then have &(p→f1) ∈ {&(p→f1)..+size-td t1}
  using first-in-intvl by blast
  moreover
  from field-lookup-wf-size-desc-gt [OF fl2] have 0 < size-td t2 by simp
  then have &(q→f2) ∈ {&(q→f2)..+size-td t2}
  using first-in-intvl by blast
  ultimately
  have &(p→f1) ≠ &(q→f2)
  by auto
  thus ?thesis
  by (auto simp add: field-lvalue-def field-offset-def field-offset-untyped-def size-of-def
fl1 fl2)
qed

```


lemma *field-lvalue-same-root-conv*:
assumes $f1$: *field-lookup* (*typ-info-t* $TYPE('a::mem-type)$) $f1\ 0 = Some(t1, n1)$
assumes $f2$: *field-lookup* (*typ-info-t* $TYPE('a::mem-type)$) $f2\ 0 = Some(t2, n2)$
shows $(\&(p::'a\ ptr \rightarrow f1) = \&(p \rightarrow f2)) \longleftrightarrow (n1 = n2)$
proof –
have max : *size-of* $TYPE('a) < addr-card$ **by** (*rule max-size*)
from $f1$
have $n1-bound$: $n1 < addr-card$
using *field-lookup-offset-eq field-offset-addr-card* **by** *blast*

from $f2$
have $n2-bound$: $n2 < addr-card$
using *field-lookup-offset-eq field-offset-addr-card* **by** *blast*
from $n1-bound\ n2-bound\ field-lookup-offset-size\ [OF\ f1]\ field-lookup-offset-size\ [OF\ f2]$
show *?thesis*
apply (*simp add: field-lvalue-def field-lookup-offset-eq [OF f1] field-lookup-offset-eq [OF f2]*)
by (*metis len-of-addr-card unat-of-nat-eq*)
qed

lemma *field-lvalue-same-type-conv'*:
assumes $valid-p$: $d, g \models_t (p::'a::mem-type\ ptr)$
assumes $valid-q$: $d, g \models_t (q::'a::mem-type\ ptr)$
assumes $f1$: $f1 \in set\ (all-field-names-no-padding\ (typ-info-t\ TYPE('a)))$
assumes $f2$: $f2 \in set\ (all-field-names-no-padding\ (typ-info-t\ TYPE('a)))$
shows $(\&(p \rightarrow f1) = \&(q \rightarrow f2)) \longleftrightarrow (p = q \wedge field-offset\ (TYPE('a))\ f1 = field-offset\ TYPE('a)\ f2)$
proof –
have $set\ (all-field-names-no-padding\ (typ-info-t\ TYPE('a))) \subseteq set\ (all-field-names\ (typ-uinfo-t\ TYPE('a)))$
using *subset-all-field-names-no-padding-all-field-names all-field-names-typ-uinfo-t-conv*
by *metis*
then
show *?thesis* **using** *field-lvalue-same-type-conv [OF valid-p valid-q] f1 f2*
by *blast*
qed

lemma *field-lvalue-same-root-conv'*:
assumes $f1$: $f1 \in set\ (all-field-names\ (typ-uinfo-t\ TYPE('a::mem-type)))$
assumes $f2$: $f2 \in set\ (all-field-names\ (typ-uinfo-t\ TYPE('a)))$
shows $(\&(p::'a\ ptr \rightarrow f1) = \&(p \rightarrow f2)) \longleftrightarrow (field-offset\ (TYPE('a))\ f1 = field-offset\ TYPE('a)\ f2)$
proof –
from *all-field-names-field-lookup [OF f1]*

obtain $t1\ n1$ **where** $f1: \text{field-lookup } (\text{typ-uinfo-t } \text{TYPE}('a))\ f1\ 0 = \text{Some } (t1, n1)$
by *blast*
from *all-field-names-field-lookup* [*OF* $f2$]
obtain $t2\ n2$ **where** $f2: \text{field-lookup } (\text{typ-uinfo-t } \text{TYPE}('a))\ f2\ 0 = \text{Some } (t2, n2)$
by *blast*

have $\text{max}: \text{size-of } \text{TYPE}('a) < \text{addr-card}$ **by** (*rule max-size*)

from *field-lookup-offset-size'* [*OF* $f1$]
have $n1 < \text{size-of } \text{TYPE}('a)$
apply (*simp add: size-of-def*)
by (*metis add-diff-cancel-right' add-leD2 cancel-comm-monoid-add-class.diff-cancel*
field-lookup-wf-size-desc-gt
less-le local.f1 not-add-less2 wf-size-desc-typ-uinfo-t-simp)
with max **have** $n1 < \text{addr-card}$ **by** *arith*

moreover

from *field-lookup-offset-size'* [*OF* $f2$]
have $n2 < \text{size-of } \text{TYPE}('a)$
apply (*simp add: size-of-def*)
by (*metis add-diff-cancel-right' add-leD2 cancel-comm-monoid-add-class.diff-cancel*
field-lookup-wf-size-desc-gt
le-imp-less-or-eq less-not-refl2 local.f2 not-add-less2 wf-size-desc-typ-uinfo-t-simp)
with max **have** $n2 < \text{addr-card}$ **by** *arith*

ultimately

show *?thesis*

apply (*simp add: field-offset-def field-offset-untyped-def f1 f2 field-lvalue-def*
size-of-def)
by (*metis len-of-addr-card unat-of-nat-eq*)

qed

lemma *field-lvalue-same-root-conv''*:

assumes $f1: f1 \in \text{set } (\text{all-field-names-no-padding } (\text{typ-info-t } \text{TYPE}('a::\text{mem-type})))$

assumes $f2: f2 \in \text{set } (\text{all-field-names-no-padding } (\text{typ-info-t } \text{TYPE}('a)))$

shows $(\&(p::'a\ \text{ptr} \rightarrow f1) = \&(p \rightarrow f2)) \longleftrightarrow (\text{field-offset } (\text{TYPE}('a))\ f1 = \text{field-offset } \text{TYPE}('a)\ f2)$

proof –

have $\text{set } (\text{all-field-names-no-padding } (\text{typ-info-t } \text{TYPE}('a::\text{mem-type}))) \subseteq$

$\text{set } (\text{all-field-names } (\text{typ-uinfo-t } \text{TYPE}('a::\text{mem-type})))$

using *subset-all-field-names-no-padding-all-field-names*

all-field-names-typ-uinfo-t-conv

```

    by metis
  with field-lvalue-same-root-conv' f1 f2 show ?thesis
  by blast
qed

```

lemma *disj-ptr-span-field-neg*:

```

  assumes disj: ptr-span (p::'a::mem-type ptr)  $\cap$  ptr-span (q::'b::mem-type ptr) =
  {}
  assumes fl1: field-lookup (typ-info-t TYPE('a)) f1 0 = Some (t1, n1)
  assumes fl2: field-lookup (typ-info-t TYPE('b)) f2 0 = Some (t2, n2)
  shows  $\&(p \rightarrow f1) \neq \&(q \rightarrow f2)$ 
proof -
  from field-tag-sub [OF fl1, of p] have  $\{\&(p \rightarrow f1)..\text{size-td } t1\} \subseteq \text{ptr-span } p$  .
  moreover
  from field-tag-sub [OF fl2, of q] have  $\{\&(q \rightarrow f2)..\text{size-td } t2\} \subseteq \text{ptr-span } q$  .
  moreover
  from field-lookup-wf-size-desc-gt [OF fl1] have  $0 < \text{size-td } t1$  by simp
  then have  $\&(p \rightarrow f1) \in \{\&(p \rightarrow f1)..\text{size-td } t1\}$ 
    using first-in-intvl by blast
  moreover
  from field-lookup-wf-size-desc-gt [OF fl2] have  $0 < \text{size-td } t2$  by simp
  then have  $\&(q \rightarrow f2) \in \{\&(q \rightarrow f2)..\text{size-td } t2\}$ 
    using first-in-intvl by blast
  ultimately
  show  $\&(p \rightarrow f1) \neq \&(q \rightarrow f2)$ 
    using disj
    by force
qed

```

lemma *disj-ptr-span-field-neg''*:

```

  assumes disj: ptr-span (p::'a::mem-type ptr)  $\cap$  ptr-span (q::'b::mem-type ptr) =
  {}
  assumes f1: f1  $\in$  set (all-field-names (typ-uinfo-t TYPE('a)))
  assumes f2: f2  $\in$  set (all-field-names (typ-uinfo-t TYPE('b')))
  shows  $\&(p \rightarrow f1) \neq \&(q \rightarrow f2)$ 
proof -
  from all-field-names-field-lookup [OF f1]
  obtain t1 n1 where fl1: field-lookup (typ-uinfo-t TYPE('a')) f1 0 = Some (t1,
n1)
    by blast
  from all-field-names-field-lookup [OF f2]
  obtain t2 n2 where fl2: field-lookup (typ-uinfo-t TYPE('b')) f2 0 = Some (t2,
n2)
    by blast
  from field-tag-sub' [OF fl1, of p] have  $\{\&(p \rightarrow f1)..\text{size-td } t1\} \subseteq \text{ptr-span } p$  .
  moreover
  from field-tag-sub' [OF fl2, of q] have  $\{\&(q \rightarrow f2)..\text{size-td } t2\} \subseteq \text{ptr-span } q$  .
  moreover

```

```

from field-lookup-wf-size-desc-gt [OF fl1] have  $0 < \text{size-td } t1$  by simp
then have  $\&(p \rightarrow f1) \in \{\&(p \rightarrow f1) .. + \text{size-td } t1\}$ 
  using first-in-intvl by blast
moreover
from field-lookup-wf-size-desc-gt [OF fl2] have  $0 < \text{size-td } t2$  by simp
then have  $\&(q \rightarrow f2) \in \{\&(q \rightarrow f2) .. + \text{size-td } t2\}$ 
  using first-in-intvl by blast
ultimately
show  $\&(p \rightarrow f1) \neq \&(q \rightarrow f2)$ 
  using disj
  by force
qed

```

```

lemma disj_ptr_span_field_neq':
  assumes disj:  $\text{ptr-span } (p::'a::\text{mem-type ptr}) \cap \text{ptr-span } (q::'b::\text{mem-type ptr}) = \{\}$ 
  assumes f1:  $f1 \in \text{set } (\text{all-field-names-no-padding } (\text{typ-into-t } \text{TYPE}('a)))$ 
  assumes f2:  $f2 \in \text{set } (\text{all-field-names-no-padding } (\text{typ-into-t } \text{TYPE}('b)))$ 
  shows  $\&(p \rightarrow f1) \neq \&(q \rightarrow f2)$ 
proof –
  have  $\text{set } (\text{all-field-names-no-padding } (\text{typ-into-t } \text{TYPE}('a))) \subseteq \text{set } (\text{all-field-names } (\text{typ-into-t } \text{TYPE}('a)))$ 
    using subset-all-field-names-no-padding-all-field-names
    all-field-names-typ-into-t-conv
    by metis
  moreover
  have  $\text{set } (\text{all-field-names-no-padding } (\text{typ-into-t } \text{TYPE}('b))) \subseteq \text{set } (\text{all-field-names } (\text{typ-into-t } \text{TYPE}('b)))$ 
    using subset-all-field-names-no-padding-all-field-names
    all-field-names-typ-into-t-conv
    by metis
  ultimately
  show thesis using disj f1 f2 disj_ptr_span_field_neq'' by blast
qed

```

```

lemma disj_ptr_span_field_disj_ptr_span:
  assumes disj:  $\text{ptr-span } (p::'a::\text{mem-type ptr}) \cap \text{ptr-span } (q::'b::\text{mem-type ptr}) = \{\}$ 
  assumes f1:  $f1 \in \text{set } (\text{field-names-u } (\text{typ-into-t } \text{TYPE}('a)) (\text{typ-into-t } (\text{TYPE}('c::\text{mem-type}))))$ 
  assumes f2:  $f2 \in \text{set } (\text{field-names-u } (\text{typ-into-t } \text{TYPE}('b)) (\text{typ-into-t } (\text{TYPE}('d::\text{mem-type}))))$ 
  shows  $\text{ptr-span } (\text{PTR}('c) \&(p \rightarrow f1)) \cap \text{ptr-span } (\text{PTR}('d) (\&(q \rightarrow f2))) = \{\}$ 
proof –
  from f1
  obtain n1 where fl1: field-lookup (typ-into-t TYPE('a)) f1 0 = Some (typ-into-t
    (TYPE('c::mem-type)), n1)
    apply (simp add: typ-into-t-def)
    by (smt (verit) field-lookup-export-into-Some field-names-SomeD2 field-names-u-field-names-export-into-c
wf-desc)
  from f2

```

obtain $n2$ **where** $f12$: *field-lookup* (*typ-ufinfo-t* $TYPE('b)$) $f2\ 0 = \text{Some}$ (*typ-ufinfo-t* ($TYPE('d::\text{mem-type})$), $n2$)
apply (*simp add: typ-ufinfo-t-def*)
by (*smt (verit) field-lookup-export-ufinfo-Some field-names-SomeD2 field-names-u-field-names-export-ufinfo-c wf-desc*)

from *field-tag-sub'* [*OF* $f1$, *of* p] **have** $\{\&(p \rightarrow f1)..\text{+size-of } TYPE('c)\} \subseteq \text{ptr-span } p$
by (*simp add: size-of-tag*)
moreover
from *field-tag-sub'* [*OF* $f2$, *of* q] **have** $\{\&(q \rightarrow f2)..\text{+size-of } TYPE('d)\} \subseteq \text{ptr-span } q$
by (*simp add: size-of-tag*)
ultimately show *?thesis*
using *disj*
by *auto*
qed

lemma *disj-ptr-span-field-disj-ptr-span'*:

assumes *disj*: $\text{ptr-span } (p::'a::\text{mem-type } \text{ptr}) \cap \text{ptr-span } (q::'b::\text{mem-type } \text{ptr}) = \{\}$
assumes $f1$: *field-lookup* (*typ-ufinfo-t* ($TYPE('a)$)) $f1\ 0 = \text{Some}$ (s , n)
assumes *match-s*: *export-ufinfo* $s = \text{typ-ufinfo-t}$ ($TYPE('c::\text{mem-type})$)
assumes $f2$: *field-lookup* (*typ-ufinfo-t* ($TYPE('b)$)) $f2\ 0 = \text{Some}$ (t , m)
assumes *match-t*: *export-ufinfo* $t = \text{typ-ufinfo-t}$ ($TYPE('d::\text{mem-type})$)
shows $\{\&(p \rightarrow f1)..\text{+size-of } TYPE('c)\} \cap \{\&(q \rightarrow f2)..\text{+size-of } TYPE('d)\} = \{\}$

proof –

from $f1$ *match-s*
have $f1'$: $f1 \in \text{set}$ (*field-names-u* (*typ-ufinfo-t* $TYPE('a)$) (*typ-ufinfo-t* ($TYPE('c::\text{mem-type})$)))
by (*simp add: field-lookup-all-field-names(1) field-lookup-field-ti set-field-names-u-all-field-names-conv*)
from $f2$ *match-t*
have $f2'$: $f2 \in \text{set}$ (*field-names-u* (*typ-ufinfo-t* $TYPE('b)$) (*typ-ufinfo-t* ($TYPE('d::\text{mem-type})$)))
by (*simp add: field-lookup-all-field-names(1) field-lookup-field-ti set-field-names-u-all-field-names-conv*)
from *disj-ptr-span-field-disj-ptr-span* [*OF* $f1'$ $f2'$]
show *?thesis*
by *simp*

qed

lemma *disj-ptr-span-field-disj-ptr-span''*:

assumes *disj*: $\text{ptr-span } (p::'a::\text{mem-type } \text{ptr}) \cap \text{ptr-span } (q::'b::\text{mem-type } \text{ptr}) \equiv \{\}$
assumes $f1$: *field-lookup* (*typ-ufinfo-t* ($TYPE('a)$)) $f1\ 0 = \text{Some}$ (s , n)
assumes *match-s*: *export-ufinfo* $s = \text{typ-ufinfo-t}$ ($TYPE('c::\text{mem-type})$)
assumes $f2$: *field-lookup* (*typ-ufinfo-t* ($TYPE('b)$)) $f2\ 0 = \text{Some}$ (t , m)
assumes *match-t*: *export-ufinfo* $t = \text{typ-ufinfo-t}$ ($TYPE('d::\text{mem-type})$)
shows $\{\&(p \rightarrow f1)..\text{+size-of } TYPE('c)\} \cap \{\&(q \rightarrow f2)..\text{+size-of } TYPE('d)\} = \{\}$
using *disj-ptr-span-field-disj-ptr-span'* [*OF* - $f1$ *match-s* $f2$ *match-t*]
by (*simp add: disj*)

lemma *root-ptr-valid-disj-field-lvalue-conv*:
assumes *valid-p*: *root-ptr-valid* *d* (*p*::'*a*::*mem-type ptr*)
assumes *valid-q*: *root-ptr-valid* *d* (*q*::'*b*::*mem-type ptr*)
assumes *neq*: *typ-uinfo-t* *TYPE*('a') \neq *typ-uinfo-t* *TYPE*('b')
assumes *f1*: *f1* \in *set* (*all-field-names* (*typ-uinfo-t* *TYPE*('a')))
assumes *f2*: *f2* \in *set* (*all-field-names* (*typ-uinfo-t* *TYPE*('b')))
shows $\&(p \rightarrow f1) = \&(q \rightarrow f2) = \text{False}$
using *root-ptr-valid-type-neq-disjoint* [*OF valid-p valid-q neq*] *f1 f2*
by (*simp add: disj-ptr-span-field-neq'*)

lemma *root-ptr-valid-disj-field-lvalue-conv'*:
assumes *valid-p*: *root-ptr-valid* *d* (*p*::'*a*::*mem-type ptr*)
assumes *valid-q*: *root-ptr-valid* *d* (*q*::'*b*::*mem-type ptr*)
assumes *neq*: *typ-uinfo-t* *TYPE*('a') \neq *typ-uinfo-t* *TYPE*('b')
assumes *f1*: *f1* \in *set* (*all-field-names* (*typ-info-t* *TYPE*('a')))
assumes *f2*: *f2* \in *set* (*all-field-names* (*typ-info-t* *TYPE*('b')))
shows $\&(p \rightarrow f1) = \&(q \rightarrow f2) = \text{False}$
proof –
have *set* (*all-field-names* (*typ-info-t* *TYPE*('a'))) = *set* (*all-field-names* (*typ-uinfo-t* *TYPE*('a')))
by (*simp add: all-field-names-typ-uinfo-t-conv*)
moreover
have *set* (*all-field-names* (*typ-info-t* *TYPE*('b'))) = *set* (*all-field-names* (*typ-uinfo-t* *TYPE*('b')))
by (*simp add: all-field-names-typ-uinfo-t-conv*)
ultimately show *?thesis* **using** *root-ptr-valid-disj-field-lvalue-conv* [*OF valid-p valid-q neq*] *f1 f2*
by *simp*
qed

lemma *intvl-non-zero-non-empty*: $0 < n \implies \{p..+n\} \neq \{\}$
by (*metis empty-iff first-in-intvl less-numeral-extra*(3))

lemma *disj-inter-swap*: $A \cap B = \{\} \implies B \cap A = \{\}$
by *blast*

lemma *h-val-clift-field*:
clift *hp* (*Ptr* $\&(p \rightarrow f)$) = *Some* *v* $\implies v = \text{h-val}$ (*hrs-mem* *hp*) (*Ptr* $\&(p \rightarrow f)$)
by (*simp add: h-val-clift'*)

lemma *valid-root-footprints-no-overlap*:
assumes *valid-root-footprint* *d* (*ptr-val* (*p*::'*a*::*mem-type ptr*)) (*typ-uinfo-t* *TYPE*('a'))
assumes *valid-root-footprint* *d* (*ptr-val* (*q*::'*b*::*mem-type ptr*)) (*typ-uinfo-t* *TYPE*('b'))
assumes *typ-uinfo-t* *TYPE*('a') \neq *typ-uinfo-t* *TYPE*('b')
shows *ptr-span* *p* \cap *ptr-span* *q* = $\{\}$
by (*metis antisym-conv* *assms*(1) *assms*(2) *assms*(3) *disj-inter-swap*
export-uinfo-size size-of-def typ-uinfo-t-def valid-root-footprint-overlap-sub-typ
valid-root-footprint-valid-footprint)

lemma *valid-root-footprints-overlap*:
assumes *valid-root-footprint* d (*ptr-val* ($p::'a::\text{mem-type ptr}$)) (*typ-uinfo-t* $\text{TYPE}('a)$)
assumes *valid-root-footprint* d (*ptr-val* ($q::'b::\text{mem-type ptr}$)) (*typ-uinfo-t* $\text{TYPE}('b)$)
assumes $\text{ptr-span } p \cap \text{ptr-span } q \neq \{\}$
shows (*typ-uinfo-t* $\text{TYPE}('a)$) = (*typ-uinfo-t* $\text{TYPE}('b)$)
by (*meson* *assms*(1) *assms*(2) *assms*(3) *valid-root-footprints-no-overlap*)

lemma *field-lookup-same-type-empty*:
field-lookup $t f n = \text{Some } (s, m) \implies s = t \longleftrightarrow f = []$
field-lookup-struct $st f n = \text{Some } (s, m) \implies f \neq []$
field-lookup-list $ts f n = \text{Some } (s, m) \implies f \neq []$
field-lookup-tuple $td f n = \text{Some } (s, m) \implies f \neq []$
apply (*induct* t **and** st **and** ts **and** td *arbitrary*: $f n s m$ **and** $f n s m$ **and** $f n s m$ **and** $f n s m$)
apply (*auto split*: *if-split-asm* *dest!*: *td-set-struct-field-lookup-structD* *td-set-struct-size-lte*)
done

lemma *root-ptr-valid-root-only*:
assumes *valid*: *root-ptr-valid* d ($p::'a::\text{mem-type ptr}$)
assumes *non-empty*: $f \neq []$
assumes $f: f \in \text{set } (\text{field-names-u } (\text{typ-uinfo-t } \text{TYPE}('a)) (\text{typ-uinfo-t } \text{TYPE}('b)))$

shows *root-ptr-valid* d ($\text{PTR}('b::\text{mem-type}) \ \&(p \rightarrow f)$) = *False*

proof –

{
assume *root-ptr-valid* d ($\text{PTR}('b::\text{mem-type}) \ \&(p \rightarrow f)$)
hence *valid-root*: *valid-root-footprint* d (*ptr-val* ($\text{PTR}('b::\text{mem-type}) \ \&(p \rightarrow f)$))
(*typ-uinfo-t* $\text{TYPE}('b)$)
by (*simp* *add*: *root-ptr-valid-def*)

have wf : *wf-desc* (*typ-info-t* $\text{TYPE}('a)$) **by** *simp*

from *valid* **have** *valid-p*: *valid-root-footprint* d (*ptr-val* p) (*typ-uinfo-t* $\text{TYPE}('a)$)
by (*simp* *add*: *root-ptr-valid-def*)

from f **have** $f \in \text{set } (\text{field-names } (\text{typ-info-t } \text{TYPE}('a)) (\text{typ-uinfo-t } \text{TYPE}('b)))$
by (*simp* *add*: *field-names-u-field-names-export-uinfo-conv*(1) *typ-uinfo-t-def*)

from *field-names-SomeD2* [*OF this* wf] **obtain** $s n$ **where**

fl : *field-lookup* (*typ-info-t* $\text{TYPE}('a)$) $f \ 0 = \text{Some } (s, n)$ **and** s : *export-uinfo*
 $s = \text{typ-uinfo-t } \text{TYPE}('b)$
by *blast*

from *field-lookup-export-uinfo-Some* [*OF* fl] s

have fl' : *field-lookup* (*typ-uinfo-t* $\text{TYPE}('a)$) $f \ 0 = \text{Some } (\text{typ-uinfo-t } \text{TYPE}('b), n)$
by (*simp* *add*: *typ-uinfo-t-def*)

```

from field-lookup-same-type-empty(1) [OF this] non-empty
have typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('b) by simp
from valid-root-footprints-no-overlap [OF valid-p valid-root this]
have disjnt (ptr-span p) (ptr-span (PTR('b) &(p→f))) by (simp add: disjnt-def)
moreover
from field-tag-sub' [OF fl'] have {&(p→f)..+size-td (typ-uinfo-t TYPE('b))}
⊆ ptr-span p .
ultimately
have False
by (metis Int-absorb Int-absorb1 disj-ptr-span-ptr-neq disjnt-def ptr-val.ptr-val-def
size-of-tag)
} thus ?thesis by blast
qed

```

```

lemma root-ptr-valid-root-only':
assumes valid: root-ptr-valid d (p::'a::mem-type ptr)
assumes non-empty: f ≠ []
assumes f: f ∈ set (field-names (typ-info-t TYPE('a)) (export-uinfo (typ-info-t
TYPE('b))))
shows root-ptr-valid d (PTR('b::mem-type) &(p→f)) = False
proof –
have set (field-names (typ-info-t TYPE('a)) (export-uinfo (typ-info-t TYPE('b))))
=
set (field-names-u (typ-uinfo-t TYPE('a)) (typ-uinfo-t TYPE('b)))
by (simp add: fold-typ-uinfo-t set-field-names-all-field-names-conv set-field-names-u-all-field-names-conv)
with root-ptr-valid-root-only [OF valid non-empty] f
show ?thesis
by simp
qed

```

```

lemma disj-subset:
assumes  $A \cap B = \{\}$ 
assumes  $A' \subseteq A$ 
assumes  $B' \subseteq B$ 
shows  $A' \cap B' = \{\}$ 
using assms
by blast

```

```

lemma disj-intvl-subset:
assumes disj: {p..+n1} ∩ {q..+m1} = {}
assumes le1: n2 + off1 ≤ n1
assumes le2: m2 + off2 ≤ m1
shows  $\{p + \text{of-nat off1}..+n2\} \cap \{q + \text{of-nat off2}..+m2\} = \{\}$ 
apply (rule disj-subset [OF disj])
apply (rule intvl-le [OF le1])
apply (rule intvl-le [OF le2])
done

```



```

lemma disj-intvl-field':
  assumes disj: {ptr-val p..+n1} ∩ {ptr-val q..+m1} = {}
  assumes f: &(p→f) = ptr-val p + (of-nat off1)
  assumes g: &(q→g) = ptr-val q + (of-nat off2)
  assumes le1: n2 + off1 ≤ n1
  assumes le2: m2 + off2 ≤ m1
  shows {&(p→f)..+n2} ∩ {&(q→g)..+m2} = {}
  apply (simp add: f g)
  apply (rule disj-intvl-subset [OF disj le1 le2])
  done

```

```

lemma disj-intvl-field:
  assumes disj: {ptr-val (p::'a::mem-type ptr)..+n1} ∩ {ptr-val (q::'b::mem-type ptr)..+m1} = {}
  assumes le1: n2 + (field-offset TYPE('a) f) ≤ n1
  assumes le2: m2 + (field-offset TYPE('b) g) ≤ m1
  shows {&(p→f)..+n2} ∩ {&(q→g)..+m2} = {}
  apply (simp add: field-lvalue-def)
  apply (rule disj-intvl-subset [OF disj le1 le2])
  done

```

```

lemma intvl-fields-disj1:
  assumes f: &(p→f) = ptr-val p + off1
  assumes g: &(p→g) = ptr-val p + off2
  assumes n-sz: unat off1 + n ≤ addr-card
  assumes m-sz: unat off2 + m ≤ addr-card
  assumes le: unat off1 ≤ unat off2
  assumes disj: unat off1 + n ≤ unat off2
  shows {&(p→f)..+n} ∩ {&(p→g)..+m} = {}
  proof –
  {
    fix k k'
    assume k-n: k < n
    assume k'-m: k' < m
    assume eq: off1 + word-of-nat k = off2 + word-of-nat k'
    have False
    proof –
      from eq n-sz m-sz k-n k'-m have unat off1 + k = unat off2 + k'
      by (metis (no-types, opaque-lifting) word-bits-def add.right-neutral
        len-of-addr-card less-le-trans nat-add-left-cancel-less of-nat-add unat-0
        unat-of-nat-eq word-arith-nat-add)
      with disj k-n show False by arith
    qed
  } note lem = this
  show ?thesis

```

```

  apply (simp add: f g)
  apply (clarsimp simp add: intvl-def)

```

```

    using lem
    by auto
qed

```

lemma *intvl-fields-disj*:

```

assumes f: &(p→f) = ptr-val p + off1
assumes g: &(p→g) = ptr-val p + off2
assumes n-sz: unat off1 + n ≤ addr-card
assumes m-sz: unat off2 + m ≤ addr-card
assumes disj: if unat off1 ≤ unat off2 then unat off1 + n ≤ unat off2 else unat
off2 + m ≤ unat off1
shows {&(p→f)..+n} ∩ {&(p→g)..+m} = {}
proof (cases unat off1 ≤ unat off2)
  case True
    show ?thesis
      apply (rule intvl-fields-disj1 [OF f g n-sz m-sz])
      using disj True
      by simp-all
  next
    case False
      have {&(p→g)..+m} ∩ {&(p→f)..+n} = {}
        apply (rule intvl-fields-disj1 [OF g f m-sz n-sz])
        using disj False
        by simp-all
      thus ?thesis by blast
qed

```

lemma *intvl-fields-disj-calculation*:

```

assumes f: &(p→f) = ptr-val p + off1
assumes g: &(p→g) = ptr-val p + off2
assumes uoff1: unat off1 = uoff1
assumes uoff2: unat off2 = uoff2
assumes n-sz: uoff1 + n ≤ addr-card
assumes m-sz: uoff2 + m ≤ addr-card
assumes disj: if uoff1 ≤ uoff2 then uoff1 + n ≤ uoff2 else uoff2 + m ≤ uoff1
shows {&(p→f)..+n} ∩ {&(p→g)..+m} = {}
  using intvl-fields-disj [OF f g] uoff1 uoff2 n-sz m-sz disj by simp

```

lemma *disjoint-field-lvalue-propagation-right*:

```

fixes q::'a::mem-type ptr
assumes disj: A ∩ {ptr-val q..+m} = {}
assumes fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (t, k)
assumes m: m = size-of TYPE('a)
assumes n: n = size-td t
shows A ∩ {&(q→f)..+n} = {}
using disj fl m n
by (meson disjoint-iff field-tag-sub subsetD)

```

lemma *disjoint-field-lvalue-propagation-left*:

fixes $q::'a::\text{mem-type ptr}$
assumes $\text{disj}: \{\text{ptr-val } q..+m\} \cap A = \{\}$
assumes $\text{fl}: \text{field-lookup } (\text{typ-info-t } \text{TYPE}('a)) \text{ f } 0 = \text{Some } (t, k)$
assumes $m: m = \text{size-of } \text{TYPE}('a)$
assumes $n: n = \text{size-td } t$
shows $\{\&(q \rightarrow f)..+n\} \cap A = \{\}$
using disj fl m n
by (*meson disjoint-iff field-tag-sub subsetD*)

lemma *disjoint-field-lvalue-propagation-both*:

fixes $q::'a::\text{mem-type ptr}$
fixes $p::'b::\text{mem-type ptr}$
assumes $\text{disj}: \{\text{ptr-val } q..+m1\} \cap \{\text{ptr-val } p..+m2\} = \{\}$
assumes $\text{fl1}: \text{field-lookup } (\text{typ-info-t } \text{TYPE}('a)) \text{ f1 } 0 = \text{Some } (t1, k1)$
assumes $\text{fl2}: \text{field-lookup } (\text{typ-info-t } \text{TYPE}('b)) \text{ f2 } 0 = \text{Some } (t2, k2)$
assumes $m1: m1 = \text{size-of } \text{TYPE}('a)$
assumes $n1: n1 = \text{size-td } t1$
assumes $m2: m2 = \text{size-of } \text{TYPE}('b)$
assumes $n2: n2 = \text{size-td } t2$
shows $\{\&(q \rightarrow f1)..+n1\} \cap \{\&(p \rightarrow f2)..+n2\} = \{\}$
using $\text{disj fl1 fl2 m1 n1 m2 n2}$
by (*meson disjoint-iff field-tag-sub subsetD*)

lemmas *disjoint-field-lvalue-propagation =*

disjoint-field-lvalue-propagation-left
disjoint-field-lvalue-propagation-right
disjoint-field-lvalue-propagation-both

lemma *disjoint-field-lvalue-neq*:

fixes $q::'a::\text{mem-type ptr}$
fixes $p::'b::\text{mem-type ptr}$
assumes $\text{disj}: \{\text{ptr-val } q..+m1\} \cap \{\text{ptr-val } p..+m2\} = \{\}$
assumes $\text{fl1}: \text{field-lookup } (\text{typ-info-t } \text{TYPE}('a)) \text{ f1 } 0 = \text{Some } (t1, k1)$
assumes $\text{fl2}: \text{field-lookup } (\text{typ-info-t } \text{TYPE}('b)) \text{ f2 } 0 = \text{Some } (t2, k2)$
assumes $m1: m1 = \text{size-of } \text{TYPE}('a)$
assumes $n1: n1 = \text{size-td } t1$
assumes $m2: m2 = \text{size-of } \text{TYPE}('b)$
assumes $n2: n2 = \text{size-td } t2$
shows $\&(q \rightarrow f1) = \&(p \rightarrow f2) = \text{False}$
using $\text{disjoint-field-lvalue-propagation-both } [\text{OF } \text{disj fl1 fl2 m1 n1 m2 n2 }] \text{ n1 n2}$
by (*metis field-lookup-wf-size-desc-gt intvl-start-inter local.fl1 local.fl2 wf-size-desc*)

lemma *overlap-field-disj-contradiction*:

fixes $q::'a::\text{mem-type ptr}$
assumes $\text{overlap}: A \cap \{\&(q \rightarrow f)..+n\} \neq \{\}$
assumes $\text{disj}: A \cap \{\text{ptr-val } q..+m\} = \{\}$
assumes $\text{fl}: \text{field-lookup } (\text{typ-info-t } \text{TYPE}('a)) \text{ f } 0 = \text{Some } (t, k)$

```

assumes m: m = size-of TYPE('a)
assumes n: n = size-td t
shows False
using overlap disj fl m n
by (meson disjoint-iff field-tag-sub subsetD)

thm root-ptr-valid-non-prefix-disj

lemma overlap-field-prefix-left:
  fixes p::'a::mem-type ptr
  assumes overlap:  $\{\&(p \rightarrow f) \dots + n1\} \cap \{\&(p \rightarrow g) \dots + n2\} \neq \{\}$ 
  assumes less: length f < length g
  assumes f: field-lookup (typ-info-t TYPE('a)) f 0 = Some (t1, k1)
  assumes g: field-lookup (typ-info-t TYPE('a)) g 0 = Some (t2, k2)
  assumes n1: n1 = size-td t1
  assumes n2: n2 = size-td t2
  shows prefix f g
proof (cases prefix f g)
  case True
    then show ?thesis by simp
next
  case False
    from less have not-prefix:  $\neg$  prefix g f
      by (simp add: More-Sublist.not-prefix-longer)
    have wf:wf-desc (typ-info-t TYPE('a)) by simp
    have le-addr-card: size-td (typ-info-t TYPE('a)) < addr-card
      by (metis max-size size-of-def)
    from field-lookup-offset-size' [OF f] have t1-bound: size-td t1 + k1  $\leq$  size-td
      (typ-info-t TYPE('a)).
    with le-addr-card have t1-bound': size-td t1 + k1 < addr-card
      by simp
    from field-lookup-offset-size' [OF g] have t2-bound: size-td t2 + k2  $\leq$  size-td
      (typ-info-t TYPE('a)).
    with le-addr-card have t2-bound': size-td t2 + k2 < addr-card
      by simp

from field-lookup-non-prefix-disj [OF wf f g not-prefix False] overlap n1 n2
have False
  using f g
  apply (simp add: field-lvalue-def intvl-def)
  apply (safe; clarsimp simp add: unat-of-nat-eq)
  subgoal premises prems for k ka
  proof –
    from prems have k1 + k = k2 + ka
    by (smt (verit, ccfv-threshold) Abs-fnat-hom-add add commute add-mono-thms-linordered-field(2)

      of-nat-lt-size-of order-less-le-trans size-of-def t1-bound t2-bound)
    with prems show False by simp
qed

```

```

subgoal premises prems for k ka
proof –
  from prems have  $k1 + k = k2 + ka$ 
  by (smt (verit, ccfv-threshold) Abs-fnat-hom-add add.commute add-mono-thms-linordered-field(2)

    of-nat-lt-size-of order-less-le-trans size-of-def t1-bound t2-bound)
  with prems show False by simp
qed
done
thus ?thesis by simp
qed

```

```

lemma overlap-field-prefix-right:
  fixes p::'a::mem-type ptr
  assumes overlap:  $\{\&(p \rightarrow g) \dots + n2\} \cap \{\&(p \rightarrow f) \dots + n1\} \neq \{\}$ 
  assumes less:  $\text{length } f < \text{length } g$ 
  assumes f: field-lookup (typ-info-t TYPE('a)) f 0 = Some (t1, k1)
  assumes g: field-lookup (typ-info-t TYPE('a)) g 0 = Some (t2, k2)
  assumes n1:  $n1 = \text{size-td } t1$ 
  assumes n2:  $n2 = \text{size-td } t2$ 
  shows prefix f g
  using overlap-field-prefix-left
  using f g less n1 n2 overlap by blast

```

```

lemma field-lvalue-eq-non-prefix:
  fixes p::'a::mem-type ptr
  assumes fld-eq:  $\&(p \rightarrow f) = \&(p \rightarrow g)$ 
  assumes f: field-lookup (typ-info-t TYPE('a)) f 0 = Some (t1, k1)
  assumes g: field-lookup (typ-info-t TYPE('a)) g 0 = Some (t2, k2)
  assumes f-g:  $\neg \text{prefix } f \ g$ 
  assumes g-f:  $\neg \text{prefix } g \ f$ 
  shows  $f = g$ 
proof –
  have wf:wf-desc (typ-info-t TYPE('a)) by simp
  show ?thesis using field-lookup-non-prefix-disj [OF wf f g g-f f-g] fld-eq f g
  apply (simp add: field-lvalue-def)
  by (metis field-lookup-wf-size-desc-pres(1) field-lvalue-same-root-conv fld-eq
    less-add-same-cancel1 nat-neq-iff wf-size-desc wf-size-desc-gt(1))
qed

```

```

lemma field-lvalue-eq-non-prefix-conv:
  fixes p::'a::mem-type ptr
  assumes f: field-lookup (typ-info-t TYPE('a)) f 0 = Some (t1, k1)
  assumes g: field-lookup (typ-info-t TYPE('a)) g 0 = Some (t2, k2)
  assumes f-g:  $\neg \text{prefix } f \ g$ 
  assumes g-f:  $\neg \text{prefix } g \ f$ 
  shows  $\&(p \rightarrow f) = \&(p \rightarrow g) \longleftrightarrow f = g$ 
  apply standard
  apply (drule field-lvalue-eq-non-prefix [OF - f g f-g g-f])

```

```

apply simp
apply simp
done

```

```

thm cvalid-field-pres''

```

```

lemma hrs-mem-update-comp: hrs-mem-update f o hrs-mem-update g = hrs-mem-update
(f o g)
apply (rule ext)
apply (auto simp add: hrs-mem-update-def)
done

```

```

lemma hrs-mem-update-comp': hrs-mem-update f (hrs-mem-update g s) = hrs-mem-update
(f o g) s
using hrs-mem-update-comp
by (metis comp-eq-dest-lhs)

```

```

named-theorems plift-defs and
plift-cond-simps and
plift-iff and
plift-eqI and
ptr-valid-h-t-valid and
plift-heap-update and
ptr-valid-intros and
h-val-field-plift and
the-plift-h-val-conv

```

```

lemma map-option-the-conv: f (the (map-option g x)) = f (the x)  $\longleftrightarrow$  ( $\forall v. x =$ 
Some v  $\longrightarrow f (g v) = f v$ )
by (metis handy-if-lemma option.exhaust-sel option.map-sel option.simps(8))

```

```

definition the-default:: 'a  $\Rightarrow$  'a option  $\Rightarrow$  'a where
the-default x v = (case v of Some z  $\Rightarrow$  z | None  $\Rightarrow$  x)

```

```

lemma the-default-simps[simp]:
the-default x None = x
the-default x (Some z) = z
by (auto simp add: the-default-def)

```

```

locale wf-ptr-valid =
fixes ptr-valid::heap-typ-desc  $\Rightarrow$  'a::mem-type ptr-guard
assumes ptr-valid-h-t-valid[ptr-valid-h-t-valid]:  $\bigwedge d. \text{ptr-valid } d \text{ } p \implies d \models_t p$ 
begin

```

```

definition plift:: heap-raw-state  $\Rightarrow$  'a typ-heap
where [plift-defs]:plift h = lift-t (ptr-valid (hrs-htd h)) h

```

lemma *plift-Some-iff* [*plift-iff*]:
shows $(\exists v. \text{plift } h \ p = \text{Some } v) \longleftrightarrow \text{ptr-valid } (\text{hrs-htd } h) \ p$
using *ptr-valid-h-t-valid*
apply (*cases h*)
apply (*auto simp add: plift-def TypHeap.lift-t-if hrs-htd-def h-t-valid-def*)
done

lemma *plift-None-iff* [*plift-iff*]:
shows $(\text{plift } h \ p = \text{None}) \longleftrightarrow \neg \text{ptr-valid } (\text{hrs-htd } h) \ p$
using *ptr-valid-h-t-valid*
apply (*cases h*)
apply (*auto simp add: plift-def TypHeap.lift-t-if hrs-htd-def h-t-valid-def*)
done

lemma *plift-None*: $\neg \text{ptr-valid } (\text{hrs-htd } h) \ p \implies \text{plift } h \ p = \text{None}$
by (*simp add: plift-None-iff*)

lemma *ptr-valid-c-guard*: $\text{ptr-valid } d \ p \implies \text{c-guard } p$
by (*auto intro: ptr-valid-h-t-valid h-t-valid-c-guard*)

lemma *plift-Some-ptr-valid*[*plift-cond-simps*]: $\text{plift } h \ p = \text{Some } v \implies \text{ptr-valid } (\text{hrs-htd } h) \ p$
using *plift-Some-iff* **by** *blast*

lemma *plift-Some-h-t-valid*[*plift-cond-simps*]: $\text{plift } h \ p = \text{Some } v \implies \text{hrs-htd } h \models_t p$
using *plift-Some-ptr-valid ptr-valid-h-t-valid*
by *blast*

lemma *plift-Some-c-guard*[*plift-cond-simps*]: $\text{plift } h \ p = \text{Some } v \implies \text{c-guard } p$
using *plift-Some-h-t-valid h-t-valid-c-guard*
by *blast*

lemma *ptr-valid-heap-update-conv*[*simp*]:
 $\text{ptr-valid } (\text{hrs-htd } ((\text{hrs-mem-update } (\text{heap-update } p \ v) \ s))) \ q \longleftrightarrow \text{ptr-valid } (\text{hrs-htd } s) \ q$
by (*cases s*) (*auto simp add: hrs-htd-def hrs-mem-update-def*)

lemma *ptr-valid-heap-update-padding-conv*[*simp*]:
 $\text{ptr-valid } (\text{hrs-htd } ((\text{hrs-mem-update } (\text{heap-update-padding } p \ v \ bs) \ s))) \ q \longleftrightarrow \text{ptr-valid } (\text{hrs-htd } s) \ q$
by (*cases s*) (*auto simp add: hrs-htd-def hrs-mem-update-def*)

lemma *ptr-valid-heap-update-pres*: $\text{ptr-valid } (\text{hrs-htd } s) \ q \implies \text{ptr-valid } (\text{hrs-htd } ((\text{hrs-mem-update } (\text{heap-update } p \ v) \ s))) \ q$
by (*simp add: ptr-valid-heap-update-conv*)

lemma *ptr-valid-heap-update-padding-pres*: $\text{ptr-valid } (\text{hrs-htd } s) \ q \implies$

ptr-valid (*hrs-htd* ((*hrs-mem-update* (*heap-update-padding* *p v bs*))) *q*)
by (*simp add: ptr-valid-heap-update-conv*)

lemma *plift-Some-h-val*:
assumes *Some: plift h p = Some v*
shows *h-val (hrs-mem h) p = v*
using *plift-Some-c-guard [OF Some] plift-Some-ptr-valid [OF Some] Some h-val-clift'*
apply (*simp add: plift-def*)
by (*smt (verit, del-insts) c-type-class.lift-def hrs-mem-def prod.collapse typ-simps(8)*)

lemma *ptr-valid-plift-Some-h-val*:
assumes *ptr-valid (hrs-htd h) p*
shows *plift h p = Some (h-val (hrs-mem h) p)*
using *assms plift-Some-h-val plift-Some-iff* **by** *blast*

lemma *the-plift-hval-conv: ptr-valid (hrs-htd h) p \implies the (plift h p) = h-val (hrs-mem h) p*
by (*metis option.collapse plift-None-iff wf-ptr-valid.plift-Some-h-val wf-ptr-valid-axioms*)

lemma *the-default-plift-hval-conv[the-plift-h-val-conv]*:
ptr-valid (hrs-htd h) p \implies the-default c-type-class.zero (plift h p) = h-val (hrs-mem h) p
by (*simp add: ptr-valid-plift-Some-h-val*)

lemma *valid-h-val-the-plift-conv*:
assumes *valid: ptr-valid (hrs-htd h) p*
shows *h-val (hrs-mem h) p = the (plift h p)*
using *valid plift-Some-h-val plift-iff*
by (*cases plift h p*) *auto*

lemma *valid-h-val-the-default-plift-conv*:
assumes *valid: ptr-valid (hrs-htd h) p*
shows *h-val (hrs-mem h) p = the-default c-type-class.zero (plift h p)*
using *valid plift-Some-h-val plift-iff*
by (*cases plift h p*) *auto*

lemma *plift-Some-clift-Some*:
assumes *Some: plift h p = Some v*
shows *clift h p = Some v*
using *plift-Some-h-t-valid [OF Some]*
h-t-valid-clift-Some-iff [of h p]
h-val-clift' [of h p] plift-Some-h-val [OF Some]
by *auto*

lemma *clift-None-plift-None*:
assumes *clift h p = None*
shows *plift h p = None*

using *assms*
by (*metis plift-None-iff plift-Some-iff wf-ptr-valid.plift-Some-clift-Some wf-ptr-valid-axioms*)

lemma *plift-clift-conv*:

assumes *ptr-valid (hrs-htd h) p*

shows *plift h p = clift h p*

by (*metis assms plift-Some-clift-Some plift-Some-iff*)

lemma *the-plift-hval-fun-upd-eqI [plift-eqI]*:

fixes *p::'a::mem-type ptr*

fixes *q::'a::mem-type ptr*

assumes *ptr-valid (hrs-htd h) p*

shows (*the-default c-type-class.zero (plift*
(hrs-mem-update (heap-update p v) h) q)) =

((λp. the-default c-type-class.zero (plift h p))(p := v)) q

by (*smt (verit, best) assms clift-heap-update fun-upd-apply hrs-htd-mem-update*

local.ptr-valid-h-t-valid plift-clift-conv the-default-simps(2) wf-ptr-valid.plift-None-iff wf-ptr-valid-axioms)

lemma *the-plift-hval-fun-upd-padding-eqI [plift-eqI]*:

fixes *p::'a::mem-type ptr*

fixes *q::'a::mem-type ptr*

assumes *ptr-valid (hrs-htd h) p*

assumes *length bs = size-of TYPE('a)*

shows (*the-default c-type-class.zero (plift*
(hrs-mem-update (heap-update-padding p v bs) h) q)) =

((λp. the-default c-type-class.zero (plift h p))(p := v)) q

by (*smt (verit, ccfv-SIG) CTypes.mem-type-simps(2) assms(1) assms(2) fun-upd-apply*
h-val-def

h-val-heap-update-padding heap-list-update-disjoint-same heap-update-padding-def

hrs-htd-mem-update hrs-mem-update ptr-valid-same-type-cases the-default-plift-hval-conv

wf-ptr-valid.plift-None-iff wf-ptr-valid.ptr-valid-h-t-valid wf-ptr-valid-axioms)

lemma *field-the-plift-hval-fun-upd-eqI [plift-eqI]*:

fixes *p::'a::mem-type ptr*

fixes *q::'a::mem-type ptr*

assumes *ptr-valid (hrs-htd h) p ∧ f x = v*

shows *f (the-default c-type-class.zero (plift*
(hrs-mem-update (heap-update p x) h) q)) =

((λp. f (the-default c-type-class.zero (plift h p))(p := v)) q

by (*simp add: assms(1) the-plift-hval-fun-upd-eqI*)

lemma *field-the-plift-hval-fun-upd-padding-eqI [plift-eqI]*:

fixes *p::'a::mem-type ptr*

fixes *q::'a::mem-type ptr*

assumes *ptr-valid (hrs-htd h) p ∧ f x = v*

```

assumes length bs = size-of TYPE('a)
shows f (the-default c-type-class.zero (plift
  (hrs-mem-update (heap-update-padding p x bs) h) q)) =
  ((λp. f (the-default c-type-class.zero (plift h p)))(p := v)) q
by (simp add: assms the-plift-hval-fun-upd-padding-eqI)

lemma the-plift-hval-eqI [plift-eqI]:
  fixes p::'b::c-type ptr
  fixes q::'a::mem-type ptr
  assumes eq: ptr-valid (hrs-htd h) q  $\implies$  hrs-htd h  $\models_t$  q  $\implies$ 
    (h-val ((heap-update p v) (hrs-mem h)) q) = (h-val (hrs-mem h) q)
  shows (the-default c-type-class.zero (plift
    (hrs-mem-update (heap-update p v) h) q)) =
    (the-default c-type-class.zero (plift h q))
proof (cases (plift h q))
  case None
  then show ?thesis using plift-None-iff
    by (metis hrs-htd-mem-update)
next
  case (Some x)
  from plift-Some-ptr-valid [OF Some]
  have ptr-valid: ptr-valid (hrs-htd h) q.
  from plift-Some-h-t-valid [OF Some]
  have h-t-valid: hrs-htd h  $\models_t$  q.
  from eq [OF ptr-valid h-t-valid]
  have h-val-eq: (h-val (heap-update p v (hrs-mem h)) q) = (h-val (hrs-mem h) q).

  with ptr-valid-heap-update-conv plift-Some-iff ptr-valid eq plift-Some-h-val
  show ?thesis
    by (smt (verit) hrs-mem-update option.sel)
qed

lemma the-plift-hval-padding-eqI [plift-eqI]:
  fixes p::'b::c-type ptr
  fixes q::'a::mem-type ptr
  assumes lbs: length bs = size-of TYPE('b)
  assumes eq: ptr-valid (hrs-htd h) q  $\implies$  hrs-htd h  $\models_t$  q  $\implies$ 
    (h-val ((heap-update-padding p v bs) (hrs-mem h)) q) = (h-val (hrs-mem h) q)

  shows (the-default c-type-class.zero (plift
    (hrs-mem-update (heap-update-padding p v bs) h) q)) =
    (the-default c-type-class.zero (plift h q))
proof (cases (plift h q))
  case None
  then show ?thesis using plift-None-iff
    by (metis hrs-htd-mem-update)
next
  case (Some x)
  from plift-Some-ptr-valid [OF Some]

```

have *ptr-valid*: *ptr-valid* (*hrs-htd* *h*) *q*.
from *plift-Some-h-t-valid* [*OF Some*]
have *h-t-valid*: *hrs-htd* *h* \models_t *q*.
from *eq* [*OF ptr-valid h-t-valid*]
have *h-val-eq*: (*h-val* (*heap-update-padding* *p v bs* (*hrs-mem* *h*)) *q*) = (*h-val* (*hrs-mem* *h*) *q*).

with *ptr-valid-heap-update-padding-conv plift-Some-iff ptr-valid eq plift-Some-h-val*
show *?thesis*
by (*smt* (*z3*) *hrs-mem-update option.sel*)
qed

lemma *field-the-plift-hval-eqI* [*plift-eqI*]:

fixes *p*::'*b*::*c-type ptr*
fixes *q*::'*a*::*mem-type ptr*
assumes *eq*: *ptr-valid* (*hrs-htd* *h*) *q* \implies *hrs-htd* *h* \models_t *q* \implies
 f (*h-val* ((*heap-update* *p v*) (*hrs-mem* *h*)) *q*) = f (*h-val* (*hrs-mem* *h*) *q*)
shows f (*the-default c-type-class.zero* (*plift*
(*hrs-mem-update* (*heap-update* *p v*) *h*) *q*)) =
 f (*the-default c-type-class.zero* (*plift* *h q*))

proof (*cases* (*plift* *h q*))

case *None*

then show *?thesis* **using** *plift-None-iff*

by (*metis hrs-htd-mem-update*)

next

case (*Some x*)

from *plift-Some-ptr-valid* [*OF Some*]

have *ptr-valid*: *ptr-valid* (*hrs-htd* *h*) *q*.

from *plift-Some-h-t-valid* [*OF Some*]

have *h-t-valid*: *hrs-htd* *h* \models_t *q*.

from *eq* [*OF ptr-valid h-t-valid*]

have *h-val-eq*: f (*h-val* (*heap-update* *p v* (*hrs-mem* *h*)) *q*) = f (*h-val* (*hrs-mem* *h*) *q*).

with *ptr-valid-heap-update-conv plift-Some-iff ptr-valid eq plift-Some-h-val*

show *?thesis*

by (*metis hrs-mem-f the-default-simps(2)*)

qed

lemma *field-the-plift-hval-padding-eqI* [*plift-eqI*]:

fixes *p*::'*b*::*c-type ptr*

fixes *q*::'*a*::*mem-type ptr*

assumes *lbs*: *length* *bs* = *size-of* *TYPE*('b)

assumes *eq*: *ptr-valid* (*hrs-htd* *h*) *q* \implies *hrs-htd* *h* \models_t *q* \implies

f (*h-val* ((*heap-update-padding* *p v bs*) (*hrs-mem* *h*)) *q*) = f (*h-val* (*hrs-mem* *h*) *q*)

shows f (*the-default c-type-class.zero* (*plift*

(*hrs-mem-update* (*heap-update-padding* *p v bs*) *h*) *q*)) =

f (*the-default c-type-class.zero* (*plift* *h q*))

```

proof (cases (plift h q))
  case None
  then show ?thesis using plift-None-iff
    by (metis hrs-htd-mem-update)
next
  case (Some x)
  from plift-Some-ptr-valid [OF Some]
  have ptr-valid: ptr-valid (hrs-htd h) q.
  from plift-Some-h-t-valid [OF Some]
  have h-t-valid: hrs-htd h  $\models_t$  q.
  from eq [OF ptr-valid h-t-valid]
  have h-val-eg: f (h-val (heap-update-padding p v bs (hrs-mem h)) q) = f (h-val
(hrs-mem h) q).

  with ptr-valid-heap-update-padding-conv plift-Some-iff ptr-valid eq plift-Some-h-val
  show ?thesis
    by (metis hrs-mem-f the-default-simps(2))
qed

lemma plift-heap-update [plift-heap-update]:
   $\llbracket$  ptr-valid (hrs-htd h) p  $\rrbracket \implies$ 
    plift (hrs-mem-update (heap-update p v) h)
      = (plift h)(p := Some (v::'a::mem-type))
  using plift-Some-clift-Some ptr-valid-h-t-valid
  by (metis (no-types, opaque-lifting) h-t-valid-guard-subst hrs-htd-def hrs-htd-mem-update

    hrs-mem-def hrs-mem-update lift-t-heap-update plift-def prod.collapse)

lemma plift-heap-update-padding [plift-heap-update]:
   $\llbracket$  ptr-valid (hrs-htd h) p; length bs = size-of TYPE('a)  $\rrbracket \implies$ 
    plift (hrs-mem-update (heap-update-padding p v bs) h)
      = (plift h)(p := Some (v::'a::mem-type))
  apply (rule ext)
  apply (clarsimp simp add: fun-upd-def, intro conjI)
  subgoal by (simp add: h-val-heap-update-padding hrs-mem-update wf-ptr-valid.ptr-valid-plift-Some-hval
wf-ptr-valid-axioms)
  using plift-Some-clift-Some ptr-valid-h-t-valid
  by (smt (verit, best) fun-upd-apply hrs-htd-mem-update the-default-plift-hval-conv

    the-plift-hval-fun-upd-padding-eqI wf-ptr-valid.plift-None-iff wf-ptr-valid.ptr-valid-plift-Some-hval
wf-ptr-valid-axioms)

lemma h-val-field-plift [h-val-field-plift]:
  fixes pa :: 'a :: mem-type ptr
  assumes cl: plift hp pa = Some v
  and ft: field-ti TYPE('a) f = Some t
  and eu: export-uinfo t = typ-uinfo-t TYPE('b :: mem-type)
  shows h-val (hrs-mem hp) (Ptr &(pa→f) :: 'b :: mem-type ptr) = from-bytes

```

```

(access-ti0 t v)
  using cl fl eu
  using h-val-field-clift' plift-Some-clift-Some by blast

lemma plift-disjoint-region:
  fixes p: 'a :: mem-type ptr
  fixes q: 'b :: mem-type ptr
  assumes disj: ptr-span q ∩ ptr-span p = {}
  shows plift (hrs-mem-update (heap-update q v) m) p = plift m p
  using disj
  by (metis Int-commute h-val-update-regions-disjoint hrs-htd-mem-update
      hrs-mem-heap-update option.collapse plift-None-iff the-plift-hval-conv)

lemma plift-disjoint-region-padding:
  fixes p: 'a :: mem-type ptr
  fixes q: 'b :: mem-type ptr
  assumes disj: ptr-span q ∩ ptr-span p = {}
  assumes lbs: length bs = size-of TYPE('b)
  shows plift (hrs-mem-update (heap-update-padding q v bs) m) p = plift m p
  using disj lbs
  by (metis (no-types, lifting) CTypes.mem-type-simps(2) disj h-val-def
      heap-list-update-disjoint-same heap-update-padding-def hrs-htd-mem-update
      hrs-mem-update wf-ptr-valid.plift-None-iff wf-ptr-valid.ptr-valid-plift-Some-hval
      wf-ptr-valid-axioms)

lemma map-option-the-plift-h-val-conv:
  f (the (map-option g (plift h p))) = f (the (plift h p)) ↔ (ptr-valid (hrs-htd h)
  p → f (g (h-val (hrs-mem h) p)) = f (h-val (hrs-mem h) p))
  apply (subst map-option-the-conv)
  apply standard
  using ptr-valid-plift-Some-hval apply blast
  using plift-Some-h-val plift-Some-ptr-valid by blast

lemma invalid-plift-heap-update-skip:
  assumes invalid-p: ¬ ptr-valid (hrs-htd h) p
  assumes typed-p: hrs-htd h ⊨t p
  shows plift (hrs-mem-update (heap-update p v) h) q = plift h q
  proof (cases p = q)
  case True
  with invalid-p have invalid-q: ¬ ptr-valid (hrs-htd h) q by simp
  from invalid-q have plift h q = None
  by (simp add: plift-None)
  moreover
  from invalid-q have plift (hrs-mem-update (heap-update p v) h) q = None
  by (simp add: plift-None)
  ultimately show ?thesis by simp
next
case False
note neq-p-q = this

```

```

show ?thesis
proof (cases ptr-valid (hrs-htd h) q)
  case True
    from ptr-valid-h-t-valid [OF True]
    have hrs-htd h  $\models_t$  q .
    from ptr-valid-same-type-cases [OF typed-p this] neq-p-q
    have ptr-span p  $\cap$  ptr-span q = {} by blast
    then show ?thesis
      by (simp add: plift-disjoint-region)
  next
    case False
    from False have plift h q = None
      by (simp add: plift-None)
    moreover
    from False have plift (hrs-mem-update (heap-update p v) h) q = None
      by (simp add: plift-None)
    ultimately show ?thesis by simp
qed
qed

lemma invalid-plift-heap-update-padding-skip:
  assumes invalid-p:  $\neg$  ptr-valid (hrs-htd h) p
  assumes typed-p: hrs-htd h  $\models_t$  p
  assumes lbs: length bs = size-of TYPE('a)
shows plift (hrs-mem-update (heap-update-padding p v bs) h) q = plift h q
proof (cases p = q)
  case True
    with invalid-p have invalid-q:  $\neg$  ptr-valid (hrs-htd h) q by simp
    from invalid-q have plift h q = None
      by (simp add: plift-None)
    moreover
    from invalid-q have plift (hrs-mem-update (heap-update-padding p v bs) h) q =
None
      by (simp add: plift-None)
    ultimately show ?thesis by simp
  next
    case False
    note neq-p-q = this
    show ?thesis
    proof (cases ptr-valid (hrs-htd h) q)
      case True
        from ptr-valid-h-t-valid [OF True]
        have hrs-htd h  $\models_t$  q .
        from ptr-valid-same-type-cases [OF typed-p this] neq-p-q
        have ptr-span p  $\cap$  ptr-span q = {} by blast
        then show ?thesis
          using lbs
          by (simp add: plift-disjoint-region-padding)
      next

```

```

    case False
  from False have plift h q = None
    by (simp add: plift-None)
  moreover
  from False have plift (hrs-mem-update (heap-update-padding p v bs) h) q =
  None
    by (simp add: plift-None)
  ultimately show ?thesis by simp
qed
qed

```

end

```

global-interpretation simple-lift: wf-ptr-valid root-ptr-valid rewrites simple-lift.plift
= simple-lift
  apply (unfold-locales)
  apply (simp add: root-ptr-valid-h-t-valid)
  apply (rule ext, rule ext)
  by (metis root-ptr-valid-h-t-valid hrs-mem-def lift-t-if prod.collapse simple-lift-def

      wf-ptr-valid.intro wf-ptr-valid.plift-None-iff wf-ptr-valid.plift-def)

```

named-theorems *ptr-valid-stack-alloc* and *ptr-valid-stack-release* and *plift-stack-alloc*
and *plift-stack-release*

```

locale ptr-valid-stack-alloc = wf-ptr-valid +
  assumes alloc[ptr-valid-stack-alloc]:
     $\bigwedge d' p q. (p, d') \in \text{stack-alloc } \mathcal{S} \text{ TYPE}('a::\text{mem-type}) d \implies$ 
       $\text{ptr-valid } d' q \longleftrightarrow (q = p) \vee \text{ptr-valid } d q$ 
begin

```

```

lemma plift-stack-alloc-update:
  assumes alloc:  $(p, d) \in \text{stack-alloc } \mathcal{S} \text{ TYPE}('a::\text{mem-type}) (\text{hrs-htd } m)$ 
  shows  $(\text{plift } (\text{hrs-htd-update } (\lambda-. d) m))(p \mapsto x) = (\lambda q. \text{if } p = q \text{ then } \text{Some } x \text{ else } \text{plift } m q)$ 
  apply (rule ext)
  apply clarsimp
  using ptr-valid-stack-alloc [OF alloc]
  by (metis hrs-htd-update hrs-mem-htd-update option.collapse plift-None-iff
      wf-ptr-valid.valid-h-val-the-plift-conv wf-ptr-valid-axioms)

```

```

lemma plift-stack-alloc-sim[plift-stack-alloc]:
  assumes alloc:  $(p, d) \in \text{stack-alloc } \mathcal{S} \text{ TYPE}('a::\text{mem-type}) (\text{hrs-htd } m)$ 
  shows  $(\text{plift } (\text{hrs-mem-update } (\text{heap-update } p x) (\text{hrs-htd-update } (\lambda-. d) m))) =$ 
     $(\lambda q. \text{if } p = q \text{ then } \text{Some } x \text{ else } \text{plift } m q)$ 
  using plift-stack-alloc-update [OF alloc]
  by (simp add: hrs-htd-update plift-heap-update ptr-valid-stack-alloc [OF alloc])

```

```

end

locale ptr-valid-stack-release = wf-ptr-valid +
  assumes release[ptr-valid-stack-release]:
     $\bigwedge d p q. \text{root-ptr-valid } d p \implies \text{typ-uinfo-t } TYPE('a) \neq \text{typ-uinfo-t } TYPE(\text{stack-byte})$ 
 $\implies$ 
    ptr-valid (stack-release p d) q  $\longleftrightarrow ((q \neq p) \wedge \text{ptr-valid } d q)$ 
begin

lemma plift-stack-release[plift-stack-release]:
  assumes root-valid: root-ptr-valid (hrs-htd m) (p::'a::mem-type ptr)
  assumes p: typ-uinfo-t TYPE('a)  $\neq$  typ-uinfo-t TYPE(stack-byte)
  shows plift (hrs-htd-update (stack-release p) m) q =
    (if p = q then None else plift m q)
using ptr-valid-stack-release [OF root-valid p]
by (metis (no-types, lifting) hrs-htd-update hrs-mem-htd-update option.collapse
  valid-h-val-the-plift-conv wf-ptr-valid.plift-None-iff wf-ptr-valid-axioms)
end

locale ptr-valid-stack = ptr-valid-stack-alloc + ptr-valid-stack-release

global-interpretation simple-lift: ptr-valid-stack root-ptr-valid
  apply (unfold-locales)
  apply (smt (verit) disjoint-iff ptr-force-type-same-cleared-region root-ptr-valid-def

    root-ptr-valid-h-t-valid s-footprintD s-footprintI2 size-of-def stack-alloc-cases
    stack-alloc-preserves-root-ptr-valid stack-release-other
    stack-release-stack-alloc-inverse typ-uinfo-size valid-root-footprint-def)
  apply (smt (verit) disjoint-iff in-ptr-span-itself intvlI intvl-inter root-ptr-valid-def

    root-ptr-valid-h-t-valid root-ptr-valid-neq-disjoint root-ptr-valid-same-type-cases

    simple-heap-diff-types-impl-diff-ptrs size-of-def stack-release-other
    stack-release-root-ptr-valid-footprint typ-uinfo-size valid-root-footprint-def)
  done

lemma valid-root-footprint-domain:
  assumes valid-root-footprint d a t
  assumes  $\bigwedge x. x \in \{a \dots + \text{size-td } t\} \implies d' x = d x$ 
  shows valid-root-footprint d' a t
  by (metis assms(1) assms(2) intvlI valid-root-footprint-def)

lemma valid-root-footprint-cases:
  assumes valid-a: valid-root-footprint d a T
  assumes root-p: valid-root-footprint d b S
  shows  $(a = b \wedge S = T) \vee \{b \dots + \text{size-td } S\} \cap \{a \dots + \text{size-td } T\} = \{\}$ 
proof (cases  $\{b \dots + \text{size-td } S\} \cap \{a \dots + \text{size-td } T\} = \{\}$ )

```



```

    case True
  thus ?thesis by simp
next
  case False
  note overlap = this

  show ?thesis
  proof (cases S = T)
    case True
    from valid-root-footprint-neq-disjoint [OF valid-a root-p] overlap
    have a = b
      by (metis inf-commute size-of-tag)
    then show ?thesis by (simp add: True)
  next
    case False
    with valid-root-footprint-type-neq-disjoint [OF root-p valid-a False] overlap
    have False
      by (simp add: size-of-def)
    thus ?thesis
      by simp
  qed
qed

```

```

lemma valid-root-footprint-domain-cases:
  assumes valid-a: valid-root-footprint d' a t
  assumes other-eq:  $\bigwedge x. x \notin \{a \text{ ..+ size-td } t\} \implies d' x = d x$ 
  assumes root-p: valid-root-footprint d' b s
  shows
     $(a = b \wedge s = t \vee$ 
       $\text{valid-root-footprint } d \text{ } b \text{ } s)$ 
  proof (cases  $\{b \text{ ..+ size-td } s\} \cap \{a \text{ ..+ size-td } t\} = \{\}$ )
    case True
    with other-eq have  $\bigwedge x. x \in \{b \text{ ..+ size-td } s\} \implies d' x = d x$  by auto
    with valid-root-footprint-domain [OF root-p]
    have valid-root-footprint d b s
      by auto
    thus ?thesis by blast
  next
    case False
    note overlap = this

    show ?thesis
    proof (cases s = t)
      case True
      from valid-root-footprint-neq-disjoint [OF valid-a root-p] overlap
      have a = b
        by (metis inf-commute size-of-tag)
      then show ?thesis by (simp add: True)
    next

```

```

case False
with valid-root-footprint-type-neq-disjoint [OF root-p valid-a False] overlap
have False
  by (simp add: size-of-def)
thus ?thesis
  by simp
qed
qed

```

```

lemma h-t-valid-ptr-span-preservation:
fixes p::'a::mem-type ptr
assumes valid: d  $\models_t$  p
assumes eq:  $\bigwedge a. a \in \text{ptr-span } p \implies d' a = d a$ 
shows d'  $\models_t$  p
  using valid eq
  by (auto simp add: h-t-valid-def valid-footprint-def Let-def intvlI size-of-def)

```

```

lemma
fixes p::'a::mem-type ptr
fixes q::'a::mem-type ptr
assumes root-ptr-valid d p
assumes d  $\models_t$  q
shows  $p = q \vee \text{ptr-span } p \cap \text{ptr-span } q = \{\}$ 
by (meson assms(1) assms(2) root-ptr-valid-same-type-cases)

```

```

lemma stack-alloc-typing-other:
fixes q::'b::mem-type ptr
assumes stack-alloc: (p, d')  $\in$  stack-alloc  $\mathcal{S}$  (TYPE('a::mem-type)) d
assumes no-stack: typ-uinfo-t (TYPE('b))  $\neq$  typ-uinfo-t (TYPE(stack-byte))
assumes other:  $\neg$  TYPE('b)  $\leq_\tau$  (TYPE('a))
shows d'  $\models_t$  q = d  $\models_t$  q
proof
  assume valid-d'-q: d'  $\models_t$  q
  show d  $\models_t$  q
  proof (cases d  $\models_t$  q)
    case True
    then show ?thesis by simp
  next
  case False
  have invalid-d-q:  $\neg$  d  $\models_t$  q using False .

```

```

from stack-alloc obtain
  d': d' = ptr-force-type p d and
  typ-uinfo-t (TYPE('a))  $\neq$  typ-uinfo-t (TYPE(stack-byte)) and
  ptr-span p  $\subseteq$   $\mathcal{S}$  and
  no-stack-d:  $\forall a \in \text{ptr-span } p. \text{root-ptr-valid } d$  (PTR (stack-byte) a) and
  ptr-aligned p and c-guard p and root-p: root-ptr-valid d' p
by (cases rule: stack-alloc-cases)
from valid-d'-q obtain

```

```

    footprint-d'-q: valid-footprint d' (ptr-val q) (typ-uinfo-t TYPE('b)) and
    c-guard-q: c-guard q
    by (auto simp add: h-t-valid-def)
  with invalid-d-q
  have no-footprint-d-q:  $\neg$  (valid-footprint d (ptr-val q) (typ-uinfo-t TYPE('b)))
    by (simp add: d' h-t-valid-def)
  from root-p valid-d'-q other have ptr-span q  $\cap$  ptr-span p = {}
    by (meson disj-inter-swap root-ptr-valid-not-subtype-disjoint)
  with no-footprint-d-q ptr-force-type-valid-footprint-disjoint2 [OF footprint-d'-q
    [simplified d']]
    have False
      by (simp add: size-of-tag)
    then show ?thesis by simp
  qed
next
assume d  $\models_t$  q from stack-alloc-preserves-typing [OF stack-alloc no-stack this]
show d'  $\models_t$  q .
qed

```

lemma $TYPE('b::c\text{-type}) \leq_\tau (TYPE('a::c\text{-type})) \implies \text{typ-uinfo-t } TYPE('b) \neq \text{typ-uinfo-t } TYPE('a)$
 $\implies \neg TYPE('a) \leq_\tau (TYPE('b))$

using *typ-uinfo-eq-sub-typ-conv(3)* by *blast*

named-theorems *stack-alloc-ptr-valid-parent* and *stack-release-ptr-valid-parent*

lemma *stack-alloc-root-ptr-valid-other:*

```

  fixes q::'b::mem-type ptr
  assumes stack-alloc: (p, d')  $\in$  stack-alloc  $\mathcal{S}$  (TYPE('a::mem-type)) d
  assumes neq: typ-uinfo-t TYPE('b)  $\neq$  typ-uinfo-t TYPE('a)
  assumes sub: TYPE('a)  $\leq_\tau$  (TYPE('b))
  shows root-ptr-valid d' (q::'b::mem-type ptr) = root-ptr-valid d q
proof
  assume valid-d': root-ptr-valid d' q
  show root-ptr-valid d q
    using neq stack-alloc stack-alloc-root-ptr-valid-new-cases valid-d' by blast
next
  assume root-ptr-valid d q
  then show root-ptr-valid d' q
    by (smt (verit, best) sub stack-alloc stack-alloc-cases stack-release-root-ptr-valid-cases

```

stack-release-stack-alloc-inverse sub-typ-def sub-typ-stack-byte)

qed

lemma *stack-release-root-ptr-valid-other:*

```

  fixes p::'a::mem-type ptr
  fixes q::'b::mem-type ptr

```

assumes *root*: *root-ptr-valid* d p
assumes *non-stack*: *typ-uinfo-t* ($TYPE('a)$) \neq *typ-uinfo-t* ($TYPE(stack\text{-}byte)$)
assumes *neq*: *typ-uinfo-t* $TYPE('b::mem\text{-}type)$ \neq *typ-uinfo-t* $TYPE('a)$
assumes *sub*: $TYPE('a) \leq_{\tau} (TYPE('b))$
shows *root-ptr-valid* (*stack-release* p d) ($q::'b::mem\text{-}type$ ptr) = *root-ptr-valid* d q
by (*metis neq non-stack root root-ptr-valid-type-neq-disjoint stack-release-root-ptr-valid1 stack-release-root-ptr-valid2 sub sub-typ-def sub-typ-stack-byte*)

lemma *be x -impl-forall-conv*: $(\exists x \in A. P\ x) \longrightarrow Q \iff (\forall x. x \in A \longrightarrow P\ x \longrightarrow Q)$
by *auto*

end

20.3 More Stack Typing

theory *Stack-Typing*
imports *L2Defs Runs-To-VCG*
begin

definition *equal-upto*:: $'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$ **where**
equal-upto $S\ f\ g = (\forall x. x \notin S \longrightarrow f\ x = g\ x)$

lemma *equal-upto-refl*[*simp, intro*]: *equal-upto* $S\ f\ f$
by (*simp add: equal-upto-def*)

lemma *symp-equal-upto*: *symp* (*equal-upto* S)
by (*simp add: equal-upto-def symp-def*)

lemma *equal-upto-commute*: *equal-upto* $S\ f\ g \iff$ *equal-upto* $S\ g\ f$
using *symp-equal-upto*
by (*metis sympE*)

lemma *equal-upto-trans*[*trans*]: *equal-upto* $S\ f\ g \implies$ *equal-upto* $S\ g\ h \implies$ *equal-upto* $S\ f\ h$
by (*simp add: equal-upto-def*)

lemma *equal-upto-mono*: $S \subseteq T \implies$ *equal-upto* $S\ f\ g \implies$ *equal-upto* $T\ f\ g$
by (*meson equal-upto-def subset-iff*)

definition *equal-on*:: $'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$ **where**
equal-on $S\ f\ g = (\forall x. x \in S \longrightarrow f\ x = g\ x)$

lemma *equal-on-UNIV-iff*[*simp*]: *equal-on* $UNIV\ f\ g \iff f = g$
by (*auto simp add: equal-on-def*)

lemma *equal-on-refl* [*simp, intro*]: *equal-on* $S\ f\ f$

```

    by (auto simp add: equal-on-def)

lemma symp-equal-on: symp (equal-on S)
  by (simp add: equal-on-def symp-def)

lemma equal-on-commute: equal-on S f g  $\longleftrightarrow$  equal-on S g f
  using symp-equal-on
  by (metis sympE)

lemma equal-on-trans[trans]: equal-on S f g  $\implies$  equal-on S g h  $\implies$  equal-on S f h
  by (simp add: equal-on-def)

lemma equal-on-mono:  $S \subseteq T \implies$  equal-on T f g  $\implies$  equal-on S f g
  by (meson equal-on-def subset-iff)

lemma equal-on-equal-upto-eq: equal-on S f g  $\implies$  equal-upto S f g  $\implies$   $f = g$ 
  by (auto simp add: equal-on-def equal-upto-def)

named-theorems unchanged-typing-on-simps and unchanged-typing

declare mex-def [unchanged-typing-on-simps]
declare meq-def [unchanged-typing-on-simps]

ML <
  structure Unchanged-Typing =
  struct
  fun unchanged-typing-tac splitter ctxt =
    let
      val unchanged-typing-on-simps = Named-Theorems.get ctxt @{named-theorems
unchanged-typing-on-simps}
      val unchanged-typing = Named-Theorems.get ctxt @{named-theorems unchanged-typing}
      val vcg-attrs = map (Attrib.attribute ctxt) @{attributes [runs-to-vcg] }
      val (-, ctxt) = ctxt |> fold-map (Thm.proof-attributes vcg-attrs) unchanged-typing
      val ctxt = ctxt addsimps unchanged-typing-on-simps

      val trace-tac = Runs-To-VCG.no-trace-tac
      fun solver-tac ctxt = ALLGOALS (asm-full-simp-tac ctxt)
    in
      Runs-To-VCG.runs-to-vcg-tac splitter ( $\sim 1$ ) trace-tac
      {do-nosplit = false, no-unsafe-hyp-subst = false} solver-tac ctxt
    end
  end
  >

lemma stack-allocs-releases-equal-on-stack:
(p, d2)  $\in$  stack-allocs n S TYPE('a::mem-type) d1  $\implies$ 
  equal-on S d2 d3  $\implies$  equal-on S d1 (stack-releases n p d3)
  by (smt (verit, best) equal-on-def stack-releases-footprint stack-releases-other)

```

stack-releases-stack-allocs-inverse)

lemma *stack-allocs-releases-equal-on-typing*:

$(p, d2) \in \text{stack-allocs } n \mathcal{S} \text{ TYPE}('a::\text{mem-type}) d1 \implies$
 $\text{equal-on } X d2 d3 \implies \text{equal-on } X d1 (\text{stack-releases } n p d3)$

by (*smt* (*verit*, *best*) *equal-on-def stack-releases-footprint stack-releases-other stack-releases-stack-allocs-inverse*)

lemma *stack-allocs-releases-equal-on-typing'*:

$(p, d2) \in \text{stack-allocs } n \mathcal{S} \text{ TYPE}('a::\text{mem-type}) d1 \implies$
 $\text{equal-on } X' d2 d3 \implies X \subseteq X' \implies \text{equal-on } X d1 (\text{stack-releases } n p d3)$

using *stack-allocs-releases-equal-on-typing*

using *equal-on-mono* **by** *blast*

context *heap-typing-state*

begin

definition *unchanged-typing-on:: addr set \Rightarrow 's \Rightarrow 's \Rightarrow bool* **where**

unchanged-typing-on S s t = equal-on S (htd s) (htd t)

lemma *unchanged-typing-on-UNIV-iff*: *unchanged-typing-on UNIV s t \longleftrightarrow htd s = htd t*

by (*auto simp add: unchanged-typing-on-def*)

lemma *unchanged-typing-on-UNIV-I*: *htd s = htd t \implies unchanged-typing-on UNIV s t*

by (*simp add: unchanged-typing-on-UNIV-iff*)

lemma *typing-eq-unchanged-typing-on*: *htd s = htd t \implies unchanged-typing-on S s t*

by (*auto simp add: unchanged-typing-on-def equal-on-def*)

lemma *unchanged-typing-on-relf[*simp*]*: *unchanged-typing-on S s s*

by (*auto simp add: unchanged-typing-on-def equal-on-def*)

lemma *unchanged-typing-on-conv[*unchanged-typing-on-simps*]*: *unchanged-typing-on S s t \longleftrightarrow ($\forall a \in S. \text{htd } s a = \text{htd } t a$)*

by (*auto simp add: unchanged-typing-on-def equal-on-def*)

lemma *unchanged-typing-on-cong*: $(\bigwedge a. a \in S \implies \text{htd } s a = \text{htd } s' a) \implies$
 $(\bigwedge a. a \in S \implies \text{htd } t a = \text{htd } t' a)$

$\implies \text{unchanged-typing-on } S s t \longleftrightarrow \text{unchanged-typing-on } S s' t$

by (*simp add: unchanged-typing-on-def equal-on-def*)

lemma *typing-eq-left-unchanged-typing-on*: *htd s1 = htd s2 \implies*

unchanged-typing-on S s1 t \longleftrightarrow unchanged-typing-on S s2 t

by (*auto simp add: unchanged-typing-on-def equal-on-def*)

```

lemma symp-unchanged-typing-on: symp (unchanged-typing-on S)
  using symp-equal-on
  by (metis symp-def unchanged-typing-on-def)

lemma unchanged-typing-on-commute: unchanged-typing-on S s t  $\longleftrightarrow$  unchanged-typing-on
S t s
  using symp-unchanged-typing-on
  by (metis sympE)

lemma unchanged-typing-on-trans[trans]:
  assumes s-t: unchanged-typing-on S s t
  assumes t-u: unchanged-typing-on S t u
  shows unchanged-typing-on S s u
  by (meson equal-on-trans s-t t-u unchanged-typing-on-def)

lemma unchanged-typing-on-mono:  $S \subseteq T \implies$  unchanged-typing-on T s t  $\implies$ 
unchanged-typing-on S s t
  using equal-on-mono
  by (smt (verit) unchanged-typing-on-def)

lemma unchanged-typing-on-root-ptr-valid-preservation:
  fixes p::'a::mem-type ptr
  assumes unchanged-typing-on S s t
  assumes ptr-span  $p \subseteq S$ 
  assumes root-ptr-valid (htd s) p
  shows root-ptr-valid (htd t) p
  using assms
  by (simp add: equal-on-def root-ptr-valid-domain subset-iff unchanged-typing-on-def)

simproc-setup unchanged-typing-nondet ( $\langle c \cdot s \ ?\{\lambda r t. \text{unchanged-typing-on } S s$ 
t\}\rangle = \langle
  let
    val prop-to-meta-eq =  $\@{\text{lemma } \langle P \implies (P \equiv \text{True}) \rangle \text{ for } P \text{ by auto}}$ 
    fun try-prove ctxt prop = try (Goal.prove-internal ctxt  $\[]$  prop)
      (fn - => Unchanged-Typing.unchanged-typing-tac NONE ctxt)
  in
    fn phi => fn ctxt => fn ct =>
      let
        val - = Utils.verbose-msg 3 ctxt (fn - => unchanged-typing-nondet invoked
on:  $\^{\@{\text{make-string}}}$  ct)
      in
        try-prove ctxt instantiate  $\langle P = ct \text{ in } cterm \langle \text{Trueprop } P \rangle \text{ for } P \rangle$ 
|> Option.map (fn thm => prop-to-meta-eq OF [thm])
      end

  end
   $\rangle$ 
declare [simproc del: unchanged-typing-nondet]

```

end

context *stack-heap-state*

begin

lemma *with-fresh-stack-ptr-unchanged-typing*[*unchanged-typing*]:

assumes $f[\text{runs-to-vcg}]: \bigwedge p s. (f p) \cdot s \text{ ?}\{\lambda r t. \text{typing.unchanged-typing-on } S s t\}$

shows $(\text{with-fresh-stack-ptr } n I (L2\text{-VARS } f nm)) \cdot s \text{ ?}\{\lambda r t. \text{typing.unchanged-typing-on } S s t\}$

unfolding *with-fresh-stack-ptr-def on-exit-def on-exit'-def L2-VARS-def*

apply (*runs-to-vcg*)

subgoal for $x d vs r t$

thm *typing.typing-eq-left-unchanged-typing-on*

thm *typing.typing-eq-left-unchanged-typing-on* [*of - (htd-upd* ($\lambda \cdot. d$) s)]

apply (*subst* (*asm*) *typing.typing-eq-left-unchanged-typing-on* [*of - (htd-upd* ($\lambda \cdot. d$) s)])

apply *simp*

apply (*simp add: typing.unchanged-typing-on-def stack-allocs-releases-equal-on-stack*)

done

done

end

lemma *override-on-merge*: $\text{override-on } (\text{override-on } f g B) g A = \text{override-on } f g (A \cup B)$

by (*auto simp add: override-on-def fun-eq-iff*)

lemma *override-on-id* [*simp*]: $\text{override-on } f f A = f$

by (*auto simp add: override-on-def*)

lemma *override-cancel-snd* [*simp*]: $\text{override-on } f (\text{override-on } g1 g0 A) A = \text{override-on } f g0 A$

by (*auto simp add: override-on-def*)

lemma *override-cancel-subset-snd* : $A \subseteq B \implies \text{override-on } f (\text{override-on } g1 g0 B) A = \text{override-on } f g0 A$

by (*auto simp add: override-on-def fun-eq-iff*)

lemma *override-cancel-fst*[*simp*] : $\text{override-on } (\text{override-on } f1 f0 A) g A = \text{override-on } f1 g A$

by (*auto simp add: override-on-def fun-eq-iff*)

lemma *override-cancel-subset-fst* : $A \subseteq B \implies \text{override-on } (\text{override-on } f1 f0 A) g B = \text{override-on } f1 g B$

by (*auto simp add: override-on-def fun-eq-iff*)

lemma *equal-on-stack-free-preservation*:

assumes *eq*: $\text{equal-on } S d2 d1$

assumes *stack-free* $d1 \subseteq S$

assumes *stack-free* $d2 \subseteq S$

shows $\text{stack-free } d2 = \text{stack-free } d1$

using *assms*
by (*smt* (*verit*, *best*) *Abs-fnat-hom-0 Collect-cong One-nat-def add.right-neutral*
equal-on-def less-Suc0 mem-Collect-eq ptr-val.simps
root-ptr-valid-def size-of-stack-byte(3) stack-free-def subsetD valid-root-footprint-def)

lemma *equal-upto-disjoint-h-val*: *equal-upto P h' h* \implies
ptr-span p \cap *P* = {} \implies
h-val h' (p::'a::mem-type ptr) = *h-val h p*
by (*smt* (*verit*, *best*) *disjoint-iff equal-upto-def h-val-def heap-list-h-eq2*)

context *heap-state*
begin

definition *equal-upto-heap-on*:: *addr set* \Rightarrow *'s* \Rightarrow *'s* \Rightarrow *bool* **where**
equal-upto-heap-on S s t = (\exists *T H*.
t = *hmem-upd* (λ -. *H*) (*htd-upd* (λ -. *T*) *s*) \wedge
equal-upto S (*hmem s*) *H* \wedge
equal-upto S (*htd s*) *T*)

definition *zero-heap-on*:: *addr set* \Rightarrow *'s* \Rightarrow *'s* **where**
zero-heap-on A s = *hmem-upd* (λ *h*. *override-on h zero-heap A*) (*htd-upd* (λ *htd*.
override-on htd stack-byte-typing A) *s*)

lemma *equal-upto-heap-on-equal-upto-htd*: *equal-upto-heap-on S s t* \implies *equal-upto*
S (*htd s*) (*htd t*)
by (*auto simp add: equal-upto-heap-on-def*)

lemma *equal-upto-heap-on-equal-upto-hmem*: *equal-upto-heap-on S s t* \implies *equal-upto*
S (*hmem s*) (*hmem t*)
by (*auto simp add: equal-upto-heap-on-def*)

lemma *zero-heap-on-empty[simp]*: *zero-heap-on* {} *s* = *s*
by (*auto simp add: zero-heap-on-def*)

lemma *equal-upto-heap-on-zero-heap-on-subset*: *A* \subseteq *B* \implies *equal-upto-heap-on B*
s (*zero-heap-on A s*)
apply (*clarsimp simp add: zero-heap-on-def equal-upto-heap-on-def override-on-def*)
by (*smt* (*verit*, *best*) *equal-upto-def equal-upto-mono heap.upd-cong hmem-htd-upd*
typing.upd-cong)

lemma *equal-upto-heap-on-zero-heap-on[simp]*: *equal-upto-heap-on A s* (*zero-heap-on*
A s)
by (*simp add: equal-upto-heap-on-zero-heap-on-subset*)

lemma *equal-upto-heap-on-refl[simp, intro]*: *equal-upto-heap-on S s s*
by (*force simp add: equal-upto-heap-on-def*)

lemma *override-on-heap-update-other*:
fixes $p::'a::\text{mem-type ptr}$
shows $\text{ptr-span } p \subseteq A \implies \text{override-on } (\text{heap-update } p \ v \ h) \ f \ A = \text{override-on } h \ f \ A$
apply (*clarsimp simp add: override-on-def fun-eq-iff*)
by (*smt (verit, ccfv-SIG) CTypes.mem-type-simps(2) heap-list-length heap-update-def heap-update-nmem-same subsetD*)

lemma *override-on-heap-update-commute*:
fixes $p::'a::\text{mem-type ptr}$
shows $\text{ptr-span } p \cap A = \{\} \implies \text{override-on } (\text{heap-update } p \ v \ h) \ f \ A = \text{heap-update } p \ v \ (\text{override-on } h \ f \ A)$
apply (*clarsimp simp add: override-on-def fun-eq-iff heap-update-def heap-update-nmem-same orthD2*)
by (*smt (verit, best) disjoint-iff heap-list-h-eq2 heap-list-length heap-update-list-value len max-size*)

lemma *override-on-heap-update-padding-other*:
fixes $p::'a::\text{mem-type ptr}$
shows $\text{ptr-span } p \subseteq A \implies \text{length } bs = \text{size-of } \text{TYPE}('a) \implies \text{override-on } (\text{heap-update-padding } p \ v \ bs \ h) \ f \ A = \text{override-on } h \ f \ A$
apply (*clarsimp simp add: override-on-def fun-eq-iff*)
by (*smt (verit) CTypes.mem-type-simps(2) heap-update-nmem-same heap-update-padding-def subsetD*)

lemma *zero-heap-on-heap-update-other*:
fixes $p::'a::\text{mem-type ptr}$
shows $\text{ptr-span } p \subseteq A \implies \text{zero-heap-on } A \ (\text{hmem-upd } (\text{heap-update } p \ v) \ s) = \text{zero-heap-on } A \ s$
by (*simp add: zero-heap-on-def heap-commute comp-def override-on-heap-update-other*)

lemma *override-on-stack-byte-typing-stack-allocs*:
assumes $\text{alloc}: (p, d) \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a::\text{mem-type}) \ d0$
assumes $\text{subset}: \text{stack-free } d0 \subseteq A$
shows $\text{override-on } d \ \text{stack-byte-typing } A = \text{override-on } d0 \ \text{stack-byte-typing } A$
using alloc subset
apply (*simp add: override-on-def fun-eq-iff*)
by (*meson in-mono stack-allocs-other stack-allocs-stack-subset-stack-free*)

lemma *zero-heap-on-stack-allocs*:
assumes $\text{alloc}: (p, d) \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a::\text{mem-type}) \ (\text{htd } s)$
assumes $\text{subset}: \text{stack-free } (\text{htd } s) \subseteq A$
shows $\text{zero-heap-on } A \ (\text{htd-upd } (\lambda-. d) \ s) = \text{zero-heap-on } A \ s$
apply (*simp add: zero-heap-on-def comp-def*)
using $\text{override-on-stack-byte-typing-stack-allocs} \ [OF \ \text{alloc subset}]$
by (*metis (no-types, lifting) typing.upd-cong*)

lemma *zero-heap-on-stack-releases*:

fixes p : 'a::mem-type ptr
assumes $subset$: $\{ptr\text{-val } p..+n * \text{size-of } TYPE('a::mem\text{-type})\} \subseteq A$
shows $zero\text{-heap-on } A (h\text{td-upd } (stack\text{-releases } n \ p) \ s) = zero\text{-heap-on } A \ s$
using $subset$
by (*simp add: zero-heap-on-def comp-def stack-releases-def override-on-merge Un-absorb2*)

lemma *zero-heap-on-stack-releases'*:

fixes p : 'a::mem-type ptr
assumes $guard$: $(\bigwedge i. i < n \implies c\text{-null-guard } (p +_p \text{int } i))$
shows $zero\text{-heap-on } (stack\text{-free } (stack\text{-releases } n \ p \ (h\text{td } s)) \cup A) (h\text{td-upd } (stack\text{-releases } n \ p) \ s) =$
 $zero\text{-heap-on } (stack\text{-free } (h\text{td } s) \cup \{ptr\text{-val } p..+n * \text{size-of } TYPE('a)\} \cup A) \ s$
apply (*subst stack-free-stack-releases*)
apply (*simp add: guard*)
apply (*subst zero-heap-on-stack-releases*)
apply *blast*
apply (*simp add: sup-commute*)
done

We want to express that certain portions of the heap (typing and values) are 'irrelevant', in particular regarding (free) stack space. The notion of 'irrelevant' is a bit vague, it means that the behaviour of the program does not depend on those locations and also that it does not modify those locations. Moreover, we prefer an abstraction function rather than an relation between states to avoid admissibility issues for refinement.

definition *frame:: addr set \Rightarrow 's \Rightarrow 's \Rightarrow 's* **where**

frame $A \ s_0 \ s =$
 $hmem\text{-upd } (\lambda h. \text{override-on } h \ (hmem \ s_0) \ (A \cup \text{stack-free } (h\text{td } s)))$
 $(h\text{td-upd } (\lambda d. \text{override-on } d \ (h\text{td } s_0) \ (A - \text{stack-free } (h\text{td } s))) \ s)$

The standard use case we have in mind is $A \cap \text{stack-free } (h\text{td } s) = \{\}$, hence $A - \text{stack-free } (h\text{td } s) = A$, but nevertheless the intuition is:

- stack free typing from s is preserved, the framed sate has at least as many stack free addresses as the original one. So we can simulate any stack allocation.
- heap values for stack free and A are taken from reference state s_0 , this captures that we do not depend on the original values in s for those addresses.
- typing for allocations in A is taken from reference state s_0 , this captures that we do not depend on the original typing in s for those addresses.

By taking the same reference state s_0 to frame two states s and s' , we can express that the 'irrelevant' parts of the heap did not change in the respective framed states $\text{frame } A \ s_0 \ s$ and $\text{frame } A \ s_0 \ s'$.

definition *raw-frame*:: $\text{addr set} \Rightarrow 's \Rightarrow 's \Rightarrow 's$ **where**

raw-frame $A \ s_0 \ s =$
 $\text{hmem-upd } (\lambda h. \text{override-on } h \ (\text{hmem } s_0) \ A)$
 $(\text{htd-upd } (\lambda d. \text{override-on } d \ (\text{htd } s_0) \ A) \ s)$

lemma *frame-heap-independent-selector*:

$(\bigwedge f \ s. \text{sel } (\text{hmem-upd } f \ s) = \text{sel } s) \implies (\bigwedge f \ s. \text{sel } (\text{htd-upd } f \ s) = \text{sel } s) \implies$
 $\text{sel } (\text{frame } A \ s_0 \ s) = \text{sel } s$
by (*auto simp add: frame-def*)

lemma *frame-id[simp]*: $\text{frame } A \ s \ s = s$

unfolding *frame-def*
by (*simp add: heap.upd-same typing.upd-same*)

lemma *frame-hmem-UNIV[simp]*: $\text{hmem } (\text{frame } \text{UNIV} \ s_0 \ s) = \text{hmem } s_0$

unfolding *frame-def*
by *simp*

lemma *hmem-frame*: $\text{hmem } (\text{frame } A \ s_0 \ s) = (\lambda a. \text{if } a \in A \cup \text{stack-free } (\text{htd } s) \text{ then } \text{hmem } s_0 \ a \ \text{else } \text{hmem } s \ a)$

unfolding *frame-def*
by (*auto simp add: fun-eq-iff*)

lemma *htd-frame*: $\text{htd } (\text{frame } A \ s_0 \ s) = (\lambda a. \text{if } a \in (A - \text{stack-free } (\text{htd } s)) \text{ then } \text{htd } s_0 \ a \ \text{else } \text{htd } s \ a)$

unfolding *frame-def*
by (*auto simp add: fun-eq-iff*)

lemma *stack-free-htd-frame*: $\text{stack-free } (\text{htd } s) \subseteq \text{stack-free } (\text{htd } (\text{frame } A \ s_0 \ s))$

apply (*clarsimp simp add: htd-frame stack-free-def root-ptr-valid-def*)
by (*smt (verit) One-nat-def add.right-neutral less-Suc0 mem-Collect-eq orthD2 semiring-1-class.of-nat-0 size-of-stack-byte(3) valid-root-footprint-def*)

lemma *stack-free-htd-frame'*: $\text{stack-free } (\text{htd } s_0) \cap A = \{\} \implies \text{stack-free } (\text{htd } (\text{frame } A \ s_0 \ s)) = \text{stack-free } (\text{htd } s)$

by (*auto simp add: htd-frame stack-free-def root-ptr-valid-def valid-root-footprint-def*)

lemma *equal-on-hmem-frame*: $\text{equal-on } (A \cup \text{stack-free } (\text{htd } s)) \ (\text{hmem } (\text{frame } A \ s_0 \ s)) \ (\text{hmem } s_0)$

by (*auto simp add: equal-on-def frame-def equal-upto-def override-on-def fun-eq-iff*)

lemma *equal-upto-hmem-frame*: $\text{equal-upto } (A \cup \text{stack-free } (\text{htd } s)) \ (\text{hmem } (\text{frame } A \ s_0 \ s)) \ (\text{hmem } s)$

by (*auto simp add: equal-on-def frame-def equal-upto-def override-on-def fun-eq-iff*)

lemma *equal-on-htd-frame*: $stack\text{-}free\ (htd\ s) \cap A = \{\} \implies equal\text{-}on\ A\ (htd\ (frame\ A\ s_0\ s))\ (htd\ s_0)$
by (*auto simp add: equal-on-def frame-def equal-upto-def override-on-def fun-eq-iff*)

lemma *equal-on-htd-stack-free-frame*: $stack\text{-}free\ (htd\ s) \cap A = \{\} \implies$
 $equal\text{-}on\ (stack\text{-}free\ (htd\ s))\ (htd\ (frame\ A\ s_0\ s))\ (htd\ s)$
by (*auto simp add: equal-on-def frame-def equal-upto-def override-on-def fun-eq-iff*)

lemma *equal-upto-htd-frame*: $equal\text{-}upto\ A\ (htd\ (frame\ A\ s_0\ s))\ (htd\ s)$
by (*auto simp add: equal-on-def frame-def equal-upto-def override-on-def fun-eq-iff*)

lemma *equal-upto-heap-on-frame*: $equal\text{-}upto\text{-}heap\text{-}on\ (A \cup stack\text{-}free\ (htd\ s))\ (frame\ A\ s_0\ s)\ s$
apply (*clarsimp simp add: equal-upto-heap-on-def frame-def equal-upto-def override-on-def fun-eq-iff*)
by (*smt (verit, ccfv-threshold) K-record-comp heap.upd-compose heap.upd-same heap-commute typing.upd-compose typing.upd-same*)

lemma *frame-heap-update*:
fixes $p::'a::mem\text{-}type\ ptr$
shows $ptr\text{-}span\ p \subseteq A \implies frame\ A\ s_0\ (hmem\text{-}upd\ (heap\text{-}update\ p\ v)\ s) = frame\ A\ s_0\ s$
apply (*clarsimp simp add: frame-def*)
apply (*subst heap-commute [symmetric]*)
apply (*simp add: comp-def*)
apply (*subst override-on-heap-update-other*)
apply *auto*
done

lemma *frame-heap-update-disjoint*:
fixes $p::'a::mem\text{-}type\ ptr$
shows $ptr\text{-}span\ p \cap A = \{\} \implies ptr\text{-}span\ p \cap stack\text{-}free\ (htd\ s) = \{\} \implies$
 $frame\ A\ s_0\ (hmem\text{-}upd\ (heap\text{-}update\ p\ v)\ s) = hmem\text{-}upd\ (heap\text{-}update\ p\ v)\ (frame\ A\ s_0\ s)$
apply (*clarsimp simp add: frame-def*)
apply (*subst heap-commute [symmetric]*)
apply (*simp add: comp-def*)
apply (*subst override-on-heap-update-commute*)
apply *blast*
apply *blast*
done

lemma *h-val-frame-disjoint'*:
fixes $p::'a::mem\text{-}type\ ptr$
shows $ptr\text{-}span\ p \cap A = \{\} \implies ptr\text{-}span\ p \cap stack\text{-}free\ (htd\ s) = \{\} \implies$
 $ptr\text{-}span\ p = ptr\text{-}span\ p' \implies$
 $h\text{-}val\ (hmem\ (frame\ A\ s_0\ s))\ p' = h\text{-}val\ (hmem\ s)\ p'$
apply (*clarsimp simp add: frame-def override-on-def*)

by (smt (verit, del-insts) disjoint-iff h-val-def heap-list-h-eq2 hmem-htd-upd)

lemma *h-val-frame-disjoint*:

fixes $p::'a::\text{mem-type ptr}$

shows $\text{ptr-span } p \cap A = \{\} \implies \text{ptr-span } p \cap \text{stack-free (htd } s) = \{\} \implies$

$\text{h-val (hmem (frame } A \ s_0 \ s)) } p = \text{h-val (hmem } s) } p$

apply (erule (1) *h-val-frame-disjoint'*)

by (rule *refl*)

lemma *h-val-frame-disjoint-globals*:

fixes $p::'a::\text{mem-type ptr}$

assumes $\mathcal{G} \cap A = \{\} \ \mathcal{G} \cap \text{stack-free (htd } s) = \{\}$

assumes $\text{ptr-span } p \subseteq \mathcal{G}$

shows $\text{h-val (hmem (frame } A \ s_0 \ s)) } p = \text{h-val (hmem } s) } p$

apply (rule *h-val-frame-disjoint*)

subgoal using *assms* by *blast*

subgoal using *assms* by *blast*

done

lemma *frame-heap-update-padding*:

fixes $p::'a::\text{mem-type ptr}$

shows $\text{ptr-span } p \subseteq A \implies \text{length } bs = \text{size-of TYPE('a)} \implies \text{frame } A \ s_0$

$(\text{hmem-upd (heap-update-padding } p \ v \ bs) \ s) = \text{frame } A \ s_0 \ s$

apply (clarsimp simp add: *frame-def*)

apply (subst *heap-commute [symmetric]*)

apply (simp add: *comp-def*)

apply (subst *override-on-heap-update-padding-other*)

apply *auto*

done

lemma *frame-stack-alloc*:

fixes $p::'a::\text{mem-type ptr}$

assumes *subset*: $\text{ptr-span } p \subseteq \text{stack-free (htd } s)$

assumes *disjoint*: $\text{stack-free (htd } s) \cap A = \{\}$

assumes *alloc*: $\text{stack-free (ptr-force-type } p \ (\text{htd } s)) = \text{stack-free (htd } s) - \text{ptr-span}$

p

shows

$\text{frame (ptr-span } p \cup A) (\text{htd-upd } (\lambda d. \text{override-on } d \ \text{stack-byte-typing (ptr-span } p)) \ t_0)$

$(\text{htd-upd } (\lambda -. \text{ptr-force-type } p \ (\text{htd } s)) \ s) = \text{frame } A \ t_0 \ s$

proof –

from *alloc subset disjoint*

have *: $\text{ptr-span } p \cup A \cup \text{stack-free (ptr-force-type } p \ (\text{htd } s)) = A \cup \text{stack-free (htd } s)$

by *auto*

from *subset disjoint alloc*

have **: *override-on (ptr-force-type } p \ (\text{htd } s))*

$(\text{override-on (htd } t_0) \ \text{stack-byte-typing (ptr-span } p)) (\text{ptr-span } p \cup A - \text{stack-free (ptr-force-type } p \ (\text{htd } s))) = \text{override-on (htd } s) \ (\text{htd } t_0) (A - \text{stack-free$

```

(htd s)
  apply (clarsimp simp add: override-on-def fun-eq-iff ptr-force-type-d)
  by (smt (verit, ccfv-threshold) add.right-neutral mem-Collect-eq old.prod.exhaust

      prod.sel(1) prod.sel(2) ptr-val.ptr-val-def root-ptr-valid-def semiring-1-class.of-nat-0
      stack-byte-typing-footprint stack-free-def subset-iff valid-root-footprint-def)
  show ?thesis
  apply (clarsimp simp add: frame-def comp-def * **)
  by (metis (no-types, lifting) typing.upd-cong)
qed

```

lemma *frame-stack-release*:

```

  fixes p::'a::mem-type ptr
  assumes disjoint-free: ptr-span p  $\cap$  stack-free (htd s) = {}
  assumes disjoint-old: ptr-span p  $\cap$  A = {}
  assumes c-null-guard: c-null-guard p
  assumes lbs: length bs = size-of TYPE ('a)
  shows frame (ptr-span p  $\cup$  A) (htd-upd ( $\lambda d$ . override-on d stack-byte-typing
    (ptr-span p)) t0) s =
    frame A t0 (hmem-upd (heap-update-list (ptr-val p) bs) (htd-upd (stack-releases
    (Suc 0) p) s))
proof –
  from disjoint-free c-null-guard
  have *: (ptr-span p  $\cup$  A  $\cup$  stack-free (htd s)) = A  $\cup$  stack-free (stack-releases
    (Suc 0) p (htd s))
  by (metis (no-types, opaque-lifting) Un-assoc add.right-neutral mult-0 stack-free-stack-release

```

stack-release-def stack-releases-def sup-commute times-nat.simps(2))

```

from disjoint-free disjoint-old c-null-guard
have **: override-on (htd s) (override-on (htd t0) stack-byte-typing (ptr-span p))
  (ptr-span p  $\cup$  A – stack-free (htd s)) =
  override-on (stack-releases (Suc 0) p (htd s)) (htd t0) (A – stack-free
  (stack-releases (Suc 0) p (htd s)))
  apply (clarsimp simp add: override-on-def fun-eq-iff stack-byte-typing-footprint
  stack-releases-footprint stack-releases-other,
    intro conjI impI)
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal by (metis UnE add.right-neutral mult-is-0 stack-free-stack-release
  stack-release-def stack-releases-def times-nat.simps(2))
  using stack-free-stack-releases-mono by blast

```

```

have ***: override-on (heap-update-list (ptr-val p) bs (hmem s)) (hmem t0) (A
 $\cup$  stack-free (stack-releases (Suc 0) p (htd s)))
  =
  override-on (hmem s) (hmem t0) (A  $\cup$  stack-free (stack-releases (Suc 0) p

```

```

(htd s)))
  using disjoint-old lbs *
  apply (clarsimp simp add: override-on-def fun-eq-iff stack-byte-typing-footprint
stack-releases-footprint stack-releases-other)
  by (metis Un-iff heap-update-nmem-same lbs)

show ?thesis
  apply (simp add: frame-def comp-def *)
  apply (simp add: heap-commute)
  using ** ***
  by (metis (no-types, lifting) heap.get-upd heap.upd-compose heap.upd-cong htd-hmem-upd

      typing.get-upd typing.upd-compose typing.upd-cong)
qed

lemma frame-stack-release-keep:
  fixes p::'a::mem-type ptr
  assumes disjoint-free: ptr-span p  $\cap$  stack-free (htd s) = {}
  assumes disjoint-old: ptr-span p  $\cap$  A = {}
  assumes c-null-guard: c-null-guard p
  assumes lbs: length bs = size-of TYPE('a)
  shows hmem-upd (heap-update-list (ptr-val p) (heap-list (hmem t0) (size-of TYPE('a))
(ptr-val p)))
    ( htd-upd (stack-releases (Suc 0) p) (frame A t0 s) ) =
      frame A t0
        (hmem-upd (heap-update-list (ptr-val p) bs)
          (htd-upd (stack-releases (Suc 0) p) s))

proof -
  from disjoint-free c-null-guard
  have *: (ptr-span p  $\cup$  A  $\cup$  stack-free (htd s)) = A  $\cup$  stack-free (stack-releases
(Suc 0) p (htd s))
  by (metis (no-types, opaque-lifting) Un-assoc add.right-neutral mult-0 stack-free-stack-release

      stack-release-def stack-releases-def sup-commute times-nat.simps(2))

  from disjoint-free disjoint-old c-null-guard
  have **: stack-releases (Suc 0) p (override-on (htd s) (htd t0) (A - stack-free
(htd s))) =
    override-on (stack-releases (Suc 0) p (htd s)) (htd t0) (A - stack-free
(stack-releases (Suc 0) p (htd s)))
  apply (clarsimp simp add: override-on-def fun-eq-iff stack-byte-typing-footprint
stack-releases-footprint stack-releases-other,
    intro conjI impI)
  subgoal by (smt (verit, del-insts) Un-iff add.right-neutral mult-is-0 stack-free-stack-release

      stack-release-def stack-release-other stack-releases-def times-nat.simps(2))
  subgoal by (smt (verit, best) UnE less-Suc0 ptr-add-0-id semiring-1-class.of-nat-0
stack-free-stack-releases

```



```

      stack-releases-footprint stack-releases-other)
    done

  have ***: heap-update-list (ptr-val p) (heap-list (hmem t0) (size-of TYPE('a))
(ptr-val p))
      (override-on (hmem s) (hmem t0) (A ∪ stack-free (htd s))) =
      override-on (heap-update-list (ptr-val p) bs (hmem s)) (hmem t0)
      (A ∪ stack-free (stack-releases (Suc 0) p (htd s)))
  using lbs disjoint-old disjoint-free *
  apply (clarsimp simp add: override-on-def fun-eq-iff heap-update-nmem-same
orthD2 heap-update-list-value )
  apply (smt (verit, ccfv-SIG) Un-iff heap-list-length heap-update-list-id' heap-update-list-value
max-size)
  done
  show ?thesis
  apply (simp add: frame-def comp-def * heap-commute )
  using ** ***
  by (metis (no-types, lifting) heap.upd-cong htd-hmem-upd lense.upd-cong typ-
ing.lense-axioms)
  qed

```

lemma *symp-equal-upto-heap-on*: *symp (equal-upto-heap-on S)*

proof

fix *s t*

assume *equal-upto-heap-on S s t*

then obtain *T H* **where**

t: *t = hmem-upd (λ-. H) (htd-upd (λ-. T) s)* **and**

eq-H: *equal-upto S (hmem s) H* **and**

eq-T: *equal-upto S (htd s) T*

by (*auto simp add: equal-upto-heap-on-def*)

show *equal-upto-heap-on S t s*

proof –

define *H'* **where** *H' = hmem s*

define *T'* **where** *T' = htd s*

have *s = hmem-upd (λ-. H') (htd-upd (λ-. T') t)*

unfolding *H'-def T'-def*

apply (*subst t*)

apply (*simp add: heap-commute heap.upd-same typing.upd-same*)

done

moreover

from *eq-H* **have** *equal-upto S (hmem t) H'*

by (*simp add: H'-def equal-upto-commute t*)

moreover

from *eq-T* **have** *equal-upto S (htd t) T'*

by (*simp add: T'-def equal-upto-commute t*)

ultimately show ?thesis

by (*auto simp add: equal-upto-heap-on-def*)

qed

qed

lemma *equal-upto-heap-on-commute*: *equal-upto-heap-on S s t* \longleftrightarrow *equal-upto-heap-on S s t*

using *symp-equal-upto-heap-on*

by (*metis sympE*)

lemma *equal-upto-heap-on-trans*[*trans*]:

assumes *s-t*: *equal-upto-heap-on S s t*

assumes *t-u*: *equal-upto-heap-on S t u*

shows *equal-upto-heap-on S s u*

proof –

from *s-t* **obtain** *T H* **where**

t: *t = hmem-upd* (λ -. *H*) (*htd-upd* (λ -. *T*) *s*) **and**

eq-H: *equal-upto S* (*hmem s*) *H* **and**

eq-T: *equal-upto S* (*htd s*) *T*

by (*auto simp add: equal-upto-heap-on-def*)

from *t-u* **obtain** *T' H'* **where**

u: *u = hmem-upd* (λ -. *H'*) (*htd-upd* (λ -. *T'*) *t*) **and**

eq-H': *equal-upto S* (*hmem t*) *H'* **and**

eq-T': *equal-upto S* (*htd t*) *T'*

by (*auto simp add: equal-upto-heap-on-def*)

have *u = hmem-upd* (λ -. *H'*) (*htd-upd* (λ -. *T'*) *s*)

apply (*subst u*)

apply (*subst t*)

apply (*simp add: heap-commute comp-def*)

done

moreover

from *eq-H'*

have *equal-upto S* (*hmem s*) *H'*

apply –

apply (*subst (asm) t*)

apply *simp*

using *eq-H equal-upto-trans* **by** *blast*

moreover

from *eq-T'*

have *equal-upto S* (*htd s*) *T'*

apply –

apply (*subst (asm) t*)

apply *simp*

using *eq-T equal-upto-trans* **by** *blast*

ultimately show *?thesis*

by (*auto simp add: equal-upto-heap-on-def*)

qed

lemma *equal-upto-heap-on-mono*: $S \subseteq T \implies \text{equal-upto-heap-on } S \ s \ t \implies \text{equal-upto-heap-on } T \ s \ t$
using *equal-upto-mono*
by (*smt (verit) equal-upto-heap-on-def*)
end

lemma *runs-to-partial-L2-modify*[*runs-to-vcg*]:
 $(L2\text{-modify } f) \cdot s \ ?\{ \lambda r \ t. \ r = \text{Result } () \wedge t = f \ s \}$
unfolding *L2-defs*
apply *runs-to-vcg*
done

lemma *runs-to-partial-L2-unknown*[*runs-to-vcg*]:
 $(\bigwedge x. P (\text{Result } x) \ s) \implies (L2\text{-unknown } ns) \cdot s \ ?\{ P \}$
unfolding *L2-defs*
apply *runs-to-vcg*
apply *simp*
done

lemma *runs-to-partial-L2-seq*[*runs-to-vcg*]:
 $f \cdot s \ ?\{ \lambda r \ t. (\forall a. r = \text{Result } a \implies g \ a \cdot t \ ?\{ Q \}) \wedge (\forall e. r = \text{Exn } e \implies Q (\text{Exn } e) \ t) \} \implies$
 $(L2\text{-seq } f \ g) \cdot s \ ?\{ Q \}$
unfolding *L2-defs*
apply *runs-to-vcg*
done

lemma (*in heap-typing-state*) *runs-to-partial-L2-seq*[*unchanged-typing*]:
assumes [*runs-to-vcg*]: $f \cdot s \ ?\{ \lambda r \ t. \ \text{unchanged-typing-on } S \ s \ t \}$
assumes [*runs-to-vcg*]: $\bigwedge r \ s. (g \ r) \cdot s \ ?\{ \lambda r \ t. \ \text{unchanged-typing-on } S \ s \ t \}$
shows $(L2\text{-seq } f \ g) \cdot s \ ?\{ \lambda r \ t. \ \text{unchanged-typing-on } S \ s \ t \}$
by (*runs-to-vcg*) (*auto simp add: unchanged-typing-on-simps*)

lemma *runs-to-partial-L2-gets*[*runs-to-vcg*]:
 $P (\text{Result } (f \ s)) \ s \implies (L2\text{-gets } f \ ns) \cdot s \ ?\{ P \}$
unfolding *L2-defs*
apply *runs-to-vcg*
done

lemma *runs-to-partial-L2-condition*[*runs-to-vcg*]:
 $(P \ s \implies f \cdot s \ ?\{ Q \}) \implies (\neg P \ s \implies g \cdot s \ ?\{ Q \}) \implies (L2\text{-condition } P \ f \ g) \cdot s \ ?\{ Q \}$
unfolding *L2-defs*
apply *runs-to-vcg*
apply *auto*

done

lemma *runs-to-partial-L2-catch*[*runs-to-vcg*]:

assumes [*runs-to-vcg*]: $f \cdot s \text{ ?}\{\lambda r t. (\forall e. r = \text{Exn } e \longrightarrow ((g \ e) \cdot t \text{ ?}\{P\})\}) \wedge$
 $(\forall v'. r = \text{Result } v' \longrightarrow P (\text{Result } v') t) \}$
shows (*L2-catch* *f g*) $\cdot s \text{ ?}\{P\}$
unfolding *L2-defs*
using *assms*
apply *runs-to-vcg*
done

lemma (in *heap-typing-state*) *runs-to-partial-L2-catch*[*unchanged-typing*]:

assumes [*runs-to-vcg*]: $f \cdot s \text{ ?}\{\lambda r t. \text{unchanged-typing-on } S \ s \ t\}$
assumes [*runs-to-vcg*]: $\bigwedge r s. (g \ r) \cdot s \text{ ?}\{\lambda r t. \text{unchanged-typing-on } S \ s \ t\}$
shows (*L2-catch* *f g*) $\cdot s \text{ ?}\{\lambda r t. \text{unchanged-typing-on } S \ s \ t\}$
by (*runs-to-vcg*) (*auto simp add: unchanged-typing-on-simps*)

lemma *runs-to-partial-L2-try*[*runs-to-vcg*]:

assumes [*runs-to-vcg*]:
 $f \cdot s \text{ ?}\{\lambda r. Q (\text{unnest-exn } r) \}$
shows (*L2-try* *f*) $\cdot s \text{ ?}\{Q\}$
unfolding *L2-defs*
apply *runs-to-vcg*
done

lemma (in *heap-typing-state*) *runs-to-partial-L2-try*[*unchanged-typing*]:

assumes [*runs-to-vcg*]: $f \cdot s \text{ ?}\{\lambda r t. \text{unchanged-typing-on } S \ s \ t\}$
shows (*L2-try* *f*) $\cdot s \text{ ?}\{\lambda r t. \text{unchanged-typing-on } S \ s \ t\}$
by (*runs-to-vcg*)

lemma *runs-to-partial-L2-while*:

assumes *B*: $\bigwedge r s. I (\text{Result } r) \ s \Longrightarrow C \ r \ s \Longrightarrow (B \ r) \cdot s \text{ ?}\{I\}$
assumes *Qr*: $\bigwedge r s. I (\text{Result } r) \ s \Longrightarrow \neg C \ r \ s \Longrightarrow Q (\text{Result } r) \ s$
assumes *Ql*: $\bigwedge e s. I (\text{Exn } e) \ s \Longrightarrow Q (\text{Exn } e) \ s$
assumes *I*: $I (\text{Result } r) \ s$
shows (*L2-while* *C B r ns*) $\cdot s \text{ ?}\{Q\}$
unfolding *L2-defs*
by (*rule runs-to-partial-whileLoop-exn [where I=I, OF I Qr Ql B]*)

lemma *runs-to-partial-L2-while-same-post*:

assumes *B*: $\bigwedge r s. Q (\text{Result } r) \ s \Longrightarrow C \ r \ s \Longrightarrow (B \ r) \cdot s \text{ ?}\{Q\}$
assumes *I*: $Q (\text{Result } r) \ s$
shows (*L2-while* *C B r ns*) $\cdot s \text{ ?}\{Q\}$
using *assms*
by – (*rule runs-to-partial-L2-while*)

lemma (in *heap-typing-state*) *runs-to-partial-L2-while-unchanged-typing*[*unchanged-typing*]:

assumes $B[\text{runs-to-vcg}]: \bigwedge r s. C r s \implies (B r) \cdot s \text{ ?}\{\lambda r t. \text{unchanged-typing-on } S s t\}$
shows $(L2\text{-while } C B r ns) \cdot s \text{ ?}\{\lambda r t. \text{unchanged-typing-on } S s t\}$
supply $\text{runs-to-partial-L2-while-same-post } [\text{runs-to-vcg}]$
apply (runs-to-vcg)
apply $(\text{auto simp add: unchanged-typing-on-simps})$
done

lemma $\text{runs-to-partial-L2-throw}[\text{runs-to-vcg}]$:

assumes $P (Exn e) s$
shows $(L2\text{-throw } e ns) \cdot s \text{ ?}\{ P \}$
unfolding $L2\text{-defs}$
using assms
by $(\text{rule runs-to-partial-throw})$

lemma $\text{runs-to-partial-L2-spec}[\text{runs-to-vcg}]: (L2\text{-spec } R) \cdot s \text{ ?}\{\lambda r t. (s, t) \in R\}$

unfolding $L2\text{-defs}$
by runs-to-vcg

lemma $\text{runs-to-partial-state-L2-assume}[\text{runs-to-vcg}]$:

$(L2\text{-assume } R) \cdot s \text{ ?}\{\lambda r t. \exists x. r = \text{Result } x \wedge (x, t) \in R s\}$
unfolding $L2\text{-defs}$
apply (runs-to-vcg)
done

lemma $\text{runs-to-partial-L2-guard}[\text{runs-to-vcg}]$:

shows $(L2\text{-guard } P) \cdot s \text{ ?}\{\lambda r t. r = \text{Result } () \wedge s = t \wedge P s\}$
unfolding $L2\text{-defs}$
apply (runs-to-vcg)
done

definition $\text{GUARDED-ASSM} :: \text{bool} \implies \text{bool}$ **where** $[\text{remove-ASSMs}]: \text{GUARDED-ASSM } P \iff P$

lemma $\text{GUARDED-ASSM-D}: \text{GUARDED-ASSM } P \implies P$ **by** $(\text{simp add: GUARDED-ASSM-def})$

lemma $\text{runs-to-partial-L2-guarded}[\text{runs-to-vcg}]$:

$(\text{GUARDED-ASSM } (P s) \implies c \cdot s \text{ ?}\{Q\}) \implies (L2\text{-guarded } P c) \cdot s \text{ ?}\{Q\}$
unfolding $L2\text{-guarded-def GUARDED-ASSM-def}$
by runs-to-vcg

lemma $\text{runs-to-partial-L2-fail-conv}[\text{simp}]: L2\text{-fail} \cdot s \text{ ?}\{ R \} \iff \text{True}$

by $(\text{simp add: L2-fail-def})$

lemma $\text{runs-to-partial-L2-fail}[\text{runs-to-vcg}]: L2\text{-fail} \cdot s \text{ ?}\{ R \}$

by simp

lemma $\text{runs-to-partial-L2-call}[\text{runs-to-vcg}]$:

assumes $[\text{runs-to-vcg}]: m \cdot s \text{ ?}\{\lambda r. Q (\text{case } r \text{ of } Exn e \Rightarrow Exn (f e) \mid \text{Result } r)$

```

⇒ Result r) }
shows (L2-call m f ns) · s ?{ Q }
unfolding L2-call-def
apply runs-to-vcg
apply (simp add: map-exn-def)
done

end

```

20.4 In Out Parameter Refinement

```

theory In-Out-Parameters
imports
  L2ExceptionRewrite
  L2Peephole
  TypHeapSimple
  Stack-Typing
begin

```

```

lemma map-exn-catch-conv: map-value (map-exn f) m = (m <catch> (λr. throw
(f r)))
  apply (rule spec-monad-eqI)
  apply (clarsimp simp add: runs-to-iff)
  apply (auto simp add: runs-to-def-old map-exn-def split: xval-splits)
done

```

```

abbreviation L2-return x ns ≡ liftE (L2-VARS (return x) ns)

```

```

lemma L2-return-L2-gets-conv: L2-return x ns = L2-gets (λ-. x) ns
  unfolding L2-defs L2-VARS-def
  by (simp add: gets-return)

```

```

lemma return-L2-gets-conv: (return x) = L2-gets (λ-. x) []
  unfolding L2-defs L2-VARS-def
  by (simp add: gets-return)

```

```

definition (in heap-state)

```

```

  IO-modify-heap-padding::'a::mem-type ptr ⇒ ('s ⇒ 'a) ⇒ ('b::default, unit, 's)
spec-monad where
  IO-modify-heap-padding p v =
    state-select {(s, t). ∃ bs. length bs = size-of (TYPE('a)) ∧ t = hmem-upd
(heap-update-padding p (v s) bs) s}

```

```

lemma (in heap-state) liftE-IO-modify-heap-padding: liftE (IO-modify-heap-padding
p v) = (IO-modify-heap-padding p v)
  unfolding IO-modify-heap-padding-def

```

apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

abbreviation (*in heap-state*) *IO-modify-heap-paddingE*::
'a::mem-type ptr \Rightarrow (*'s* \Rightarrow *'a*) \Rightarrow (*'b, unit, 's*) *exn-monad* **where**
IO-modify-heap-paddingE p v \equiv *liftE (IO-modify-heap-padding p v)*

lemma (*in heap-state*) *no-fail-IO-modify-padding[simp]*: *succeeds (IO-modify-heap-padding p v) s*
unfolding *IO-modify-heap-padding-def*
apply (*simp*)
using *length-to-bytes-p* **by** *blast*

lemma (*in heap-state*) *no-fail-IO-modify-paddingE[simp]*: *succeeds (IO-modify-heap-paddingE p v) s*
unfolding *IO-modify-heap-padding-def*
apply (*simp*)
using *length-to-bytes-p* **by** *blast*

named-theorems *refines-right-eq*

lemma (*in heap-state*) *IO-modify-heap-paddingE-root-refines'*:
fixes *p::'a::xmem-type ptr*
fixes *fld-update::('b::xmem-type* \Rightarrow *'b*) \Rightarrow *'a* \Rightarrow *'a*
assumes *fg-cons: fg-cons fld (fld-update* \circ ($\lambda x -. x$))
assumes *fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (adjust-ti (typ-info-t TYPE('b::xmem-type)) fld (fld-update* \circ ($\lambda x -. x$)), *n*)
assumes *cgrd: c-guard p*
shows *refines*
(IO-modify-heap-paddingE (PTR('b) &(p→f)) v)
(IO-modify-heap-paddingE p (λs. fld-update (λ-. v s) (h-val (hmem s)
p)))
s s ((=))

proof –
{
fix *r t*
assume *exec: reaches (IO-modify-heap-paddingE (PTR('b) &(p→f)) v) s r t*
have *reaches (IO-modify-heap-paddingE p (λs. fld-update (λ-. v s) (h-val (hmem s) p))) s r t*
proof –
from *exec* **obtain** *bs* **where**
r: r = Result () **and** *bs: length bs = size-of TYPE('b)* **and**
t: t = hmem-upd (heap-update-padding (PTR('b) &(p→f)) (v s) bs) s
unfolding *liftE-IO-modify-heap-padding*
by (*auto simp add: IO-modify-heap-padding-def*)
note *root-conv = heap-update-padding-field-root-conv'' [OF fl fg-cons cgrd bs,*
where *v=v s* **and** *hp = hmem s]*
from *bs cgrd fl* **have** ***: *length (super-update-bs bs (heap-list (hmem s) (size-of*

```

TYPE('a) (ptr-val p) n) = size-of TYPE('a)
  apply (subst length-super-update-bs)
  subgoal
    by (metis add.commute field-lookup-offset-size heap-list-length size-of-tag
typ-desc-size-update-ti typ-winfo-size)
  subgoal
    using heap-list-length by blast
  done
  show ?thesis
  unfolding liftE-IO-modify-heap-padding
  apply (clarsimp simp add: IO-modify-heap-padding-def r)
  using root-conv heap.upd-cong * t
  by blast
qed
} note * = this
show ?thesis
  using *
  by (auto simp add: refines-def-old )
qed

```

```

lemma (in heap-state) IO-modify-heap-paddingE-root-refines':
  fixes p::'a::xmem-type ptr
  fixes fld-update::('b::xmem-type  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'a
  assumes fg-cons: fg-cons fld (fld-update  $\circ$  ( $\lambda x$  -. x))
  assumes fl: field-ti TYPE('a) f = Some (adjust-ti (typ-info-t TYPE('b::xmem-type))
fld (fld-update  $\circ$  ( $\lambda x$  -. x)))
  assumes cgrd: c-guard p
  shows refines
    (IO-modify-heap-paddingE (PTR('b) &(p $\rightarrow$ f)) v)
    (IO-modify-heap-paddingE p ( $\lambda s$ . (fld-update ( $\lambda$ -. v s)) (h-val (hmem s)
p)))
    s s ((=))

```

```

proof -
  from fl obtain n where
    field-lookup (typ-info-t TYPE('a)) f 0 = Some (adjust-ti (typ-info-t TYPE('b::xmem-type))
fld (fld-update  $\circ$  ( $\lambda x$  -. x)), n)
  using field-ti-field-lookupE by blast
  from IO-modify-heap-paddingE-root-refines' [OF fg-cons this cgrd]
  show ?thesis .
qed

```

```

lemma (in heap-state) IO-modify-heap-paddingE-root-refines [refines-right-eq]:
  fixes p::'a::xmem-type ptr
  fixes fld-update::('b::xmem-type  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'a
  assumes v':  $\bigwedge s$ . v' s = ((fld-update ( $\lambda$ -. v s)) (h-val (hmem s) p))
  assumes fg-cons: fg-cons fld (fld-update  $\circ$  ( $\lambda x$  -. x))
  assumes fl: field-ti TYPE('a) f = Some (adjust-ti (typ-info-t TYPE('b::xmem-type))
fld (fld-update  $\circ$  ( $\lambda x$  -. x)))
  assumes cgrd: c-guard p

```


shows *refines*
 (*IO-modify-heap-paddingE* (*PTR*('b) &(p→f)) v)
 (*IO-modify-heap-paddingE* p v')
 s s ((=))
unfolding v'
by (*rule IO-modify-heap-paddingE-root-refines''* [*OF fg-cons fl cgrd*])

lemma *refines-subst-right*:
assumes *f-g*: *refines f g s t Q*
assumes *refines-eq*: *refines g g' t t ((=))*
shows *refines f g' s t Q*
using *f-g refines-eq*
by (*force simp add: refines-def-old*)

lemma *refines-right-eq-id*: *refines f f s s ((=))*
by (*force simp add: refines-def-old*)

lemma *refines-subst-left*:
assumes *f-g*: *refines f g s t Q*
assumes *f-eq*: *run f s = run f' s*
shows *refines f' g s t Q*
using *f-g f-eq*
apply (*auto simp add: refines-def-old reaches-def succeeds-def*)
done

lemma *refines-right-eq-L2-seq* [*refines-right-eq*]:
assumes *f1*: *refines f1 g1 s s ((=))*
assumes *f2*: $\bigwedge v t. \text{refines } (f2\ v) (g2\ v) t t ((=))$
shows *refines (L2-seq f1 f2) (L2-seq g1 g2) s s ((=))*
unfolding *L2-defs*
apply (*rule refines-bind-bind-exn* [*OF f1*])
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal using *f2* **by** *auto*
done

lemma *refines-right-eq-L2-guard'*:
assumes $Q\ s \implies P\ s$
assumes $Q\ s \implies \text{refines } X\ X'\ s s (=)$
shows *refines (L2-seq (L2-guard P) (λ-. X)) (L2-seq (L2-guard Q) (λ-. X')) s s ((=))*
unfolding *L2-defs*
apply (*rule refines-bind'*[*OF*
refines-guard[*THEN refines-strengthen*[*OF - runs-to-partial-guard runs-to-partial-guard*]]])
apply (*auto simp: assms*)
done

lemma *refines-right-eq-L2-guard*:
assumes $Q \implies P$
assumes $Q \implies \text{refines } X \ X' \ s \ s \ (=)$
shows $\text{refines } (L2\text{-seq } (L2\text{-guard } (\lambda\cdot. P)) \ (\lambda\cdot. X)) \ (L2\text{-seq } (L2\text{-guard } (\lambda\cdot. Q)) \ (\lambda\cdot. X')) \ s \ s \ (=)$
using *assms*
by (*rule refines-right-eq-L2-guard'*)

lemma *select-UNIV-L2-unknown-conv*: $(\text{select } UNIV) = L2\text{-unknown } ns$
unfolding *L2-defs*
by *simp*

lemma *select-singleton-conv*: $((\text{select } (\{x\})) \ggg = g) = g \ x$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *hd-UNIV*: $hd \text{ ' } UNIV \subseteq UNIV$
by (*rule subset-UNIV*)

lemma *hd-singleton*: $hd \text{ ' } \{[x]\} \subseteq \{x\}$
by *simp*

lemma *refines-select-right-witness*:
assumes $x \in X$
assumes $\text{refines } f \ (g \ x) \ s \ t \ Q$
shows $\text{refines } f \ ((\text{select } X) \ggg = g) \ s \ t \ Q$
using *assms*
using *assms*
by (*fastforce simp add: refines-def-old succeeds-bind reaches-bind*)

lemma *refines-bindE-right*:
assumes $f: \text{refines } f \ f' \ s \ s' \ Q$
assumes $ll: \bigwedge e \ e' \ t \ t'. \ Q \ (Exn \ e, \ t) \ (Exn \ e', \ t') \implies R \ (Exn \ e, \ t) \ (Exn \ e', \ t')$
assumes $lr: \bigwedge e \ v' \ t \ t'. \ Q \ (Exn \ e, \ t) \ (Result \ v', \ t') \implies \text{refines } (\text{throw } e) \ (g' \ v') \ t \ t' \ R$
assumes $rl: \bigwedge v \ e' \ t \ t'. \ Q \ (Result \ v, \ t) \ (Exn \ e', \ t') \implies \text{refines } ((\text{return } v)) \ (\text{throw } e') \ t \ t' \ R$
assumes $rr: \bigwedge v \ v' \ t \ t'. \ Q \ (Result \ v, \ t) \ (Result \ v', \ t') \implies \text{refines } ((\text{return } v)) \ (g' \ v') \ t \ t' \ R$
shows $\text{refines } f \ (f' \ \ggg = g') \ s \ s' \ R$
proof –
have $eq: f = (f \ \ggg = (\lambda v. \text{return } v))$
by *simp*
show *?thesis*
apply (*subst eq*)

```

    using assms
    by (rule refines-bind-bind-exn)
qed

```

```

lemma refines-exec-modify-step-right:
  assumes refines (return x) g s (upd t) Q
  shows refines (return x) (do{ - <- (modify (upd)); g }) s t Q
  using assms
  by (auto simp add: refines-def-old reaches-bind succeeds-bind)

```

```

lemma (in heap-state) refines-exec-IO-modify-heap-padding-step-right:
  fixes p:: 'a::mem-type ptr
  assumes
    refines (return x) g s
    (hmem-upd (heap-update-padding p v (heap-list (hmem s) (size-of TYPE('a))
(ptr-val p)) ) t) Q)
  shows refines (return x) (do { - <- IO-modify-heap-paddingE p (λ-. v); g }) s t
Q
  using assms unfolding liftE-IO-modify-heap-padding
  apply (clarsimp simp add: refines-def-old IO-modify-heap-padding-def reaches-bind
succeeds-bind )
  apply (metis heap-list-length)
  done

```

```

lemma (in heap-state) refines-exec-IO-modify-heap-padding-single-step-right:
  fixes p:: 'a::mem-type ptr
  assumes Q (Result (), s)
    (Result (), hmem-upd (heap-update-padding p (v t) (heap-list (hmem s)
(size-of TYPE('a)) (ptr-val p))) t)
  shows refines (return ())
    (IO-modify-heap-paddingE p v) s t Q
  using assms unfolding liftE-IO-modify-heap-padding
  apply (clarsimp simp add: refines-def-old IO-modify-heap-padding-def reaches-bind
succeeds-bind)
  by (metis heap-list-length)

```

```

lemma (in heap-state) refines-exec-IO-modify-heap-padding-single-step-right-InL:
  fixes p:: 'a::mem-type ptr
  assumes Q (Exn e, s) (Exn e', hmem-upd (heap-update-padding p (v t) (heap-list
(hmem s) (size-of TYPE('a)) (ptr-val p))) t)
  shows refines (throw e)
    (do { - <- IO-modify-heap-paddingE p v;
      L2-throw e' ns
    })
    s t Q
  unfolding L2-defs unfolding liftE-IO-modify-heap-padding
  using assms
  apply (clarsimp simp add: refines-def-old IO-modify-heap-padding-def reaches-bind

```

succeeds-bind Exn-def [symmetric]
apply (*metis heap-list-length*)
done

lemma *refines-exec-gets-right*:
assumes Q (*Result* x , s) (*Result* (g t), t)
shows *refines* (*return* x) (*gets* g) s t Q
using *assms*
by (*auto simp add: refines-def-old*)

lemma *refines-exec-L2-return-right*:
assumes Q (*Result* x , s) (*Result* w , t)
shows *refines* (*return* (x)) (*L2-return* w ns) s t Q
using *assms* **unfolding** *L2-VARS-def*
by (*auto simp add: refines-def-old*)

lemma *f-catch-throw*: (f *<catch>* *throw*) = f
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add: runs-to-iff*)
apply (*clarsimp simp add: runs-to-def-old*)
by (*metis Exn-def default-option-def exception-or-result-cases not-None-eq*)

lemma *refines-L2-catch-right*:
assumes f : *refines* f g s t Q
assumes *Res-Res*: $\bigwedge v v' s' t'. Q$ (*Result* v , s') (*Result* v' , t') $\implies R$ (*Result* v , s') (*Result* v' , t')
assumes *Res-Exn*: $\bigwedge v e' s' t'. Q$ (*Result* v , s') (*Exn* e' , t') \implies *refines* (*return* v) (h e') s' t' R
assumes *Exn-Res*: $\bigwedge e v' s' t'. Q$ (*Exn* e , s') (*Result* v' , t') $\implies R$ (*Exn* e , s') (*Result* v' , t')
assumes *Exn-Exn*: $\bigwedge e e' s' t'. Q$ (*Exn* e , s') (*Exn* e' , t') \implies *refines* (*throw* e) (h e') s' t' R
shows *refines* f (*L2-catch* g h) s t R **unfolding** *L2-defs*
apply (*subst f-catch-throw[symmetric]*)
apply (*rule refines-catch [OF f]*)
subgoal using *Exn-Exn* **by** *auto*
subgoal using *Exn-Res* **by** (*auto simp add: refines-def-old Exn-def [symmetric]*)
subgoal using *Res-Exn* **by** (*auto simp add: refines-def-old Exn-def [symmetric]*)
subgoal using *Res-Res* **by** *auto*
done

lemma *L2-seq-gets-app-conv*: run (*L2-seq* (*L2-gets* f ns) g) s = run (g (f s)) s
unfolding *L2-defs*
apply (*auto simp add: run-bind*)
done

lemma *refines-project-right*:

```

assumes f-g: refines f g s t Q
assumes run g t = run (g' (prj t)) t
shows refines f (L2-seq (L2-gets prj ns) g^) s t Q
unfolding L2-defs
using assms
apply (clarsimp simp add: refines-def-old reaches-bind succeeds-bind)
apply (auto simp add: succeeds-def reaches-def)
done

```

```

lemma refines-project-guard-right:
assumes f-g: refines f (L2-seq (L2-guard P) (λ-. g)) s t Q
assumes P t ⇒ run g t = run (g' (prj t)) t
shows refines f (L2-seq (L2-guard P) (λ-. (L2-seq (L2-gets prj ns) g^))) s t Q
using assms unfolding L2-defs
apply (clarsimp simp add: refines-def-old reaches-bind succeeds-bind)
apply (auto simp add: succeeds-def reaches-def)
done

```

named-rules *cguard-assms and alloc-assms and modifies-assms and disjoint-assms*
and

disjoint-alloc and disjoint-stack-free and stack-ptr and h-val-globals-frame-eq
and

rel-alloc-independent-globals

synthesize-rules *refines-in-out*

add-synthesize-pattern *refines-in-out =*

```

<
  [(Binding.make (concrete-function, here), fn ctxt => fn i => fn t =>
    (case t of
      @{term-pat Trueprop (refines (L2-modify (λs. ?glob-upd -)) - - - -)} =>
glob-upd
      | @{term-pat Trueprop (refines (L2-modify (?glob-upd -)) - - - -)} => glob-upd
      | @{term-pat Trueprop (refines ?f - - - -)} => f
      | t => t))]
>

```

lemma *refines-yield-both[simp]: refines (return a) (return b) s t R ↔ R (Result a, s) (Result b, t)*

by (*auto simp add: refines-def-old*)

lemma *disjoint-union-distrib: A ∩ (B ∪ C) = {} ↔ A ∩ B = {} ∧ A ∩ C = {}*

by *blast*

lemma *inter-commute: A ∩ B = B ∩ A by blast*

lemma *disjoint-symmetric: A ∩ B = {} ⇒ B ∩ A = {}*

by *auto*

lemma *disjoint-symmetric': A ∩ B ≡ {} ⇒ B ∩ A = {}*

by auto

definition *IOcorres*::

$(s \Rightarrow \text{bool}) \Rightarrow$
 $(s \Rightarrow (e, a) \text{ xval} \Rightarrow s \Rightarrow \text{bool}) \Rightarrow$
 $(s \Rightarrow t) \Rightarrow$
 $(a \Rightarrow s \Rightarrow b) \Rightarrow$
 $(e \Rightarrow s \Rightarrow e1) \Rightarrow$
 $(e1, b, t) \text{ exn-monad} \Rightarrow$
 $(e, a, s) \text{ exn-monad} \Rightarrow \text{bool}$ **where**
 $\text{IOcorres } P \ Q \ st \ rx \ ex \ f_a \ f_c \equiv \text{corresXF-post } st \ rx \ ex \ P \ Q \ f_a \ f_c$

lemma *IOcorres-id*: $\text{IOcorres } (\lambda -. \text{True}) (\lambda - - -. \text{True}) (\lambda s. s) (\lambda r -. r) (\lambda r -. r)$
ff

by (auto simp add: *IOcorres-def* *corresXF-post-def* *split: xval-splits prod.splits*)

lemmas *IOcorres-skip* = *IOcorres-id*

lemma *IOcorres-trivial-from-local-var-extract*:

$L2\text{corres } st \ rx \ ex \ P \ A \ C \Longrightarrow \text{IOcorres } (\lambda -. \text{True}) (\lambda - - -. \text{True}) (\lambda s. s) (\lambda r -. r)$
 $(\lambda r -. r) \ A \ A$

by (rule *IOcorres-id*)

lemma *admissible-IOcorres* [*corres-admissible*]:

ccpo.admissible *Inf* (\geq) ($\lambda f_a. \text{IOcorres } P \ Q \ st \ rx \ ex \ f_a \ f_c$)

unfolding *IOcorres-def*

by (rule *admissible-nondet-ord-corresXF-post*)

lemma *IOcorres-top* [*corres-top*]: $\text{IOcorres } P \ Q \ st \ rx \ ex \ \top \ f_c$

unfolding *IOcorres-def*

by (rule *corresXF-post-top*)

lemma *distinct-addresses-ptr-val-lemma*:

$n < \text{addr-card} \Longrightarrow \text{ptr-val } p + \text{word-of-nat } n \notin (\lambda x. \text{ptr-val } p + \text{word-of-nat } x) \text{ ‘}$
 $\{0..<n\}$

by auto (*metis* *addr-card-len-of-conv* *nless-le* *order-le-less-trans* *unat-of-nat-eq*)

lemma *distinct-addresses-ptr-lemma*:

assumes *bound*: $n \leq \text{addr-card}$

shows *distinct* ($\text{map } (\lambda i. \text{ptr-val } p + \text{of-nat } i) [0..<n]$)

using *bound*

proof (*induct* *n* *arbitrary*: *p*)

case 0

then show *?case* by *simp*

next

case (*Suc* *n*)

from *Suc* **have** *hyp*: *distinct* ($\text{map } (\lambda i. \text{ptr-val } p + \text{word-of-nat } i) [0..<n]$) **by**

auto

show *?case*

apply (*subst* *upt-Suc-snoc*)

```

apply (subst map-append)
apply (subst distinct-append)
apply (simp add: hyp)
using Suc.prem
apply (simp add: distinct-addresses-ptr-val-lemma)
done
qed

```

lemma *array-intvl-Suc-split*: $\{a..+Suc\ n * m\} = \{a..+n * m\} \cup \{a + (word-of-nat\ (n*m))..+m\}$
by (metis add-diff-cancel-right' intvl-split le-add2 mult-Suc)

definition *rel-singleton-stack* :: 'a::c-type ptr \Rightarrow heap-mem \Rightarrow unit \Rightarrow 'a \Rightarrow bool
where
rel-singleton-stack p h = (λ (::unit) v.
v = h-val h p)

lemma *domain-rel-singleton-stack*:
equal-on (ptr-span p) h h' \Longrightarrow *rel-singleton-stack* p h = *rel-singleton-stack* p h'
apply (clarsimp simp add: fun-eq-iff *rel-singleton-stack-def*)
apply (metis (mono-tags, lifting) equal-on-def h-val-def heap-list-h-eq2)+
done

named-theorems *rel-stack-intros* and *rel-stack-simps*

lemma *rel-singleton-stack-simp* [*rel-stack-simps*]:
rel-singleton-stack p h x v \longleftrightarrow v = h-val h p
by (auto simp add: *rel-singleton-stack-def*)

lemma *rel-stack-refl* [*rel-stack-intros*]: (λ -. (=)) h x x
by auto

lemma *rel-singleton-stackI* [*rel-stack-intros*]: *rel-singleton-stack* p h x (h-val h p)
by (auto simp add: *rel-singleton-stack-def*)

lemma *rel-singleton-stack-condI* [*rel-stack-intros*]: h-val h p = y \Longrightarrow *rel-singleton-stack* p h x y
by (auto simp add: *rel-singleton-stack-def*)

lemma *fun-of-rel-singleton-stack*[*fun-of-rel-intros*]: fun-of-rel (*rel-singleton-stack* p h) (λ -. (h-val h p))
by (auto simp add: fun-of-rel-def *rel-singleton-stack-def*)

lemma *funp-rel-singleton-stack*[*funp-intros, corres-admissible*]: *funp* (*rel-singleton-stack* *p h*)

by (*auto simp add: rel-singleton-stack-def funp-def fun-of-rel-def*)

definition *rel-push* ::

'a::c-type ptr \Rightarrow (*heap-mem* \Rightarrow *'b* \Rightarrow *'c* \Rightarrow *bool*) \Rightarrow *heap-mem* \Rightarrow
'b \Rightarrow *'a* \times *'c* \Rightarrow *bool*

where

rel-push p R h = (λr (*v, x*).
 R *h r x* \wedge
 $v = h\text{-val } h$ *p*)

lemma *rel-singleton-stack-rel-push-conv*: *rel-singleton-stack p* = (λh *x y. rel-push* *p* ($\lambda - - . True$) *h x* (*y, ()*))

by (*auto simp add: rel-singleton-stack-def rel-push-def*)

lemma *rel-push-simp*[*rel-stack-simps*]: *rel-push p R h r* (*v, x*) \longleftrightarrow

R *h r x* \wedge $v = h\text{-val } h$ *p*

by (*auto simp add: rel-push-def*)

lemma *fun-of-rel-puhsh* [*fun-of-rel-intros*]:

fun-of-rel (R *h*) *f* \Longrightarrow *fun-of-rel* (*rel-push p R h*) ($\lambda x. (h\text{-val } h$ *p, f x*)

by (*auto simp add: fun-of-rel-def rel-push-def*)

lemma *funp-rel-push*[*funp-intros, corres-admissible*]: *funp* (R *h*) \Longrightarrow *funp* (*rel-push* *p R h*)

by (*force simp add: funp-def rel-push-def fun-of-rel-def*)

lemma *rel-push-stackI* [*rel-stack-intros*]: Q *h x y* \Longrightarrow *rel-push p Q h x* ($(h\text{-val } h$ *p, y*)

by (*auto simp add: rel-push-def*)

lemma *rel-push-stack-condI* [*rel-stack-intros*]: $h\text{-val } h$ *p = v* \Longrightarrow Q *h x y* \Longrightarrow *rel-push p Q h x* (*v, y*)

by (*auto simp add: rel-push-def*)

definition *rel-sum-stack* $L R h = rel\text{-sum } (L h) (R h)$

definition *rel-xval-stack* $L R h = rel\text{-xval } (L h) (R h)$

lemma *rel-sum-stack-expand-sum-eq*:

$(rel\text{-sum-stack } (\lambda - . (=)) R) = (rel\text{-sum-stack } (rel\text{-sum-stack } (\lambda - . (=)) (\lambda - . (=))) R)$

by (*auto simp add: rel-sum-stack-def fun-eq-iff rel-sum.simps split: sum.splits*)

lemma *rel-xval-stack-expand-sum-eq*:

$(rel\text{-xval-stack } (\lambda - . (=)) R) = (rel\text{-xval-stack } (rel\text{-sum-stack } (\lambda - . (=)) (\lambda - . (=))) R)$

R)
by (*auto simp add: rel-sum-stack-def rel-xval-stack-def fun-eq-iff rel-sum.simps sum.rel-eq split: xval-splits sum.splits*)

lemma *rel-sum-stack-expand-sum-bot*:
 $(\lambda - - . \text{False}) = (\text{rel-sum-stack } (\lambda - - . \text{False}) (\lambda - - . \text{False}))$
by (*auto simp add: rel-sum-stack-def fun-eq-iff rel-sum.simps split: sum.splits*)

lemma *rel-xval-stack-expand-xval-bot*:
 $(\lambda - - . \text{False}) = (\text{rel-xval-stack } (\lambda - - . \text{False}) (\lambda - - . \text{False}))$
by (*auto simp add: rel-xval-stack-def fun-eq-iff rel-xval.simps split: xval-splits*)

lemma *rel-sum-stack-entail*:
assumes $L: \bigwedge v v'. L' h v v' \implies L h v v'$
assumes $R: \bigwedge v v'. R' h v v' \implies R h v v'$
assumes *rel-sum-stack* $L' R' h x y$
shows *rel-sum-stack* $L R h x y$
using *assms*
by (*auto simp add: rel-sum-stack-def rel-sum.simps split: sum.splits*)

lemma *rel-xval-stack-entail*:
assumes $L: \bigwedge v v'. L' h v v' \implies L h v v'$
assumes $R: \bigwedge v v'. R' h v v' \implies R h v v'$
assumes *rel-xval-stack* $L' R' h x y$
shows *rel-xval-stack* $L R h x y$
using *assms*
by (*auto simp add: rel-xval-stack-def rel-xval.simps*)

lemma *rel-sum-stack-intros*:
 $L h l1 l2 \implies \text{rel-sum-stack } L R h (\text{Inl } l1) (\text{Inl } l2)$
 $R h r1 r2 \implies \text{rel-sum-stack } L R h (\text{Inr } r1) (\text{Inr } r2)$
by (*auto simp add: rel-sum-stack-def*)

lemma *rel-xval-stack-intros*:
 $L h l1 l2 \implies \text{rel-xval-stack } L R h (\text{Exn } l1) (\text{Exn } l2)$
 $R h r1 r2 \implies \text{rel-xval-stack } L R h (\text{Result } r1) (\text{Result } r2)$
by (*auto simp add: rel-xval-stack-def*)

lemma *fun-of-rel-sum-stack[fun-of-rel-intros]*:
 $\text{fun-of-rel } (L h) f-l \implies \text{fun-of-rel } (R h) f-r \implies \text{fun-of-rel } (\text{rel-sum-stack } L R h)$
 $(\text{sum-map } f-l f-r)$
unfolding *rel-sum-stack-def*
by (*auto intro: fun-of-rel-intros*)

lemma *fun-of-rel-xval-stack[fun-of-rel-intros]*:

$fun\text{-of-rel } (L\ h)\ f\text{-l} \implies fun\text{-of-rel } (R\ h)\ f\text{-r} \implies fun\text{-of-rel } (rel\text{-xval-stack } L\ R\ h)$
 $(map\text{-xval } f\text{-l } f\text{-r})$

unfolding $rel\text{-xval-stack-def}$
by $(auto\ intro: fun\text{-of-rel-intros})$

lemma $funp\text{-rel-sum-stack}[funp\text{-intros, corres-admissible}]$: $funp\ (L\ h) \implies funp\ (R\ h) \implies funp\ (rel\text{-sum-stack } L\ R\ h)$

unfolding $rel\text{-sum-stack-def}$ **by** $(auto\ intro: funp\text{-intros})$

lemma $funp\text{-rel-xval-stack}[funp\text{-intros, corres-admissible}]$: $funp\ (L\ h) \implies funp\ (R\ h) \implies funp\ (rel\text{-xval-stack } L\ R\ h)$

unfolding $rel\text{-xval-stack-def}$ **by** $(auto\ intro: funp\text{-intros})$

lemma $rel\text{-sum-stack-cases}$:

$rel\text{-sum-stack } L\ R\ h\ x\ y =$
 $((\exists v\ w. x = Inl\ v \wedge y = Inl\ w \wedge L\ h\ v\ w) \vee$
 $(\exists v\ w. x = Inr\ v \wedge y = Inr\ w \wedge R\ h\ v\ w))$
by $(auto\ simp\ add: rel\text{-sum-stack-def } rel\text{-sum.simps})$

lemma $rel\text{-xval-stack-cases}$:

$rel\text{-xval-stack } L\ R\ h\ x\ y =$
 $((\exists v\ w. x = Exn\ v \wedge y = Exn\ w \wedge L\ h\ v\ w) \vee$
 $(\exists v\ w. x = Result\ v \wedge y = Result\ w \wedge R\ h\ v\ w))$
by $(auto\ simp\ add: rel\text{-xval-stack-def } rel\text{-xval.simps})$

lemma $rel\text{-sum-stack-simps } [simp]$:

$rel\text{-sum-stack } L\ R\ h\ (Inl\ l_1)\ (Inl\ l_2) = L\ h\ l_1\ l_2$
 $rel\text{-sum-stack } L\ R\ h\ (Inr\ r_1)\ (Inr\ r_2) = R\ h\ r_1\ r_2$
 $rel\text{-sum-stack } L\ R\ h\ (Inl\ l_1)\ (Inr\ r_2) = False$
 $rel\text{-sum-stack } L\ R\ h\ (Inr\ r_1)\ (Inl\ l_2) = False$
 $rel\text{-sum-stack } L\ R\ h\ (Inl\ l_1)\ y = (\exists w. y = Inl\ w \wedge L\ h\ l_1\ w)$
 $rel\text{-sum-stack } L\ R\ h\ (Inr\ r_1)\ y = (\exists w. y = Inr\ w \wedge R\ h\ r_1\ w)$
 $rel\text{-sum-stack } L\ R\ h\ x\ (Inl\ l_2) = (\exists v. x = Inl\ v \wedge L\ h\ v\ l_2)$
 $rel\text{-sum-stack } L\ R\ h\ x\ (Inr\ r_2) = (\exists v. x = Inr\ v \wedge R\ h\ v\ r_2)$
by $(auto\ simp\ add: rel\text{-sum-stack-def } rel\text{-sum.simps})$

lemma $rel\text{-xval-stack-simps } [simp]$:

$rel\text{-xval-stack } L\ R\ h\ (Exn\ l_1)\ (Exn\ l_2) = L\ h\ l_1\ l_2$
 $rel\text{-xval-stack } L\ R\ h\ (Result\ r_1)\ (Result\ r_2) = R\ h\ r_1\ r_2$
 $rel\text{-xval-stack } L\ R\ h\ (Exn\ l_1)\ (Result\ r_2) = False$
 $rel\text{-xval-stack } L\ R\ h\ (Result\ r_1)\ (Exn\ l_2) = False$
 $rel\text{-xval-stack } L\ R\ h\ (Exn\ l_1)\ y = (\exists w. y = Exn\ w \wedge L\ h\ l_1\ w)$
 $rel\text{-xval-stack } L\ R\ h\ (Result\ r_1)\ y = (\exists w. y = Result\ w \wedge R\ h\ r_1\ w)$
 $rel\text{-xval-stack } L\ R\ h\ x\ (Exn\ l_2) = (\exists v. x = Exn\ v \wedge L\ h\ v\ l_2)$
 $rel\text{-xval-stack } L\ R\ h\ x\ (Result\ r_2) = (\exists v. x = Result\ v \wedge R\ h\ v\ r_2)$
by $(auto\ simp\ add: rel\text{-xval-stack-def } rel\text{-xval.simps})$

lemma *rel-sum-stack-eq-collapse*: $(rel\text{-}sum\text{-}stack (\lambda\cdot. (=)) (\lambda\cdot. (=))) = ((\lambda\cdot. (=)))$

by (*auto simp add: rel-sum-stack-cases fun-eq-iff*)

lemma *rel-xval-stack-eq-collapse*: $(rel\text{-}xval\text{-}stack (\lambda\cdot. (=)) (\lambda\cdot. (=))) = ((\lambda\cdot. (=)))$

apply (*clarsimp simp add: rel-xval-stack-cases fun-eq-iff*)

by (*metis Exn-def default-option-def exception-or-result-cases not-Some-eq*)

lemma *rel-sum-stack-InlI*: $L\ h\ l1\ l2 \implies rel\text{-}sum\text{-}stack\ L\ R\ h\ (Inl\ l1)\ (Inl\ l2)$

by (*simp*)

lemma *rel-xval-stack-ExnI*: $L\ h\ l1\ l2 \implies rel\text{-}xval\text{-}stack\ L\ R\ h\ (Exn\ l1)\ (Exn\ l2)$

by (*simp*)

lemma *rel-sum-stack-InrI*: $R\ h\ r1\ r2 \implies rel\text{-}sum\text{-}stack\ L\ R\ h\ (Inr\ r1)\ (Inr\ r2)$

by (*simp*)

lemma *rel-xval-stack-ResultI*: $R\ h\ r1\ r2 \implies rel\text{-}xval\text{-}stack\ L\ R\ h\ (Result\ r1)\ (Result\ r2)$

by (*simp*)

definition *rel-exit* $Q\ h = (\lambda v\ w. \exists x. v = Nonlocal\ x \wedge Q\ h\ x\ w)$

lemma *rel-exit-simps*[*simp*]:

rel-exit $Q\ h\ (Nonlocal\ x)\ y = Q\ h\ x\ y$

rel-exit $Q\ h\ Break\ y = False$

rel-exit $Q\ h\ Continue\ y = False$

rel-exit $Q\ h\ Return\ y = False$

rel-exit $Q\ h\ (Goto\ l)\ y = False$

by (*auto simp add: rel-exit-def*)

lemma *rel-exit-intro*: $Q\ h\ x\ y \implies rel\text{-}exit\ Q\ h\ (Nonlocal\ x)\ y$

by (*auto simp add: rel-exit-def*)

lemma *rel-xval-stack-rel-exit-intro*:

assumes $\bigwedge x. rel\text{-}xval\text{-}stack\ (rel\text{-}exit\ Q)\ R\ h\ (Exn\ (Nonlocal\ (the\text{-}Nonlocal\ x)))$
(*Exn* x)

assumes $rel\text{-}exit\ Q\ h\ e\ e'$

shows $rel\text{-}xval\text{-}stack\ (rel\text{-}exit\ Q)\ R\ h\ (Exn\ (Nonlocal\ (the\text{-}Nonlocal\ e)))\ (Exn\ e')$

using *assms*

by (*auto simp add: rel-exit-def*)

lemma *rel-xval-stack-rel-exit-intro'*:

assumes $\bigwedge x. rel\text{-}xval\text{-}stack\ (rel\text{-}exit\ Q)\ R\ h\ (Exn\ (Nonlocal\ x))\ (Exn\ x)$

assumes $Q\ h\ e\ e'$

shows $rel\text{-}xval\text{-}stack\ (rel\text{-}exit\ Q)\ R\ h\ (Exn\ (Nonlocal\ e))\ (Exn\ e')$

```

using assms
by (auto simp add: rel-exit-def)

lemma rel-exit-False-conv [simp]: rel-exit ( $\lambda - -. \text{False}$ )  $h e e' \longleftrightarrow \text{False}$ 
by (auto simp add: rel-exit-def)

lemma rel-exit-FalseE: rel-exit ( $\lambda - -. \text{False}$ )  $h e e' \Longrightarrow L$ 
by simp

lemma rel-sum-stack-generalise-left:
  rel-sum-stack  $L R h v w \Longrightarrow (\bigwedge v w. L h v w \Longrightarrow L' h v w) \Longrightarrow \text{rel-sum-stack } L' R h v w$ 
by (auto simp add: rel-sum-stack-def rel-sum.simps)

lemma rel-xval-stack-generalise-left:
  rel-xval-stack  $L R h v w \Longrightarrow (\bigwedge v w. L h v w \Longrightarrow L' h v w) \Longrightarrow \text{rel-xval-stack } L' R h v w$ 
by (auto simp add: rel-xval-stack-def rel-xval.simps)

lemmas generalise-unreachable-exitE =
  rel-exit-FalseE
  rel-sum-stack-generalise-left
  rel-xval-stack-generalise-left

lemma fun-of-rel-rel-exit: fun-of-rel ( $L h$ ) f-l  $\Longrightarrow \text{fun-of-rel } (\text{rel-exit } L h) (\lambda v. \text{case } v \text{ of Nonlocal } x \Rightarrow \text{f-l } x \mid - \Rightarrow \text{undefined})$ 
by (auto simp add: fun-of-rel-def split: c-exntype.splits)

lemma funp-rel-exit [funp-intros, corres-admissible]: funp ( $L h$ )  $\Longrightarrow \text{funp } (\text{rel-exit } L h)$ 
using fun-of-rel-rel-exit
by (metis funp-def)

named-theorems equal-upto-simps
named-theorems refines-stack-intros

lemma equal-uptoI:
  assumes eq:  $\bigwedge x. x \notin A \Longrightarrow f x = g x$ 
  shows equal-upto  $A f g$ 
  using eq
  by (auto simp add: equal-upto-def)

lemma equal-upto-heap-update[equal-upto-simps]:
  fixes p:: 'a::mem-type ptr
  assumes (ptr-span  $p$ )  $\subseteq A$ 
  shows equal-upto  $A (\text{heap-update } p v h1) h2 = \text{equal-upto } A h1 h2$ 
  using assms

```

by (smt (verit, best) CTypes.mem-type-simps(2) equal-upto-def heap-list-length heap-update-def heap-update-nmem-same subset-iff)

lemma equal-upto-complement: equal-upto B f g = equal-on (– B) f g
by (simp add: equal-on-def equal-upto-def)

lemma equal-upto-update-left-equalize:
equal-on (– F) (f s) s \implies equal-on F (f s) t \implies equal-upto (F \cup A) s t =
equal-upto A (f s) t
by (smt (verit) Compl-iff Un-iff equal-on-def equal-upto-def)

lemma admissible-const-snd:
assumes admissible-fst: ccpo.admissible \cap ($\lambda x y. y \subseteq x$) ($\lambda X. \exists w \in X. Q w$)
shows ccpo.admissible \cap ($\lambda x y. y \subseteq x$) ($\lambda X. \exists w. (w, v) \in X \wedge Q w$)

proof (rule ccpo.admissibleI[rule-format])

fix C::('a \times 'b) set set

assume chain: Complete-Partial-Order.chain ($\lambda x y. y \subseteq x$) C

assume nonempty: C \neq {}

assume chain-prop: $\bigwedge X. X \in C \implies \exists w. (w, v) \in X \wedge Q w$

show $\exists w. (w, v) \in \bigcap C \wedge Q w$

proof –

define C' where C' = Set.filter ($\lambda(r, t). t = v$) ' C

have subset: $\bigcap C' \subseteq \bigcap C$

using C'-def by fastforce

moreover

have snd-C': $\bigwedge X t. X \in C' \implies t \in \text{snd } ' X \implies t = v$

using C'-def

by auto

moreover

from chain have chain-C': Complete-Partial-Order.chain ($\lambda x y. y \subseteq x$) C'

using chain-prop C'-def

by (simp add: chain-imageI subset-iff)

from nonempty chain-prop have nonempty-C': C' \neq {}

using C'-def

by blast

from chain-C' have result-chain: Complete-Partial-Order.chain ($\lambda x y. y \subseteq x$)
($(\lambda X. \text{fst } ' X) ' C'$)

by (simp add: chain-imageI image-mono)

from nonempty-C' have nonempty-result-chain: ($(\lambda X. \text{fst } ' X) ' C'$) \neq {} by
auto

{

fix X::'a set

assume X \in ($'$) fst ' C'

then obtain X0 where X: X = fst ' Set.filter ($\lambda(r, t). t = v$) X0 and X0:
X0 \in C

by (auto simp add: C'-def)

from chain-prop [OF X0] obtain w where w: (w, v) \in X0 and Q: Q w by
blast

```

from  $w \in X$  have  $w \in X$ 
  apply clarsimp
  by (metis (mono-tags, lifting) case-prodI fst-conv image-iff member-filter)
  with  $Q$ 
  have  $\exists w \in X. Q \ w \ ..$ 
} note result-chain-prop = this
from ccpo.admissibleD [OF admissible-fst result-chain nonempty-result-chain
result-chain-prop]
obtain  $r'$  where
   $r': r' \in \bigcap ((\lambda X. \text{fst } 'X) ' C')$  and  $Q: Q \ r'$ 
  by auto
from  $r' \text{ snd-} C'$  have  $(r', v) \in \bigcap C'$ 
  by fastforce
with  $Q \text{ subset}$  show ?thesis by blast
qed
qed

```

```

lemma admissible-funp: funp Q  $\implies$  ccpo.admissible  $\bigcap$  ( $\lambda x \ y. y \subseteq x$ ) ( $\lambda X. \exists w \in X. Q \ v \ w$ )
  apply (clarsimp simp add: funp-def fun-of-rel-def)
  by (smt (verit, del-Insts) InterI admissible-const ccpo.admissible-def)

```

```

lemma admissible-funp-conj: funp Q  $\implies$  ccpo.admissible  $\bigcap$  ( $\lambda x \ y. y \subseteq x$ ) ( $\lambda X. \exists w \in X. Q \ v \ w \wedge P \ w$ )
  apply (clarsimp simp add: funp-def fun-of-rel-def)
  by (smt (verit, del-Insts) InterI admissible-const ccpo.admissible-def)

```

```

lemma override-on-frame-raw-frame-lemma1:
  assumes disj-A-B: A  $\cap$  B = {}
  assumes sf: stack-free d  $\cap$  B = {}
  shows
    override-on d (override-on d' d B) (A  $\cup$  B - stack-free d) =
    override-on d d' (A - stack-free d)
  using disj-A-B sf
  by (auto simp add: override-on-def fun-eq-iff)

```

```

lemma override-on-frame-raw-frame-lemma2:
  assumes disj-A-B: A  $\cap$  B = {}
  assumes sf: stack-free d  $\cap$  B = {}
  shows
    override-on h' (override-on h h' B) (A  $\cup$  B  $\cup$  stack-free d) =
    override-on h' h (A  $\cup$  stack-free d)
  using disj-A-B sf
  by (auto simp add: override-on-def fun-eq-iff)

```

```

lemma disjoint-stack-free-equal-upto-trans:
  equal-upto A d' d  $\implies$ 
  P  $\cap$  A = {}  $\implies$  stack-free d  $\cap$  P = {}  $\implies$ 

```

$stack\text{-}free\ d' \cap P = \{\}$
apply (*clarsimp simp add: equal-upto-def*)
using *root-ptr-valid-def stack-free-def valid-root-footprint-def* **by** *fastforce*

lemma *disjoint-stack-free-equal-on-trans*:
 $equal\text{-}on\ S\ d'\ d \implies$
 $P \subseteq S \implies stack\text{-}free\ d \cap P = \{\} \implies$
 $stack\text{-}free\ d' \cap P = \{\}$
apply (*clarsimp simp add: equal-on-def*)
using *root-ptr-valid-def stack-free-def valid-root-footprint-def* **by** *fastforce*

lemma *refines-L2-guard-right'*:
assumes $P\ t \implies refines\ f\ g\ s\ t\ Q$
shows $refines\ f\ (L2\text{-}seq\ (L2\text{-}guard\ P)\ (\lambda\cdot.\ g))\ s\ t\ Q$
using *assms unfolding L2-defs*
by (*auto simp add: refines-def-old reaches-bind succeeds-bind*)

lemma *refines-L2-guard-right''*:
assumes $P\ t \implies refines\ f\ (L2\text{-}seq\ (L2\text{-}guard\ P)\ (\lambda\cdot.\ g))\ s\ t\ Q$
shows $refines\ f\ (L2\text{-}seq\ (L2\text{-}guard\ P)\ (\lambda\cdot.\ g))\ s\ t\ Q$
using *assms unfolding L2-defs*
by (*auto simp add: refines-def-old reaches-bind succeeds-bind*)

lemma *refines-L2-guard-right'''*:
assumes $P\ t \implies refines\ f\ (L2\text{-}seq\ (L2\text{-}seq\ (L2\text{-}guard\ P)\ (\lambda\cdot.\ g))\ X)\ s\ t\ Q$
shows $refines\ f\ (L2\text{-}seq\ (L2\text{-}seq\ (L2\text{-}guard\ P)\ (\lambda\cdot.\ g))\ X)\ s\ t\ Q$
using *assms unfolding L2-defs*
by (*auto simp add: refines-def-old reaches-bind succeeds-bind*)

lemma *refines-L2-guard-right''''*:
assumes $P\ t \implies$
 $refines\ f$
 $(L2\text{-}seq$
 $(L2\text{-}catch\ (L2\text{-}seq\ (L2\text{-}guard\ P)\ (\lambda\cdot.\ g))\ X)$
 $Y)\ s\ t\ Q$
shows $refines\ f$
 $(L2\text{-}seq$
 $(L2\text{-}catch\ (L2\text{-}seq\ (L2\text{-}guard\ P)\ (\lambda\cdot.\ g))\ X)$
 $Y)\ s\ t\ Q$
using *assms unfolding L2-defs*
by (*auto simp add: refines-def-old reaches-bind succeeds-bind reaches-catch succeeds-catch*)

lemma *refines-L2-guard-rightE*:
assumes $P\ t$
assumes $refines\ f\ (L2\text{-}seq\ (L2\text{-}guard\ P)\ (\lambda\cdot.\ g))\ s\ t\ Q$
shows $refines\ f\ g\ s\ t\ Q$
using *assms unfolding L2-defs*

by (*auto simp add: refines-def-old reaches-bind succeeds-bind*)

lemma *refines-L2-guard-rightE'*:

assumes $P\ t$

assumes *refines* $f\ (L2\text{-seq}\ (L2\text{-guard}\ P)\ (\lambda\cdot.\ g))\ s\ t\ Q$

shows *refines* $f\ (L2\text{-seq}\ (L2\text{-guard}\ P)\ (\lambda\cdot.\ g))\ s\ t\ Q$

using *assms unfolding L2-defs*

by (*auto simp add: refines-def-old reaches-bind succeeds-bind*)

lemma *refines-L2-guard-right*:

$(P \implies \text{refines } f\ g\ s\ t\ Q) \implies \text{refines } f\ (L2\text{-seq}\ (L2\text{-guard}\ (\lambda\cdot.\ P))\ (\lambda\cdot.\ g))\ s\ t\ Q$

unfolding *L2-defs*

by (*auto simp add: refines-def-old reaches-bind succeeds-bind*)

context *heap-state*

begin

lemma *equal-upto-heap-on-heap-update[equal-upto-simps]*:

fixes $p:: 'a::\text{mem-type}\ \text{ptr}$

assumes $\text{ptr-span } p \subseteq A$

shows *equal-upto-heap-on* $A\ (\text{hmem-upd}\ (\text{heap-update } p\ v)\ s)\ t = \text{equal-upto-heap-on } A\ s\ t$

using *assms equal-upto-heap-update*

by (*smt (verit, ccfv-threshold) equal-upto-heap-on-def equal-upto-heap-on-refl*

equal-upto-heap-on-trans heap.get-upd heap.upd-cong heap.upd-same hmem-htd-upd

sympD

symp-equal-upto-heap-on)

lemma *equal-upto-heap-stack-release'*:

fixes $p:: 'a::\text{mem-type}\ \text{ptr}$

assumes $\text{ptr-span } p \subseteq A$

shows *equal-upto-heap-on* $A\ (\text{htd-upd}\ (\text{stack-releases}\ (\text{Suc } 0)\ p)\ s)\ s$

by (*metis (no-types, lifting) One-nat-def add-mult-distrib add-right-cancel assms*

equal-upto-heap-on-zero-heap-on heap-state.equal-upto-heap-on-trans heap-state-axioms

plus-1-eq-Suc sympD symp-equal-upto-heap-on times-nat.simps(2) zero-heap-on-stack-releases)

lemma *equal-upto-heap-stack-release[equal-upto-simps]*:

fixes $p:: 'a::\text{mem-type}\ \text{ptr}$

assumes $\text{ptr-span } p \subseteq A$

shows *equal-upto-heap-on* $A\ (\text{htd-upd}\ (\text{stack-releases}\ (\text{Suc } 0)\ p)\ s)\ t = \text{equal-upto-heap-on } A\ s\ t$

using *equal-upto-heap-stack-release'*

by (*metis assms equal-upto-heap-on-trans sympD symp-equal-upto-heap-on*)

lemma *equal-upto-heap-on-htd-upd*:

equal-upto-heap-on $A\ s\ t \implies$

$equal\text{-}upto\ A\ (f\ (htd\ s))\ (htd\ t) \implies$
 $equal\text{-}upto\text{-}heap\text{-}on\ A\ (htd\text{-}upd\ f\ s)\ t$
by (*smt* (*verit*, *best*) *equal-upto-heap-on-def* *hmem-htd-upd* *htd-hmem-upd*
lense.upd-compose *typing.get-upd* *typing.lense-axioms* *typing.upd-cong*)

lemma *equal-upto-heap-on-hmem*: $equal\text{-}upto\text{-}heap\text{-}on\ A\ s\ t \implies equal\text{-}upto\ A\ (hmem\ s)\ (hmem\ t)$
by (*auto simp add: equal-upto-heap-on-def*)

lemma *equal-upto-heap-on-sym*: $equal\text{-}upto\text{-}heap\text{-}on\ A\ s\ t = equal\text{-}upto\text{-}heap\text{-}on\ A\ t\ s$
by (*metis* *heap-state.symp-equal-upto-heap-on* *heap-state-axioms* *sympD*)

lemma *equal-upto-heap-stack-alloc'*:
fixes *p*: *'a::mem-type ptr*
shows $equal\text{-}upto\text{-}heap\text{-}on\ (ptr\text{-}span\ p)$
 $(hmem\text{-}upd\ (heap\text{-}update\ p\ v)\ (htd\text{-}upd\ (\lambda\cdot.\ ptr\text{-}force\text{-}type\ p\ (htd\ s))\ s))$
 s
by (*simp add: equal-upto-def* *equal-upto-heap-on-heap-update*
equal-upto-heap-on-htd-upd *ptr-force-type-d* *sympD* *symp-equal-upto-heap-on*)

lemma *equal-upto-heap-stack-alloc*:
fixes *p*: *'a::mem-type ptr*
assumes $length\ bs = size\text{-}of\ TYPE('a)$
shows $equal\text{-}upto\text{-}heap\text{-}on\ (ptr\text{-}span\ p)$
 $(hmem\text{-}upd\ (heap\text{-}update\text{-}padding\ p\ v\ bs)\ (htd\text{-}upd\ (\lambda\cdot.\ ptr\text{-}force\text{-}type\ p\ (htd\ s))\ s))$
 s
apply (*subst* *equal-upto-heap-on-sym*)
using *assms*
apply (*clarsimp simp add: equal-upto-def* *equal-upto-heap-on-def*)
by (*smt* (*verit*, *del-insts*) *CTypes.mem-type-simps(2)* *heap.lense-axioms* *heap-update-nmem-same*
heap-update-padding-def *hmem-htd-upd* *lense.upd-cong* *ptr-force-type-d* *typing.lense-axioms*)

definition

$rel\text{-}alloc::\ addr\ set \Rightarrow addr\ set \Rightarrow addr\ set \Rightarrow 's \Rightarrow 's \Rightarrow 's \Rightarrow bool$
where
 $rel\text{-}alloc\ S\ M\ A\ t_0 \equiv \lambda s\ t.$
 $stack\text{-}free\ (htd\ s) \subseteq S \wedge$
 $stack\text{-}free\ (htd\ s) \cap A = \{\}\ \wedge$
 $stack\text{-}free\ (htd\ s) \cap M = \{\}\ \wedge$
 $t = frame\ A\ t_0\ s$

lemma *rel-alloc-fold-frame*: $rel\text{-}alloc\ S\ M\ A\ t_0\ s\ t \implies frame\ A\ t_0\ s = t$
by (*auto simp add: rel-alloc-def*)

lemma *rel-alloc-modifies-antimono*: $rel\text{-}alloc\ S\ M2\ A\ t_0\ s\ t \implies M1 \subseteq M2 \implies$

rel-alloc $S M A t_0 s t$
by (*auto simp add: rel-alloc-def*)

lemma *rel-alloc-stack-free-disjoint*:
rel-alloc $S M A t_0 s t \implies \text{ptr-span } p \subseteq A \implies \text{ptr-span } p \cap \text{stack-free } (\text{htd } s) = \{\}$
by (*auto simp add: rel-alloc-def*)

lemma *rel-alloc-stack-free-disjoint-trans*:
rel-alloc $S M A t_0 s' t \implies \text{htd } s' = \text{htd } s \implies \text{ptr-span } p \subseteq A \implies \text{ptr-span } p \cap \text{stack-free } (\text{htd } s) = \{\}$
by (*auto simp add: rel-alloc-def*)

lemma *rel-alloc-stack-free-disjoint'*:
rel-alloc $S M A t_0 s t \implies \text{ptr-span } p \subseteq A \implies \text{stack-free } (\text{htd } s) \cap \text{ptr-span } p = \{\}$
by (*auto simp add: rel-alloc-def*)

lemma *rel-alloc-stack-free-disjoint-trans'*:
rel-alloc $S M A t_0 s' t \implies \text{htd } s' = \text{htd } s \implies \text{ptr-span } p \subseteq A \implies \text{stack-free } (\text{htd } s) \cap \text{ptr-span } p = \{\}$
by (*auto simp add: rel-alloc-def*)

lemma *rel-alloc-stack-free-disjoint-field-lvalue*:
fixes $p:: 'a::\text{mem-type ptr}$
assumes *rel-alloc* $S M A t_0 s t \text{ ptr-span } p \subseteq A$
assumes *field-lookup* (*typ-info-t* $\text{TYPE}('a)$) $f 0 = \text{Some } (T, n)$
assumes *export-uinfo* $T = \text{typ-uinfo-t } (\text{TYPE}('b))$
shows $\{\&(p \rightarrow f)..\text{+size-of } \text{TYPE}('b::\text{c-type})\} \cap \text{stack-free } (\text{htd } s) = \{\}$
using *rel-alloc-stack-free-disjoint field-tag-sub assms export-size-of*
by *fastforce*

lemma *rel-alloc-stack-free-disjoint-field-lvalue-trans*:
fixes $p:: 'a::\text{mem-type ptr}$
assumes *rel-alloc* $S M A t_0 s' t \text{ htd } s' = \text{htd } s \text{ ptr-span } p \subseteq A$
assumes *field-lookup* (*typ-info-t* $\text{TYPE}('a)$) $f 0 = \text{Some } (T, n)$
assumes *export-uinfo* $T = \text{typ-uinfo-t } (\text{TYPE}('b))$
shows $\{\&(p \rightarrow f)..\text{+size-of } \text{TYPE}('b::\text{c-type})\} \cap \text{stack-free } (\text{htd } s) = \{\}$
using *rel-alloc-stack-free-disjoint field-tag-sub assms export-size-of*
by *fastforce*

lemma *rel-alloc-stack-free-disjoint-field-lvalue'*:
fixes $p:: 'a::\text{mem-type ptr}$
assumes *rel-alloc* $S M A t_0 s t \text{ ptr-span } p \subseteq A$
assumes *field-lookup* (*typ-info-t* $\text{TYPE}('a)$) $f 0 = \text{Some } (T, n)$
assumes *export-uinfo* $T = \text{typ-uinfo-t } (\text{TYPE}('b))$
shows $\text{stack-free } (\text{htd } s) \cap \{\&(p \rightarrow f)..\text{+size-of } \text{TYPE}('b::\text{c-type})\} = \{\}$
using *rel-alloc-stack-free-disjoint-field-lvalue [OF assms]* **by** *blast*

lemma *rel-alloc-stack-free-disjoint-field-lvalue-trans'*:

fixes p : 'a::mem-type ptr

assumes $rel\text{-}alloc\ S\ M\ A\ t_0\ s'\ t\ htd\ s' = htd\ s\ ptr\text{-}span\ p \subseteq A$

assumes $field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('a))\ f\ 0 = Some\ (T, n)$

assumes $export\text{-}uinfo\ T = typ\text{-}uinfo\text{-}t\ (TYPE('b))$

shows $stack\text{-}free\ (htd\ s) \cap \{\&(p \rightarrow f)..\text{+size-of}\ TYPE('b::c\text{-}type)\} = \{\}$

using $rel\text{-}alloc\text{-}stack\text{-}free\text{-}disjoint\text{-}field\text{-}lvalue\text{-}trans\ [OF\ assms]$ **by** *blast*

lemma *h-val-rel-alloc-disjoint*:

fixes p : 'a::mem-type ptr

shows $rel\text{-}alloc\ S\ M\ A\ t_0\ s\ t \implies ptr\text{-}span\ p \cap A = \{\} \implies ptr\text{-}span\ p \cap stack\text{-}free\ (htd\ s) = \{\} \implies h\text{-}val\ (hmem\ t)\ p = h\text{-}val\ (hmem\ s)\ p$

apply (*simp add: rel-alloc-def h-val-frame-disjoint*)

done

definition *rel-stack::*

$addr\ set \Rightarrow addr\ set \Rightarrow addr\ set \Rightarrow 's \Rightarrow 's \Rightarrow (heap\text{-}mem \Rightarrow 'b \Rightarrow 'c \Rightarrow bool) \Rightarrow$

$('b \times 's) \Rightarrow ('c \times 's) \Rightarrow bool$

where

$rel\text{-}stack\ S\ M\ A\ s\ t_0\ R = (\lambda(v, s')\ (w, t').$

$R\ (hmem\ s')\ v\ w \wedge$

$rel\text{-}alloc\ S\ M\ A\ t_0\ s'\ t' \wedge$

$equal\text{-}upto\ (M \cup stack\text{-}free\ (htd\ s'))\ (hmem\ s')\ (hmem\ s) \wedge$

$htd\ s' = htd\ s)$

lemma *rel-stack-unchanged-typing*: $rel\text{-}alloc\ S\ M'\ A\ t_0\ s\ t \implies rel\text{-}stack\ S\ M\ A\ s\ t_0\ R\ (v, s')\ (w, t') \implies$

$equal\text{-}on\ S\ (htd\ t')\ (htd\ t)$

by (*auto simp add: rel-alloc-def rel-stack-def Diff-triv equal-on-def htd-frame inf-commute*)

lemma *rel-stack-unchanged-heap*: $rel\text{-}alloc\ S\ M'\ A\ t_0\ s\ t \implies rel\text{-}stack\ S\ M\ A\ s\ t_0\ R\ (v, s')\ (w, t') \implies$

$equal\text{-}upto\ (M \cup stack\text{-}free\ (htd\ s'))\ (hmem\ t')\ (hmem\ t)$

apply (*clarsimp simp add: rel-alloc-def rel-stack-def Diff-triv equal-on-def htd-frame inf-commute*)

by (*smt (verit, ccfv-threshold) equal-on-def equal-on-stack-free-preservation equal-upto-def hmem-frame*)

lemma *rel-stack-unchanged-stack-free*:

$rel\text{-}alloc\ S\ M'\ A\ t_0\ s\ t \implies rel\text{-}stack\ S\ M\ A\ s\ t_0\ R\ (v, s')\ (w, t') \implies$

$stack\text{-}free\ (htd\ s') = stack\text{-}free\ (htd\ s)$

apply (*auto simp add: rel-alloc-def rel-stack-def Diff-triv htd-frame inf-commute*)

done

lemma *rel-stack-unchanged-stack-free'*:

assumes $stack$: $rel\text{-}alloc\ S\ M'\ A\ t_0\ s\ t$

assumes *rel-stack*: *rel-stack* $S M A s t_0 R (v, s') (w, t')$
shows *stack-free* (*htd* t') = *stack-free* (*htd* t)
using *stack rel-stack*
apply (*clarsimp simp add: rel-alloc-def rel-stack-def Diff-triv htd-frame inf-commute*)
subgoal
using *rel-stack-unchanged-stack-free* [*OF stack rel-stack*] *equal-on-stack-free-preservation*
by (*auto simp add: frame-def stack-free-def root-ptr-valid-def valid-root-footprint-def*)
done

lemma *rel-stack-unchanged-heap'*:
assumes *alloc*: *rel-alloc* $S \{\} A t_0 s t$
assumes *stack*: *rel-stack* $S M A s t_0 R (v, s') (w, t')$
shows *equal-upto* ($M \cup \text{stack-free (htd } t')$) (*hmem* t') (*hmem* t)
apply (*rule equal-upto-mono* [*OF - rel-stack-unchanged-heap*] [*OF alloc stack*])
using *stack-free-htd-frame stack*
apply (*auto simp add: rel-stack-def rel-alloc-def subset-iff*)
done

lemma *rel-stack-Exn*[*simp*]:
rel-stack $S M A s t_0 (\text{rel-xval-stack } L R) (\text{Exn } v, s') (\text{Exn } w, t') = \text{rel-stack } S M A s t_0 L (v, s') (w, t')$
by (*simp add: rel-stack-def*)

lemma *rel-stack-Exn-Result*[*simp*]:
rel-stack $S M A s t_0 (\text{rel-xval-stack } L R) (\text{Exn } v, s') (\text{Result } w, t') = \text{False}$
by (*simp add: rel-stack-def*)

lemma *rel-stack-Result*[*simp*]:
rel-stack $S M A s t_0 (\text{rel-xval-stack } L R) (\text{Result } v, s') (\text{Result } w, t') = \text{rel-stack } S M A s t_0 R (v, s') (w, t')$
by (*simp add: rel-stack-def*)

lemma *rel-zero-stack-Result-Exn*[*simp*]:
rel-stack $S M A s t_0 (\text{rel-xval-stack } L R) (\text{Result } v, s') (\text{Exn } w, t') = \text{False}$
by (*simp add: rel-stack-def*)

lemma *admissible-fail*: *ccpo.admissible* *Inf* ($\lambda x y. y \leq x$) ($\lambda A. \neg \text{succeeds } A t$)
apply (*rule ccpo.admissibleI*)
apply (*clarsimp simp add: ccpo.admissible-def chain-def succeeds-def reaches-def split: prod.splits xval-splits*)
apply *transfer*
by (*auto simp add: Inf-post-state-def top-post-state-def*)

lemma *fun-lub-lem*: ($\lambda(A::('f \Rightarrow (('d, 'e) \text{exception-or-result} \times 'f) \text{post-state}) \text{set}) x::'f.$
 $\sqcap f::'f \Rightarrow (('d, 'e) \text{exception-or-result} \times 'f) \text{post-state} \in A. f x) = \text{fun-lub}$
(*Inf*)
apply (*clarsimp simp add: fun-lub-def [abs-def]*)
by (*simp add: image-def*)

lemma *fun-ord-lem*:

```

    (λ(a::'f ⇒ (('d, 'e) exception-or-result × 'f) post-state)
      b::'f ⇒ (('d, 'e) exception-or-result × 'f) post-state. ∀ x::'f. b x ≤ a x)
= fun-ord (≥)
apply (simp add: fun-ord-def [abs-def])
done

```

lemma *admissible-refines-funp*:

```

assumes *: funp R
shows cppo.admissible Inf (≥) (λA. refines C A s t R)
apply (rule cppo.admissibleI)

apply (clarsimp simp add: cppo.admissible-def chain-def
  refines-def-old reaches-def succeeds-def)
apply (intro allI impI conjI)
subgoal
  apply transfer
  by (auto simp add: Inf-post-state-def top-post-state-def split: if-split-asm)
subgoal
  using *
  apply transfer
  apply (clarsimp simp add: funp-def fun-of-rel-def Inf-post-state-def top-post-state-def
  image-def vimage-def
    split: if-split-asm)
  by (metis post-state.distinct(1) outcomes.simps(2))
done

```

lemma *fun-of-rel-stack*:

```

assumes f: ∧h. fun-of-rel (Q h) (f h)
shows fun-of-rel (rel-stack S M A s t0 Q) (λ(r, s). (f (hmem s) r, frame A t0 s))
using f
by (auto simp add: fun-of-rel-def rel-stack-def rel-alloc-def)

```

lemma *funp-rel-stack*:

```

assumes funp: ∧h. funp (Q h)
shows funp (rel-stack S M A s t0 Q)
proof –
from funp obtain f where f: ∧h. fun-of-rel (Q h) (f h)
apply (clarsimp simp add: funp-def )
by metis
from fun-of-rel-stack [OF f]
have fun-of-rel (rel-stack S M A s t0 Q) (λ(r, s). (f (hmem s) r, frame A t0 s)).
then show ?thesis
  unfolding funp-def
  by blast
qed

```

```

theorem admissible-refines-rel-stack[corres-admissible]:
  assumes funp:  $\bigwedge h. \text{funp } (Q \ h)$ 
  shows ccpo.admissible Inf ( $\geq$ ) ( $\lambda g. \text{refines } f \ g \ s' \ t' \ (\text{rel-stack } S \ M \ A \ s \ t_0 \ Q)$ )
  apply (rule admissible-refines-funp)
  apply (rule funp-rel-stack)
  apply (rule funp)
  done

lemma admissible-rel-stack-eq: ccpo.admissible  $\cap$  ( $\lambda x \ y. \ y \subseteq x$ ) ( $\lambda X. \exists w \in X. (\lambda-. (=)) \ h \ v \ w$ )
  by (auto simp add: ccpo.admissible-def)

lemma admissible-rel-singleton-stack:
  shows ccpo.admissible  $\cap$  ( $\lambda x \ y. \ y \subseteq x$ ) ( $\lambda X. \exists w \in X. (\text{rel-singleton-stack } p) \ h \ v \ w$ )
  by (auto simp add: ccpo.admissible-def rel-singleton-stack-def)

lemma admissible-rel-push:
  assumes admiss:  $\bigwedge h \ v. \text{ccpo.admissible} \cap (\lambda x \ y. \ y \subseteq x) (\lambda X. \exists w \in X. Q \ h \ v \ w)$ 
  shows ccpo.admissible  $\cap$  ( $\lambda x \ y. \ y \subseteq x$ ) ( $\lambda X. \exists w \in X. (\text{rel-push } p \ Q) \ h \ v \ w$ )
proof –
  have ccpo.admissible  $\cap$  ( $\lambda x \ y. \ y \subseteq x$ ) ( $\lambda X. \exists w. (h\text{-val } h \ p, \ w) \in X \wedge Q \ h \ v \ w$ )
  proof (rule ccpo.admissibleI[rule-format])
    fix C::('c  $\times$  'b) set set
    assume chain: Complete-Partial-Order.chain ( $\lambda x \ y. \ y \subseteq x$ ) C
    assume nonempty: C  $\neq$  {}
    assume chain-prop: ( $\bigwedge X. X \in C \implies \exists w. (h\text{-val } h \ p, \ w) \in X \wedge Q \ h \ v \ w$ )
    show  $\exists w. (h\text{-val } h \ p, \ w) \in \bigcap C \wedge Q \ h \ v \ w$ 
    proof –
      define C' where C' = Set.filter ( $\lambda(v, w). \ v = h\text{-val } h \ p$ ) ' C
      have subset:  $\bigcap C' \subseteq \bigcap C$ 
      using C'-def by fastforce
      from chain have chain-C': Complete-Partial-Order.chain ( $\lambda x \ y. \ y \subseteq x$ ) C'
      using chain-prop C'-def
      by (simp add: chain-imageI subset-iff)
      have fst-C':  $\bigwedge X \ t. X \in C' \implies t \in \text{fst } ' X \implies t = h\text{-val } h \ p$ 
      using C'-def
      by auto
      from nonempty chain-prop have nonempty-C': C'  $\neq$  {}
      using C'-def
      by blast
      from chain-C' have result-chain: Complete-Partial-Order.chain ( $\lambda x \ y. \ y \subseteq x$ ) (( $\lambda X. \text{snd } ' X$ ) ' C')
      by (simp add: chain-imageI image-mono)
      from nonempty-C' have nonempty-result-chain: (( $\lambda X. \text{snd } ' X$ ) ' C')  $\neq$  {}
    by auto
    from chain-prop have result-chain-prop: ( $\bigwedge X. X \in (\cdot) \ \text{snd } ' C' \implies \exists w \in X. Q \ h \ v \ w$ )
  
```

```

    using C'-def
    by auto (metis (mono-tags, lifting) case-prodI member-filter snd-conv)
    from ccpo.admissibleD [OF admiss [of h v] result-chain nonempty-result-chain
result-chain-prop]
    obtain w where
      w: w ∈ ∩ ((λX. snd ' X) ' C') and Q: Q h v w
    by auto
    from w fst-C' have (h-val h p, w) ∈ ∩ C'
    by fastforce
    with Q subset show ?thesis by blast
  qed
qed
then show ?thesis
  by (clarsimp simp add: rel-push-def split-paired-Bex)
qed

```

lemma *admissible-rel-sum-stack*:

```

  assumes admiss-L: ∧h v. ccpo.admissible ∩ (λx y. y ⊆ x) (λX. ∃w ∈ X. L h v
w)
  assumes admiss-R: ∧h v. ccpo.admissible ∩ (λx y. y ⊆ x) (λX. ∃w ∈ X. R h
v w)
  shows ccpo.admissible ∩ (λx y. y ⊆ x) (λX. ∃w ∈ X. (rel-sum-stack L R) h v
w)
proof –
  have ccpo.admissible ∩ (λx y. y ⊆ x) (λX. ∃w ∈ X. rel-sum (L h) (R h) v w)
proof (rule ccpo.admissibleI[rule-format])
  fix C::('c + 'e) set set
  assume chain: Complete-Partial-Order.chain (λx y. y ⊆ x) C
  assume nonempty: C ≠ {}
  assume chain-prop: ∧X. X ∈ C ⇒ ∃w ∈ X. rel-sum (L h) (R h) v w
  show ∃w ∈ ∩ C. rel-sum (L h) (R h) v w
proof (cases v)
  case (Inl l)
  define C' where C' = Set.filter (Sum-Type.isl) ' C
  have subset: ∩ C' ⊆ ∩ C
    using C'-def by fastforce
  from chain have chain-C': Complete-Partial-Order.chain (λx y. y ⊆ x) C'
    using chain-prop C'-def
  by (simp add: chain-imageI subset-iff)
  from nonempty chain-prop have nonempty-C': C' ≠ {}
  using C'-def
  by blast
  have prop-C': ∧X t. X ∈ C' ⇒ t ∈ X ⇒ Sum-Type.isl t
  using C'-def
  by auto
  from Inl chain-prop have chain-prop':
    ∧X. X ∈ C' ⇒ ∃w ∈ X. ∃l'. w = Inl l' ∧ L h l l'
  by (fastforce simp add: rel-sum.simps C'-def)
  have ∃w ∈ ∩ C. ∃l'. w = Inl l' ∧ L h l l'

```

```

proof –
  from chain-C' have result-chain: Complete-Partial-Order.chain ( $\lambda x y. y \subseteq x$ ) (( $\lambda X. \text{Sum-Type.proj1 } 'X$ ) ' C')
    by (simp add: chain-imageI image-mono)
  from nonempty-C' have nonempty-result-chain: (( $\lambda X. \text{Sum-Type.proj1 } 'X$ ) ' C')  $\neq \{\}$  by auto

  from chain-prop' have result-chain-prop:  $\bigwedge X. X \in (\lambda X. \text{Sum-Type.proj1 } 'X) ' C' \implies \exists w \in X. L\ h\ l\ w$ 
    using image-iff by fastforce
  from ccpo.admissibleD [OF admiss-L [of h l] result-chain nonempty-result-chain result-chain-prop]
    obtain w where
      w:  $w \in \bigcap ((\cdot) \text{proj1 } 'C')$  and L: L h l w
    by auto
  from w prop-C' have Inl w  $\in \bigcap C'$  by fastforce
  with L subset show ?thesis by blast
qed
then show ?thesis by (simp add: Inl rel-sum.simps)
next
case (Inr r)
define C' where C' = Set.filter (Not o Sum-Type.isl) ' C
have subset:  $\bigcap C' \subseteq \bigcap C$ 
  using C'-def by fastforce
from chain have chain-C': Complete-Partial-Order.chain ( $\lambda x y. y \subseteq x$ ) C'
  using chain-prop C'-def
  by (simp add: chain-imageI subset-iff)
from nonempty chain-prop have nonempty-C': C'  $\neq \{\}$ 
  using C'-def
  by blast
have prop-C':  $\bigwedge X t. X \in C' \implies t \in X \implies \neg (\text{Sum-Type.isl } t)$ 
  using C'-def
  by auto
from Inr chain-prop have chain-prop':
   $\bigwedge X. X \in C' \implies \exists w \in X. \exists r'. w = \text{Inr } r' \wedge R\ h\ r\ r'$ 
  by (fastforce simp add: rel-sum.simps C'-def)
have  $\exists w \in \bigcap C. \exists r'. w = \text{Inr } r' \wedge R\ h\ r\ r'$ 
proof –
  from chain-C' have result-chain: Complete-Partial-Order.chain ( $\lambda x y. y \subseteq x$ ) (( $\lambda X. \text{Sum-Type.proj1 } 'X$ ) ' C')
    by (simp add: chain-imageI image-mono)
  from nonempty-C' have nonempty-result-chain: (( $\lambda X. \text{Sum-Type.proj1 } 'X$ ) ' C')  $\neq \{\}$  by auto

  from chain-prop' have result-chain-prop:  $\bigwedge X. X \in (\lambda X. \text{Sum-Type.proj1 } 'X) ' C' \implies \exists w \in X. R\ h\ r\ w$ 
    using image-iff by fastforce
  from ccpo.admissibleD [OF admiss-R [of h r] result-chain nonempty-result-chain result-chain-prop]

```


obtain w **where**
 $w: w \in \bigcap ((\cdot) \text{ projr } \cdot C')$ **and** $L: R h r w$
by *auto*
from $w \text{ prop-}C'$ **have** $\text{Inr } w \in \bigcap C'$ **by** *fastforce*
with $L \text{ subset}$ **show** *?thesis* **by** *blast*
qed
then show *?thesis* **by** (*simp add: Inr rel-sum.simps*)
qed
then show *?thesis* **by** (*simp add: rel-sum-stack-def*)
qed

lemma *IOcorres-refines-conv:*

assumes *rel-to-prj*: $\bigwedge h. \text{fun-of-rel } (R h) (\text{prj } h)$
assumes *rel-to-prjE*: $\bigwedge h. \text{fun-of-rel } (L h) (\text{prjE } h)$
shows *IOcorres*
 $(\lambda s. \text{rel-alloc } \mathcal{S} M A t_0 s (\text{frame } A t_0 s) \wedge P s)$
 $(\lambda s r s'. \text{stack-free } (\text{htd } s') \subseteq \mathcal{S} \wedge \text{stack-free } (\text{htd } s') \cap A = \{\} \wedge \text{stack-free}$
 $(\text{htd } s') \cap M1 = \{\} \wedge$
 $\text{equal-upto } (M1 \cup \text{stack-free } (\text{htd } s')) (\text{hmem } s') (\text{hmem } s) \wedge$
 $\text{htd } s' = \text{htd } s \wedge$
 $(\text{case } r \text{ of } \text{Exn } l \Rightarrow \exists e. l = \text{Nonlocal } e \wedge L (\text{hmem } s') e (\text{prjE}$
 $(\text{hmem } s') e) | \text{Result } x \Rightarrow R (\text{hmem } s') x (\text{prj } (\text{hmem } s') x)))$
 $(\text{frame } A t_0)$
 $(\lambda r s. \text{prj } (\text{hmem } s) r)$
 $(\lambda r s. \text{prjE } (\text{hmem } s) (\text{the-Nonlocal } r))$
 $g f$
 \longleftrightarrow
 $(\forall s t. \text{rel-alloc } \mathcal{S} M A t_0 s t \longrightarrow P s \longrightarrow \text{refines } f g s t (\text{rel-stack } \mathcal{S} M1 A$
 $s t_0 (\text{rel-xval-stack } (\text{rel-exit } L) R)))$
apply *standard*
subgoal
apply (*clarsimp simp add: IOcorres-def corresXF-post-def*)
apply (*rule refinesI*)
subgoal
using *rel-alloc-fold-frame* **by** *blast*

subgoal for $s t v s'$

apply (*erule-tac x=s in allE*)
apply (*subst (asm) (1 2) rel-alloc-def*)
apply *safe*
apply (*clarsimp*)
apply (*erule-tac x=v in allE*)
apply (*erule-tac x=s' in allE*)
subgoal
apply (*clarsimp split: xval-splits*)
subgoal for x
apply (*rule exI[where x=Exn (prjE (hmem s') x)]*)

```

    apply (rule exI[where x=frame A t0 s'])
    apply (clarsimp simp add: rel-stack-def rel-alloc-def fun-of-rel-def)
    done
  subgoal for x
    apply (rule exI[where x=Result (prj (hmem s') x)])
    apply (rule exI[where x=frame A t0 s'])
    apply (clarsimp simp add: rel-stack-def rel-alloc-def fun-of-rel-def)
    done
  done
done
done
subgoal
  apply (clarsimp simp add: IOcorres-def corresXF-post-def)
  subgoal for s
    apply (rule conjI)
  subgoal
    apply clarsimp
    subgoal for v s'
      apply (erule-tac x=s in allE)
      apply (erule-tac x=(frame A t0 s) in allE)
      apply (clarsimp simp add: rel-alloc-def refines-def-old)
      apply (erule-tac x=v in allE)
      apply (erule-tac x=s' in allE)
    subgoal
      apply (cases v)
    subgoal
      using rel-to-prjE by (clarsimp simp add: rel-stack-def rel-alloc-def
fun-of-rel-def rel-exit-def default-option-def
      Exn-def [symmetric])
    subgoal
      using rel-to-prj by (clarsimp simp add: rel-stack-def rel-alloc-def
fun-of-rel-def default-option-def
      Exn-def [symmetric])
    done
  done
done
done
subgoal
  by (auto simp add: refines-def-old)
done
done
done

```

lemmas IOcorres-to-refines = iffD1 [OF IOcorres-refines-conv, rule-format]

lemmas refines-to-IOcorres = iffD2 [OF IOcorres-refines-conv, rule-format]

lemma refines-rel-xval-stack-generalise-exit:

$$\begin{aligned} & \text{refines } f \ g \ s \ t \ (\text{rel-stack } \mathcal{S} \ M \ A \ s \ t_0 \ (\text{rel-xval-stack } L \ R)) \implies (\bigwedge h \ v \ w. L \ h \ v \ w \\ & \implies L' \ h \ v \ w) \implies \\ & \text{refines } f \ g \ s \ t \ (\text{rel-stack } \mathcal{S} \ M \ A \ s \ t_0 \ (\text{rel-xval-stack } L' \ R)) \end{aligned}$$

by (*erule refines-weaken*) (*auto simp add: rel-stack-def rel-xval-stack-def rel-xval.simps*)

lemma *L2-gets-rel-stack'*:

assumes R (*hmem* s) (e s) (e' (*frame* A t_0 s))

assumes *rel-alloc* \mathcal{S} M A t_0 s t

shows *refines* (*L2-gets* e ns) (*L2-gets* e' ns) s t (*rel-stack* \mathcal{S} $\{\}$ A s t_0 (*rel-xval-stack* L R))

using *assms*

by (*auto simp add: refines-def-old L2-defs rel-stack-def rel-alloc-def*)

lemma *L2-gets-rel-stack*:

assumes R (*hmem* s) (e s) (e' (*frame* A t_0 s))

assumes *rel-alloc* \mathcal{S} M A t_0 s t

shows *refines* (*L2-gets* e ns) (*L2-gets* e' ns) s t (*rel-stack* \mathcal{S} $\{\}$ A s t_0 (*rel-xval-stack* L R))

using *assms*

by (*auto simp add: refines-def-old L2-defs rel-stack-def rel-alloc-def*)

lemma *L2-gets-rel-stack-guarded*:

assumes $G \implies R$ (*hmem* s) (e s) (e' (*frame* A t_0 s))

assumes *rel-alloc* \mathcal{S} M A t_0 s t

shows *refines* (*L2-gets* e ns) (*L2-seq* (*L2-guard* (λ -. G)) (λ -. (*L2-gets* e' ns))) s t (*rel-stack* \mathcal{S} $\{\}$ A s t_0 (*rel-xval-stack* L R))

using *assms*

by (*auto simp add: refines-def-old L2-defs rel-stack-def rel-alloc-def succeeds-bind reaches-bind*)

lemma *L2-gets-constant-trivial-rel-stack*:

assumes *rel-alloc* \mathcal{S} M A t_0 s t

shows *refines* (*L2-gets* (λ -. c) ns) (*L2-gets* (λ -. c) ns) s t (*rel-stack* \mathcal{S} $\{\}$ A s t_0 (*rel-xval-stack* L (λ -. (=))))

using *assms*

by (*auto simp add: refines-def-old L2-defs rel-stack-def rel-alloc-def*)

lemma *L2-gets-constant-rel-stack*:

assumes R (*hmem* s) c c'

assumes *rel-alloc* \mathcal{S} M A t_0 s t

shows *refines* (*L2-gets* (λ -. c) ns) (*L2-gets* (λ -. c') ns) s t (*rel-stack* \mathcal{S} $\{\}$ A s t_0 (*rel-xval-stack* L R))

using *assms*

by (*auto simp add: refines-def-old L2-defs rel-stack-def rel-alloc-def*)

lemma *L2-throw-rel-stack*:

assumes L (*hmem* s) c c'

assumes *rel-alloc* \mathcal{S} M A t_0 s t

shows *refines* (*L2-throw* c ns) (*L2-throw* c' ns) s t (*rel-stack* \mathcal{S} $\{\}$ A s t_0 (*rel-xval-stack* L R))

using *assms*

by (*auto simp add: refines-def-old L2-defs rel-stack-def rel-alloc-def Exn-def [symmetric]*)

lemma *L2-try-rel-stack:*

assumes *s-t: rel-alloc S M A t₀ s t*

assumes *refines f g s t (rel-stack S M1 A s t₀ (rel-xval-stack (rel-sum-stack L R) R))*

shows *refines (L2-try f) (L2-try g) s t (rel-stack S M1 A s t₀ (rel-xval-stack L R))*

using *assms*

apply (*clarsimp simp add: refines-def-old L2-defs rel-stack-def rel-alloc-def reaches-try*)

subgoal for *s' r'*

apply (*cases r'*)

apply (*clarsimp simp add: default-option-def Exn-def [symmetric]*)

subgoal for *y*

apply (*cases y*)

subgoal by *fastforce*

subgoal by *fastforce*

done

subgoal

apply *fastforce*

done

done

done

lemma *L2-try-rel-stack-merge1:*

assumes $\bigwedge h v v'. R' h v v' \implies R h v v'$

assumes *s-t: rel-alloc S M A t₀ s t*

assumes *refines f g s t (rel-stack S M1 A s t₀ (rel-xval-stack (rel-sum-stack L R') R))*

shows *refines (L2-try f) (L2-try g) s t (rel-stack S M1 A s t₀ (rel-xval-stack L R))*

using *assms*

apply (*clarsimp simp add: refines-def-old L2-defs rel-stack-def rel-alloc-def reaches-try*)

subgoal for *s' r'*

apply (*cases r'*)

apply (*clarsimp simp add: default-option-def Exn-def [symmetric]*)

subgoal for *y*

apply (*cases y*)

subgoal by *fastforce*

subgoal by *fastforce*

done

subgoal

apply *fastforce*

done

done

done

lemma *L2-try-rel-stack-merge2:*

assumes $\bigwedge h v v'. R' h v v' \implies R h v v'$

assumes s - t : *rel-alloc* $\mathcal{S} M A t_0 s t$
assumes *refines* $f g s t$ (*rel-stack* $\mathcal{S} M1 A s t_0$ (*rel-xval-stack* (*rel-sum-stack* $L R$) R'))
shows *refines* (*L2-try* f) (*L2-try* g) $s t$ (*rel-stack* $\mathcal{S} M1 A s t_0$ (*rel-xval-stack* $L R$))
using *assms*
apply (*clarsimp simp add: refines-def-old L2-defs rel-stack-def rel-alloc-def reaches-try*)
subgoal for $s' r'$
apply (*cases* r')
apply (*clarsimp simp add: default-option-def Exn-def [symmetric]*)
subgoal for y
apply (*cases* y)
subgoal by *fastforce*
subgoal by *fastforce*
done
subgoal
apply *fastforce*
done
done
done

lemma *L2-guard-rel-stack*:
assumes e' (*frame* $A t_0 s$) $\implies e s$
assumes *rel-alloc* $\mathcal{S} M A t_0 s t$
shows *refines* (*L2-guard* e) (*L2-guard* e') $s t$ (*rel-stack* $\mathcal{S} \{\}$ $A s t_0$ (*rel-xval-stack* $L (\lambda \cdot (=))$))
using *assms*
by (*auto simp add: refines-def-old L2-defs rel-stack-def rel-alloc-def*)

lemma *L2-modify-heap-update-rel-stack'*:
fixes $p::'a::$ *mem-type ptr*
assumes R (*heap-update* $p v$ (*hmem* s)) $() (e' (\text{frame } A t_0 s))$
assumes *ptr-span* $p \subseteq A$
assumes *rel-alloc* $\mathcal{S} (\text{ptr-span } p) A t_0 s t$
shows *refines*
 $(\text{L2-modify } (\text{hmem-upd } (\text{heap-update } p v)))$
 $(\text{L2-gets } e' ns)$
 $s t$
 $(\text{rel-stack } \mathcal{S} (\text{ptr-span } p) A s t_0 (\text{rel-xval-stack } L R))$
using *assms*
by (*auto simp add: refines-def-old L2-defs rel-stack-def rel-alloc-def frame-heap-update equal-upto-heap-update*)

lemma *L2-modify-heap-update-rel-stack*:
fixes $p::'a::$ *mem-type ptr*
assumes R (*heap-update* $p (v s)$ (*hmem* s)) $() (e' (\text{frame } A t_0 s))$
assumes *rel-alloc* $\mathcal{S} M A t_0 s t$
assumes *ptr-span* $p \subseteq A$
assumes *ptr-span* $p \subseteq M$

shows *refines*
 (*L2-modify* ($\lambda s. \text{hmem-upd} (\text{heap-update } p (v \ s)) \ s$))
 (*L2-gets* $e' \ ns$)
 $s \ t$
 (*rel-stack* $\mathcal{S} (\text{ptr-span } p) \ A \ s \ t_0 (\text{rel-xval-stack } L \ R)$)
using *assms*
by (*auto simp add: refines-def-old L2-defs rel-stack-def rel-alloc-def*
frame-heap-update equal-upto-heap-update)

lemma *L2-modify-keep-heap-update-rel-stack:*

fixes $p::'a:: \text{mem-type } \text{ptr}$
assumes $v \ s = v' (\text{frame } A \ t_0 \ s)$
assumes *rel-alloc* $\mathcal{S} \ M \ A \ t_0 \ s \ t$
assumes $\text{ptr-span } p \cap A = \{\}$
assumes $\text{ptr-span } p \cap \text{stack-free} (\text{htd } s) = \{\}$
assumes $\text{ptr-span } p \subseteq M$
shows *refines*
 (*L2-modify* ($\lambda s. \text{hmem-upd} (\text{heap-update } p (v \ s)) \ s$))
 (*L2-modify* ($\lambda s. \text{hmem-upd} (\text{heap-update } p (v' \ s)) \ s$))
 $s \ t$
 (*rel-stack* $\mathcal{S} (\text{ptr-span } p) \ A \ s \ t_0 (\text{rel-xval-stack } L (\lambda-. (=))))$)
using *assms*
by (*auto simp add: refines-def-old L2-defs rel-stack-def rel-alloc-def*
frame-heap-update disjoint-subset2 equal-upto-heap-update
frame-heap-update-disjoint)

lemma *L2-modify-keep-heap-update-rel-stack-guarded:*

fixes $p::'a:: \text{mem-type } \text{ptr}$
assumes $G \implies v \ s = v' (\text{frame } A \ t_0 \ s)$
assumes *rel-alloc* $\mathcal{S} \ M \ A \ t_0 \ s \ t$
assumes $G \implies \text{ptr-span } p \cap A = \{\}$
assumes $G \implies \text{ptr-span } p \cap \text{stack-free} (\text{htd } s) = \{\}$
assumes $G \implies \text{ptr-span } p \subseteq M$
shows *refines*
 (*L2-modify* ($\lambda s. \text{hmem-upd} (\text{heap-update } p (v \ s)) \ s$))
 (*L2-seq* (*L2-guard* ($\lambda-. G$)) ($\lambda-. (\text{L2-modify} (\lambda s. \text{hmem-upd} (\text{heap-update}$
 $p (v' \ s)) \ s))))$))
 $s \ t$
 (*rel-stack* $\mathcal{S} (\text{ptr-span } p) \ A \ s \ t_0 (\text{rel-xval-stack } L (\lambda-. (=))))$)
apply (*rule refines-L2-guard-right*)
apply (*rule L2-modify-keep-heap-update-rel-stack*)
using *assms* **by** *auto*

lemma *L2-call-rel-stack':*

assumes *rel-alloc* $\mathcal{S} \ M \ A \ t_0 \ s \ t$
assumes $L': \bigwedge e \ e' \ s. L (\text{hmem } s) \ e \ e' \implies L' (\text{hmem } s) (\text{emb } e) (\text{emb}' e')$
assumes $f: \text{refines } f \ g \ s \ t$
 (*rel-stack* $\mathcal{S} \ M1 \ A \ s \ t_0 (\text{rel-xval-stack } L \ R)$)

shows *refines*
 (L2-call *f emb ns*)
 (L2-call *g emb' ns'*)
s t
 (rel-stack \mathcal{S} *M1 A s t₀* (rel-xval-stack *L' R*))
using *assms*
apply (*clarsimp simp add: L2-call-def refines-def-old reaches-map-value*)
subgoal for *s' r'*
apply (*cases r'*)
subgoal
by (*fastforce simp add: default-option-def Exn-def [symmetric]*
rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps)
subgoal
by (*fastforce simp add: rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps*)
done
done

lemma *L2-call-rel-stack''*:
assumes *rel-alloc S M A t₀ s t*
assumes *L': $\bigwedge e e' s. L (hmem s) e e' \implies rel-sum-stack L R' (hmem s) (emb e)$*
(emb' e')
assumes *f: refines f g s t*
(rel-stack S M1 A s t₀ (rel-xval-stack L R))
shows *refines*
 (L2-call *f emb ns*)
 (L2-call *g emb' ns'*)
s t
 (rel-stack \mathcal{S} *M1 A s t₀* (rel-xval-stack (rel-sum-stack *L R'*) *R*))
apply (*rule L2-call-rel-stack' [OF assms(1) - f]*)
by (*rule L'*)

lemma *L2-call-rel-stack*:
assumes *rel-alloc S M A t₀ s t*
assumes *L': $\bigwedge e e' s. L (hmem s) e e' \implies L' (hmem s) (emb e) (emb e')$*
assumes *f: refines f g s t*
(rel-stack S M1 A s t₀ (rel-xval-stack L R))
shows *refines*
 (L2-call *f emb ns*)
 (L2-call *g emb ns'*)
s t
 (rel-stack \mathcal{S} *M1 A s t₀* (rel-xval-stack *L' R*))
apply (*rule L2-call-rel-stack' [OF assms (1) - f]*)
by (*rule L'*)

Currently exceptions on the function level are terminal and are propagated to the toplevel. So the result relation for *Inl* is equality.

lemma *L2-call-rel-stack-eq-InL*:
assumes *rel-alloc S M A t₀ s t*

assumes f : *refines* f g s t
 $(rel\text{-}stack \mathcal{S} M1 A s t_0 (rel\text{-}xval\text{-}stack (\lambda\text{-} (=)) R))$
shows *refines*
 $(L2\text{-}call f emb ns)$
 $(L2\text{-}call g emb ns')$
 s t
 $(rel\text{-}stack \mathcal{S} M1 A s t_0 (rel\text{-}xval\text{-}stack (\lambda\text{-} (=)) R))$
apply (*rule* $L2\text{-}call\text{-}rel\text{-}stack'$ [*OF* *assms*(1) - f])
by *simp*

lemma $L2\text{-}call\text{-}rel\text{-}stack\text{-}bot\text{-}InL$:

assumes $rel\text{-}alloc \mathcal{S} M A t_0 s t$
assumes f : *refines* f g s t
 $(rel\text{-}stack \mathcal{S} M1 A s t_0 (rel\text{-}xval\text{-}stack (\lambda\text{-} - \text{-} False) R))$
shows *refines*
 $(L2\text{-}call f emb ns)$
 $(L2\text{-}call g emb ns')$
 s t
 $(rel\text{-}stack \mathcal{S} M1 A s t_0 (rel\text{-}xval\text{-}stack (\lambda\text{-} - \text{-} False) R))$
apply (*rule* $L2\text{-}call\text{-}rel\text{-}stack'$ [*OF* *assms*(1) - f])
by *simp*

lemma $L2\text{-}seq\text{-}rel\text{-}stack''$:

assumes $s\text{-}t$: $rel\text{-}alloc \mathcal{S} M A t_0 s t$
assumes $f1\text{-}g1$: *refines* $f1$ $g1$ s t ($rel\text{-}stack \mathcal{S} M1 A s t_0 (rel\text{-}xval\text{-}stack L R1)$)
assumes $f2\text{-}g2$: $\bigwedge s' t' v w. R1 (hmem s') v w \implies rel\text{-}alloc \mathcal{S} M A t_0 s' t' \implies$
 $equal\text{-}upto (M1 \cup stack\text{-}free (htd s')) (hmem s') (hmem s) \implies$
 $htd s' = htd s \implies$

$refines (f2 v) (g2' s' w) s' t' (rel\text{-}stack \mathcal{S} M2 A s' t_0 (rel\text{-}xval\text{-}stack L R))$

assumes $g2'\text{-}g2$: $\bigwedge s' t' v w. R1 (hmem s') v w \implies rel\text{-}alloc \mathcal{S} M A t_0 s' t' \implies$
 $equal\text{-}upto (M1 \cup stack\text{-}free (htd s')) (hmem s') (hmem s) \implies$
 $htd s' = htd s \implies$
 $g2' s' w = g2 w$

assumes $M1$: $M1 \subseteq M$

assumes $M2$: $M2 \subseteq M$

shows *refines* ($L2\text{-}seq f1 f2$) ($L2\text{-}seq g1 g2$) s t ($rel\text{-}stack \mathcal{S} (M1 \cup M2) A s t_0$)
 $(rel\text{-}xval\text{-}stack L R)$

proof –

from $f2\text{-}g2$ $g2'\text{-}g2$

have $f2\text{-}g2$: $\bigwedge s' t' v w. R1 (hmem s') v w \implies rel\text{-}alloc \mathcal{S} M A t_0 s' t' \implies$
 $equal\text{-}upto (M1 \cup stack\text{-}free (htd s')) (hmem s') (hmem s) \implies$
 $htd s' = htd s \implies$

$refines (f2 v) (g2 w) s' t' (rel\text{-}stack \mathcal{S} M2 A s' t_0 (rel\text{-}xval\text{-}stack L R))$

by *metis*

show *?thesis*


```

apply (rule refines-L2-seq)
  apply (rule f1-g1)
  apply (simp-all)
subgoal
  using s-t M1 M2
  apply (clarsimp simp add: rel-stack-def, intro conjI)
subgoal
  by (smt (verit, best) Un-commute disjoint-subset2 disjoint-union-distrib
    equal-on-stack-free-preservation rel-alloc-def sup.coboundedI2)
subgoal
  using equal-upto-mono
  by (metis inf-sup-ord(3) order-le-less sup-mono)
done
apply (rule refines-mono [OF - f2-g2 ])
subgoal
  by (auto simp add: rel-stack-def rel-alloc-def equal-upto-def)
subgoal
  by (auto simp add: rel-stack-def rel-alloc-def)
subgoal
  using M1 M2 s-t
  by (auto simp add: rel-stack-def rel-alloc-def)
subgoal
  by (auto simp add: rel-stack-def)
subgoal
  by (auto simp add: rel-stack-def)
done
qed

lemma L2-seq-rel-stack:
  assumes s-t: rel-alloc  $\mathcal{S}$  M A t0 s t
  assumes f1-g1: refines f1 g1 s t (rel-stack  $\mathcal{S}$  M1 A s t0 (rel-xval-stack L R1))
  assumes f2-g2:  $\bigwedge s' t' v w. R1$  (hmem s') v w  $\implies$  rel-alloc  $\mathcal{S}$  M A t0 s' t'  $\implies$ 
    equal-upto (M1  $\cup$  stack-free (htd s')) (hmem s') (hmem s)  $\implies$ 
    htd s' = htd s  $\implies$ 

    refines (f2 v) (g2' w s') s' t' (rel-stack  $\mathcal{S}$  M2 A s' t0 (rel-xval-stack L R))
  assumes g2'-g2:  $\bigwedge s' t' v w. R1$  (hmem s') v w  $\implies$  rel-alloc  $\mathcal{S}$  M A t0 s' t'  $\implies$ 
    equal-upto (M1  $\cup$  stack-free (htd s')) (hmem s') (hmem s)  $\implies$ 
    htd s' = htd s  $\implies$ 
    g2' w s' = g2 w

  assumes M1: M1  $\subseteq$  M
  assumes M2: M2  $\subseteq$  M
  assumes M': M' = M1  $\cup$  M2
  shows refines (L2-seq f1 f2) (L2-seq g1 g2) s t (rel-stack  $\mathcal{S}$  M' A s t0 (rel-xval-stack
L R))
proof -
  from f2-g2 g2'-g2
  have f2-g2:  $\bigwedge s' t' v w. R1$  (hmem s') v w  $\implies$  rel-alloc  $\mathcal{S}$  M A t0 s' t'  $\implies$ 

```

```

      equal-upto (M1 ∪ stack-free (htd s')) (hmem s') (hmem s) ⇒
      htd s' = htd s ⇒

      refines (f2 v) (g2 w) s' t' (rel-stack S M2 A s' t0 (rel-xval-stack L R))
    by metis
  show ?thesis
  apply (rule refines-L2-seq)
    apply (rule f1-g1)
    apply (simp-all)
  subgoal
    using s-t M1 M2 M'
    apply (clarsimp simp add: rel-stack-def, intro conjI)
  subgoal
    by (metis (no-types, lifting) Int-Un-distrib rel-alloc-def sup.absorb-iff1)

  subgoal
    using equal-upto-mono
    by (metis inf-sup-ord(3) order-le-less sup-mono)
  done
  apply (rule refines-mono [OF - f2-g2 ])
  subgoal
    apply (clarsimp simp add: rel-stack-def rel-alloc-def, intro conjI)
  subgoal by auto (metis M1 M2 M' UnE disjoint-iff equal-on-stack-free-preservation)
  subgoal by (smt (verit, best) M1 M2 M' Un-iff equal-on-stack-free-preservation
equal-upto-def)
  done
  subgoal
    by (auto simp add: rel-stack-def rel-alloc-def)
  subgoal
    using M1 M2 s-t
    apply (auto simp add: rel-stack-def rel-alloc-def)
  done
  subgoal
    by (auto simp add: rel-stack-def)
  subgoal
    by (auto simp add: rel-stack-def)
  done
qed

lemma L2-seq-rel-stack-g2-normalised:
  assumes s-t: rel-alloc S M A t0 s t
  assumes f1-g1: refines f1 g1 s t (rel-stack S M1 A s t0 (rel-xval-stack L R1))
  assumes f2-g2: ∧s' t' v w. R1 (hmem s') v w ⇒ rel-alloc S M A t0 s' t' ⇒
    equal-upto (M1 ∪ stack-free (htd s')) (hmem s') (hmem s) ⇒
    htd s' = htd s ⇒
    refines (f2 v) (g2 w) s' t' (rel-stack S M2 A s' t0 (rel-xval-stack L R))
  assumes M1: M1 ⊆ M
  assumes M2: M2 ⊆ M
  assumes M': M' = M1 ∪ M2

```

shows *refines* (*L2-seq* *f1 f2*) (*L2-seq* *g1 g2*) *s t* (*rel-stack* \mathcal{S} $M' A$ $s t_0$ (*rel-xval-stack* $L R$))

by (*rule* *L2-seq-rel-stack* [*OF* *s-t* *f1-g1* *f2-g2* - $M1 M2 M'$]) *auto*

lemma *L2-seq-rel-stack'*:

assumes *s-t*: *rel-alloc* \mathcal{S} $M A t_0 s t$

assumes *f1-g1*: *refines* *f1 g1 s t* (*rel-stack* \mathcal{S} $M1 A s t_0$ (*rel-xval-stack* $L R1$))

assumes *f2-g2*: $\bigwedge s' t' v w. R1$ (*hmem* s') $v w \implies$ *rel-alloc* \mathcal{S} $M A t_0 s' t' \implies$
refines (*f2 v*) (*g2 w*) $s' t'$ (*rel-stack* \mathcal{S} $M2 A s' t_0$ (*rel-xval-stack* $L R$))

assumes $M1: M1 \subseteq M$

assumes $M2: M2 \subseteq M$

assumes $M': M' = M1 \cup M2$

shows *refines* (*L2-seq* *f1 f2*) (*L2-seq* *g1 g2*) *s t* (*rel-stack* \mathcal{S} $M' A s t_0$ (*rel-xval-stack* $L R$))

proof –

show *?thesis*

apply (*rule* *refines-L2-seq*)

apply (*rule* *f1-g1*)

apply (*simp-all*)

subgoal

using *s-t* $M1 M2 M'$

apply (*clarsimp simp add: rel-stack-def, intro conjI*)

subgoal

by (*metis* (*no-types, lifting*) *Int-Un-distrib rel-alloc-def sup.absorb-iff1*)

subgoal

using *equal-upto-mono*

by (*metis* *inf-sup-ord(3) order-le-less sup-mono*)

done

apply (*rule* *refines-mono* [*OF* - *f2-g2*])

subgoal

apply (*clarsimp simp add: rel-stack-def rel-alloc-def, intro conjI*)

subgoal by *auto* (*metis* $M1 M2 M' UnE$ *disjoint-iff equal-on-stack-free-preservation*)

subgoal by (*smt* (*verit, best*) $M1 M2 M' Un$ -*iff equal-on-stack-free-preservation*

equal-upto-def)

done

subgoal

by (*auto simp add: rel-stack-def rel-alloc-def*)

subgoal

using $M1 M2 s-t$

apply (*auto simp add: rel-stack-def rel-alloc-def*)

done

done

qed

lemma *frame-raw-frame-union*:

assumes *disj-A-B*: $A \cap B = \{\}$

assumes *sf*: *stack-free* (*htd* s) $\cap B = \{\}$

shows $\text{frame } (A \cup B) (\text{raw-frame } B \ s \ t_0) \ s = \text{frame } A \ t_0 \ s$
apply (*clarsimp simp add: frame-def raw-frame-def*)
using *override-on-frame-raw-frame-lemma1 [OF disj-A-B sf]*
override-on-frame-raw-frame-lemma2 [OF disj-A-B sf]
by (*metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong*)

lemma *refines-narrow-frame'*:

assumes *disj-B-M*: $B \cap M = \{\}$
assumes *disj-A-B*: $A \cap B = \{\}$
assumes *spec*: $\bigwedge t_0 \ s \ t. \text{rel-alloc } \mathcal{S} \ M1 \ (A \cup B) \ t_0 \ s \ t \implies$
refines f g s t (rel-stack S M (A U B) s t_0 Q)
assumes *sf*: $\text{stack-free } (\text{htd } s) \cap B = \{\}$
assumes *rel-alloc*: $\text{rel-alloc } \mathcal{S} \ M1 \ A \ t_0 \ s \ t$
shows *refines f g s t (rel-stack S M A s t_0 Q)*

proof –

from *rel-alloc* **have** $t = \text{frame } A \ t_0 \ s$ **by** (*auto simp add: rel-alloc-def*)
define t_0' **where** $t_0' = \text{raw-frame } B \ s \ t_0$

have $t = \text{frame } (A \cup B) \ t_0' \ s$
using *frame-raw-frame-union [OF disj-A-B sf]*
by (*simp add: t_0'-def t*)

with *rel-alloc sf* **have** *rel-alloc'*: $\text{rel-alloc } \mathcal{S} \ M1 \ (A \cup B) \ t_0' \ s \ t$
by (*auto simp add: t_0'-def rel-alloc-def*)

from *spec [OF rel-alloc']* **have** *refines'*: $\text{refines } f \ g \ s \ t \ (\text{rel-stack } \mathcal{S} \ M \ (A \cup B) \ s \ t_0' \ Q)$.

show *?thesis*

proof (*rule refinesI*)

show $\text{succeeds } g \ t \implies \text{succeeds } f \ s$ **using** *refines'*

by (*auto simp add: refines-def-old*)

next

fix $v \ s'$

assume *succeeds*: $\text{succeeds } g \ t \ \text{succeeds } f \ s$

assume *result*: $\text{reaches } f \ s \ v \ s'$

show $\exists w \ t'. \text{reaches } g \ t \ w \ t' \wedge \text{rel-stack } \mathcal{S} \ M \ A \ s \ t_0 \ Q \ (v, \ s') \ (w, \ t')$

proof –

from *succeeds result refines'* **obtain** $w \ t'$

where

res: $\text{reaches } g \ t \ w \ t'$ **and**

stack: $\text{rel-stack } \mathcal{S} \ M \ (A \cup B) \ s \ t_0' \ Q \ (v, \ s') \ (w, \ t')$

by (*fastforce simp add: refines-def-old*)

from *stack* **obtain**

mem-eq: $\text{equal-upto } (M \cup \text{stack-free } (\text{htd } s')) \ (\text{hmem } s') \ (\text{hmem } s)$ **and**

htd-eq1: $\text{equal-upto } M \ (\text{htd } s') \ (\text{htd } s)$ **and**

htd-eq2: $\text{equal-on } \mathcal{S} \ (\text{htd } s') \ (\text{htd } s)$

by (*auto simp add: rel-stack-def*)

```

from stack
have sf-eq: stack-free (htd s') = stack-free (htd s)
  using disj-A-B by (auto simp add: rel-stack-def)

have htd-eq:
  override-on (htd s') (override-on (htd t0) (htd s) B) (A ∪ B - stack-free
(htd s)) =
  override-on (htd s') (htd t0) (A - stack-free (htd s))
  using disj-A-B disj-B-M sf sf-eq htd-eq1
  apply (clarsimp simp add: override-on-def fun-eq-iff)
  by (auto simp add: equal-upto-def orthD1)

have hmem-eq:
  override-on (hmem s') (override-on (hmem t0) (hmem s) B) (A ∪ B ∪
stack-free (htd s)) =
  override-on (hmem s') (hmem t0) (A ∪ stack-free (htd s))
  using disj-A-B disj-B-M sf sf-eq mem-eq
  by (auto simp add: override-on-def fun-eq-iff disjoint-iff equal-upto-def)

from stack have rel-stack': rel-stack  $\mathcal{S}$  M A s t0 Q (v, s') (w, t')
  apply (simp add: t0'-def)
  apply (simp add: rel-stack-def rel-alloc-def)
  apply (clarsimp simp add: frame-def raw-frame-def sf-eq)
  using htd-eq mem-eq
  by auto (metis (no-types, lifting) heap.upd-cong hmem-eq hmem-htd-upd
typing.upd-cong)

  then show ?thesis using res
  by auto
qed
qed
qed

lemma refines-narrow-frame:
  assumes subset:  $B \subseteq A$ 
  assumes disj-B-M:  $B \cap M = \{\}$ 
  assumes spec:  $\bigwedge t_0 s t. \text{rel-alloc } \mathcal{S} M1 A t_0 s t \implies \text{refines } f g s t (\text{rel-stack } \mathcal{S} M$ 
A s t0 Q)
  assumes sf: stack-free (htd s) ∩ B =  $\{\}$ 
  assumes rel-alloc: rel-alloc  $\mathcal{S} M1 (A - B) t_0 s t$ 
  shows refines f g s t (rel-stack  $\mathcal{S} M (A - B) s t_0 Q$ )
proof -
  from subset obtain A1 where A:  $A = A1 \cup B$  and disj:  $A1 \cap B = \{\}$  and
A1:  $A1 = A - B$ 
  by (metis Diff-Diff-Int Diff-disjoint Un-Diff-Int inter-sub)
  from refines-narrow-frame'[OF disj-B-M disj spec [simplified A] sf] rel-alloc
  show ?thesis

```

by (auto simp add: A1)
qed

lemma *refines-widen-modifies*:
assumes *rel-alloc* $S M A t_0 s t$
assumes *refines* $f g s t$ (*rel-stack* $S M' A s t_0 Q$)
assumes $M' \subseteq M$
shows *refines* $f g s t$ (*rel-stack* $S M A s t_0 Q$)
using *assms*
apply (*clarsimp simp add: refines-def-old rel-stack-def rel-alloc-def split: prod.splits xval-splits*)
by (*smt (verit) UnI2 Un-assoc equal-on-stack-free-preservation equal-upto-def sup.absorb-iff1*)

lemma *refines-widen-modifies'*:
assumes *rel-alloc* $S M A t_0 s t \implies$ *refines* $f g s t$ (*rel-stack* $S M' A s t_0 Q$)
assumes $M' \subseteq M$
assumes *rel-alloc* $S M A t_0 s t$
shows *refines* $f g s t$ (*rel-stack* $S M A s t_0 Q$)
using *refines-widen-modifies assms* **by** *blast*

lemma *refines-widen-modifies''*:
assumes *rel-alloc* $S M A t_0 s t$
assumes *refines* $f g s t$ (*rel-stack* $S M1 A s t_0 Q$)
assumes $M1 \subseteq M2$
assumes $M2 \subseteq M$
shows *refines* $f g s t$ (*rel-stack* $S M2 A s t_0 Q$)
using *refines-widen-modifies assms rel-alloc-modifies-antimono*
by *blast*

lemma *refines-widen-modifies-weaken*:
assumes *alloc*: *rel-alloc* $S M A t_0 s t$
assumes *f*: *refines* $f g s t$ (*rel-stack* $S M1 A s t_0 Q'$)
assumes *M1-M2*: $M1 \subseteq M2$
assumes *M2-M*: $M2 \subseteq M$
assumes *weaken*: $\bigwedge h r r'. Q' h r r' \implies Q h r r'$
shows *refines* $f g s t$ (*rel-stack* $S M2 A s t_0 Q$)
proof –
have *refines* $f g s t$ (*rel-stack* $S M1 A s t_0 Q$)
apply (*rule refines-weaken [OF f]*)
using *weaken* **by** (*auto simp add: rel-stack-def*)
with *alloc M1-M2 M2-M refines-widen-modifies rel-alloc-modifies-antimono refines-weaken*
show *?thesis*
by *blast*
qed

lemma *L2-condition-rel-stack*:

```

assumes  $s$ - $t$ : rel-alloc  $\mathcal{S}$   $M$   $A$   $t_0$   $s$   $t$ 
assumes  $c$ - $c'$ :  $c$   $s$  =  $c'$  (frame  $A$   $t_0$   $s$ )
assumes  $f1$ - $g1$ : refines  $f1$   $g1$   $s$   $t$  (rel-stack  $\mathcal{S}$   $M1$   $A$   $s$   $t_0$  (rel-xval-stack  $L$   $R$ ))
assumes  $f2$ - $g2$ : refines  $f2$   $g2$   $s$   $t$  (rel-stack  $\mathcal{S}$   $M2$   $A$   $s$   $t_0$  (rel-xval-stack  $L$   $R$ ))
assumes  $M1$ :  $M1 \subseteq M$ 
assumes  $M2$ :  $M2 \subseteq M$ 
assumes  $M'$ :  $M' = M1 \cup M2$ 
shows refines (L2-condition  $c$   $f1$   $f2$ ) (L2-condition  $c'$   $g1$   $g2$ )  $s$   $t$  (rel-stack  $\mathcal{S}$   $M'$ 
 $A$   $s$   $t_0$  (rel-xval-stack  $L$   $R$ ))
unfolding L2-condition-def
apply (rule refines-condition)
subgoal using  $s$ - $t$   $c$ - $c'$  by (auto simp add: rel-alloc-def)
subgoal apply (rule refines-widen-modifies [OF -  $f1$ - $g1$ ]) using  $s$ - $t$   $M1$   $M2$   $M'$ 
rel-alloc-modifies-antimono by auto
subgoal apply (rule refines-widen-modifies [OF -  $f2$ - $g2$ ]) using  $s$ - $t$   $M1$   $M2$   $M'$ 
rel-alloc-modifies-antimono by auto
done

```

lemma *L2-while-rel-stack''*:

```

assumes  $s$ - $t$ : rel-alloc  $\mathcal{S}$   $M$   $A$   $t_0$   $s$   $t$ 
assumes  $R$ - $i$ :  $R$  (hmem  $s$ )  $i$   $i'$ 
assumes refines-condition:  $\bigwedge s$   $v$   $v'$ .  $R$  (hmem  $s$ )  $v$   $v' \implies c' v'$  (frame  $A$   $t_0$   $s$ ) =
 $c v s$ 
assumes bdy:  $\bigwedge s$   $t$   $v$   $v'$ .  $R$  (hmem  $s$ )  $v$   $v' \implies$  rel-alloc  $\mathcal{S}$   $M1$   $A$   $t_0$   $s$   $t \implies c v s$ 
 $\implies c' v' t \implies$ 
refines ( $f$   $v$ ) ( $g$   $v'$ )  $s$   $t$  (rel-stack  $\mathcal{S}$   $M1$   $A$   $s$   $t_0$  (rel-xval-stack  $L$   $R$ ))
assumes  $M1$ :  $M1 \subseteq M$ 
shows refines (L2-while  $c$   $f$   $i$   $ns$ ) (L2-while  $c'$   $g$   $i'$   $ns'$ )  $s$   $t$  (rel-stack  $\mathcal{S}$   $M1$   $A$   $s$   $t_0$ 
(rel-xval-stack  $L$   $R$ ))
unfolding L2-while-def
apply (rule refines-whileLoop-exn)
subgoal by (auto simp add: rel-stack-def refines-condition rel-alloc-def)
subgoal for  $v$   $s'$   $w$   $t'$ 
apply clarsimp
apply (rule refines-mono [OF - bdy])
subgoal
apply (clarsimp simp add: rel-stack-def )
subgoal by (metis equal-upto-trans)
done
apply (auto simp add: rel-stack-def)
done
apply (auto simp add: s-t R-i rel-stack-def rel-alloc-modifies-antimono [OF -
 $M1$ ])
done

```

lemma *L2-while-rel-stack'''*:

```

assumes  $s$ - $t$ : rel-alloc  $\mathcal{S}$   $M$   $A$   $t_0$   $s$   $t$ 
assumes  $R$ - $i$ :  $R$  (hmem  $s$ )  $i$   $i'$ 

```

assumes *refines-condition*: $\bigwedge s v v'. R (hmem\ s) v v' \implies c' v' (frame\ A\ t_0\ s) = c\ v\ s$
assumes *bdy*: $\bigwedge s t v v'. R (hmem\ s) v v' \implies rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t \implies c' v' t \implies c\ v\ s$
assumes *M1*: $M1 \subseteq M$
shows *refines* (*L2-while* *c f i ns*) (*L2-while* *c' g i' ns'*) *s t* (*rel-stack* $\mathcal{S}\ M1\ A\ s\ t_0$ (*rel-xval-stack* *L R*))
unfolding *L2-while-def*
apply (*rule refines-whileLoop-exn*)
subgoal by (*auto simp add: rel-stack-def refines-condition rel-alloc-def*)
subgoal for *v s' w t'*
apply *clarsimp*
apply (*rule refines-mono [OF - bdy]*)
apply (*clarsimp simp add: rel-stack-def*)
apply (*metis equal-upto-trans*)
using *s-t by (auto simp add: rel-stack-def rel-alloc-def)*
apply (*auto simp add: s-t R-i rel-stack-def rel-alloc-modifies-antimono [OF - M1]*)
done

lemma *L2-while-rel-stack'*:

assumes *s-t*: *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t$
assumes *refines-condition*: $\bigwedge s v v'. R (hmem\ s) v v' \implies c' v' (frame\ A\ t_0\ s) = c\ v\ s$
assumes *R-i*: $R (hmem\ s) i\ i'$
assumes *bdy*: $\bigwedge s t v v'. R (hmem\ s) v v' \implies rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t \implies rel\text{-}stack\ \mathcal{S}\ M1\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ L\ R)$
assumes *M1*: $M1 \subseteq M$
shows *refines* (*L2-while* *c f i ns*) (*L2-while* *c' g i' ns'*) *s t* (*rel-stack* $\mathcal{S}\ M1\ A\ s\ t_0$ (*rel-xval-stack* *L R*))
unfolding *L2-while-def*
apply (*rule refines-whileLoop-exn*)
subgoal by (*auto simp add: rel-stack-def refines-condition rel-alloc-def*)
subgoal for *v s' w t'*
apply *clarsimp*
apply (*rule refines-mono [OF - bdy]*)
apply (*clarsimp simp add: rel-stack-def*)
apply (*metis equal-upto-trans*)
subgoal by (*auto simp add: rel-stack-def*)
subgoal using *s-t by (simp add: rel-stack-def) (metis rel-alloc-def)*
done
apply (*auto simp add: s-t R-i rel-stack-def rel-alloc-modifies-antimono [OF - M1]*)
done

lemma *L2-while-rel-stack*:

assumes *s-t*: *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t$
assumes *R-i*: $R (hmem\ s) i\ i'$

assumes *bdy*: $\bigwedge s' t' v w. R (hmem\ s')\ v\ w \implies rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s'\ t' \implies c\ v$
 $s' \implies c' w\ t' \implies$
 $equal\text{-}upto\ (M1 \cup\ stack\text{-}free\ (htd\ s'))\ (hmem\ s')\ (hmem\ s) \implies$
 $htd\ s' = htd\ s \implies$
 $refines\ (f\ v)\ (g'\ w\ s')\ s'\ t'\ (rel\text{-}stack\ \mathcal{S}\ M1\ A\ s'\ t_0\ (rel\text{-}xval\text{-}stack\ L\ R))$
assumes *refines-condition*: $\bigwedge s' t' v w. R (hmem\ s')\ v\ w \implies rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0$
 $s'\ t' \implies$
 $equal\text{-}upto\ (M1 \cup\ stack\text{-}free\ (htd\ s'))\ (hmem\ s')\ (hmem\ s) \implies htd\ s' =$
 $htd\ s \implies$
 $c' w\ t' = c\ v\ s'$
assumes *g'-g*: $\bigwedge s' t' v w. R (hmem\ s')\ v\ w \implies rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s'\ t' \implies$
 $equal\text{-}upto\ (M1 \cup\ stack\text{-}free\ (htd\ s'))\ (hmem\ s')\ (hmem\ s) \implies$
 $htd\ s' = htd\ s \implies$
 $g'\ w\ s' = g\ w$
assumes *M1*: $M1 \subseteq M$
shows *refines* (*L2-while c f i ns*) (*L2-while c' g i' ns'*) *s t* (*rel-stack S M1 A s t0*
(rel-xval-stack L R))
unfolding *L2-while-def*
apply (*rule refines-mono* [**where** $R = \lambda(r, s')\ (w, t')$.
 $equal\text{-}upto\ (M1 \cup\ stack\text{-}free\ (htd\ s'))\ (hmem\ s')\ (hmem\ s) \wedge$
 $htd\ s' = htd\ s \wedge$
 $(rel\text{-}stack\ \mathcal{S}\ M1\ A\ s'\ t_0\ (rel\text{-}xval\text{-}stack\ L\ R))\ (r, s')\ (w, t')$])
subgoal
by (*auto simp add: rel-stack-def*)
apply (*rule refines-whileLoop-exn*)
subgoal using *s-t* **by** (*auto simp add: rel-stack-def refines-condition rel-alloc-def*)
subgoal for *v s' w t'*
apply *clarsimp*
apply (*rule refines-weaken* [**where** $R=(rel\text{-}stack\ \mathcal{S}\ M1\ A\ s'\ t_0\ (rel\text{-}xval\text{-}stack$
 $L\ R))$])
apply (*subst g'-g* [*of s' v w t', symmetric*])
apply (*simp add: rel-stack-def*)

apply (*simp add: rel-stack-def*)
using *rel-alloc-def s-t*
apply *force*
apply *simp*
apply *simp*
apply (*rule bdy*)
apply (*simp add: rel-stack-def*)
apply (*simp add: rel-stack-def*)
using *s-t* **apply** (*metis equal-on-stack-free-preservation rel-alloc-def*)
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply (*clarsimp simp add: rel-stack-def*)
apply (*metis equal-upto-trans*)

done
apply (*smt* (*verit*) *L2-split-fixup-3* *M1* *R-i* *equal-upto-refl* *heap-state.rel-stack-def* *heap-state-axioms* *rel-alloc-modifies-antimono* *rel-xval-stack-simps(2)* *s-t*)
apply (*auto simp add: s-t R-i rel-stack-def rel-alloc-modifies-antimono [OF - M1]*)
done

lemma *L2-while-rel-stack-g-normalised:*

assumes *s-t: rel-alloc S M A t₀ s t*
assumes *R-i: R (hmem s) i i'*
assumes *bdy: $\bigwedge s' t' v w. R (hmem s') v w \implies rel-alloc S M A t_0 s' t' \implies c v s' \implies c' w t' \implies$*
equal-upto (M1 \cup stack-free (htd s')) (hmem s') (hmem s) \implies
htd s' = htd s \implies
refines (f v) (g w) s' t' (rel-stack S M1 A s' t₀ (rel-xval-stack L R))
assumes *refines-condition: $\bigwedge s' t' v w. R (hmem s') v w \implies rel-alloc S M A t_0 s' t' \implies$*
equal-upto (M1 \cup stack-free (htd s')) (hmem s') (hmem s) $\implies htd s' =$
htd s \implies
c' w t' = c v s'
assumes *M1: M1 \subseteq M*
shows *refines (L2-while c f i ns) (L2-while c' g i' ns') s t (rel-stack S M1 A s t₀ (rel-xval-stack L R))*
apply (*rule L2-while-rel-stack [OF s-t R-i bdy refines-condition - M1]*)
by *auto*

lemma *L2-while-rel-stack-g-normalised-guarded:*

assumes *s-t: rel-alloc S M A t₀ s t*
assumes *R-i: R (hmem s) i i'*
assumes *bdy: $\bigwedge s' t' v w. R (hmem s') v w \implies rel-alloc S M A t_0 s' t' \implies c v s' \implies c' w t' \implies$*
equal-upto (M1 \cup stack-free (htd s')) (hmem s') (hmem s) \implies
htd s' = htd s \implies
refines (f v) (g w) s' t' (rel-stack S M1 A s' t₀ (rel-xval-stack L R))
assumes *refines-condition: $\bigwedge s' t' v w. R (hmem s') v w \implies rel-alloc S M A t_0 s' t' \implies$*
equal-upto (M1 \cup stack-free (htd s')) (hmem s') (hmem s) $\implies htd s' =$
htd s $\implies G' w t' \implies$
c' w t' = c v s'
assumes *M1: M1 \subseteq M*
assumes *G-G': G t = G' i' t*
shows *refines (L2-while c f i ns)*
(L2-seq (L2-guard G)
($\lambda-. (L2-while c' (\lambda v. L2-seq (g v) (\lambda res. L2-seq (L2-guard (G' res))$
($\lambda-. L2-gets (\lambda-. res) ns'))$) i' ns'))) s t (rel-stack S M1 A s t₀ (rel-xval-stack L
R))

```

apply (rule refines-mono [where R =  $\lambda(r, s') (w, t').$ 
  equal-upto (M1  $\cup$  stack-free (htd s')) (hmem s') (hmem s)  $\wedge$ 
  htd s' = htd s  $\wedge$ 
  (rel-stack S M1 A s' t0 (rel-xval-stack L R)) (r, s') (w, t') ])
subgoal
  by (auto simp add: rel-stack-def)
unfolding L2-defs gets-return
apply (rule refines-whileLoop-guard-right)
subgoal using s-t by (auto simp add: rel-stack-def refines-condition rel-alloc-def)
subgoal for v s' w t'
  apply clarsimp
  apply (rule refines-weaken [where R=(rel-stack S M1 A s' t0 (rel-xval-stack
L R))])
  apply (rule bdy)
  subgoal by (auto simp add: rel-stack-def)
  subgoal using rel-alloc-def s-t by (auto simp add: rel-stack-def)
  subgoal by simp
  subgoal by simp
  subgoal using s-t by simp
  subgoal by simp
  subgoal
    apply (clarsimp simp add: rel-stack-def)
    apply (metis equal-upto-trans)
    done
  done
  subgoal by (auto simp add: rel-stack-def)
subgoal using R-i s-t by (auto simp add: rel-stack-def rel-alloc-modifies-antimono
[OF - M1])
subgoal using G-G' by simp
done

```

lemma L2-unknown-rel-stack:

```

assumes s-t: rel-alloc S M A t0 s t
shows refines (L2-unknown ns) (L2-unknown ns) s t (rel-stack S {} A s t0
(rel-xval-stack L ( $\lambda.$  (=))))
using s-t
by (auto simp add: L2-unknown-def refines-def-old rel-stack-def rel-alloc-modifies-antimono)

```

lemma L2-fail-rel-stack:

```

assumes s-t: rel-alloc S M A t0 s t
shows refines (L2-fail) (L2-fail) s t (rel-stack S {} A s t0 (rel-xval-stack L R))
using s-t
by (auto simp add: L2-fail-def refines-def-old rel-stack-def rel-alloc-modifies-antimono)

```

lemma L2-skip-rel-stack:

```

assumes s-t: rel-alloc S M A t0 s t
shows refines L2-skip L2-skip s t (rel-stack S {} A s t0 (rel-xval-stack L ( $\lambda.$ 
(=))))

```

using $s-t$
by (*auto simp add: L2-gets-def refines-def-old rel-stack-def rel-alloc-modifies-antimono*)

lemma *L2-spec-rel-stack*:

assumes $s-t: \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t$
assumes $\bigwedge t'. (\text{frame } A \ t_0 \ s, \ t') \in r' \implies \exists s'. (s, \ s') \in r$
assumes $\bigwedge s'. (s, \ s') \in r \implies (\text{frame } A \ t_0 \ s, \ \text{frame } A \ t_0 \ s') \in r'$
assumes $\bigwedge s'. (s, \ s') \in r \implies \text{equal-upto } (M \cup \text{stack-free } (\text{htd } s')) \ (\text{hmem } s')$
(*hmem s*)
assumes $\bigwedge s'. (s, \ s') \in r \implies \text{htd } s' = \text{htd } s$
assumes $\bigwedge s'. (s, \ s') \in r \implies \text{stack-free } (\text{htd } s') \subseteq \mathcal{S}$
assumes $\bigwedge s'. (s, \ s') \in r \implies \text{stack-free } (\text{htd } s') \cap A = \{\}$
assumes $\bigwedge s'. (s, \ s') \in r \implies \text{stack-free } (\text{htd } s') \cap M = \{\}$
shows *refines* (*L2-spec r*) (*L2-spec r'*) $s \ t$ (*rel-stack* $\mathcal{S} \ M \ A \ s \ t_0$ (*rel-xval-stack L*
($\lambda \cdot$ ($=$))))
using *assms*
by (*auto simp add: L2-spec-def refines-def-old rel-stack-def rel-alloc-def succeeds-bind reaches-bind*)

lemma *L2-spec-rel-stack'*:

assumes $s-t: \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t$
assumes $\bigwedge t'. \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t \implies (\text{frame } A \ t_0 \ s, \ t') \in r' \implies \exists s'. (s, \ s') \in r$
assumes $\bigwedge s'. \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t \implies (s, \ s') \in r \implies (\text{frame } A \ t_0 \ s, \ \text{frame } A \ t_0 \ s') \in r'$
assumes $\bigwedge s'. \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t \implies (s, \ s') \in r \implies \text{equal-upto } (M \cup \text{stack-free } (\text{htd } s')) \ (\text{hmem } s') \ (\text{hmem } s)$
assumes $\bigwedge s'. \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t \implies (s, \ s') \in r \implies \text{htd } s' = \text{htd } s$
assumes $\bigwedge s'. \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t \implies (s, \ s') \in r \implies \text{stack-free } (\text{htd } s') \subseteq \mathcal{S}$
assumes $\bigwedge s'. \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t \implies (s, \ s') \in r \implies \text{stack-free } (\text{htd } s') \cap A = \{\}$
assumes $\bigwedge s'. \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t \implies (s, \ s') \in r \implies \text{stack-free } (\text{htd } s') \cap M = \{\}$
shows *refines* (*L2-spec r*) (*L2-spec r'*) $s \ t$ (*rel-stack* $\mathcal{S} \ M \ A \ s \ t_0$ (*rel-xval-stack L*
($\lambda \cdot$ ($=$))))
using $s-t$ *assms*(2–8) [*OF s-t*] **by** (*rule L2-spec-rel-stack*)

lemma *L2-spec-rel-stack-same*:

assumes $s-t: \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t$
assumes $\bigwedge t'. \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t \implies (\text{frame } A \ t_0 \ s, \ t') \in r \implies \exists s'. (s, \ s') \in r$
assumes $\bigwedge s'. \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t \implies (s, \ s') \in r \implies (\text{frame } A \ t_0 \ s, \ \text{frame } A \ t_0 \ s') \in r$
assumes $\bigwedge s'. \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t \implies (s, \ s') \in r \implies \text{equal-upto } (M \cup \text{stack-free } (\text{htd } s')) \ (\text{hmem } s') \ (\text{hmem } s)$
assumes $\bigwedge s'. \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t \implies (s, \ s') \in r \implies \text{htd } s' = \text{htd } s$
assumes $\bigwedge s'. \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t \implies (s, \ s') \in r \implies \text{stack-free } (\text{htd } s') \subseteq \mathcal{S}$

assumes $\bigwedge s'. \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t \implies (s, s') \in r \implies \text{stack-free } (\text{htd } s') \cap A$
 $= \{\}$
assumes $\bigwedge s'. \text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t \implies (s, s') \in r \implies \text{stack-free } (\text{htd } s') \cap M$
 $= \{\}$
shows $\text{refines } (L2\text{-spec } r) \ (L2\text{-spec } r) \ s \ t \ (\text{rel-stack } \mathcal{S} \ M \ A \ s \ t_0 \ (\text{rel-xval-stack } L$
 $(\lambda-. (=))))$
using *assms*
by (*rule L2-spec-rel-stack'*)

lemma *L2-spec-rel-stack-heap-agnostic:*

assumes *s-t: rel-alloc* $\mathcal{S} \ M \ A \ t_0 \ s \ t$
assumes *hmem-unchanged:* $\bigwedge s \ s'. (s, s') \in r \implies \text{hmem } s' = \text{hmem } s$
assumes *htd-unchanged:* $\bigwedge s \ s'. (s, s') \in r \implies \text{htd } s' = \text{htd } s$
assumes *heap-irrelevant:* $\bigwedge s \ s' \ f \ g. (s, s') \in r \implies (\text{hmem-upd } g \ (\text{htd-upd } f \ s),$
 $\text{hmem-upd } g \ (\text{htd-upd } f \ s')) \in r$
shows $\text{refines } (L2\text{-spec } r) \ (L2\text{-spec } r) \ s \ t \ (\text{rel-stack } \mathcal{S} \ \{\} \ A \ s \ t_0 \ (\text{rel-xval-stack } L$
 $(\lambda-. (=))))$
proof –
have $\text{refines } (L2\text{-spec } r) \ (L2\text{-spec } r) \ s \ t \ (\text{rel-stack } \mathcal{S} \ M \ A \ s \ t_0 \ (\text{rel-xval-stack } L$
 $(\lambda-. (=))))$
apply (*rule L2-spec-rel-stack-same* [*OF s-t*])
subgoal premises *prems* **for** *t'*
proof –
from *prems* **have** $r: (\text{frame } A \ t_0 \ s, \ t') \in r$ **by** *simp*
have *s-eq:* $s = \text{hmem-upd } (\lambda-. \text{hmem } s) \ ((\text{htd-upd } (\lambda-. \text{htd } s) \ (\text{frame } A \ t_0 \ s)))$
apply (*simp add: frame-def*)
by (*metis equal-upto-heap-on-def equal-upto-heap-on-frame frame-def heap.get-upd*
 $\text{htd-hmem-upd typing.get-upd}$)
show *?thesis*
apply (*subst s-eq*)
apply (*rule exI*)
apply (*rule heap-irrelevant*)
apply (*rule r*)
done
qed
subgoal
apply (*simp add: frame-def*)
using *heap-irrelevant htd-unchanged*
by *simp*
subgoal
using *hmem-unchanged* **by** *simp*
subgoal
using *htd-unchanged* **by** *simp*
subgoal
using *htd-unchanged* **by** (*simp add: rel-alloc-def*)
subgoal
using *htd-unchanged* **by** (*simp add: rel-alloc-def*)
subgoal
using *htd-unchanged* **by** (*simp add: rel-alloc-def*)

```

done
then show ?thesis
  using hmem-unchanged htd-unchanged
  apply (clarsimp simp add: L2-spec-def refines-def-old rel-stack-def rel-alloc-def
reaches-bind
  succeeds-bind)
  apply (meson UNIV-I image-eqI)
  apply force+
done
qed

```

```

lemma L2-assume-rel-stack:
  assumes s-t: rel-alloc S M A t0 s t
  assumes  $\bigwedge v s'. (v, s') \in f s \implies$ 
     $\exists w. (w, \text{frame } A t_0 s') \in g (\text{frame } A t_0 s) \wedge \text{rel-stack } S M A s t_0 R (v, s') (w,$ 
 $\text{frame } A t_0 s')$ 
  shows refines (L2-assume f) (L2-assume g) s t (rel-stack S M A s t0 (rel-xval-stack
L R))
  using assms
  apply (clarsimp simp add: L2-assume-def refines-def-old rel-alloc-def rel-stack-def)
  by metis

```

```

lemma L2-assume-rel-stack-heap-agnostic:
  assumes s-t: rel-alloc S M A t0 s t
  assumes hmem-unchanged:  $\bigwedge a s s'. (a, s') \in f s \implies \text{hmem } s' = \text{hmem } s$ 
  assumes htd-unchanged:  $\bigwedge a s s'. (a, s') \in f s \implies \text{htd } s' = \text{htd } s$ 
  assumes heap-irrelevant:  $\bigwedge a s s' g h. (a, s') \in f s \implies (a, \text{hmem-upd } h (\text{htd-upd } g s')) \in f (\text{hmem-upd } h (\text{htd-upd } g s))$ 
  shows refines (L2-assume f) (L2-assume f) s t (rel-stack S {} A s t0 (rel-xval-stack
L ( $\lambda \cdot (=)$ )))
  apply (rule L2-assume-rel-stack)
  subgoal
    using s-t
  by (auto simp add: L2-assume-def refines-def-old rel-stack-def rel-alloc-def)
  subgoal for v s'
    apply (rule exI[where x=v])
    apply (intro conjI)
  subgoal
    using frame-def heap-irrelevant htd-unchanged by presburger
  subgoal
    apply (clarsimp simp: rel-stack-def frame-def)
    apply (intro conjI)
  subgoal using  $\langle \text{rel-alloc } S \{ \} A t_0 s t \rangle$  frame-def htd-unchanged rel-alloc-def
by force
  subgoal using hmem-unchanged by force
  subgoal using htd-unchanged by blast

```

done
done
done

lemma *refines-rel-stack-embed-result'*:

assumes *rel-alloc*: *rel-alloc* $\mathcal{S} M A t_0 s t$
assumes *f*: *refines* *f g s t* (*rel-stack* $\mathcal{S} M1 A s t_0$ (*rel-xval-stack* $L R$))
assumes *R1*: $\bigwedge v s' w t'. \text{rel-alloc } \mathcal{S} M A t_0 s' t' \implies R (\text{hmem } s') v w \implies$
 $\text{equal-upto } (M1 \cup \text{stack-free } (\text{htd } s')) (\text{hmem } s') (\text{hmem } s) \implies$
 $\text{htd } s' = \text{htd } s \implies$
 $R1 (\text{hmem } s') v (\text{emb } w)$
assumes *M2-M*: $M2 \subseteq M$
assumes *M1-M2*: $M1 \subseteq M2$
shows *refines* *f* (*L2-seq* *g* (*ETA-TUPLED* ($\lambda x. L2\text{-gets } (\lambda-. \text{emb } x) ns$))) *s t*
(*rel-stack* $\mathcal{S} M2 A s t_0$ (*rel-xval-stack* $L R1$))
unfolding *L2-defs* *ETA-TUPLED-def*
apply (*subst* *bind-return* [*symmetric*, *of f*])
apply (*rule* *refines-bind-bind-exn* [*OF f*])
subgoal
using *M1-M2* *M2-M* *rel-alloc*
apply (*clarsimp* *simp* *add*: *rel-stack-def* *rel-alloc-def*)
by *auto* (*meson* *Un-mono* *equal-upto-mono* *equalityE*)
subgoal **by** *auto*
subgoal **by** *auto*
subgoal
apply (*clarsimp* *simp* *add*: *gets-return* *rel-stack-def*)
apply (*intro* *conjI*)
subgoal **using** *R1* *rel-alloc* **by** (*auto* *simp* *add*: *rel-alloc-def*)
subgoal **using** *M2-M* *rel-alloc* **by** (*auto* *simp* *add*: *rel-alloc-def*)
subgoal **using** *M1-M2* **by** (*meson* *equal-upto-mono* *equalityD2* *sup.mono*)
done
done

lemma *refines-rel-stack-root-upd-result*:

assumes *rel-alloc*: *rel-alloc* $\mathcal{S} M A t_0 s t$
assumes *f*: $P t \implies \text{refines } f (L2\text{-seq } (L2\text{-guard } P) (\lambda-. g)) s t$ (*rel-stack* $\mathcal{S} M1$
 $A s t_0$ (*rel-xval-stack* $L R$))
assumes *R1*: $\bigwedge v s' w t'. \text{rel-alloc } \mathcal{S} M A t_0 s' t' \implies R (\text{hmem } s') v w \implies P t$
 \implies
 $\text{equal-upto } (M1 \cup \text{stack-free } (\text{htd } s')) (\text{hmem } s') (\text{hmem } s) \implies$
 $\text{htd } s' = \text{htd } s \implies$
 $R1 (\text{hmem } s') v (\text{emb } w)$
assumes *M2-M*: $P t \implies M2 \subseteq M$
assumes *M1-M2*: $P t \implies M1 \subseteq M2$
shows *refines* *f* (*L2-seq* (*L2-guard* *P*) ($\lambda-. L2\text{-seq } g$ (*ETA-TUPLED* ($\lambda x. L2\text{-gets}$
 $(\lambda-. \text{emb } x) ns$)))) *s t* (*rel-stack* $\mathcal{S} M2 A s t_0$ (*rel-xval-stack* $L R1$))
using *refines-rel-stack-embed-result'* [*OF* *assms*(1) - - *assms*(4,5), *of f g L R R1*
emb]

using *assms*(2,3)
by (*auto simp add: refines-bind-guard-right-iff L2-defs ETA-TUPLED-def*)

lemma *refines-rel-stack-embed-exit*:

assumes *rel-alloc*: *rel-alloc* $\mathcal{S} M A t_0 s t$
assumes *f*: *refines* *f g s t* (*rel-stack* $\mathcal{S} M1 A s t_0$ (*rel-xval-stack* $L R$))
assumes *L1*: $\bigwedge v s' w t'. \text{rel-alloc } \mathcal{S} M A t_0 s' t' \implies L (\text{hmem } s') v w \implies$
 $\text{equal-upto } (M1 \cup \text{stack-free } (\text{htd } s')) (\text{hmem } s') (\text{hmem } s) \implies$
 $\text{htd } s' = \text{htd } s \implies$
 $L1 (\text{hmem } s') v (\text{emb } w)$
assumes *M2-M*: $M2 \subseteq M$
assumes *M1-M2*: $M1 \subseteq M2$
shows *refines*
f (*L2-catch* *g* (*ETA-TUPLED* ($\lambda x. \text{L2-throw } (\text{emb } x) ns$))) *s t*
(*rel-stack* $\mathcal{S} M2 A s t_0$ (*rel-xval-stack* $L1 R$))
unfolding *L2-defs ETA-TUPLED-def*
apply (*subst f-catch-throw* [*symmetric, of f*])
apply (*rule refines-catch* [*OF assms*(2)])
subgoal
apply (*clarsimp simp add: rel-stack-def, intro conjI*)
subgoal using *L1 rel-alloc* **by** (*auto simp add: rel-alloc-def*)
subgoal using *M2-M rel-alloc* **by** (*auto simp add: rel-alloc-def*)
subgoal using *M1-M2* **by** (*meson equal-upto-mono equalityD2 sup.mono*)
done
subgoal by *auto*
subgoal by *auto*
subgoal
apply (*simp add: rel-stack-def*)
apply (*intro conjI*)
subgoal using *M2-M rel-alloc* **by** (*auto simp add: rel-alloc-def*)
subgoal using *M1-M2* **by** *auto* (*meson equal-upto-mono equalityD2 sup.mono*)
done
done

lemma *refines-rel-stack-embed-both*:

assumes *stack*: *rel-alloc* $\mathcal{S} M A t_0 s t$
assumes *f-g*: *refines* *f g s t* (*rel-stack* $\mathcal{S} M1 A s t_0$ (*rel-xval-stack* $L R$))
assumes *L*: $\bigwedge v s' w t'. \text{rel-alloc } \mathcal{S} M A t_0 s' t' \implies L (\text{hmem } s') v w \implies$
 $\text{equal-upto } (M1 \cup \text{stack-free } (\text{htd } s')) (\text{hmem } s') (\text{hmem } s) \implies$
 $\text{htd } s' = \text{htd } s \implies$
 $L1 (\text{hmem } s') v (\text{embL } w)$
assumes *R*: $\bigwedge v s' w t'. \text{rel-alloc } \mathcal{S} M A t_0 s' t' \implies R (\text{hmem } s') v w \implies$
 $\text{equal-upto } (M1 \cup \text{stack-free } (\text{htd } s')) (\text{hmem } s') (\text{hmem } s) \implies$
 $\text{htd } s' = \text{htd } s \implies$
 $R1 (\text{hmem } s') v (\text{embR } w)$
assumes *M2-M*: $M2 \subseteq M$
assumes *M1-M2*: $M1 \subseteq M2$
shows *refines*
f


```

(L2-seq
  (L2-catch g (ETA-TUPLED ( $\lambda x. L2\text{-throw (embL } x) ns$ )))
  (ETA-TUPLED ( $\lambda x. L2\text{-gets } (\lambda \cdot. embR } x) ns$ ))) s t
  (rel-stack  $\mathcal{S}$  M2 A s t0 (rel-xval-stack L1 R1))
  apply (rule refines-rel-stack-embed-result' [OF stack - - M2-M M1-M2, where
R=R'])
  subgoal
    apply (rule refines-rel-stack-embed-exit [OF stack f-g ])
    subgoal
      by (rule L)
    subgoal using M1-M2 M2-M by auto
    subgoal by auto
    done
  subgoal
    by (rule R)
  done
end

```

```

lemma refines-assume-result-and-state-left:
  assumes  $\bigwedge s' v. (v, s') \in A s \implies \text{refines } (f v) g s' t Q$ 
  shows refines (do { v <- assume-result-and-state A; f v }) g s t Q
  using assms
  apply (auto simp add: refines-def-old succeeds-bind reaches-bind split: xval-splits)
  done

```

```

lemma refines-assume-result-and-state-right:
  assumes  $A t \neq \{\}$ 
  assumes  $\bigwedge t' v. (v, t') \in A t \implies \text{refines } f (g v) s t' Q$ 
  shows refines f (do { v <- assume-result-and-state A; g v }) s t Q
  using assms
  apply (clarsimp simp add: refines-def-old succeeds-bind reaches-bind
    split: exception-or-result-splits )
  by auto (metis (no-types, opaque-lifting) Result-eq-Result is-Result-simps(1) is-Result-simps(2))

```

```

lemma refines-assume-result-and-state-both:
  assumes  $B t = \{\} \implies A s = \{\}$ 
  assumes  $\bigwedge s' v t' w. (v, s') \in A s \implies (w, t') \in B t \implies \text{refines } (f v) (g w) s' t' Q$ 
  shows refines (do { v <- assume-result-and-state A; f v }) (do { w <- assume-result-and-state B; g w }) s t Q
  apply (rule refines-bind')
  apply (simp add: refines-assume-result-and-state-iff)
  using assms
  by (force simp add: sim-set-def)

```

```

lemma refines-assume-result-and-state-both-same-val':
  assumes  $\bigwedge s' v. (v, s') \in A s \implies \exists t'. (v, t') \in B t \wedge \text{refines } (f v) (g v) s' t' Q$ 
  shows refines (do { v <- assume-result-and-state A; f v }) (do { v <- as-

```

sume-result-and-state B; g v } $s\ t\ Q$
apply (*rule refines-bind'*)
apply (*simp add: refines-assume-result-and-state-iff*)
using *assms*
by (*force simp add: sim-set-def*)

lemma *refines-assume-result-and-state-both-same-val:*
assumes $\bigwedge v\ s'. (v, s') \in A\ s \implies \exists t'. (v, t') \in B\ t$
assumes $\bigwedge s'\ v\ t'. (v, s') \in A\ s \implies (v, t') \in B\ t \implies \text{refines } (f\ v)\ (g\ v)\ s'\ t'\ Q$
shows *refines* (*do* { $v <- \text{assume-result-and-state } A; f\ v$ }) (*do* { $v <- \text{assume-result-and-state } B; g\ v$ }) $s\ t\ Q$
apply (*rule refines-bind'*)
apply (*simp add: refines-assume-result-and-state-iff*)
using *assms*
by (*force simp add: sim-set-def*)

lemma *refines-assume-result-and-state-both-same-val-frame:*
assumes $f: t = \text{frm } s$
assumes $\bigwedge s'\ v. (v, s') \in A\ s \implies (v, \text{frm } s') \in B\ t$
assumes $\bigwedge s'\ v. (v, s') \in A\ s \implies \text{refines } (f\ v)\ (g\ v)\ s'\ (\text{frm } s')\ Q$
shows *refines* (*do* { $v <- \text{assume-result-and-state } A; f\ v$ }) (*do* { $v <- \text{assume-result-and-state } B; g\ v$ }) $s\ t\ Q$
apply (*rule refines-bind'*)
apply (*simp add: refines-assume-result-and-state-iff*)
using *assms*
by (*force simp add: sim-set-def*)

lemma *refines-on-exit-left:*
assumes $f\text{-}g: \text{refines } f\ g\ s0\ t\ Q'$
assumes 1: $\bigwedge s. \exists s'. (s, s') \in \text{cleanup}$
assumes 2: $\bigwedge s\ v\ t\ w\ s'. Q'\ (v, s)\ (w, t) \implies (s, s') \in \text{cleanup} \implies Q\ (v, s')\ (w, t)$
shows *refines* (*on-exit* $f\ \text{cleanup}$) $g\ s0\ t\ Q$
unfolding *on-exit-def*
apply (*subst on-exit'-skip*[*of* g , *symmetric*])
apply (*rule refines-on-exit'*[*OF* $f\text{-}g$ [*THEN* *refines-weaken*]])
apply (*auto simp: refines-yield-right-iff runs-to-iff 1 2*)
done

lemma *refines-on-exit-same-cleanup:*
assumes $f\text{-}g: \text{refines } f\ g\ s0\ t\ Q'$
assumes 1: $\bigwedge s. \exists s'. (s, s') \in \text{cleanup}$
assumes 2: $\bigwedge s\ v\ t\ w\ s'\ t'. Q'\ (v, s)\ (w, t) \implies (s, s') \in \text{cleanup} \implies (t, t') \in \text{cleanup} \implies Q\ (v, s')\ (w, t')$
shows *refines* (*on-exit* $f\ \text{cleanup}$) (*on-exit* $g\ \text{cleanup}$) $s0\ t\ Q$
unfolding *on-exit-def*
apply (*rule refines-on-exit'*[*OF* $f\text{-}g$ [*THEN* *refines-weaken*]])

```

using 1 2
apply clarsimp
apply (rule refines-state-select)
  apply force
  apply force
done

```

lemma *refines-on-exit-same-cleanup-choice*:

```

assumes f-g: refines f g s0 t Q'
assumes 1:  $\bigwedge s. \exists s'. (s, s') \in \text{cleanup}$ 
assumes 2:  $\bigwedge s v t w s'. Q' (v, s) (w, t) \implies (s, s') \in \text{cleanup} \implies \exists t'. (t, t') \in \text{cleanup} \wedge Q (v, s') (w, t')$ 
shows refines (on-exit f cleanup) (on-exit g cleanup) s0 t Q
unfolding on-exit-def
apply (rule refines-on-exit'[OF f-g[THEN refines-weaken]])
using 1 2
apply clarsimp
apply (rule refines-state-select)
apply auto
done

```

lemma *refines-runs-to-partial-fuse-both*:

```

assumes sim: refines f f' s s' Q
assumes runs-to-f:  $f \cdot s \ ?\{P1\}$ 
assumes runs-to-f':  $f' \cdot s' \ ?\{P2\}$ 
shows refines f f' s s'  $(\lambda(r,t) (r',t')). Q (r,t) (r',t') \wedge P1 r t \wedge P2 r' t'$ 
using sim runs-to-f runs-to-f'
apply (force simp add: refines-def-old runs-to-partial-def-old )
done

```

definition *domain-bound* $A Q = (\forall h h'. \text{equal-on } A h h' \longrightarrow Q h = Q h')$

lemma *domain-bound-equal-on*: $\text{domain-bound } A Q \implies \text{equal-on } A h h' \implies Q h = Q h'$

by (*auto simp add: domain-bound-def*)

lemma *domain-bound-equal-on-subset*: $\text{domain-bound } A Q \implies A \subseteq A' \implies \text{equal-on } A' h h' \implies Q h = Q h'$

using *domain-bound-equal-on equal-on-mono* **by** *blast*

lemma *domain-bound-heap-update-list*:

```

fixes p::'a::mem-type ptr
assumes domain-bound A Q
assumes length bs = size-of TYPE('a)
assumes ptr-span p  $\cap A = \{\}$ 
shows  $Q (\text{heap-update-list } (\text{ptr-val } p) bs h) = Q h$ 
using assms domain-bound-equal-on
by (smt (verit, best) equal-on-def heap-update-nmem-same orthD2)

```

named-theorems *domain-bound-intros*

lemma *domain-bound-mono*: $A \subseteq A' \implies \text{domain-bound } A \ Q \implies \text{domain-bound } A' \ Q$

by (*auto simp add: domain-bound-def equal-on-mono*)

lemma *domain-bound-eq*: $\text{domain-bound } \{\} (\lambda-. (=))$

by (*auto simp add: domain-bound-def*)

lemma *domain-bound-rel-sum-stack*[*domain-bound-intros*]: $\text{domain-bound } A \ L \implies \text{domain-bound } A \ R \implies \text{domain-bound } A \ (\text{rel-sum-stack } L \ R)$

by (*auto simp add: domain-bound-def rel-sum-stack-def*)

lemma *domain-bound-rel-xval-stack*[*domain-bound-intros*]: $\text{domain-bound } A \ L \implies \text{domain-bound } A \ R \implies \text{domain-bound } A \ (\text{rel-xval-stack } L \ R)$

by (*auto simp add: domain-bound-def rel-xval-stack-def*)

lemma *domain-bound-unreachable-exit* [*domain-bound-intros*]:

$\text{domain-bound } A \ (\text{rel-exit } (\lambda- - -. \text{False}))$

by (*auto simp add: domain-bound-def rel-exit-def*)

lemma *domain-bound-bot*[*domain-bound-intros*]: $\text{domain-bound } A \ (\lambda- - -. \text{False})$

by (*auto simp add: domain-bound-def*)

lemma *domain-bound-eq'*[*domain-bound-intros*]: $\text{domain-bound } A \ (\lambda-. (=))$

using *domain-bound-eq domain-bound-mono* **by** *blast*

lemma *domain-bound-top*[*domain-bound-intros*]: $\text{domain-bound } A \ (\lambda- - -. \text{True})$

by (*auto simp add: domain-bound-def*)

lemma *domain-bound-rel-singleton-stack*: $\text{domain-bound } (\text{ptr-span } p) \ (\text{rel-singleton-stack } p)$

using *domain-rel-singleton-stack* **by** (*auto simp add: domain-bound-def*)

lemma *domain-bound-rel-exit*[*domain-bound-intros*]:

$\text{domain-bound } A \ Q \implies \text{domain-bound } A \ (\text{rel-exit } Q)$

by (*auto simp add: rel-exit-def domain-bound-def*)

lemma *domain-bound-rel-singleton-stack'*[*domain-bound-intros*]:

$\text{ptr-span } p \subseteq A \implies \text{domain-bound } A \ (\text{rel-singleton-stack } p)$

using *domain-bound-rel-singleton-stack domain-bound-mono* **by** *blast*

lemma *domain-rel-push*: $\text{domain-bound } A \ Q \implies \text{equal-on } (\text{ptr-span } p \cup A) \ h \ h' \implies \text{rel-push } p \ Q \ h = \text{rel-push } p \ Q \ h'$

apply (*simp add: domain-bound-def rel-push-def fun-eq-iff*)

by (*metis Un-upper1 domain-rel-singleton-stack equal-on-mono rel-singleton-stack-def sup-commute*)

lemma *domain-bound-rel-push*: $\text{domain-bound } A \ Q \implies \text{domain-bound } (\text{ptr-span } p \cup A) \ (\text{rel-push } p \ Q)$

using *domain-rel-push domain-bound-def* **by** *blast*

lemma *domain-bound-rel-push'[domain-bound-intros]*: $\text{ptr-span } p \subseteq A \implies \text{domain-bound } A \ Q \implies \text{domain-bound } A \ (\text{rel-push } p \ Q)$

using *domain-bound-rel-push*

by (*metis (no-types, lifting) subset-Un-eq*)

context *stack-heap-state*

begin

lemma *with-fresh-stack-ptr-rel-stack''*:

assumes *stack*: $\text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t$

assumes *domain-bound*: $\text{domain-bound } A \ Q$

assumes *f*: $\bigwedge p \ s' \ t_0.$

$\llbracket \text{ptr-span } p \subseteq \mathcal{S}; \text{ptr-span } p \subseteq \text{stack-free } (\text{htd } s);$

$\text{ptr-span } p \cap A = \{\}; \text{ptr-span } p \cap M = \{\};$

$\text{root-ptr-valid } (\text{htd } s') \ p;$

$[h\text{-val } (\text{hmem } s') \ p] \in I \ s;$

$\text{equal-upto-heap-on } (\text{ptr-span } p) \ s' \ s;$

$\text{stack-free } (\text{htd } s') \subseteq \text{stack-free } (\text{htd } s);$

$\text{rel-alloc } \mathcal{S} \ (\text{ptr-span } p \cup M) \ (\text{ptr-span } p \cup A) \ t_0 \ s' \ t \rrbracket$

$\implies \text{refines } (f \ p) \ g \ s' \ t \ (\text{rel-stack } \mathcal{S} \ (\text{ptr-span } p \cup M1) \ (\text{ptr-span } p \cup A) \ s' \ t_0 \ Q)$

assumes *M1-M*: $M1 \subseteq M$

shows $\text{refines } (\text{with-fresh-stack-ptr } (\text{Suc } 0) \ I \ f) \ g \ s \ t \ (\text{rel-stack } \mathcal{S} \ M1 \ A \ s \ t_0 \ Q)$

unfolding *with-fresh-stack-ptr-def*

apply (*rule refines-assume-result-and-state-left*)

apply *clarsimp*

subgoal for *p d vs bs*

apply (*rule refines-on-exit-left*)

apply (*rule f [of - - htd-upd ($\lambda d. \text{override-on } d \ \text{stack-byte-typing } (\text{ptr-span } p)) \ t_0]$*)

subgoal

by (*erule stack-allocs-cases*) (*auto*)

subgoal

using *stack-allocs-stack-subset-stack-free* **by** *fastforce*

subgoal

apply (*erule stack-allocs-cases*)

using *stack*

by (*auto simp add: rel-alloc-def stack-free-def*)

subgoal

apply (*erule stack-allocs-cases*)

using *stack*

by (*auto simp add: rel-alloc-def stack-free-def*)

```

subgoal
  apply (erule stack-allocs-cases)
  using stack
  by (auto simp add: rel-alloc-def)
subgoal
  apply (erule stack-allocs-cases)
  using stack
  by (metis h-val-heap-update-padding heap.get-upd length-1-conv nth-Cons-0)
subgoal
  apply (erule stack-allocs-cases)
  apply (auto simp add: equal-upto-heap-stack-alloc)

  done
subgoal
  by (simp add: stack-free-stack-allocs)
subgoal
  apply (frule stack-free-stack-allocs)
  apply (erule stack-allocs-cases)
  using stack M1-M
  apply (simp add: rel-alloc-def)
  apply (simp add: frame-heap-update-padding)
  using frame-stack-alloc
  apply (subst frame-stack-alloc)
  apply (auto simp add: stack-free-def)
  done
subgoal
  by blast
subgoal
  apply clarsimp
  apply (clarsimp simp add: rel-stack-def)
  apply (clarsimp simp add: rel-alloc-def)
  apply (intro conjI)
  subgoal
    apply (erule stack-allocs-cases)
    using domain-bound stack domain-bound-heap-update-list
    apply (simp add: rel-alloc-def)
    by (simp add: disjoint-subset domain-bound-heap-update-list)
  subgoal
  by (metis (no-types, lifting) Int-Un-eq(1) distrib-inf-le equal-on-stack-free-preservation

    in-mono inf-idem rel-alloc-def stack stack-free-stack-allocs stack-free-stack-releases-mono'

    stack-releases-stack-allocs-inverse subset-drop-Diff-strg)
  subgoal
    by (metis (no-types, lifting) Int-Un-eq(1) distinct-element distrib-inf-le
      equal-on-stack-free-preservation inf.idem order-antisym-conv rel-alloc-def
      stack stack-free-stack-allocs stack-free-stack-releases-mono' stack-releases-stack-allocs-inverse
      subset-drop-Diff-strg)
  subgoal

```

```

apply (erule stack-allocs-cases)
using M1-M
  by (smt (verit, best) UnE c-guard-def disj-subset disjoint-union-distrib
inter-commute
      orthD2 rel-alloc-def stack stack-free-stack-releases)

subgoal
  apply (erule stack-allocs-cases)
  apply (subst frame-stack-release)
  subgoal by auto
    subgoal by auto (metis IntI empty-iff mem-Collect-eq rel-alloc-def stack
stack-free-def)
    subgoal by auto (meson c-null-guard-def)
    apply auto
  done
subgoal
  apply (subst stack-free-stack-releases)
subgoal
  by (meson c-guard-def stack-allocs-cases)
subgoal
  apply clarsimp
by (simp add: equal-upto-def heap-update-nmem-same heap-update-padding-def)
done
subgoal
  using stack-allocs-releases-equal-on-stack
  by (smt (verit, del-insts) UnE add.right-neutral equal-upto-def mult-is-0
      stack-releases-footprint stack-releases-other stack-releases-stack-allocs-inverse
      times-nat.simps(2))
  done
done
done

```

lemma *gen-with-fresh-stack-ptr-rel-stack*:

```

assumes I': hd ' I s  $\subseteq$  I' — There are two cases of local variables: I s = ( $\lambda$ -.
UNIV) (uninitialized) and I s = ( $\lambda$ -. {v}) (initialized)
assumes stack: rel-alloc S M A t0 s t
assumes domain-bound: domain-bound A Q
assumes f:  $\bigwedge p s' t_0 v$ .
   $\llbracket$ ptr-span p  $\subseteq$  S; ptr-span p  $\subseteq$  stack-free (htd s);
  ptr-span p  $\cap$  A = {}; ptr-span p  $\cap$  M = {};
  root-ptr-valid (htd s') p;
  h-val (hmem s') p = v;
  [v]  $\in$  I s;
  equal-upto-heap-on (ptr-span p) s' s;
  stack-free (htd s')  $\subseteq$  stack-free (htd s);
  rel-alloc S (ptr-span p  $\cup$  M) (ptr-span p  $\cup$  A) t0 s' t $\rrbracket$ 
 $\implies$  refines (f p) (g v) s' t (rel-stack S (ptr-span p  $\cup$  M1) (ptr-span p  $\cup$  A) s'
t0 Q)
assumes M1-M: M1  $\subseteq$  M

```

```

shows refines (with-fresh-stack-ptr (Suc 0) I f) (select I' >>= g) s t (rel-stack
S M1 A s t0 Q)
unfolding with-fresh-stack-ptr-def
apply (rule refines-assume-result-and-state-left)
apply clarsimp
subgoal for p d vs bs
  apply (rule refines-on-exit-left)
  apply (rule refines-select-right-witness [where x=hd vs])
  subgoal using I'
    by auto
  apply (rule f [of - - - htd-upd ( $\lambda d.$  override-on d stack-byte-typing (ptr-span
p)) t0])
subgoal
  by (erule stack-allocs-cases) (auto)
subgoal
  using stack-allocs-stack-subset-stack-free by fastforce
subgoal
  apply (erule stack-allocs-cases)
  using stack
  by (auto simp add: rel-alloc-def stack-free-def)
subgoal
  apply (erule stack-allocs-cases)
  using stack
  by (auto simp add: rel-alloc-def stack-free-def)
subgoal
  apply (erule stack-allocs-cases)
  using stack
  by (auto simp add: rel-alloc-def)
subgoal
  apply (erule stack-allocs-cases)
  apply (cases vs)
  apply (auto simp add: h-val-heap-update-padding)
  done
subgoal
  by (cases vs) auto
subgoal
  apply (erule stack-allocs-cases)
  apply (auto simp add: equal-upto-heap-stack-alloc)
  done
subgoal
  by (simp add: stack-free-stack-allocs)
subgoal
  apply (frule stack-free-stack-allocs)
  apply (erule stack-allocs-cases)
  using stack M1-M
  apply (simp add: rel-alloc-def)
  apply (simp add: frame-heap-update-padding)
  using frame-stack-alloc
  apply (subst frame-stack-alloc)

```



```

    apply (auto simp add: stack-free-def)
  done
subgoal
  by blast
subgoal
  apply clarsimp
  apply (clarsimp simp add: rel-stack-def)
  apply (clarsimp simp add: rel-alloc-def)
  apply (intro conjI)
subgoal
  apply (erule stack-allocs-cases)
  using domain-bound stack domain-bound-heap-update-list
  apply (simp add: rel-alloc-def)
  by (simp add: disjoint-subset domain-bound-heap-update-list)
subgoal
  by (metis (no-types, lifting) Int-Un-eq(1) distrib-inf-le equal-on-stack-free-preservation

    in-mono inf-idem rel-alloc-def stack stack-free-stack-allocs stack-free-stack-releases-mono'

    stack-releases-stack-allocs-inverse subset-drop-Diff-strg)
subgoal
  by (metis (no-types, lifting) Int-Un-eq(1) distinct-element distrib-inf-le
    equal-on-stack-free-preservation inf.idem order-antisym-conv rel-alloc-def
    stack stack-free-stack-allocs stack-free-stack-releases-mono' stack-releases-stack-allocs-inverse
    subset-drop-Diff-strg)
subgoal
  apply (erule stack-allocs-cases)
  using M1-M
  by (smt (verit, best) UnE c-guard-def disj-subset disjoint-union-distrib
    inter-commute
    orthD2 rel-alloc-def stack stack-free-stack-releases)

subgoal
  apply (erule stack-allocs-cases)
  apply (subst frame-stack-release)
  subgoal by auto
  subgoal by auto (metis IntI empty-iff mem-Collect-eq rel-alloc-def stack
    stack-free-def)
  subgoal by auto (meson c-null-guard-def)
  apply auto
  done
subgoal
  apply (subst stack-free-stack-releases)
  subgoal
  by (meson c-guard-def stack-allocs-cases)
  subgoal
  apply clarsimp
  by (simp add: equal-upto-def heap-update-nmem-same heap-update-padding-def)
  done

```

subgoal
using *stack-allocs-releases-equal-on-stack*
by (*smt (verit, del-insts) UnE add.right-neutral equal-upto-def mult-is-0*
stack-releases-footprint stack-releases-other stack-releases-stack-allocs-inverse
times-nat.simps(2))
done
done
done

lemma *with-fresh-stack-ptr-rel-stack-uninitialized'*:

assumes *stack: rel-alloc S M A t₀ s t*
assumes *f: $\bigwedge p s' t_0 v.$*
 \llbracket *ptr-span p* \subseteq *S*; *ptr-span p* \subseteq *stack-free (htd s)*;
ptr-span p \cap *A* = {}; *ptr-span p* \cap *M* = {};
root-ptr-valid (htd s') *p*;
h-val (hmem s') *p* = *v*;
equal-upto-heap-on (ptr-span p) s' s;
stack-free (htd s') \subseteq *stack-free (htd s)*;
rel-alloc S (ptr-span p \cup M) (ptr-span p \cup A) t₀ s' t
 \implies *refines (f p) (g v) s' t (rel-stack S (ptr-span p \cup M1) (ptr-span p \cup A) s'*
t₀ Q)
assumes *M1-M: M1 \subseteq M*
assumes *domain-bound: domain-bound A Q*
shows *refines (with-fresh-stack-ptr (Suc 0) ($\lambda.$ UNIV) (L2-VARS f ns)) (L2-seq*
(L2-unknown ns) g) s t (rel-stack S M1 A s t₀ Q)
unfolding *L2-seq-def L2-unknown-def L2-VARS-def*
by (*rule gen-with-fresh-stack-ptr-rel-stack [OF hd-UNIV stack domain-bound f*
M1-M])

lemma *with-fresh-stack-ptr-rel-stack-uninitialized:*

assumes *stack: rel-alloc S M A t₀ s t*
assumes *f: $\bigwedge p s' t_0.$*
 \llbracket *ptr-span p* \subseteq *S*; *ptr-span p* \subseteq *stack-free (htd s)*;
ptr-span p \cap *A* = {}; *ptr-span p* \cap *M* = {};
root-ptr-valid (htd s') *p*;
equal-upto-heap-on (ptr-span p) s' s;
stack-free (htd s') \subseteq *stack-free (htd s)*;
rel-alloc S (ptr-span p \cup M) (ptr-span p \cup A) t₀ s' t
 \implies *refines (f p) (g' s' p) s' t (rel-stack S (ptr-span p \cup M1) (ptr-span p \cup A)*
s' t₀ Q)
assumes *g: $\bigwedge s' p. ptr-span p \cap A = \{\}$ \implies equal-upto (ptr-span p) (hmem s')*
(hmem s) \implies g' s' p = g (h-val (hmem s') p)
assumes *M1-M: M1 \subseteq M*
assumes *domain-bound: domain-bound A Q*
shows *refines (with-fresh-stack-ptr (Suc 0) ($\lambda.$ UNIV) (L2-VARS f ns)) (L2-seq*
(L2-unknown ns) g) s t (rel-stack S M1 A s t₀ Q)
using *with-fresh-stack-ptr-rel-stack-uninitialized' [OF stack - M1-M domain-bound]*
f g
by (*smt (verit, best) equal-upto-heap-on-hmem*)

lemma *with-fresh-stack-ptr-rel-stack-uninitialized-g-normalised*:

assumes *stack*: *rel-alloc* \mathcal{S} M A t_0 s t
assumes *f*: $\bigwedge p$ s' t_0 .
 \llbracket *ptr-span* $p \subseteq \mathcal{S}$; *ptr-span* $p \subseteq$ *stack-free* (*htd* s);
ptr-span $p \cap A = \{\}$; *ptr-span* $p \cap M = \{\}$;
root-ptr-valid (*htd* s') p ;
equal-upto-heap-on (*ptr-span* p) s' s ;
stack-free (*htd* s') \subseteq *stack-free* (*htd* s);
rel-alloc \mathcal{S} (*ptr-span* $p \cup M$) (*ptr-span* $p \cup A$) t_0 s' t
 \implies *refines* (*f* p) (*g* (*h-val* (*hmem* s') p)) s' t (*rel-stack* \mathcal{S} (*ptr-span* $p \cup M1$)
(*ptr-span* $p \cup A$) s' t_0 Q)
assumes *M1-M*: $M1 \subseteq M$
assumes *domain-bound*: *domain-bound* A Q
shows *refines* (*with-fresh-stack-ptr* (*Suc* 0) (λ -. *UNIV*) (*L2-VARS* f ns)) (*L2-seq*
(*L2-unknown* ns) g) s t (*rel-stack* \mathcal{S} $M1$ A s t_0 Q)
apply (*rule with-fresh-stack-ptr-rel-stack-uninitialized* [*OF stack f - M1-M do-*
main-bound])
apply *auto*
done

lemma *with-fresh-stack-ptr-rel-stack-initialized'*:

fixes *g*:: ($'d$, $'e$, $'s$) *exn-monad*
assumes *stack*: *rel-alloc* \mathcal{S} M A t_0 s t
assumes *f*: $\bigwedge p$ s' t_0 .
 \llbracket *ptr-span* $p \subseteq \mathcal{S}$; *ptr-span* $p \subseteq$ *stack-free* (*htd* s);
ptr-span $p \cap A = \{\}$; *ptr-span* $p \cap M = \{\}$;
root-ptr-valid (*htd* s') p ;
h-val (*hmem* s') $p = v$;
equal-upto-heap-on (*ptr-span* p) s' s ;
stack-free (*htd* s') \subseteq *stack-free* (*htd* s);
rel-alloc \mathcal{S} (*ptr-span* $p \cup M$) (*ptr-span* $p \cup A$) t_0 s' t
 \implies *refines* (*f* p) g s' t (*rel-stack* \mathcal{S} (*ptr-span* $p \cup M1$) (*ptr-span* $p \cup A$) s' t_0 Q)
assumes *M1-M*: $M1 \subseteq M$
assumes *domain-bound*: *domain-bound* A Q
shows *refines* (*with-fresh-stack-ptr* (*Suc* 0) (λ -. $\{[v]\}$) (*L2-VARS* f ns)) g s t
(*rel-stack* \mathcal{S} $M1$ A s t_0 Q)
unfolding *L2-VARS-def*
apply (*rule gen-with-fresh-stack-ptr-rel-stack* [**where** $I = (\lambda$ -. $\{[v]\})$, *OF hd-singleton*,
simplified select-singleton-conv, *OF stack domain-bound f M1-M*])
apply *auto*
done

lemma *with-fresh-stack-ptr-rel-stack-initialized*:

fixes *g*:: ($'d$, $'e$, $'s$) *exn-monad*
assumes *stack*: *rel-alloc* \mathcal{S} M A t_0 s t
assumes *f*: $\bigwedge p$ s' t_0 .
 \llbracket *ptr-span* $p \subseteq \mathcal{S}$; *ptr-span* $p \subseteq$ *stack-free* (*htd* s);
ptr-span $p \cap A = \{\}$; *ptr-span* $p \cap M = \{\}$;

root- ptr -valid (*htd* s') p ;
h-val (*hmem* s') $p = v$;
equal-upto-heap-on (*ptr-span* p) $s' s$;
stack-free (*htd* s') \subseteq *stack-free* (*htd* s);
rel-alloc \mathcal{S} (*ptr-span* $p \cup M$) (*ptr-span* $p \cup A$) $t_0 s' t$]
 \implies *refines* (*f p*) (*g' s' p*) $s' t$ (*rel-stack* \mathcal{S} (*ptr-span* $p \cup M1$) (*ptr-span* $p \cup A$)
 $s' t_0 Q$)
assumes g : $\bigwedge s' p. \text{ptr-span } p \cap A = \{\}$ \implies *equal-upto* (*ptr-span* p) (*hmem* s')
(*hmem* s) \implies *h-val* (*hmem* s') $p = v \implies g' s' p = g$
assumes $M1-M$: $M1 \subseteq M$
assumes *domain-bound*: *domain-bound* $A Q$
shows *refines* (*with-fresh-stack- ptr* (*Suc* 0) ($\lambda-. \{[v]\}$) (*L2-VARS* $f ns$)) $g s t$
(*rel-stack* \mathcal{S} $M1 A s t_0 Q$)
using *with-fresh-stack- ptr -rel-stack-initialized'* [*OF stack - M1-M domain-bound*]
 g
equal-upto-heap-on-hmem f by auto

lemma *with-fresh-stack- ptr -rel-stack-fix-initialized*:

fixes g :: ($'d, 'e, 's$) *exn-monad*
assumes *stack*: *rel-alloc* $\mathcal{S} M A t_0 s t$
assumes f : $\bigwedge p s' t_0.$
[*ptr-span* $p \subseteq \mathcal{S}$; *ptr-span* $p \subseteq$ *stack-free* (*htd* s);
ptr-span $p \cap A = \{\}$; *ptr-span* $p \cap M = \{\}$;
root- ptr -valid (*htd* s') p ;
h-val (*hmem* s') $p = v$;
equal-upto-heap-on (*ptr-span* p) $s' s$;
stack-free (*htd* s') \subseteq *stack-free* (*htd* s);
rel-alloc \mathcal{S} (*ptr-span* $p \cup M$) (*ptr-span* $p \cup A$) $t_0 s' t$]
 \implies *refines* (*f p*) (*g' s' p*) $s' t$ (*rel-stack* \mathcal{S} (*ptr-span* $p \cup M1$) (*ptr-span* $p \cup A$)
 $s' t_0 Q$)
assumes g : $\bigwedge s' p. \text{ptr-span } p \cap A = \{\}$ \implies *equal-upto* (*ptr-span* p) (*hmem* s')
(*hmem* s) \implies *h-val* (*hmem* s') $p = v \implies g' s' p = g$
assumes $M1-M$: $M1 \subseteq M$
assumes I : $I s = \{[v]\}$
assumes *domain-bound*: *domain-bound* $A Q$
shows *refines* (*with-fresh-stack- ptr* (*Suc* 0) I (*L2-VARS* $f ns$)) $g s t$ (*rel-stack* \mathcal{S}
 $M1 A s t_0 Q$)
proof –
from I **have** *eq*: *run* (*with-fresh-stack- ptr* (*Suc* 0) ($\lambda-. \{[v]\}$) (*L2-VARS* $f ns$)) s
 $=$ *run* (*with-fresh-stack- ptr* (*Suc* 0) I (*L2-VARS* $f ns$)) s
unfolding *with-fresh-stack- ptr -def*
by (*simp add: run-bind*)
from *with-fresh-stack- ptr -rel-stack-initialized* [*OF stack f g M1-M domain-bound*]
have *refines* (*with-fresh-stack- ptr* (*Suc* 0) ($\lambda-. \{[v]\}$) (*L2-VARS* $f ns$)) $g s t$
(*rel-stack* \mathcal{S} $M1 A s t_0 Q$) .
from *refines-subst-left* [*OF this*] *eq*
show *?thesis*
by *blast*
qed

lemma *with-fresh-stack-ptr-rel-stack-fix-initialized-g-normalised:*

fixes g :: ('d, 'e, 's) *exn-monad*
assumes $stack$: *rel-alloc* $\mathcal{S} M A t_0 s t$
assumes f : $\bigwedge p s' t_0$.
 $\llbracket ptr\text{-span } p \subseteq \mathcal{S}; ptr\text{-span } p \subseteq stack\text{-free } (htd s);$
 $ptr\text{-span } p \cap A = \{\}; ptr\text{-span } p \cap M = \{\};$
 $root\text{-ptr}\text{-valid } (htd s') p;$
 $h\text{-val } (hmem s') p = v;$
 $equal\text{-upto}\text{-heap}\text{-on } (ptr\text{-span } p) s' s;$
 $stack\text{-free } (htd s') \subseteq stack\text{-free } (htd s);$
 $rel\text{-alloc } \mathcal{S} (ptr\text{-span } p \cup M) (ptr\text{-span } p \cup A) t_0 s' t \rrbracket$
 $\implies refines (f p) g s' t (rel\text{-stack } \mathcal{S} (ptr\text{-span } p \cup M1) (ptr\text{-span } p \cup A) s' t_0 Q)$
assumes $M1\text{-}M$: $M1 \subseteq M$
assumes I : $I s = \{[v]\}$
assumes $domain\text{-}bound$: *domain-bound* $A Q$
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) I (*L2-VARS f ns*)) $g s t (rel\text{-stack } \mathcal{S} M1 A s t_0 Q)$
apply (*rule with-fresh-stack-ptr-rel-stack-fix-initialized* [*OF stack f - M1-M - domain-bound*])
using I
apply *auto*
done

lemma *with-fresh-stack-ptr-rel-stack:*

assumes $stack$: *rel-alloc* $\mathcal{S} M A t_0 s t$
assumes f : $\bigwedge p s' t_0$.
 $\llbracket ptr\text{-span } p \subseteq \mathcal{S}; ptr\text{-span } p \subseteq stack\text{-free } (htd s);$
 $ptr\text{-span } p \cap A = \{\}; ptr\text{-span } p \cap M = \{\};$
 $root\text{-ptr}\text{-valid } (htd s') p;$
 $[h\text{-val } (hmem s') p] \in I s;$
 $equal\text{-upto}\text{-heap}\text{-on } (ptr\text{-span } p) s' s;$
 $stack\text{-free } (htd s') \subseteq stack\text{-free } (htd s);$
 $rel\text{-alloc } \mathcal{S} (ptr\text{-span } p \cup M) (ptr\text{-span } p \cup A) t_0 s' t \rrbracket$
 $\implies refines (f p) (g' s' p) s' t (rel\text{-stack } \mathcal{S} (ptr\text{-span } p \cup M1) (ptr\text{-span } p \cup A) s' t_0 Q)$
assumes g : $\bigwedge s' p. ptr\text{-span } p \cap A = \{\} \implies equal\text{-upto } (ptr\text{-span } p) (hmem s')$
 $(hmem s) \implies [h\text{-val } (hmem s') p] \in I s \implies g' s' p = g$
assumes $M1\text{-}M$: $M1 \subseteq M$
assumes $domain\text{-}bound$: *domain-bound* $A Q$
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) I (*L2-VARS f ns*)) $g s t (rel\text{-stack } \mathcal{S} M1 A s t_0 Q)$
unfolding *L2-VARS-def*
using *with-fresh-stack-ptr-rel-stack''* [*OF stack domain-bound*] $M1\text{-}M f g equal\text{-upto}\text{-heap}\text{-on}\text{-hmem}$
by *auto*

lemma *refines-rel-stack-project-result:*

assumes *refines* $f g s t (rel\text{-stack } \mathcal{S} M A s t_0 (rel\text{-xval}\text{-stack } L R))$

```

assumes  $\bigwedge h x y. R h x y \implies R' h x (prj y)$ 
shows refines f (L2-seq g (ETA-TUPLED ( $\lambda x. L2\text{-gets } (\lambda \cdot. prj x) ns$ ))) s t
(rel-stack S M A s t0 (rel-xval-stack L R'))
using assms
apply (clarsimp simp add: refines-def-old L2-defs rel-stack-def rel-xval-stack-def
ETA-TUPLED-def
reaches-bind succeeds-bind
split: prod.splits sum.splits)
subgoal for r s'
apply (cases r)
subgoal
apply (clarsimp simp add: default-option-def Exn-def [symmetric])
by (smt (verit, del-insts) Exn-def Result-neq-Exn case-exception-or-result-Exn
rel-xval.simps)
subgoal
apply clarsimp
by (smt (verit, ccfv-SIG) Result-neq-Exn case-exception-or-result-Result rel-xval.cases
rel-xval.simps(2))
done
done

```

lemma *refines-rel-stack-adjust-result:*

```

assumes refines f g s t (rel-stack S M A s t0 (rel-xval-stack L R))
assumes  $\bigwedge s' t' v w. R (hmem s') v w \implies rel\text{-alloc } S M A t_0 s' t' \implies$ 
equal-upto (M  $\cup$  stack-free (htd s')) (hmem s') (hmem s)  $\implies$ 
equal-upto M (htd s') (htd s)  $\implies$ 
equal-on S (htd s') (htd s)  $\implies R' (hmem s') v (adj w)$ 
shows refines f (L2-seq g (ETA-TUPLED ( $\lambda x. L2\text{-gets } (\lambda \cdot. adj x) ns$ ))) s t
(rel-stack S M A s t0 (rel-xval-stack L R'))
using assms
apply (clarsimp simp add: refines-def-old L2-defs rel-stack-def rel-xval-stack-def
ETA-TUPLED-def
reaches-bind succeeds-bind
split: prod.splits sum.splits)
subgoal for r s'
apply (cases r)
subgoal
apply (clarsimp simp add: default-option-def Exn-def [symmetric])
by (smt (verit, del-insts) Exn-def Result-neq-Exn case-exception-or-result-Exn
rel-xval.simps)
subgoal for v
apply (erule-tac x=Result v in allE)
apply (erule-tac x=s' in allE)
apply (fastforce simp add: rel-xval.simps)
done
done
done

```

lemma *stack-allocs-frame*:
assumes *alloc*: $(p, d) \in \text{stack-allocs } (\text{Suc } 0) \mathcal{S} \text{ TYPE}('a::\text{mem-type}) (\text{htd } s)$
shows $\exists d. (p, d) \in \text{stack-allocs } (\text{Suc } 0) \mathcal{S} \text{ TYPE}('a::\text{mem-type}) (\text{htd } (\text{frame } A \ t_0 \ s))$
proof –
have $\text{stack-free } (\text{htd } s) \subseteq \text{stack-free } (\text{htd } (\text{frame } A \ t_0 \ s))$
by (*rule stack-free-htd-frame*)
from *stack-allocs-stack-free-mono* [*OF this alloc*]
show *?thesis* .
qed

lemma *heap-list-update-nth*:
 $\bigwedge h \ p. \text{length } v \leq \text{addr-card} \implies$
 $i < \text{length } v \implies$
 $(\text{heap-update-list } p \ v \ h) (p + \text{of-nat } i) = v!i$
proof (*induct v arbitrary: i*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
thus *?case*
by (*metis heap-list-nth heap-list-update*)
qed

lemma *stack-alloc-simulation-aux*:
fixes *p::'a::mem-type ptr*
assumes *neq-stack-byte*: $\text{typ-uinfo-t } \text{TYPE}('a) \neq \text{typ-uinfo-t } \text{TYPE}(\text{stack-byte})$
assumes *disjnt*: $\text{stack-free } (\text{htd } s) \cap A = \{\}$
assumes *stack-free*: $\forall a \in \text{ptr-span } p. \text{root-ptr-valid } (\text{htd } s) (\text{PTR}(\text{stack-byte}) \ a)$
assumes *not-null*: $0 \notin \text{ptr-span } p$
assumes *lbs*: $\text{length } bs = \text{size-of } \text{TYPE}('a)$
assumes *root-ptr*: $\text{root-ptr-valid } (\text{ptr-force-type } p \ (\text{htd } s)) \ p$
shows
 $\text{htd-upd } (\lambda x. \text{override-on } (\text{ptr-force-type } p \ (\text{htd } s)) (\text{htd } t_0) (A - \text{stack-free } (\text{ptr-force-type } p \ (\text{htd } s))))$
 $(\text{hmem-upd } (\lambda x. \text{override-on } (\text{heap-update-padding } p \ (vs \ ! \ 0) \ bs \ x) (\text{hmem } t_0)$
 $(A \cup \text{stack-free } (\text{ptr-force-type } p \ (\text{htd } s)))) \ s) =$
 $\text{htd-upd } (\lambda-. \text{ptr-force-type } p \ (\text{override-on } (\text{htd } s) (\text{htd } t_0) (A - \text{stack-free } (\text{htd } s))))$
 $(\text{hmem-upd } (\lambda x. \text{heap-update-padding } p \ (vs \ ! \ 0) \ bs \ (\text{override-on } x \ (\text{hmem } t_0)$
 $(A \cup \text{stack-free } (\text{htd } s)))) \ s)$
proof –
from *stack-free* **have** *stack-free'*: $\text{ptr-span } p \subseteq \text{stack-free } (\text{htd } s)$
by (*simp add: stack-free-def subsetI*)
with *root-ptr neq-stack-byte*
have *sf-eq*: $\text{stack-free } (\text{ptr-force-type } p \ (\text{htd } s)) = \text{stack-free } (\text{htd } s) - \text{ptr-span } p$

apply (*clarsimp simp add: root-ptr-valid-def stack-free-def, intro set-eqI iffI*)
apply (*clarsimp, intro conjI*)
apply (*smt (verit, best) ptr-force-type-d size-of-def typ-uinfo-size valid-root-footprint-domain-cases*)

apply (*metis (mono-tags, lifting) size-of-tag valid-root-footprint-type-neq*)
by (*simp add: ptr-force-type-d valid-root-footprint-def*)
have *heap-eq: override-on (heap-update-padding p (vs ! 0) bs (hmem s)) (hmem t₀) (A ∪ (stack-free (htd s) - ptr-span p))*
 $=$
heap-update-padding p (vs ! 0) bs (override-on (hmem s) (hmem t₀) (A ∪ stack-free (htd s)))
using *disjnt stack-free' lbs*
apply (*clarsimp simp add: override-on-def fun-eq-iff, intro conjI impI*)
subgoal by (*smt (verit) CTypes.mem-type-simps(2) disjoint-subset heap-update-nmem-same heap-update-padding-def orthD2*)
subgoal by (*clarsimp simp add: heap-update-nmem-same heap-update-padding-def*)
by (*auto simp add: heap-update-mem-same heap-update-padding-def*)
(smt (verit, best) CTypes.mem-type-simps(2) heap-update-nmem-same heap-update-padding-def subset-iff)

have *htd-eq: override-on (ptr-force-type p (htd s)) (htd t₀) (A - stack-free (ptr-force-type p (htd s)))*
 $=$
ptr-force-type p (override-on (htd s) (htd t₀) (A - stack-free (htd s)))

apply (*simp add: sf-eq*)
using *disjnt stack-free'*
by (*auto simp add: override-on-def fun-eq-iff ptr-force-type-split*)
show *?thesis*
apply (*simp add: sf-eq*)
using *heap-eq htd-eq*
by (*metis (no-types, lifting) heap.upd-cong sf-eq typing.upd-cong*)

qed

lemma *keep-with-fresh-stack-ptr-rel-stack'*:
assumes *stack: rel-alloc S M A t₀ s t*
assumes *I: I (frame A t₀ s) = I s*
assumes *f: $\bigwedge p s' t_0 t. \text{--- I could use the fixed } t_0$*
 \llbracket *ptr-span p \subseteq S; ptr-span p \subseteq stack-free (htd s);*
 \llbracket *ptr-span p \cap A = {}; ptr-span p \cap M = {};*
 \llbracket *root-ptr-valid (htd s') p;*
 \llbracket *[h-val (hmem s') p] \in I s;*
 \llbracket *equal-upto-heap-on (ptr-span p) s' s;*
 \llbracket *stack-free (htd s') \subseteq stack-free (htd s);*
 \llbracket *rel-alloc S (ptr-span p \cup M) A t₀ s' t;*
 \llbracket *ptr-span p \cap stack-free (htd s') = {}]
 \implies *refines (f p) (g p) s' t (rel-stack S (ptr-span p \cup M1) A s' t₀ Q)**

assumes *M1-M: M1 \subseteq M*
assumes *domain-bound: domain-bound A Q*


```

shows refines (with-fresh-stack-ptr (Suc 0) I f) (with-fresh-stack-ptr (Suc 0) I
g) s t (rel-stack S M1 A s t0 Q)
unfolding with-fresh-stack-ptr-def
apply (rule refines-assume-result-and-state-both-same-val-frame [where frm =
frame A t0])
subgoal using stack by (auto simp add: rel-alloc-def)
subgoal for s' p
  using stack I
  apply (clarsimp simp add: rel-alloc-def)
  apply (frule stack-allocs-frame [where A=A and t0=t0])
  apply clarsimp
  apply (erule stack-allocs-cases)
subgoal for d vs bs d'
  apply (rule exI[where x=d'])
  apply clarsimp
  apply (rule exI[where x=vs])
  apply (simp add: I)
  apply (erule stack-allocs-cases)
  apply clarsimp
  apply (rule exI[where x=bs])
  using stack

  apply (clarsimp simp add: frame-def heap-commute comp-def )
  apply (rule stack-alloc-simulation-aux)
  apply auto
  done
done
subgoal for s' p
  apply clarsimp
  apply (rule refines-on-exit-same-cleanup-choice)

  apply (rule f [rule-format, where t0=t0 ] )
subgoal
  by (erule stack-allocs-cases) auto
subgoal
  apply (erule stack-allocs-cases)
  by (simp add: stack-free-def subset-iff)
subgoal
  using stack rel-alloc-def stack-allocs-stack-subset-stack-free' by fastforce
subgoal using stack
  by (metis (no-types, lifting) add.right-neutral inf.orderE inf-assoc inf-bot-right
mult-is-0
rel-alloc-def stack-allocs-stack-subset-stack-free times-nat.simps(2))
subgoal
  apply (erule stack-allocs-cases)
  by (metis htd-hmem-upd typing.get-upd)
subgoal
  by (metis h-val-heap-update-padding heap.get-upd length-1-conv nth-Cons-0)
subgoal

```

by (*smt* (*verit*, *best*) *append.simps(1)* *dual-order.refl* *fold.simps(1)* *fold.simps(2)*)

heap-state.equal-upto-heap-stack-alloc *heap-state-axioms* *id-comp* *lense.upd-cong*
ptr-add-0-id
semiring-1-class.of-nat-0 *stack-allocs-cases* *typing.lense-axioms* *upt.simps(1)*
upt.simps(2))

subgoal
by (*simp* *add: stack-free-stack-allocs*)

subgoal using *stack M1-M*

by (*smt* (*verit*) *Diff-Diff-Int* *Diff-eq-empty-iff* *Int-Diff* *Int-Un-distrib* *One-nat-def*
Un-Int-eq(3)
add-mult-distrib *add-right-cancel* *htd-hmem-upd* *inf.absorb-iff2* *inf.orderE*
plus-1-eq-Suc
rel-alloc-def *stack-allocs-stack-subset-stack-free* *stack-free-stack-allocs* *times-nat.simps(2)*
typing.get-upd)

subgoal
apply (*erule* *stack-allocs-cases*)

by (*smt* (*verit*, *ccfv-threshold*) *disjoint-iff* *htd-hmem-upd* *in-ptr-span-itself*
mem-Collect-eq
root-ptr-valid-casesE *stack-free-def* *typing.get-upd*)

subgoal by *blast*

subgoal for *d vs bs s1 v t w s2*

apply *clarsimp*

subgoal for *bs1*

apply (*rule* *exI*[**where** *x = hmem-upd*
(heap-update-list (ptr-val p)
(heap-list (hmem t₀) (size-of TYPE('a)) (ptr-val p))
(htd-upd (stack-releases (Suc 0) p) t)])

apply (*rule* *conjI*)

subgoal using *heap-list-length* **by** *blast*

using *stack*

apply (*clarsimp* *simp* *add: rel-stack-def*)

apply (*intro* *conjI*)

subgoal
apply (*erule* *stack-allocs-cases*)

using *domain-bound*

by (*simp* *add: disjoint-subset* *domain-bound-heap-update-list* *rel-alloc-def*)

subgoal
apply (*clarsimp* *simp* *add: rel-alloc-def*, *intro* *conjI*)

apply (*metis* (*no-types*, *lifting*) *Int-Un-eq(1)*
in-mono *inf-idem* *stack-free-stack-allocs* *stack-free-stack-releases-mono'*
stack-releases-stack-allocs-inverse *subset-drop-Diff-strg* *sup.absorb-iff1*)

apply (*metis* (*no-types*, *lifting*) *Int-absorb2* *Int-iff* *Un-Int-eq(4)*
boolean-algebra.conj-disj-distrib *distrib-inf-le* *orthD1*
stack-free-stack-allocs *stack-free-stack-releases-mono'* *stack-releases-stack-allocs-inverse*
subset-drop-Diff-strg)

using *M1-M*

```

      apply (smt (verit) c-guard-def disjoint-subset disjoint-union-distrib
inter-commute orthD2
      stack-allocs-cases stack-free-stack-releases)

    apply (erule stack-allocs-cases)
    apply (subst frame-stack-release-keep [where bs=bs1] )
    subgoal
      by (metis disjoint-union-distrib inter-commute)
    subgoal
      by (metis add.right-neutral disjoint-subset mult-Suc mult-is-0)
    subgoal
      by (meson c-guard-def)
    subgoal
      by assumption
    subgoal by simp
    done
  subgoal
    apply (subst stack-free-stack-releases)
    subgoal
      by (meson c-guard-def stack-allocs-cases)
    subgoal
      apply clarsimp
  by (simp add: equal-upto-def heap-update-nmem-same heap-update-padding-def)
  done
  subgoal
    using stack-allocs-releases-equal-on-stack
    by (smt (verit, del-Insts) UnE add.right-neutral equal-upto-def mult-is-0
      stack-releases-footprint stack-releases-other stack-releases-stack-allocs-inverse
times-nat.simps(2))
  done
done
done
done
done

```

lemma *keep-with-fresh-stack-ptr-rel-stack:*

```

assumes stack: rel-alloc  $\mathcal{S}$   $M$   $A$   $t_0$   $s$   $t$ 
assumes  $I$ :  $I$   $s = I$  (frame  $A$   $t_0$   $s$ ) — FIXME: make refinement  $\Gamma$  similar to
condition in L2-condition / L2-while?
assumes  $f$ :  $\bigwedge p$   $s'$   $t_0$   $t$ . — I could use the fixed  $t_0$ 
[[ptr-span  $p \subseteq \mathcal{S}$ ; ptr-span  $p \subseteq$  stack-free (htd  $s$ );
ptr-span  $p \cap A = \{\}$ ; ptr-span  $p \cap M = \{\}$ ;
root-ptr-valid (htd  $s'$ )  $p$ ;
[h-val (hmem  $s'$ )  $p$ ]  $\in I$   $s$ ;
equal-upto-heap-on (ptr-span  $p$ )  $s'$   $s$ ;
stack-free (htd  $s'$ )  $\subseteq$  stack-free (htd  $s$ );
rel-alloc  $\mathcal{S}$  (ptr-span  $p \cup M)$   $A$   $t_0$   $s'$   $t$ ;
ptr-span  $p \cap$  stack-free (htd  $s'$ ) =  $\{\}$ ]

```

$\implies \text{refines } (f p) (g' s' p) s' t \text{ (rel-stack } \mathcal{S} \text{ (ptr-span } p \cup M1) A s' t_0 Q)$
assumes $g: \bigwedge s' p. \text{ptr-span } p \cap A = \{\}$ $\implies \text{equal-upto (ptr-span } p) (\text{hmem } s')$
 $(\text{hmem } s) \implies [h\text{-val } (\text{hmem } s') p] \in I s$
 $\implies g' s' p = g p$
assumes $M1\text{-}M: M1 \subseteq M$
assumes $\text{domain-bound: domain-bound } A Q$
shows $\text{refines (with-fresh-stack-ptr (Suc } 0) I (L2\text{-}VARS f ns)) \text{ (with-fresh-stack-ptr (Suc } 0) I (L2\text{-}VARS g ns)) s t \text{ (rel-stack } \mathcal{S} M1 A s t_0 Q)$
unfolding $L2\text{-}VARS\text{-def}$
using $\text{keep-with-fresh-stack-ptr-rel-stack' [OF stack } I \text{ [symmetric] - } M1\text{-}M \text{ domain-bound}] f g$
using $\text{equal-upto-heap-on-hmem by force}$

lemma $\text{keep-with-fresh-stack-ptr-rel-stack-g-normalised:}$

assumes $\text{stack: rel-alloc } \mathcal{S} M A t_0 s t$
assumes $I: I s = I \text{ (frame } A t_0 s)$ — FIXME: make refinement I' similar to condition in $L2\text{-condition} / L2\text{-while?}$
assumes $f: \bigwedge p s' t_0 t. \text{ — I could use the fixed } t_0$
 $\llbracket \text{ptr-span } p \subseteq \mathcal{S}; \text{ptr-span } p \subseteq \text{stack-free (htd } s);$
 $\text{ptr-span } p \cap A = \{\}; \text{ptr-span } p \cap M = \{\};$
 $\text{root-ptr-valid (htd } s') p;$
 $[h\text{-val } (\text{hmem } s') p] \in I s;$
 $\text{equal-upto-heap-on (ptr-span } p) s' s;$
 $\text{stack-free (htd } s') \subseteq \text{stack-free (htd } s);$
 $\text{rel-alloc } \mathcal{S} \text{ (ptr-span } p \cup M) A t_0 s' t;$
 $\text{ptr-span } p \cap \text{stack-free (htd } s') = \{\}\rrbracket$
 $\implies \text{refines } (f p) (g p) s' t \text{ (rel-stack } \mathcal{S} \text{ (ptr-span } p \cup M1) A s' t_0 Q)$
assumes $M1\text{-}M: M1 \subseteq M$
assumes $\text{domain-bound: domain-bound } A Q$
shows $\text{refines (with-fresh-stack-ptr (Suc } 0) I (L2\text{-}VARS f ns)) \text{ (with-fresh-stack-ptr (Suc } 0) I (L2\text{-}VARS g ns)) s t \text{ (rel-stack } \mathcal{S} M1 A s t_0 Q)$
apply $(\text{rule keep-with-fresh-stack-ptr-rel-stack [OF stack } I f \text{ - } M1\text{-}M \text{ domain-bound}]$
apply auto
done

lemma $\text{refines-rel-stack-adapt-right:}$

assumes $\text{stack: rel-alloc } \mathcal{S} M A t_0 s t$
assumes $M1\text{-}M: M1 \subseteq M$
assumes $f\text{-}g: \text{refines } f g s t \text{ (rel-stack } \mathcal{S} M1 A s t_0 \text{ (rel-xval-stack } L R))$
assumes $h: \bigwedge s' v t' w.$
 $R (\text{hmem } s') v w \implies$
 $\text{rel-alloc } \mathcal{S} M1 A t_0 s' t' \implies$
 $\text{equal-upto } (M1 \cup \text{stack-free (htd } s')) (\text{hmem } s') (\text{hmem } s) \implies$
 $\text{htd } s' = \text{htd } s \implies$
 $\text{refines } (L2\text{-gets } (\lambda\text{-. } v) ns) (h w) s' t' \text{ (rel-stack } \mathcal{S} M1 A s t_0 \text{ (rel-xval-stack } L R'))$
shows $\text{refines } f (L2\text{-seq } g h) s t \text{ (rel-stack } \mathcal{S} M1 A s t_0 \text{ (rel-xval-stack } L R'))$
using $\text{stack } M1\text{-}M f\text{-}g h$
apply $(\text{clarsimp simp add: refines-def-old } L2\text{-defs}$

```

    rel-stack-def rel-alloc-def succeeds-bind reaches-bind )
subgoal for  $r\ s'$ 
  apply (cases  $r$ )
subgoal for  $e$ 
  apply (clarsimp simp add: default-option-def Exn-def [symmetric])
subgoal for  $y$ 
  apply (erule-tac  $x=Exn\ y$  in  $allE$ )
  apply (erule-tac  $x=s'$  in  $allE$ )
  by (metis (mono-tags, lifting) Exn-def case-exception-or-result-Exn rel-xval-stack-simps(5))
done
subgoal for  $v$ 
  apply (erule-tac  $x=Result\ v$  in  $allE$ )
  apply (erule-tac  $x=s'$  in  $allE$ )
  apply (fastforce simp add: rel-xval.simps)
done
done
done

```

lemma *refines-handleE'-both-sides:*

```

assumes refines  $f\ g\ s\ t\ Q$ 
assumes  $\bigwedge v\ v'\ s'\ t'.\ Q\ (Result\ v,\ s')\ (Result\ v',\ t') \implies R\ (Result\ v,\ s')\ (Result\ v',\ t')$ 
assumes  $\bigwedge v\ e'\ s'\ t'.\ Q\ (Result\ v,\ s')\ (Exn\ e',\ t') \implies \text{refines}\ (return\ v)\ (h'\ e')$ 
 $s'\ t'\ R$ 
assumes  $\bigwedge e\ v'\ s'\ t'.\ Q\ (Exn\ e,\ s')\ (Result\ v',\ t') \implies \text{refines}\ (h\ e)\ (return\ v')\ s'\ t'\ R$ 
assumes  $\bigwedge e\ e'\ s'\ t'.\ Q\ (Exn\ e,\ s')\ (Exn\ e',\ t') \implies \text{refines}\ (h\ e)\ (h'\ e')\ s'\ t'\ R$ 
shows refines ( $f\ <catch>\ h$ ) ( $g\ <catch>\ h'$ )  $s\ t\ R$ 
by (rule Spec-Monad.refines-catch [OF assms(1) assms(5) assms(4) assms(3) assms(2)])

```

lemma *refines-rel-stack-map-exn:*

```

assumes  $f\ g:$  refines  $f\ g\ s\ t\ (rel-stack\ \mathcal{S}\ M\ A\ s\ t_0\ (rel-xval-stack\ L\ R))$ 
assumes  $L:$   $\bigwedge e\ e'\ s'.\ L\ (hmem\ s')\ e\ e' \implies L'\ (hmem\ s')\ (emb\ e)\ (emb\ e')$ 
shows refines ( $map\ value\ (map\ exn\ emb)\ f$ ) ( $map\ value\ (map\ exn\ emb)\ g$ )  $s\ t$ 
 $(rel-stack\ \mathcal{S}\ M\ A\ s\ t_0\ (rel-xval-stack\ L'\ R))$ 
using assms
apply (clarsimp simp add: refines-def-old L2-defs
  rel-stack-def rel-alloc-def reaches-map-value)
subgoal for  $s'\ r$ 
apply (cases  $r$ )
subgoal for  $e$ 
apply (clarsimp simp add: default-option-def Exn-def [symmetric])
subgoal for  $y$ 
apply (erule-tac  $x=Exn\ y$  in  $allE$ )
apply (erule-tac  $x=s'$  in  $allE$ )
by auto

```

```

done
subgoal for v
  apply (erule-tac x=Result v in allE)
  apply (erule-tac x=s' in allE)
  apply (auto)
done
done
done

```

```

lemma refines-rel-stack-map-exn-exit:
  assumes f-g: refines f g s t (rel-stack S M A s t0 (rel-xval-stack (rel-exit L) R))
  assumes L:  $\bigwedge e e' h. L h e e' \implies L' h (emb e) (emb' e')$ 
  shows refines (map-value (map-exn (emb o the-Nonlocal)) f) (map-value (map-exn
    (emb') g) s t (rel-stack S M A s t0 (rel-xval-stack L' R))
  using assms
  apply (clarsimp simp add: refines-def-old L2-defs
    rel-stack-def rel-alloc-def reaches-map-value )
  subgoal for s' r
    apply (cases r)
  subgoal for e
    apply (clarsimp simp add: default-option-def Exn-def [symmetric])
  subgoal for y
    apply (erule-tac x=Exn y in allE)
    apply (erule-tac x=s' in allE)
    apply (auto simp add: rel-exit-def)
  done
done
subgoal for v
  apply (erule-tac x=Result v in allE)
  apply (erule-tac x=s' in allE)
  apply (auto)
done
done
done

```

```

lemma throw-L2-throw-conv: throw x = L2-throw x []
  by (simp add: L2-throw-def)

```

```

lemma L2-catch-join-exn-conv:
  (L2-catch
    (L2-seq
      (L2-catch g (λx. L2-seq (liftE (h x)) (λ-. L2-throw (prj x) ns1)))
      (λs. liftE (g1 s)))
    (λr. L2-throw (emb r) ns2)) =
  (L2-seq
    (L2-catch g (λx. L2-seq (liftE (h x)) (λ-. L2-throw (emb (prj x)) ns2)))
    (λs. liftE (g1 s)))

```

unfolding *L2-defs*

apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add: runs-to-iff*)
apply (*auto simp add: runs-to-def-old default-option-def Exn-def*)
done

lemma *L2-call-rel-stack-embed-exit:*

assumes $L: \bigwedge e e' h. L h e e' \implies L' h (emb e) (emb' e')$

assumes *f-g: refines*

f
(*L2-seq*
(*L2-catch* *g* ($\lambda x. (L2-seq (liftE (h x)) (\lambda-. L2-throw (prj x) ns')$))
($\lambda s. liftE (g1 s)$)))
s t
(*rel-stack* \mathcal{S} *M1* *A* *s* *t*₀ (*rel-xval-stack* (*rel-exit* *L*) *R*))

shows *refines*

(*L2-call* *f* (*emb o the-Nonlocal*) *ns*)
(*L2-seq*
(*L2-catch* *g* ($\lambda x. (L2-seq (liftE (h x)) (\lambda-. L2-throw (emb' (prj x)) ns')$))
($\lambda s. liftE (g1 s)$)))
s t
(*rel-stack* \mathcal{S} *M1* *A* *s* *t*₀ (*rel-xval-stack* *L'* *R*))

proof –

from *refines-rel-stack-map-exn-exit* [**where** $L' = L'$, *OF* *f-g L*]

have *refines*

(*map-value* (*map-exn* (*emb o the-Nonlocal*)) *f*)
(*map-value* (*map-exn* (*emb'*)))
(*L2-seq*
(*L2-catch* *g* ($\lambda x. L2-seq (liftE (h x)) (\lambda-. L2-throw (prj x) ns')$))
($\lambda s. liftE (g1 s)$)))
s t (*rel-stack* \mathcal{S} *M1* *A* *s* *t*₀ (*rel-xval-stack* *L'* *R*)) .

then show *?thesis*

apply (*simp add: map-exn-catch-conv L2-catch-def [symmetric] throw-L2-throw-conv*)

apply (*simp add: L2-catch-join-exn-conv*)

apply (*simp add: L2-call-def map-exn-catch-conv L2-catch-def [symmetric]*)

L2-throw-def)

done

qed

lemma *L2-call-rel-stack-nest-exit-guarded:*

assumes *f-g: P t* \implies *refines*

f
(*L2-seq* (*L2-guard* *P*) ($\lambda-. (L2-seq (L2-catch *g* ($\lambda x. (L2-seq (liftE (h x)) (\lambda-. L2-throw (prj x) ns')$))
($\lambda s. liftE (g1 s)$)))$

s t

```

    (rel-stack  $\mathcal{S}$  M1 A s t0 (rel-xval-stack (rel-exit L) R))
assumes L:  $\bigwedge e e' h. P t \implies L h e e' \implies L' h (emb e) (emb' e')$ 
shows refines
    (L2-call f (emb o the-Nonlocal) ns)
    (L2-seq (L2-guard P) ( $\lambda$ -.
      (L2-seq
        (L2-catch g ( $\lambda x. (L2-seq (liftE (h x)) (\lambda-. L2-throw (emb' (prj x)) ns'))$ ))
        ( $\lambda s. liftE (g1 s)$ ))))
    s t
    (rel-stack  $\mathcal{S}$  M1 A s t0 (rel-xval-stack L' R))
apply (rule refines-L2-guard-right')
apply (rule L2-call-rel-stack-embed-exit [where L=L and emb=emb and emb'=
emb'])
apply (rule L, assumption+)
using f-g refines-L2-guard-rightE by fastforce

```

```

lemma bind-catch-liftE-assoc:
  (do {v  $\leftarrow$  (g <catch> ( $\lambda x. do \{$ 
    y  $\leftarrow$  liftE (h x);
    throw (exn x y)
  }));
    liftE (g1 v)
  }) =
  (do {v  $\leftarrow$  g; liftE (g1 v)} <catch> ( $\lambda x. do \{$ 
    y  $\leftarrow$  liftE (h x);
    throw (exn x y)
  })))
apply (rule spec-monad-eqI)
apply (auto simp add: runs-to-iff Exn-def[symmetric] elim!: runs-to-weaken)
done

```

```

lemma bind-catch-liftE-split-catch: (do {
  s  $\leftarrow$  g;
  liftE (g1 s)
} <catch> ( $\lambda x. do \{$ 
  -  $\leftarrow$  liftE (h x);
  throw (emb (prj x))
}))) =
((do {
  s  $\leftarrow$  g;
  liftE (g1 s)
} <catch> ( $\lambda x. do \{$ 
  -  $\leftarrow$  liftE (h x);
  throw (prj x) })))
  <catch> ( $\lambda x. throw (emb x)$ ))
apply (rule spec-monad-eqI)

```


apply (*auto simp add: runs-to-iff Exn-def[symmetric] elim!: runs-to-weaken*)
done

lemma *refines-catch-right-trans:*

fixes $f:: ('e, 'a, 's) \text{exn-monad}$
assumes $f\text{-}g: \text{refines } f\ g\ s\ t\ Q$
assumes $R: \bigwedge r\ s'\ e\ t'. Q\ (r, s')\ (\text{Exn } e, t') \implies \text{refines } (\text{yield } r)\ (h\ e)\ s'\ t'\ R$
assumes $\text{Exn-Res}: \bigwedge e\ s'\ v'\ t'. Q\ (\text{Exn } e, s')\ (\text{Result } v', t') \implies R\ (\text{Exn } e, s')$
(Result v', t')
assumes $\text{Res-Res}: \bigwedge v\ v'\ s'\ t'. Q\ (\text{Result } v, s')\ (\text{Result } v', t') \implies R\ (\text{Result } v,$
 $s')\ (\text{Result } v', t')$
shows $\text{refines } f\ (g\ \langle \text{catch} \rangle\ h)\ s\ t\ R$
apply (*subst f-catch-throw[symmetric, of f]*)
apply (*rule refines-catch [OF f-g]*)
subgoal premises prems **using** R [*OF prems(1)*] **by** *simp*
subgoal using Exn-Res **by** (*auto simp add: refines-yield-right-iff*)
subgoal premises prems **using** R [*OF prems(1)*] **by** *simp*
subgoal using Res-Res **by** (*auto simp add: refines-yield-right-iff*)
done

lemma *L2-call-rel-stack-embellish-exit:*

assumes $s\text{-}t: \text{rel-alloc } \mathcal{S}\ M\ A\ t_0\ s\ t$
assumes $f\text{-}g: P\ t \implies \text{refines}$
 f
 $(L2\text{-seq } (L2\text{-guard } P)\ (\lambda\cdot$
 $(L2\text{-seq}$
 $(L2\text{-catch } g\ (\lambda x. (L2\text{-seq } (\text{liftE } (h\ x))\ (\lambda\cdot. L2\text{-throw } (\text{prj } x)\ ns))))$
 $(\lambda s. \text{liftE } (g1\ s))))))$
 $s\ t$
 $(\text{rel-stack } \mathcal{S}\ M1\ A\ s\ t_0\ (\text{rel-xval-stack } (\text{rel-exit } L)\ R))$
assumes $L: \bigwedge s'\ t'\ e\ e'. L\ (\text{hmem } s')\ e\ e' \implies P\ t \implies \text{rel-alloc } \mathcal{S}\ M\ A\ t_0\ s'\ t'$
 \implies
 $\text{equal-upto } (M1 \cup \text{stack-free } (\text{htd } s'))\ (\text{hmem } s')\ (\text{hmem } s) \implies \text{htd } s' =$
 $\text{htd } s \implies$
 $L'\ (\text{hmem } s')\ e\ (\text{emb } e')$
assumes $M1: P\ t \implies M1 \subseteq M$
shows refines
 f
 $(L2\text{-seq } (L2\text{-guard } P)\ (\lambda\cdot$
 $(L2\text{-seq}$
 $(L2\text{-catch } g\ (\lambda x. (L2\text{-seq } (\text{liftE } (h\ x))\ (\lambda\cdot. L2\text{-throw } (\text{emb } (\text{prj } x))\ ns\text{-exit))))$
 $(\lambda s. \text{liftE } (g1\ s))))))$
 $s\ t$
 $(\text{rel-stack } \mathcal{S}\ M1\ A\ s\ t_0\ (\text{rel-xval-stack } (\text{rel-exit } L')\ R))$
unfolding $L2\text{-defs}$
apply (*clarsimp simp add: refines-bind-guard-right-iff bind-catch-liftE-assoc*)
apply (*simp add: bind-catch-liftE-split-catch*)
apply (*rule refines-catch-right-trans*)

```

apply (rule f-g [simplified L2-defs refines-bind-guard-right-iff bind-catch-liftE-assoc,
rule-format])
  apply assumption
  apply simp
subgoal
  apply (clarsimp simp add: rel-stack-def rel-exit-def)
  using L s-t by (auto simp add: rel-alloc-def)
subgoal
  by (auto simp add: rel-stack-def)
subgoal
  by (auto simp add: rel-stack-def)
done

```

definition *override-heap-on* P $t1$ $t2$ \equiv
 $hmem\text{-upd}$ ($\lambda h.$ *override-on* h ($hmem$ $t2$) P)
 $(htd\text{-upd}$ ($\lambda d.$ *override-on* d (htd $t2$) P) $t1$)

lemma *override-heap-on-empty* [simp]: *override-heap-on* $\{\}$ $t1$ $t2$ = $t1$
by (simp add: *override-heap-on-def*)

lemma *refines-rel-stack-override-heap-on-exit*:

```

fixes p:: 'a::xmem-type ptr
assumes stack: rel-alloc  $\mathcal{S}$   $M$   $A$   $t_0$   $s$   $t$ 
assumes f-g:  $\bigwedge s$   $t$   $t_0.$  rel-alloc  $\mathcal{S}$   $M$  ( $P \cup A$ )  $t_0$   $s$   $t$   $\implies$ 
  refines  $f$  ( $g$   $s$ )  $s$   $t$  (rel-stack  $\mathcal{S}$   $M1$  ( $P \cup A$ )  $s$   $t_0$   $Q$ )
assumes disj-A:  $P \cap A = \{\}$ 
assumes p-M1:  $P \subseteq M1$ 
assumes M1-M:  $M1 \subseteq M$ 
assumes disj-stack-free-s:  $P \cap \text{stack-free} (htd\ s) = \{\}$ 
shows refines  $f$  ( $g$   $s$ )  $s$   $t$ 
  (rel-stack  $\mathcal{S}$   $M1$  ( $P \cup A$ )  $s$  (override-heap-on  $P$   $t_0$   $t$ )  $Q$ )

```

proof –

```

from stack have sf-s-t: stack-free (htd  $s$ )  $\subseteq$  stack-free (htd  $t$ )
  using rel-alloc-def stack-free-htd-frame by auto

```

```

from stack have t:  $t = \text{frame } A\ t_0\ s$ 
  by (auto simp add: rel-alloc-def)

```

define $t0'$ **where** $t0' = \text{override-heap-on } P\ t_0\ t$

have eq-htd:

```

  override-on (htd  $s$ ) (htd  $t_0$ ) ( $A - \text{stack-free} (htd\ s)$ ) =
    override-on (htd  $s$ ) (htd  $t0'$ ) ( $(P \cup A) - \text{stack-free} (htd\ s)$ )
  using disj-stack-free-s disj-A  $t$ 

```

by (auto simp add: $t0'$ -def *override-heap-on-def* *override-on-def* *fun-eq-iff* *frame-def*)

have eq-hmem:

```

  override-on (hmem  $s$ ) (hmem  $t_0$ ) ( $A \cup \text{stack-free} (htd\ s)$ ) =
    override-on (hmem  $s$ ) (hmem  $t0'$ ) ( $(P \cup A) \cup \text{stack-free} (htd\ s)$ )

```

```

using disj-stack-free-s disj-A t
by (auto simp add: t0'-def override-heap-on-def override-on-def fun-eq-iff frame-def)

from t have t-t0':  $t = \text{frame } ((P \cup A)) \ t0' \ s$ 
using eq-htd eq-hmem
apply (simp add: frame-def)
by (metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong)

have stack': rel-alloc  $\mathcal{S} \ M \ ((P \cup A)) \ t0' \ s \ t$ 
using stack disj-stack-free-s disj-stack-free-s t t-t0'
by (auto simp add: rel-alloc-def)

from disj-stack-free-s disj-stack-free-s stack
have stack-free': stack-free (htd s)  $\cap (P \cup A) = \{\}$ 
by (auto simp add: rel-alloc-def)

from f-g [OF stack1]
have f-g': refines f (g s) s t
      (rel-stack  $\mathcal{S} \ M1 \ ((P \cup A)) \ s \ t0' \ Q$ ) .
then show ?thesis
by (simp add: t0'-def)
qed

```

```

lemma refines-rel-stack-override-heap-on-exit-guarded:
  fixes p: 'a::xmem-type ptr
  assumes stack: rel-alloc  $\mathcal{S} \ M \ A \ t_0 \ s \ t$ 
  assumes f-g:  $\bigwedge s \ t \ t_0. \ G \implies \text{rel-alloc } \mathcal{S} \ M \ (P \cup A) \ t_0 \ s \ t \implies$ 
    refines f (L2-seq (L2-guard ( $\lambda-. \ G$ )) ( $\lambda-. \ g \ s$ )) s t (rel-stack  $\mathcal{S} \ M1 \ (P \cup A) \ s$ 
     $t_0 \ Q$ )
  assumes disj-A:  $G \implies P \cap A = \{\}$ 
  assumes p-M1:  $G \implies P \subseteq M1$ 
  assumes M1-M:  $G \implies M1 \subseteq M$ 
  assumes disj-stack-free-s:  $G \implies P \cap \text{stack-free } (\text{htd } s) = \{\}$ 
  shows refines f (L2-seq (L2-guard ( $\lambda-. \ G$ )) ( $\lambda-. \ g \ s$ )) s t
    (rel-stack  $\mathcal{S} \ M1 \ (P \cup A) \ s \ (\text{override-heap-on } P \ t_0 \ t) \ Q$ )
proof –
  {assume guard: G
   have refines f (L2-seq (L2-guard ( $\lambda-. \ G$ )) ( $\lambda-. \ g \ s$ )) s t (rel-stack  $\mathcal{S} \ M1 \ (P \cup$ 
    $A) \ s \ (\text{override-heap-on } P \ t_0 \ t) \ Q$ )
   using refines-rel-stack-override-heap-on-exit [OF stack f-g [OF guard] disj-A
   [OF guard] p-M1 [OF guard]
    $M1-M$  [OF guard] disj-stack-free-s [OF guard]].}
   then show ?thesis
   by (rule refines-L2-guard-right')
  }
qed

```

```

lemma refines-rel-stack-override-heap-emptyI:

```

assumes *refines f g s t* (*rel-stack* \mathcal{S} M ($P \cup A$) s (*override-heap-on* P t_0 t) Q)
shows *refines f g s t* (*rel-stack* \mathcal{S} M ($P \cup \{\}$ $\cup A$) s (*override-heap-on* ($P \cup \{\}$) t_0 t) Q)
using *assms*
by *simp*

lemma *refines-rel-stack-pop-heap-both:*

fixes p : *'a::xmem-type ptr*
assumes *stack: rel-alloc* \mathcal{S} M A t_0 s t
assumes *f-g: refines f g s t*
 (*rel-stack* \mathcal{S} $M1$ (*ptr-span* $p \cup P \cup A$) s
 (*override-heap-on* (*ptr-span* $p \cup P$) t_0 t)
 (*rel-xval-stack* (*rel-exit* (*rel-push* p L)) (*rel-push* p R)))
assumes *disj-P-A: $P \cap A = \{\}$*
assumes *disj-p-A: $ptr\text{-span } p \cap A = \{\}$*
assumes *disj-p-P: $ptr\text{-span } p \cap P = \{\}$*
assumes *disj-stack-free-s-p: $ptr\text{-span } p \cap stack\text{-free } (htd\ s) = \{\}$*
assumes *disj-stack-free-s-P: $P \cap stack\text{-free } (htd\ s) = \{\}$*
shows *refines*
 f
 (*L2-seq*
 (*L2-catch* g ($\lambda x.$ (*L2-seq* (*IO-modify-heap-paddingE* p ($\lambda.$ (*fst* x))) ($\lambda.$ *L2-throw*
 (*snd* x) *ns-exit*))))
 ($\lambda x.$ (*L2-seq* (*IO-modify-heap-paddingE* p ($\lambda.$ (*fst* x))) ($\lambda.$ *L2-return* (*snd* x)
 ns)))) s t
 (*rel-stack* \mathcal{S} $M1$ ($P \cup A$) s (*override-heap-on* P t_0 t) (*rel-xval-stack* (*rel-exit* L)
 R))

proof –

from *stack have sf-s-t: stack-free (htd s) \subseteq stack-free (htd t)*
 using *rel-alloc-def stack-free-htd-frame* **by** *auto*

from *stack have t: t = frame A t₀ s*
 by (*auto simp add: rel-alloc-def*)

define $t0'$ **where** $t0' = \text{override-heap-on } (ptr\text{-span } p \cup P) t_0 t$

have *eq-htd:*

 (*override-on* (*htd* s) (*htd* t_0) ($A - stack\text{-free } (htd\ s)$) =
 (*override-on* (*htd* s) (*htd* $t0'$) ((*ptr-span* $p \cup P \cup A$) - *stack-free* (*htd* s)))
 using *disj-stack-free-s-p disj-stack-free-s-P disj-P-A disj-p-A t*
 by (*auto simp add: t0'-def override-heap-on-def override-on-def fun-eq-iff frame-def*)

have *eq-hmem:*

 (*override-on* (*hmem* s) (*hmem* t_0) ($A \cup stack\text{-free } (htd\ s)$) =
 (*override-on* (*hmem* s) (*hmem* $t0'$) ((*ptr-span* $p \cup P \cup A$) \cup *stack-free* (*htd*
 s)))
 using *disj-stack-free-s-p disj-stack-free-s-P disj-p-A disj-P-A t*
 by (*auto simp add: t0'-def override-heap-on-def override-on-def fun-eq-iff frame-def*)

```

from  $t$  have  $t-t0'$ :  $t = \text{frame } ((\text{ptr-span } p \cup P \cup A)) \ t0' \ s$ 
  using  $\text{eq-htd eq-hmem}$ 
  apply ( $\text{simp add: frame-def}$ )
  by ( $\text{metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong}$ )

have  $\text{stack}'$ :  $\text{rel-alloc } \mathcal{S} \ M \ ((\text{ptr-span } p \cup P \cup A)) \ t0' \ s \ t$ 
  using  $\text{stack disj-stack-free-s-P disj-stack-free-s-p } t \ t-t0'$ 
  by ( $\text{auto simp add: rel-alloc-def}$ )

from  $\text{disj-stack-free-s-p disj-stack-free-s-P disj-stack-free-s-P stack}$ 
have  $\text{stack-free}'$ :  $\text{stack-free } (\text{htd } s) \cap (P \cup A) = \{\}$ 
  by ( $\text{auto simp add: rel-alloc-def}$ )

show  $?thesis$ 
  unfolding  $L2\text{-seq-def}$ 
  apply ( $\text{rule refines-bindE-right [where } Q = \lambda(v, s') (w, t').$ 
     $\text{case } v \text{ of}$ 
       $\text{Exn } e \Rightarrow \exists w'. w = \text{Exn } w' \wedge \text{rel-stack } \mathcal{S} \ M1 \ (P \cup A) \ s \ (\text{override-heap-on}$ 
 $P \ t_0 \ t) \ (\text{rel-xval-stack } (\text{rel-exit } L) \ (\lambda- \ - \ . \ \text{False})) \ (\text{Exn } e, s') \ (\text{Exn } w', t')$ 
       $| \text{Result } v \Rightarrow \exists w'. w = \text{Result } w' \wedge \text{rel-stack } \mathcal{S} \ M1 \ (\text{ptr-span } p \cup P \cup A) \ s$ 
 $(\text{override-heap-on } (\text{ptr-span } p \cup P) \ t_0 \ t) \ (\text{rel-xval-stack } (\lambda- \ - \ . \ \text{False}) \ (\text{rel-push } p$ 
 $R)) \ (\text{Result } v, s') \ (\text{Result } w', t')]$ )
    apply ( $\text{rule refines-L2-catch-right}$ )
    apply ( $\text{rule f-g}$ )
  subgoal by ( $\text{clarsimp simp add: rel-stack-def } t0'\text{-def rel-alloc-def}$ )
  subgoal by  $\text{clarsimp}$ 
  subgoal by  $\text{clarsimp}$ 
  subgoal for  $e \ e' \ s' \ t'$ 
    apply ( $\text{frule rel-stack-unchanged-stack-free [OF } \text{stack}', \text{unfolded } t0'\text{-def}]$ )
    apply ( $\text{frule rel-stack-unchanged-stack-free' [OF } \text{stack}', \text{unfolded } t0'\text{-def}]$ )
    apply ( $\text{subst (asm) rel-stack-def}$ )
    apply ( $\text{clarsimp simp add: rel-exit-def}$ )
    apply ( $\text{subst (asm) rel-push-def}$ )
    apply  $\text{clarsimp}$ 
    apply ( $\text{rule refines-exec-IO-modify-heap-padding-single-step-right-InL}$ )
    using  $\text{stack}$ 
    apply ( $\text{clarsimp simp add: stack-free' rel-alloc-def rel-stack-def } t0'\text{-def [symmetric]}$ 
 $t \ [\text{symmetric}] \ t-t0' \ [\text{symmetric}]$ )
    subgoal premises  $\text{prems for } w \ x$ 
    proof –
      from  $\text{prems obtain}$ 
         $e = \text{Nonlocal } x$ 
         $e' = (\text{h-val } (\text{hmem } s') \ p, w)$ 
         $L \ (\text{hmem } s') \ x \ w$  and
         $\text{sf-s}'$ :  $\text{stack-free } (\text{htd } s') = \text{stack-free } (\text{htd } s)$  and
         $\text{sf-t}'$ :  $\text{stack-free } (\text{htd } (\text{frame } (\text{ptr-span } p \cup P \cup A) \ t0' \ s')) = \text{stack-free } (\text{htd}$ 
 $t)$  and
         $\text{stack-free } (\text{htd } s) \subseteq \mathcal{S}$ 

```

```

stack-free (htd s) ∩ (ptr-span p ∪ P ∪ A) = {}
stack-free (htd s) ∩ M1 = {}
t' = frame (ptr-span p ∪ P ∪ A) t0' s'
equal-upto (M1 ∪ stack-free (htd s)) (hmem s') (hmem s) and
htd-eq: htd s' = htd s by (simp)
have eq-typing:
  override-on (htd s') (htd t0') (ptr-span p ∪ P ∪ A - stack-free (htd s')) =
    override-on (htd s') (htd (override-heap-on P t0 t)) (P ∪ A -
stack-free (htd s'))
  using disj-P-A disj-p-A
  by (auto simp add: override-on-def fun-eq-iff t0'-def htd-eq t htd-frame
override-heap-on-def)

have eq-heap: heap-update-padding p (h-val (hmem s') p)
  (heap-list (hmem s') (size-of TYPE('a)) (ptr-val p))
  (override-on (hmem s') (hmem t0') (ptr-span p ∪ P ∪ A ∪ stack-free
(htd s'))) =
  override-on (hmem s') (hmem (override-heap-on P t0 t)) (P ∪ A ∪
stack-free (htd s'))
  using disj-P-A disj-p-A
  apply (simp add: override-on-def override-heap-on-def fun-eq-iff t0'-def)
  by (smt (verit, best) disj-p-P disj-stack-free-s-p h-val-def heap-list-length
heap-update-list-id' heap-update-list-value heap-update-padding-def
hmem-htd-upd max-size orthD2 sf-s' to-bytes-heap-list-id)

show hmem-upd
  (heap-update-padding p (h-val (hmem s') p)
  (heap-list (hmem s') (size-of TYPE('a)) (ptr-val p)))
  (frame (ptr-span p ∪ P ∪ A) t0' s') =
  frame (P ∪ A) (override-heap-on P t0 t) s'
  apply (simp add: frame-def comp-def)
  using eq-typing eq-heap
  by (metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong)
qed
done
subgoal by clarsimp
subgoal by clarsimp
subgoal by clarsimp
subgoal for v v' s' t'
  apply (clarsimp)
  apply (frule rel-stack-unchanged-stack-free [OF stack', unfolded t0'-def])
  apply (frule rel-stack-unchanged-stack-free' [OF stack', unfolded t0'-def])
  apply (subst (asm) rel-stack-def)
  apply (clarsimp simp add: rel-exit-def)
  apply (subst (asm) rel-push-def)
  apply clarsimp
  apply (rule refines-exec-IO-modify-heap-padding-step-right)
  apply (rule refines-exec-L2-return-right)

```

```

using stack
apply (clarsimp simp add: stack-free' rel-alloc-def rel-stack-def t0'-def [symmetric])
t [symmetric] t-t0' [symmetric]
subgoal premises prems for w
proof –
from prems obtain
  v' = (h-val (hmem s') p, w)
  R (hmem s') v w and
  sf-s': stack-free (htd s') = stack-free (htd s) and
  sf-t': stack-free (htd (frame (ptr-span p ∪ P ∪ A) t0' s')) = stack-free (htd
t) and
  stack-free (htd s) ⊆ S
  stack-free (htd s) ∩ (ptr-span p ∪ P ∪ A) = {}
  stack-free (htd s) ∩ M1 = {}
  t' = frame (ptr-span p ∪ P ∪ A) t0' s'
  equal-upto (M1 ∪ stack-free (htd s)) (hmem s') (hmem s) and
  htd-eq: htd s' = htd s by simp
have eq-typing:
  override-on (htd s') (htd t0') (ptr-span p ∪ P ∪ A – stack-free (htd s')) =
  override-on (htd s') (htd (override-heap-on P t0 t)) (P ∪ A –
stack-free (htd s'))
  using disj-P-A disj-p-A
  by (auto simp add: override-on-def fun-eq-iff t0'-def htd-eq t htd-frame
override-heap-on-def)

have eq-heap: heap-update-padding p (h-val (hmem s') p)
  (heap-list (hmem s') (size-of TYPE('a)) (ptr-val p))
  (override-on (hmem s') (hmem t0') (ptr-span p ∪ P ∪ A ∪ stack-free
(htd s')) =
  override-on (hmem s') (hmem (override-heap-on P t0 t)) (P ∪ A ∪
stack-free (htd s'))
using disj-P-A disj-p-A
apply (simp add: override-on-def override-heap-on-def fun-eq-iff t0'-def)
by (smt (verit, best) disj-p-P disj-stack-free-s-p h-val-def heap-list-length
heap-update-list-id' heap-update-list-value heap-update-padding-def
hmem-htd-upd max-size orthD2 sf-s' to-bytes-heap-list-id)

show hmem-upd
  (heap-update-padding p (h-val (hmem s') p)
  (heap-list (hmem s') (size-of TYPE('a)) (ptr-val p)))
  (frame (ptr-span p ∪ P ∪ A) t0' s') =
  frame (P ∪ A) (override-heap-on P t0 t) s'
apply (simp add: frame-def comp-def)
using eq-typing eq-heap
by (metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong)
qed
done
done

```

qed

lemma *refines-rel-stack-pop-heap-both-guarded*:

fixes p : 'a::xmem-type ptr
assumes $stack$: rel-alloc $\mathcal{S} M A t_0 s t$
assumes f - g : refines f (L2-seq (L2-guard (λ -. G)) (λ -. g)) $s t$
 (rel-stack $\mathcal{S} M1$ (ptr-span $p \cup P \cup A$) s
 (override-heap-on (ptr-span $p \cup P$) $t_0 t$)
 (rel-xval-stack (rel-exit (rel-push $p L$)) (rel-push $p R$)))
assumes $disj$ - P - A : $G \implies P \cap A = \{\}$
assumes $disj$ - p - A : $G \implies ptr$ -span $p \cap A = \{\}$
assumes $disj$ - p - P : $G \implies ptr$ -span $p \cap P = \{\}$
assumes $disj$ -stack-free- s - p : $G \implies ptr$ -span $p \cap stack$ -free ($htd s$) = $\{\}$
assumes $disj$ -stack-free- s - P : $G \implies P \cap stack$ -free ($htd s$) = $\{\}$
shows refines
 f
 (L2-seq
 (L2-catch (L2-seq (L2-guard (λ -. G)) (λ -. g))
 (λx . (L2-seq (IO-modify-heap-paddingE p (λ -. ($fst x$))) (λ -. L2-throw (snd
 x) ns-exit))))
 (λx . (L2-seq (IO-modify-heap-paddingE p (λ -. ($fst x$))) (λ -. L2-return (snd x)
ns)))) $s t$
 (rel-stack $\mathcal{S} M1$ ($P \cup A$) s (override-heap-on $P t_0 t$) (rel-xval-stack (rel-exit L)
 R))
proof –
 {
 assume grd : G
 have ?thesis
 by (rule refines-rel-stack-pop-heap-both [OF $stack$ f - g $disj$ - P - A [OF grd] $disj$ - p - A
[OF grd] $disj$ - p - P [OF grd]
 $disj$ -stack-free- s - p [OF grd] $disj$ -stack-free- s - P [OF grd]])
 } **then show** ?thesis
 by (rule refines-L2-guard-right''')

qed

lemma *refines-rel-stack-pop-heap-both-singleton*:

fixes p : 'a::xmem-type ptr
assumes $stack$: rel-alloc $\mathcal{S} M A t_0 s t$
assumes f - g : refines $f g s t$
 (rel-stack $\mathcal{S} M1$ (ptr-span $p \cup P \cup A$) s
 (override-heap-on (ptr-span $p \cup P$) $t_0 t$)
 (rel-xval-stack (rel-exit (rel-push $p L$)) (rel-singleton-stack p)))
assumes $disj$ - P - A : $P \cap A = \{\}$
assumes $disj$ - p - A : ptr -span $p \cap A = \{\}$
assumes $disj$ - p - P : ptr -span $p \cap P = \{\}$
assumes $disj$ -stack-free- s - p : ptr -span $p \cap stack$ -free ($htd s$) = $\{\}$
assumes $disj$ -stack-free- s - P : $P \cap stack$ -free ($htd s$) = $\{\}$
shows refines
 f


```

(L2-seq
(L2-catch  $g$  ( $\lambda x.$  (L2-seq (IO-modify-heap-paddingE  $p$  ( $\lambda.$  (fst  $x$ ))) ( $\lambda.$  L2-throw
(snd  $x$ ) ns-exit))))
( $\lambda x.$  (IO-modify-heap-paddingE  $p$  ( $\lambda.$   $x$ )))  $s$   $t$ 
(rel-stack  $\mathcal{S}$   $M1$  ( $P \cup A$ )  $s$  (override-heap-on  $P$   $t_0$   $t$ ) (rel-xval-stack (rel-exit  $L$ )
( $\lambda.$  (=))))
proof -
from stack have sf-s-t: stack-free (htd  $s$ )  $\subseteq$  stack-free (htd  $t$ )
using rel-alloc-def stack-free-htd-frame by auto

from stack have  $t$ :  $t = \text{frame } A \ t_0 \ s$ 
by (auto simp add: rel-alloc-def)

define  $t_0'$  where  $t_0' = \text{override-heap-on } (ptr\text{-span } p \cup P) \ t_0 \ t$ 

have eq-htd:
override-on (htd  $s$ ) (htd  $t_0$ ) ( $A - \text{stack-free } (htd \ s)$ ) =
override-on (htd  $s$ ) (htd  $t_0'$ ) ( $(ptr\text{-span } p \cup P \cup A) - \text{stack-free } (htd \ s)$ )
using disj-stack-free-s-p disj-stack-free-s-P disj-P-A disj-P-A t
by (auto simp add: t_0'-def override-heap-on-def override-on-def fun-eq-iff frame-def)

have eq-hmem:
override-on (hmem  $s$ ) (hmem  $t_0$ ) ( $A \cup \text{stack-free } (htd \ s)$ ) =
override-on (hmem  $s$ ) (hmem  $t_0'$ ) ( $(ptr\text{-span } p \cup P \cup A) \cup \text{stack-free } (htd \ s)$ )
using disj-stack-free-s-p disj-stack-free-s-P disj-p-A disj-P-A t
by (auto simp add: t_0'-def override-heap-on-def override-on-def fun-eq-iff frame-def)

from  $t$  have  $t-t_0'$ :  $t = \text{frame } ((ptr\text{-span } p \cup P \cup A)) \ t_0' \ s$ 
using eq-htd eq-hmem
apply (simp add: frame-def)
by (metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong)

have stack': rel-alloc  $\mathcal{S}$   $M$  ( $(ptr\text{-span } p \cup P \cup A)$ )  $t_0' \ s \ t$ 
using stack disj-stack-free-s-P disj-stack-free-s-p t t-t_0'
by (auto simp add: rel-alloc-def)

from disj-stack-free-s-p disj-stack-free-s-P disj-stack-free-s-P stack
have stack-free': stack-free (htd  $s$ )  $\cap (P \cup A) = \{\}$ 
by (auto simp add: rel-alloc-def)

show ?thesis
unfolding L2-seq-def
apply (rule refines-bindE-right [where  $Q = \lambda(v, s') (w, t')$ .
case  $v$  of
Exn  $e \Rightarrow \exists w'. w = \text{Exn } w' \wedge \text{rel-stack } \mathcal{S} \ M1 \ (P \cup A) \ s \ (\text{override-heap-on}$ 
 $P \ t_0 \ t) \ (\text{rel-xval-stack } (\text{rel-exit } L) \ (\lambda - - . \text{False})) \ (\text{Exn } e, s') \ (\text{Exn } w', t')$ 
| Result  $v \Rightarrow \exists w'. w = \text{Result } w' \wedge \text{rel-stack } \mathcal{S} \ M1 \ (ptr\text{-span } p \cup P$ 

```

$\cup A) s$ (*override-heap-on* (*ptr-span* $p \cup P$) t_0 t) (*rel-xval-stack* ($\lambda - - . \text{False}$)
(*rel-singleton-stack* p)) (*Result* v, s') (*Result* w', t'))
apply (*rule refines-L2-catch-right*)
apply (*rule f-g*)
subgoal by (*clarsimp simp add: rel-stack-def t0'-def rel-alloc-def*)
subgoal by *clarsimp*
subgoal by *clarsimp*
subgoal for $e e' s' t'$
apply (*frule rel-stack-unchanged-stack-free* [*OF stack', unfolded t0'-def*])
apply (*frule rel-stack-unchanged-stack-free'* [*OF stack', unfolded t0'-def*])
apply (*subst (asm) rel-stack-def*)
apply (*clarsimp simp add: rel-exit-def*)
apply (*subst (asm) rel-push-def*)
apply *clarsimp*
apply (*rule refines-exec-IO-modify-heap-padding-single-step-right-InL*)
using *stack*
apply (*clarsimp simp add: stack-free' rel-alloc-def rel-stack-def t0'-def* [*symmetric*]
 t [*symmetric*] $t-t0'$ [*symmetric*])
subgoal premises *prems* **for** $w x$
proof –
from *prems* **obtain**
 $e = \text{Nonlocal } x$
 $e' = (\text{h-val } (\text{hmem } s') p, w)$
 $L (\text{hmem } s') x w$ **and**
 $\text{sf-}s': \text{stack-free } (\text{htd } s') = \text{stack-free } (\text{htd } s)$ **and**
 $\text{sf-}t': \text{stack-free } (\text{htd } (\text{frame } (\text{ptr-span } p \cup P \cup A) t0' s')) = \text{stack-free } (\text{htd } t)$ **and**
 $\text{stack-free } (\text{htd } s) \subseteq \mathcal{S}$
 $\text{stack-free } (\text{htd } s) \cap (\text{ptr-span } p \cup P \cup A) = \{\}$
 $\text{stack-free } (\text{htd } s) \cap M1 = \{\}$
 $t' = \text{frame } (\text{ptr-span } p \cup P \cup A) t0' s'$
 $\text{equal-upto } (M1 \cup \text{stack-free } (\text{htd } s)) (\text{hmem } s') (\text{hmem } s)$ **and**
 $\text{htd-eq: htd } s' = \text{htd } s$ **by** (*simp*)
have *eq-typing*:
 $\text{override-on } (\text{htd } s') (\text{htd } t0') (\text{ptr-span } p \cup P \cup A - \text{stack-free } (\text{htd } s')) =$
 $\text{override-on } (\text{htd } s') (\text{htd } (\text{override-heap-on } P t_0 t)) (P \cup A -$
 $\text{stack-free } (\text{htd } s'))$
using *disj-P-A disj-p-A*
by (*auto simp add: override-on-def fun-eq-iff t0'-def htd-eq t htd-frame*
override-heap-on-def)

have *eq-heap*: $\text{heap-update-padding } p (\text{h-val } (\text{hmem } s') p)$
 $(\text{heap-list } (\text{hmem } s') (\text{size-of } \text{TYPE}(a)) (\text{ptr-val } p))$
 $(\text{override-on } (\text{hmem } s') (\text{hmem } t0') (\text{ptr-span } p \cup P \cup A \cup \text{stack-free}$
 $(\text{htd } s')) =$
 $\text{override-on } (\text{hmem } s') (\text{hmem } (\text{override-heap-on } P t_0 t)) (P \cup A \cup$
 $\text{stack-free } (\text{htd } s'))$
using *disj-P-A disj-p-A*

```

apply (simp add: override-on-def override-heap-on-def fun-eq-iff t0'-def)
by (smt (verit, best) disj-p-P disj-stack-free-s-p h-val-def heap-list-length
heap-update-list-id' heap-update-list-value heap-update-padding-def
hmem-htd-upd max-size orthD2 sf-s' to-bytes-heap-list-id)

show hmem-upd
  (heap-update-padding p (h-val (hmem s') p)
   (heap-list (hmem s') (size-of TYPE('a)) (ptr-val p)))
  (frame (ptr-span p  $\cup$  P  $\cup$  A) t0' s') =
  frame (P  $\cup$  A) (override-heap-on P t0 t) s'
apply (simp add: frame-def comp-def)
using eq-typing eq-heap
by (metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong)
qed
done

subgoal by clarsimp
subgoal by clarsimp
subgoal by clarsimp
subgoal for v v' s' t'
apply (clarsimp)
apply (frule rel-stack-unchanged-stack-free [OF stack', unfolded t0'-def])
apply (frule rel-stack-unchanged-stack-free' [OF stack', unfolded t0'-def])
apply (subst (asm) rel-stack-def)
apply (clarsimp simp add: rel-exit-def)
apply (subst (asm) rel-singleton-stack-def)
apply clarsimp
apply (rule refines-exec-IO-modify-heap-padding-single-step-right)
using stack
apply (clarsimp simp add: stack-free' rel-alloc-def rel-stack-def t0'-def [symmetric]
t [symmetric] t-t0' [symmetric])
subgoal premises prems proof –
from prems obtain
  v' = h-val (hmem s') p and
  sf-s': stack-free (htd s') = stack-free (htd s) and
  sf-t': stack-free (htd (frame (ptr-span p  $\cup$  P  $\cup$  A) t0' s')) = stack-free (htd
t) and
  stack-free (htd s)  $\subseteq$  S
  stack-free (htd s)  $\cap$  (ptr-span p  $\cup$  P  $\cup$  A) = {}
  stack-free (htd s)  $\cap$  M1 = {}
  t' = frame (ptr-span p  $\cup$  P  $\cup$  A) t0' s'
  equal-upto (M1  $\cup$  stack-free (htd s)) (hmem s') (hmem s) and
  htd-eq: htd s' = htd s by simp
have eq-typing:
  override-on (htd s') (htd t0') (ptr-span p  $\cup$  P  $\cup$  A – stack-free (htd s')) =
  override-on (htd s') (htd (override-heap-on P t0 t)) (P  $\cup$  A –
stack-free (htd s'))
using disj-P-A disj-p-A
by (auto simp add: override-on-def fun-eq-iff t0'-def htd-eq t htd-frame
override-heap-on-def)

```

```

have eq-heap: heap-update-padding p (h-val (hmem s') p)
      (heap-list (hmem s') (size-of TYPE('a)) (ptr-val p))
      (override-on (hmem s') (hmem t0') (ptr-span p ∪ P ∪ A ∪ stack-free
(htd s'))) =
      override-on (hmem s') (hmem (override-heap-on P t0 t)) (P ∪ A ∪
stack-free (htd s'))
using disj-P-A disj-p-A
apply (simp add: override-on-def override-heap-on-def fun-eq-iff t0'-def)
by (smt (verit, best) disj-p-P disj-stack-free-s-p h-val-def heap-list-length
heap-update-list-id' heap-update-list-value heap-update-padding-def
hmem-htd-upd max-size orthD2 sf-s' to-bytes-heap-list-id)

show hmem-upd
      (heap-update-padding p (h-val (hmem s') p)
      (heap-list (hmem s') (size-of TYPE('a)) (ptr-val p)))
      (frame (ptr-span p ∪ P ∪ A) t0' s') =
      frame (P ∪ A) (override-heap-on P t0 t) s'
apply (simp add: frame-def comp-def)
using eq-typing eq-heap
by (metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong)
qed
done
done
qed

```

lemma *refines-rel-stack-pop-heap-both-singleton-guarded:*

```

fixes p: 'a::xmem-type ptr
assumes stack: rel-alloc S M A t0 s t
assumes f-g: refines f (L2-seq (L2-guard (λ-. G)) (λ-. g)) s t
      (rel-stack S M1 (ptr-span p ∪ P ∪ A) s
      (override-heap-on (ptr-span p ∪ P) t0 t)
      (rel-xval-stack (rel-exit (rel-push p L)) (rel-singleton-stack p)))
assumes disj-P-A: G ⇒ P ∩ A = {}
assumes disj-p-A: G ⇒ ptr-span p ∩ A = {}
assumes disj-p-P: G ⇒ ptr-span p ∩ P = {}
assumes disj-stack-free-s-p: G ⇒ ptr-span p ∩ stack-free (htd s) = {}
assumes disj-stack-free-s-P: G ⇒ P ∩ stack-free (htd s) = {}
shows refines
      f
      (L2-seq
      (L2-catch (L2-seq (L2-guard (λ-. G)) (λ-. g))
      (λx. (L2-seq (IO-modify-heap-paddingE p (λ-. (fst x))) (λ-. L2-throw (snd
x) ns-exit))))
      (λx. (IO-modify-heap-paddingE p (λ-. x)))) s t
      (rel-stack S M1 (P ∪ A) s (override-heap-on P t0 t) (rel-xval-stack (rel-exit L)
(λ-. (=))))
proof -

```

```

{
  assume grd: G
  have ?thesis
    by (rule refines-rel-stack-pop-heap-both-singleton [OF stack f-g disj-P-A [OF
grd] disj-p-A [OF grd] disj-p-P [OF grd]
      disj-stack-free-s-p [OF grd] disj-stack-free-s-P [OF grd]])
  } then show ?thesis
    by (rule refines-L2-guard-right'''')
qed

```

lemma *refines-rel-stack-pop-heap-no-exit*:

```

fixes p: 'a::xmem-type ptr
assumes stack: rel-alloc S M A t0 s t
assumes f-g: refines f g s t
      (rel-stack S M1 (ptr-span p  $\cup P \cup A$ ) s
        (override-heap-on (ptr-span p  $\cup P$ ) t0 t)
        (rel-xval-stack (rel-exit ( $\lambda$ - - . False)) (rel-push p R)))
assumes disj-P-A:  $P \cap A = \{\}$ 
assumes disj-p-A:  $\text{ptr-span } p \cap A = \{\}$ 
assumes disj-p-P:  $\text{ptr-span } p \cap P = \{\}$ 
assumes disj-stack-free-s-p:  $\text{ptr-span } p \cap \text{stack-free (htd } s) = \{\}$ 
assumes disj-stack-free-s-P:  $P \cap \text{stack-free (htd } s) = \{\}$ 
shows refines
  f
  (L2-seq g ( $\lambda x$ . (L2-seq (IO-modify-heap-paddingE p ( $\lambda$ - . (fst x))) ( $\lambda$ - . L2-return
(snd x ns)))))) s t
  (rel-stack S M1 ( $P \cup A$ ) s (override-heap-on P t0 t) (rel-xval-stack (rel-exit ( $\lambda$ -
- . False)) R))

```

proof –

```

from stack have sf-s-t: stack-free (htd s)  $\subseteq$  stack-free (htd t)
using rel-alloc-def stack-free-htd-frame by auto

```

```

from stack have t:  $t = \text{frame } A t_0 s$ 
by (auto simp add: rel-alloc-def)

```

```

define t0' where  $t0' = \text{override-heap-on (ptr-span } p \cup P) t_0 t$ 

```

have *eq-htd*:

```

  override-on (htd s) (htd t0) (A – stack-free (htd s)) =
  override-on (htd s) (htd t0') ((ptr-span p  $\cup P \cup A$ ) – stack-free (htd s))
using disj-stack-free-s-p disj-stack-free-s-P disj-P-A disj-p-A t
by (auto simp add: t0'-def override-heap-on-def override-on-def fun-eq-iff frame-def)

```

have *eq-hmem*:

```

  override-on (hmem s) (hmem t0) (A  $\cup$  stack-free (htd s)) =
  override-on (hmem s) (hmem t0') ((ptr-span p  $\cup P \cup A$ )  $\cup$  stack-free (htd
s))
using disj-stack-free-s-p disj-stack-free-s-P disj-p-A disj-P-A t
by (auto simp add: t0'-def override-heap-on-def override-on-def fun-eq-iff frame-def)

```

```

from  $t$  have  $t \rightarrow t'$ :  $t = \text{frame } ((\text{ptr-span } p \cup P \cup A)) \ t' \ s$ 
  using  $\text{eq-htd eq-hmem}$ 
  apply ( $\text{simp add: frame-def}$ )
  by ( $\text{metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong}$ )

have  $\text{stack}'$ :  $\text{rel-alloc } \mathcal{S} \ M \ ((\text{ptr-span } p \cup P \cup A)) \ t' \ s \ t$ 
  using  $\text{stack disj-stack-free-s-P disj-stack-free-s-p t t'}$ 
  by ( $\text{auto simp add: rel-alloc-def}$ )

from  $\text{disj-stack-free-s-p disj-stack-free-s-P disj-stack-free-s-P stack}$ 
have  $\text{stack-free}'$ :  $\text{stack-free (htd } s) \cap (P \cup A) = \{\}$ 
  by ( $\text{auto simp add: rel-alloc-def}$ )

show  $?thesis$ 
  unfolding  $L2\text{-seq-def}$ 
  apply ( $\text{rule refines-bindE-right}$ )
    apply ( $\text{rule f-g}$ )
    apply  $\text{simp-all}$ 
  subgoal by ( $\text{simp add: rel-stack-def}$ )
  subgoal for  $v \ v' \ s' \ t'$ 
    apply ( $\text{frule rel-stack-unchanged-stack-free [OF stack', unfolded t'-def]}$ )
    apply ( $\text{frule rel-stack-unchanged-stack-free' [OF stack', unfolded t'-def]}$ )
    apply ( $\text{subst (asm) rel-stack-def}$ )
    apply  $\text{clarsimp}$ 
    apply ( $\text{subst (asm) rel-push-def}$ )
    apply  $\text{clarsimp}$ 
    apply ( $\text{rule refines-exec-IO-modify-heap-padding-step-right}$ )
    apply ( $\text{rule refines-exec-L2-return-right}$ )
    using  $\text{stack}$ 
    apply ( $\text{clarsimp simp add: rel-stack-def rel-push-def rel-alloc-def stack-free'}$ )
    unfolding  $t'\text{-def [symmetric]} \ t \text{ [symmetric]}$ 
    subgoal premises  $\text{prems for } w$ 
    proof –
      from  $\text{prems obtain}$ 
         $v' = (\text{h-val (hmem } s') \ p, \ w)$ 
         $R \ (\text{hmem } s') \ v \ w$  and
         $\text{sf-s' : stack-free (htd } s') = \text{stack-free (htd } s) \text{ and}$ 
         $\text{sf-t' : stack-free (htd (frame (ptr-span } p \cup P \cup A) \ t' \ s')) = \text{stack-free (htd}$ 
       $t) \text{ and}$ 
         $\text{stack-free (htd } s) \subseteq \mathcal{S}$ 
         $\text{stack-free (htd } s) \cap (\text{ptr-span } p \cup P \cup A) = \{\}$ 
         $\text{stack-free (htd } s) \cap M1 = \{\}$ 
         $t' = \text{frame (ptr-span } p \cup P \cup A) \ t' \ s'$ 
         $\text{equal-upto (M1 } \cup \text{stack-free (htd } s)) \ (\text{hmem } s') \ (\text{hmem } s) \text{ and}$ 
         $\text{htd-eq: htd } s' = \text{htd } s$  by  $\text{simp}$ 

    have  $\text{eq-typing}$ :
       $\text{override-on (htd } s') \ (\text{htd } t') \ (\text{ptr-span } p \cup P \cup A - \text{stack-free (htd } s')) =$ 

```

$\text{override-on (htd } s') \text{ (htd (override-heap-on } P \ t_0 \ t)) \ (P \cup A - \text{stack-free (htd } s'))$
using *disj-P-A disj-p-A*
by (*auto simp add: override-on-def fun-eq-iff t0'-def htd-eq t htd-frame override-heap-on-def*)

have *eq-heap: heap-update-padding p (h-val (hmem s') p)*
 $\text{(heap-list (hmem } s') \text{ (size-of TYPE('a)) (ptr-val p))}$
 $\text{(override-on (hmem } s') \text{ (hmem } t_0') \text{ (ptr-span } p \cup P \cup A \cup \text{stack-free (htd } s'))} =$
 $\text{override-on (hmem } s') \text{ (hmem (override-heap-on } P \ t_0 \ t)) \ (P \cup A \cup \text{stack-free (htd } s'))$
using *disj-P-A disj-p-A*
apply (*simp add: override-on-def override-heap-on-def fun-eq-iff t0'-def*)
by (*smt (verit, best) disj-p-P disj-stack-free-s-p h-val-def heap-list-length heap-update-list-id' heap-update-list-value heap-update-padding-def hmem-htd-upd max-size orthD2 sf-s' to-bytes-heap-list-id*)

show *hmem-upd*
 $\text{(heap-update-padding } p \text{ (h-val (hmem } s') \ p)$
 $\text{(heap-list (hmem } s') \text{ (size-of TYPE('a)) (ptr-val p))}$
 $\text{(frame (ptr-span } p \cup P \cup A) \ t_0' \ s') =$
 $\text{frame (} P \cup A \text{) (override-heap-on } P \ t_0 \ t) \ s'$
apply (*simp add: frame-def comp-def*)
using *eq-typing eq-heap*
by (*metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong*)

qed
done
done

qed

lemma *refines-rel-stack-pop-heap-no-exit-guarded:*

fixes *p:: 'a::xmem-type ptr*

assumes *stack: rel-alloc S M A t₀ s t*

assumes *f-g: refines f (L2-seq (L2-guard (λ-. G)) (λ-. g)) (λ-. g)) s t*

$\text{(rel-stack } S \ M1 \text{ (ptr-span } p \cup P \cup A) \ s$

$\text{(override-heap-on (ptr-span } p \cup P) \ t_0 \ t)$

$\text{(rel-xval-stack (rel-exit (λ- - . False)) (rel-push } p \ R)))$)

assumes *disj-P-A: G ⇒ P ∩ A = {}*

assumes *disj-p-A: G ⇒ ptr-span p ∩ A = {}*

assumes *disj-p-P: G ⇒ ptr-span p ∩ P = {}*

assumes *disj-stack-free-s-p: G ⇒ ptr-span p ∩ stack-free (htd s) = {}*

assumes *disj-stack-free-s-P: G ⇒ P ∩ stack-free (htd s) = {}*

shows *refines*

f

$\text{(L2-seq (L2-seq (L2-guard (λ-. G)) (λ-. g)) (λx. (L2-seq (IO-modify-heap-paddingE$
 $p \text{ (λ-. (fst } x))) \text{ (λ-. L2-return (snd } x) \ ns)))) \ s \ t$

$\text{(rel-stack } S \ M1 \text{ (} P \cup A \text{) } s \text{ (override-heap-on } P \ t_0 \ t) \text{ (rel-xval-stack (rel-exit (λ-$

```

- -. False)) R))
proof -
{
  assume grad: G
  have ?thesis
    by (rule refines-rel-stack-pop-heap-no-exit [OF stack f-g disj-P-A [OF grad]
disj-p-A [OF grad] disj-p-P [OF grad]
disj-stack-free-s-p [OF grad] disj-stack-free-s-P [OF grad]])
} then show ?thesis
  by (rule refines-L2-guard-right'')
qed

```

lemma *refines-rel-stack-pop-heap-no-exit-singleton*:

```

fixes p: 'a::xmem-type ptr
assumes stack: rel-alloc S M A t0 s t
assumes f-g: refines f g s t
  (rel-stack S M1 (ptr-span p ∪ P ∪ A) s
   (override-heap-on (ptr-span p ∪ P) t0 t)
   (rel-xval-stack (rel-exit (λ-. -. False)) (rel-singleton-stack p)))
assumes disj-P-A: P ∩ A = {}
assumes disj-p-A: ptr-span p ∩ A = {}
assumes disj-p-P: ptr-span p ∩ P = {}
assumes disj-stack-free-s-p: ptr-span p ∩ stack-free (htd s) = {}
assumes disj-stack-free-s-P: P ∩ stack-free (htd s) = {}
shows refines
  f
  (L2-seq g (λx. (IO-modify-heap-paddingE p (λ-. x))) s t
   (rel-stack S M1 (P ∪ A) s (override-heap-on P t0 t) (rel-xval-stack (rel-exit (λ-
- -. False)) (λ-. (=)))))

```

```

proof -
from stack have sf-s-t: stack-free (htd s) ⊆ stack-free (htd t)
  using rel-alloc-def stack-free-htd-frame by auto

```

```

from stack have t: t = frame A t0 s
  by (auto simp add: rel-alloc-def)

```

```

define t0' where t0' = override-heap-on (ptr-span p ∪ P) t0 t

```

```

have eq-htd:
  override-on (htd s) (htd t0) (A - stack-free (htd s)) =
  override-on (htd s) (htd t0') ((ptr-span p ∪ P ∪ A) - stack-free (htd s))
  using disj-stack-free-s-p disj-stack-free-s-P disj-P-A disj-p-A t
  by (auto simp add: t0'-def override-heap-on-def override-on-def fun-eq-iff frame-def)

```

```

have eq-hmem:
  override-on (hmem s) (hmem t0) (A ∪ stack-free (htd s)) =
  override-on (hmem s) (hmem t0') ((ptr-span p ∪ P ∪ A) ∪ stack-free (htd
s))
  using disj-stack-free-s-p disj-stack-free-s-P disj-p-A disj-P-A t

```



```

by (auto simp add: t0'-def override-heap-on-def override-on-def fun-eq-iff frame-def)

from t have t-t0': t = frame ((ptr-span p ∪ P ∪ A)) t0' s
  using eq-htd eq-hmem
  apply (simp add: frame-def)
  by (metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong)

have stack': rel-alloc S M ((ptr-span p ∪ P ∪ A)) t0' s t
  using stack disj-stack-free-s-P disj-stack-free-s-p t t-t0'
  by (auto simp add: rel-alloc-def)

from disj-stack-free-s-p disj-stack-free-s-P disj-stack-free-s-P stack
have stack-free': stack-free (htd s) ∩ (P ∪ A) = {}
  by (auto simp add: rel-alloc-def)

show ?thesis
  unfolding L2-seq-def
  apply (rule refines-bindE-right)
  apply (rule f-g)
  apply simp-all
  subgoal by (simp add: rel-stack-def)
  subgoal for v' s' t'
    apply (frule rel-stack-unchanged-stack-free [OF stack', unfolded t0'-def])
    apply (frule rel-stack-unchanged-stack-free' [OF stack', unfolded t0'-def])
    apply (subst (asm) rel-stack-def)
    apply clarsimp
    apply (subst (asm) rel-singleton-stack-def)
    apply clarsimp
    apply (rule refines-exec-IO-modify-heap-padding-single-step-right)
    using stack
    apply (clarsimp simp add: rel-stack-def rel-push-def rel-alloc-def stack-free')
    unfolding t0'-def [symmetric] t [symmetric]
    subgoal premises prems
    proof –
      from prems obtain
        v' = h-val (hmem s') p and
        sf-s': stack-free (htd s') = stack-free (htd s) and
        sf-t': stack-free (htd (frame (ptr-span p ∪ P ∪ A) t0' s')) = stack-free (htd
t) and
        stack-free (htd s) ⊆ S
        stack-free (htd s) ∩ (ptr-span p ∪ P ∪ A) = {}
        stack-free (htd s) ∩ M1 = {}
        t' = frame (ptr-span p ∪ P ∪ A) t0' s'
        equal-upto (M1 ∪ stack-free (htd s)) (hmem s') (hmem s) and
        htd-eq: htd s' = htd s by simp

      have eq-typing:
        override-on (htd s') (htd t0') (ptr-span p ∪ P ∪ A - stack-free (htd s')) =
          override-on (htd s') (htd (override-heap-on P t0 t)) (P ∪ A -

```

```

stack-free (htd s'))
  using disj-P-A disj-p-A
  by (auto simp add: override-on-def fun-eq-iff t0'-def htd-eq t htd-frame
override-heap-on-def)

```

```

  have eq-heap: heap-update-padding p (h-val (hmem s') p)
    (heap-list (hmem s') (size-of TYPE('a)) (ptr-val p))
    (override-on (hmem s') (hmem t0') (ptr-span p  $\cup$  P  $\cup$  A  $\cup$  stack-free
(htd s'))) =
    override-on (hmem s') (hmem (override-heap-on P t0 t)) (P  $\cup$  A  $\cup$ 
stack-free (htd s'))

```

```

  using disj-P-A disj-p-A
  apply (simp add: override-on-def override-heap-on-def fun-eq-iff t0'-def)
  by (smt (verit, best) disj-p-P disj-stack-free-s-p h-val-def heap-list-length
heap-update-list-id' heap-update-list-value heap-update-padding-def
hmem-htd-upd max-size orthD2 sf-s' to-bytes-heap-list-id)

```

```

show hmem-upd
  (heap-update-padding p (h-val (hmem s') p)
  (heap-list (hmem s') (size-of TYPE('a)) (ptr-val p)))
  (frame (ptr-span p  $\cup$  P  $\cup$  A) t0' s') =
  frame (P  $\cup$  A) (override-heap-on P t0 t) s'
  apply (simp add: frame-def comp-def)
  using eq-typing eq-heap
  by (metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong)

```

```

qed
done
done

```

```

qed

```

lemma *refines-rel-stack-pop-heap-no-exit-singleton-guarded:*

```

fixes p:: 'a::xmem-type ptr
assumes stack: rel-alloc S M A t0 s t
assumes f-g: refines f (L2-seq (L2-guard ( $\lambda$ -. G)) ( $\lambda$ -. g)) s t
  (rel-stack S M1 (ptr-span p  $\cup$  P  $\cup$  A) s
  (override-heap-on (ptr-span p  $\cup$  P) t0 t)
  (rel-xval-stack (rel-exit ( $\lambda$ -. -. False)) (rel-singleton-stack p)))
assumes disj-P-A: G  $\implies$  P  $\cap$  A = {}
assumes disj-p-A: G  $\implies$  ptr-span p  $\cap$  A = {}
assumes disj-p-P: G  $\implies$  ptr-span p  $\cap$  P = {}
assumes disj-stack-free-s-p: G  $\implies$  ptr-span p  $\cap$  stack-free (htd s) = {}
assumes disj-stack-free-s-P: G  $\implies$  P  $\cap$  stack-free (htd s) = {}
shows refines

```

```

  f
  (L2-seq (L2-seq (L2-guard ( $\lambda$ -. G)) ( $\lambda$ -. g)) ( $\lambda$ x. (IO-modify-heap-paddingE p
( $\lambda$ -. x)))) s t
  (rel-stack S M1 (P  $\cup$  A) s (override-heap-on P t0 t) (rel-xval-stack (rel-exit ( $\lambda$ -. -. False)) ( $\lambda$ -. (=))))

```

```

proof –
  {
    assume grd: G
    have ?thesis
      by (rule refines-rel-stack-pop-heap-no-exit-singleton [OF stack f-g disj-P-A [OF
grd] disj-p-A [OF grd] disj-p-P [OF grd]
          disj-stack-free-s-p [OF grd] disj-stack-free-s-P [OF grd]])
    } then show ?thesis
      by (rule refines-L2-guard-right'')
qed

```

```

lemma refines-rel-stack-shuffle-both:
  assumes refines f g s t (rel-stack S M A s t0 (rel-xval-stack (rel-exit L) R))
  assumes  $\bigwedge h e e'. L h e e' \implies L' h e$  (shuffle-exit e')
  assumes  $\bigwedge h v v'. R h v v' \implies R' h v$  (shuffle v')
  shows refines
    f
    (L2-seq
      (L2-catch g ( $\lambda e. L2-throw$  (shuffle-exit e) ns-exit))
      ( $\lambda x. L2-return$  (shuffle x) ns))
    s t
    (rel-stack S M A s t0 (rel-xval-stack (rel-exit L') R'))
  using assms
  apply (clarsimp simp add: reaches-bind reaches-catch succeeds-bind succeeds-catch
L2-defs refines-def-old L2-VARS-def
      rel-stack-def rel-alloc-def rel-xval-stack-def rel-exit-def rel-xval.simps default-option-def
Exn-def[symmetric]
      split: xval-splits prod.splits)
  subgoal for r s'
  apply (cases r)
  subgoal
    apply (clarsimp simp add: default-option-def Exn-def[symmetric])
    subgoal for y
      apply (erule-tac x=Exn y in allE)
      apply (erule-tac x=s' in allE)
      apply clarsimp
      by (smt (verit) Exn-def Exn-eq-Exn Result-neq-Exn case-exception-or-result-Exn)
    done
  subgoal for v
    apply (erule-tac x=Result v in allE)
    apply (erule-tac x=s' in allE)
    apply clarsimp
    by (smt (z3) Result-eq-Result Result-neq-Exn case-exception-or-result-Result)
  done
done

```

```

lemma refines-rel-stack-shuffle-no-exit:
  assumes refines f g s t (rel-stack S M A s t0 (rel-xval-stack (rel-exit ( $\lambda - - .$ 

```

```

False)) R))
assumes  $\bigwedge h v v'. R h v v' \implies R' h v (\text{shuffle } v')$ 
shows refines
  f
  (L2-seq
    g
    ( $\lambda x. \text{L2-return } (\text{shuffle } x) ns$ )
    s t
    (rel-stack  $\mathcal{S} M A s t_0$  (rel-xval-stack (rel-exit ( $\lambda - - . \text{False}$ ))  $R'$ ))
using assms
apply (clarsimp simp add: succeeds-bind reaches-bind L2-defs refines-def-old
L2-VARS-def
  rel-stack-def rel-alloc-def rel-sum-stack-def rel-exit-def rel-sum.simps split:
sum.splits prod.splits)
subgoal for  $r s'$ 
apply (cases r)
subgoal
apply (clarsimp simp add: default-option-def Exn-def[symmetric])
subgoal for  $y$ 

  apply (erule-tac x=Exn y in allE)
  apply (erule-tac x=s' in allE)
  apply auto
  done
done
subgoal for  $v$ 
  apply (erule-tac x=Result v in allE)
  apply (erule-tac x=s' in allE)
  apply clarsimp
  by (smt (z3) Result-eq-Result Result-neq-Exn case-exception-or-result-Result)
done
done

```

```

lemma L2-call-rel-stack-bare':
assumes f: refines f g s t
  (rel-stack  $\mathcal{S} M1 A s t_0$  (rel-xval-stack  $L R$ ))
shows refines
  f
  (L2-call  $g (\lambda x. x) ns'$ )
  s t
  (rel-stack  $\mathcal{S} M1 A s t_0$  (rel-xval-stack  $L R$ ))
using assms
by (auto simp add: L2-call-def refines-def-old reaches-map-value)

```

```

lemma L2-call-rel-stack-bare-retype-unreachable-exit':
assumes f: refines f g s t
  (rel-stack  $\mathcal{S} M1 A s t_0$  (rel-xval-stack (rel-exit ( $\lambda - - . \text{False}$ ))  $R$ ))
shows refines
  f

```

$(L2\text{-call } g \text{ emb } ns')$
 $s \ t$
 $(rel\text{-stack } \mathcal{S} \ M1 \ A \ s \ t_0 \ (rel\text{-xval-stack } (rel\text{-exit } (\lambda - \ . \ False)) \ R))$
using *assms*
apply (*clarsimp simp add: L2-call-def reaches-map-value succeeds-bind reaches-bind*
L2-defs refines-def-old L2-VARS-def
rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps rel-exit-def)
by *fastforce*

lemma *L2-call-rel-stack-bare-retype-unreachable-exit''*:

assumes *f: refines f g s t*
 $(rel\text{-stack } \mathcal{S} \ M1 \ A \ s \ t_0 \ (rel\text{-xval-stack } (rel\text{-exit } (\lambda - \ . \ False)) \ R))$
shows *refines*
 f
 $(L2\text{-seq } (L2\text{-guard } (\lambda - . \ P)) \ (\lambda - .$
 $(L2\text{-seq } (L2\text{-catch } (L2\text{-call } g \text{ emb } ns)$
 $(\lambda x . \ L2\text{-seq } (liftE \ (return \ ())) \ (\lambda - . \ L2\text{-throw } x \ ns\text{-exit}))))))$
 $(\lambda v . \ L2\text{-return } v \ ns)))$
 $s \ t$
 $(rel\text{-stack } \mathcal{S} \ M1 \ A \ s \ t_0 \ (rel\text{-xval-stack } (rel\text{-exit } (\lambda - \ . \ False)) \ R))$
using *assms*
apply (*clarsimp simp add: L2-call-def reaches-map-value succeeds-catch reaches-catch*
succeeds-bind reaches-bind L2-defs refines-def-old L2-VARS-def
rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps rel-exit-def)
by *fastforce*

lemma *L2-call-rel-stack-bare-retype-unreachable-exit*:

assumes *f: refines f g s t*
 $(rel\text{-stack } \mathcal{S} \ M1 \ A \ s \ t_0 \ (rel\text{-xval-stack } (rel\text{-exit } (\lambda - \ . \ False)) \ R))$
shows *refines*
 f
 $(L2\text{-seq } (L2\text{-guard } (\lambda - . \ P)) \ (\lambda - .$
 $(L2\text{-seq } (L2\text{-catch } (L2\text{-call } g \ \text{undefined } ns)$
 $(\lambda x . \ L2\text{-seq } (liftE \ (return \ ())) \ (\lambda - . \ L2\text{-throw } x \ ns\text{-exit}))))))$
 $(\lambda v . \ L2\text{-return } v \ ns)))$
 $s \ t$
 $(rel\text{-stack } \mathcal{S} \ M1 \ A \ s \ t_0 \ (rel\text{-xval-stack } (rel\text{-exit } (\lambda - \ . \ False)) \ R))$
by (*rule L2-call-rel-stack-bare-retype-unreachable-exit'' [OF f]*)

lemma *L2-call-rel-stack-bare-retype-unreachable-exit-extend-modifies'*:

assumes *f: refines f g s t*
 $(rel\text{-stack } \mathcal{S} \ M1 \ A \ s \ t_0 \ (rel\text{-xval-stack } (rel\text{-exit } (\lambda - \ . \ False)) \ R))$
assumes $P \implies \text{stack-free } (htd \ s) \cap M2 = \{\}$
assumes $M1\text{-}M2: P \implies M1 \subseteq M2$
shows *refines*
 f
 $(L2\text{-seq } (L2\text{-guard } (\lambda - . \ P)) \ (\lambda - .$
 $(L2\text{-seq } (L2\text{-catch } (L2\text{-call } g \ \text{emb } ns)$

```

      (λx. L2-seq (liftE (return ())) (λ-. L2-throw x ns-exit))))
    (λv. L2-return v ns)))
  s t
  (rel-stack S M2 A s t0 (rel-xval-stack (rel-exit (λ- - -. False)) R))
using assms
apply (clarsimp simp add: L2-call-def reaches-map-value succeeds-catch reaches-catch
succeeds-bind reaches-bind L2-defs refines-def-old L2-VARS-def
rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps rel-exit-def)
subgoal for r s'
apply (cases r)
subgoal
apply (clarsimp simp add: Exn-def [symmetric] default-option-def)
by (metis Result-neq-Exn)
subgoal for v
apply (erule-tac x=Result v in allE)
apply (erule-tac x=s' in allE)
apply (clarsimp simp add: Exn-def [symmetric] default-option-def)
by (smt (z3) case-xval-simps(2) equal-upto-mono map-exn-simps(2)
order-eq-refl sup-mono)
done
done

```

lemma *L2-call-rel-stack-bare-retype-unreachable-exit-extend-modifies:*

```

assumes f: refines f g s t
      (rel-stack S M1 A s t0 (rel-xval-stack (rel-exit (λ- - -. False)) R))
assumes sf: P ⇒ stack-free (htd s) ∩ M2 = {}
assumes M1-M2: P ⇒ M1 ⊆ M2
shows refines
  f
  (L2-seq (L2-guard (λ-. P)) (λ-.
    (L2-seq (L2-catch (L2-call g undefined ns)
      (λx. L2-seq (liftE (return ())) (λ-. L2-throw x ns-exit))))
    (λv. L2-return v ns)))
  s t
  (rel-stack S M2 A s t0 (rel-xval-stack (rel-exit (λ- - -. False)) R))
by (rule L2-call-rel-stack-bare-retype-unreachable-exit-extend-modifies' [OF f sf
M1-M2])

```

lemma *L2-call-rel-stack-bare:*

```

assumes f: refines f g s t
      (rel-stack S M1 A s t0 (rel-xval-stack (rel-exit L) R))
shows refines
  f
  (L2-seq (L2-guard (λ-. P)) (λ-.
    (L2-seq (L2-catch (L2-call g (λx. x) ns)
      (λx. L2-seq (liftE (return ())) (λ-. L2-throw x ns-exit))))
    (λv. L2-return v ns)))
  s t

```

```

      (rel-stack S M1 A s t0 (rel-xval-stack (rel-exit L) R))
using assms
apply (clarsimp simp add: L2-call-def reaches-map-value succeeds-catch reaches-catch
succeeds-bind reaches-bind L2-defs refines-def-old L2-VARS-def
rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps rel-exit-def)
subgoal for r s'
  apply (cases r)
  subgoal
    apply (clarsimp simp add: default-option-def Exn-def[symmetric])
    subgoal for y

      apply (erule-tac x=Exn y in allE)
      apply (erule-tac x=s' in allE)
      apply fastforce
      done
    done
  subgoal for v
    apply (erule-tac x=Result v in allE)
    apply (erule-tac x=s' in allE)
    apply fastforce
    done
  done
done

```

lemma *L2-call-rel-stack-bare-extend-modifies:*

```

assumes f: refines f g s t
      (rel-stack S M1 A s t0 (rel-xval-stack (rel-exit L) R))
assumes sf: P ⇒ stack-free (htd s) ∩ M2 = {}
assumes M1-M2: P ⇒ M1 ⊆ M2
shows refines
  f
  (L2-seq (L2-guard (λ-. P)) (λ-. (L2-seq (L2-catch (L2-call g (λx. x) ns) (λx. L2-seq (liftE (return ())) (λ-. L2-throw x ns-exit)))) (λv. L2-return v ns)))
  s t
  (rel-stack S M2 A s t0 (rel-xval-stack (rel-exit L) R))
unfolding L2-defs L2-VARS-def liftE-return
apply (clarsimp simp add: f-catch-throw L2-call-def refines-bind-guard-right-iff)
apply (rule refines-weaken [OF f])
using sf M1-M2
apply (clarsimp simp add: rel-stack-def rel-alloc-def)
by (meson Un-subset-iff Un-upper2 equal-upto-mono sup.coboundedI1)

```

lemma *refines-rel-stack-extend-modifies:*

```

assumes f: refines f (L2-seq (L2-guard (λ-. P)) (λ-. g)) s t (rel-stack S M1 A s
t0 Q)
assumes sf: P ⇒ M2 ∩ stack-free (htd s) = {}
assumes M1-M2: P ⇒ M1 ⊆ M2

```

shows *refines f (L2-seq (L2-guard ($\lambda\cdot$. P)) ($\lambda\cdot$. g)) s t (rel-stack \mathcal{S} M2 A s t₀ Q)*
using *assms*
apply (*clarsimp simp add: L2-call-def reaches-map-value succeeds-catch reaches-catch succeeds-bind reaches-bind L2-defs refines-def-old L2-VARS-def rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps rel-exit-def*)
by (*smt (verit, best) dual-order.refl equal-upto-mono inf-aci(1) sup.mono*)

lemma *refines-rel-sum-stack-pop-exit'*:

assumes *f-g:*
refines f g s t
(rel-stack \mathcal{S} M A s t₀
(rel-xval-stack
(rel-exit (rel-push p L))
R))

shows

refines f (L2-catch g ($\lambda(x, p)$. L2-throw p ns)) s t
(rel-stack \mathcal{S} M A s t₀
(rel-xval-stack
(rel-exit L)
R))

apply (*rule refines-L2-catch-right*)

apply (*rule f-g*)

apply *simp-all*

apply (*auto simp add: L2-call-def reaches-map-value succeeds-catch reaches-catch*

succeeds-bind reaches-bind L2-defs refines-def-old default-option-def Exn-def
[symmetric] L2-VARS-def
rel-push-def
rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps rel-exit-def)
done

lemma *refines-rel-sum-stack-pop-exit:*

assumes *f-g:*
refines f g s t
(rel-stack \mathcal{S} M A s t₀
(rel-xval-stack
(rel-exit (rel-push p L))
R))

shows

refines f (L2-catch g (λx . L2-throw (snd x) ns)) s t
(rel-stack \mathcal{S} M A s t₀
(rel-xval-stack
(rel-exit L)
R))

apply (*rule refines-L2-catch-right*)

apply (*rule f-g*)

apply *simp-all*

apply (*auto simp add: L2-call-def reaches-map-value succeeds-catch reaches-catch*


```

    succeeds-bind reaches-bind L2-defs refines-def-old default-option-def Exn-def
[symmetric] L2-VARS-def
    rel-push-def
    rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps rel-exit-def)
done

```

end

lemma *L2-call-fuse-handlers1*:

```

L2-seq
  (L2-catch
    (L2-seq
      (L2-catch g (λx. L2-seq (liftE (h1 x)) (λ-. L2-throw (prj1 x) ns1)))
      (λx. liftE (g1 x)))
      (λx. L2-seq (liftE (h2 x)) (λ-. L2-throw (prj2 x) ns2)))
      (λx. liftE (g2 x)) =
    (L2-seq
      (L2-catch g (λx. L2-seq (L2-seq (liftE (h1 x)) (λ-. liftE (h2 (prj1 x)))) (λ-.
L2-throw (prj2 (prj1 x) ns2))))
      (λx. L2-seq (liftE (g1 x)) (λx. liftE (g2 x))))
unfolding L2-defs
apply (rule spec-monad-eqI)
apply (auto simp add: runs-to-iff)
done

```

lemma *L2-call-fuse-handlers2*:

```

(L2-catch
  (L2-seq
    (L2-catch g (λx. L2-seq (liftE (h1 x)) (λ-. L2-throw (prj1 x) ns1)))
    (λx. liftE (g1 x)))
    (λx. L2-throw (prj2 x) ns2))
=
L2-seq
  (L2-catch g (λx. L2-seq (liftE (h1 x)) (λ-. L2-throw (prj2 (prj1 x) ns2)))
  (λx. (liftE (g1 x))))
unfolding L2-defs
apply (rule spec-monad-eqI)
apply (clarsimp simp add: runs-to-iff)
apply (auto simp add: runs-to-def-old default-option-def Exn-def split: xval-splits)
done

```

lemma *L2-call-triv-conv*: $L2\text{-call } f (\lambda x. x) ns = f$

```

unfolding L2-call-def
apply (rule spec-monad-eqI)
apply (auto simp add: runs-to-iff)

```

done

lemma *L2-catch-L2-call-conv*:

L2-catch

$(L2\text{-call } f \ (\lambda e. \ e) \ ns)$

$(\lambda e. \ L2\text{-throw } (emb \ e) \ ns') =$

$L2\text{-call } f \ (ETA\text{-TUPLED } emb) \ ns$

apply (*simp add: L2-call-triv-conv ETA-TUPLED-def*)

apply (*simp add: L2-call-def map-exn-catch-conv L2-catch-def L2-throw-def comp-def*)

done

lemma *L2-seq-return-conv*:

$L2\text{-seq } (liftE \ (return \ x)) \ (\lambda x. \ liftE \ (return \ (f \ x))) = liftE \ (return \ (f \ x))$

unfolding *L2-defs*

apply (*rule spec-monad-eqI*)

apply (*auto simp add: runs-to-iff*)

done

lemma *L2-seq-return-unused-conv*:

$L2\text{-seq } (liftE \ (return \ x)) \ (\lambda-. \ g) = g$

unfolding *L2-defs*

apply (*rule spec-monad-eqI*)

apply (*auto simp add: runs-to-iff*)

done

lemma *L2-seq-L2-return-unused-conv*:

$L2\text{-seq } (L2\text{-return } x \ ns) \ (\lambda-. \ g) = g$

unfolding *L2-defs L2-VARS-def*

apply (*rule spec-monad-eqI*)

apply (*auto simp add: runs-to-iff*)

done

lemma *L2-seq-return-id-conv*:

$L2\text{-seq } f \ (\lambda x. \ liftE \ (return \ x)) = f$

unfolding *L2-defs*

apply (*rule spec-monad-eqI*)

apply (*auto simp add: runs-to-iff*)

done

lemma *L2-seq-L2-return-id-conv*:

$L2\text{-seq } f \ (\lambda x. \ L2\text{-return } x \ ns) = f$

unfolding *L2-defs L2-VARS-def*

apply (*rule spec-monad-eqI*)

apply (*auto simp add: runs-to-iff*)

done

lemma *L2-catch-L2-guard-out-conv*: $L2\text{-catch } (L2\text{-seq } (L2\text{-guard } P) \ (\lambda-. \ X)) \ Y =$

L2-seq (L2-guard P) (λ-. L2-catch X Y)
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *L2-seq-guard-out-conv:*
L2-seq (L2-seq (L2-guard P) (λ-. X)) Y = (L2-seq (L2-guard P) (λ-. L2-seq X Y))
by (*simp add: L2-seq-assoc*)

lemma *L2-seq-L2-catch-assoc:*
L2-seq (L2-seq (L2-catch X Y) Z1) Z2 = L2-seq (L2-catch X Y) (λx. L2-seq (Z1 x) Z2)
by (*rule L2-seq-assoc*)

lemma *L2-catch-throw-id: L2-catch f (λx. L2-throw x ns) = f*
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add: runs-to-iff*)
apply (*clarsimp simp add: runs-to-def-old*)
by (*metis Exn-def default-option-def exception-or-result-cases not-None-eq*)

lemma *L2-catch-call-throw:*
(L2-catch
(L2-call f emb ns)
(λx. L2-throw (g x) ns-exit)) =
L2-call f (ETA-TUPLED (λx. (g (emb x)))) ns
unfolding *L2-defs L2-call-def ETA-TUPLED-def*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add: runs-to-iff*)
apply (*auto simp add: runs-to-def-old map-exn-def split: xval-splits*)
done

lemma *liftE-bind-L2-seq: (liftE (A >>= B)) = L2-seq (liftE A) (ETA-TUPLED*
(λx. liftE (B x)))
unfolding *L2-defs L2-call-def ETA-TUPLED-def*
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *liftE-L2-seq: L2-seq (liftE A) (λx. liftE (B x)) = (liftE (A >>= B))*
unfolding *L2-defs*
by (*simp add: liftE-bind*)

lemma *liftE-return-L2-gets-conv: liftE (return x) = L2-gets (λ-. x) []*

by (*simp add: return-L2-gets-conv*)

lemmas *L2-call-canonical-convs* =

L2-seq-return-conv
L2-call-fuse-handlers2
L2-call-fuse-handlers1
L2-guard-join-nested
liftE-L2-seq

L2-seq-L2-catch-assoc
L2-catch-L2-guard-out-conv
L2-seq-guard-out-conv

bind-assoc

lemmas *L2-call-pre-final-convs* =

L2-seq-return-unused-conv
L2-seq-L2-return-unused-conv
L2-seq-return-id-conv
L2-seq-L2-return-id-conv
L2-catch-call-throw

lemmas *L2-call-final-convs* =

L2-seq-return-unused-conv
L2-seq-L2-return-unused-conv
L2-seq-return-id-conv
L2-seq-L2-return-id-conv
L2-catch-L2-call-conv
guard-triv
liftE-bind-L2-seq
L2-gets-unbound
L2-catch-throw-id
L2-return-L2-gets-conv
liftE-return-L2-gets-conv

L2-seq-L2-catch-assoc
L2-catch-L2-guard-out-conv
L2-seq-guard-out-conv

lemma *L2-call-ETA-TUPLED1*: $L2\text{-seq } (L2\text{-catch } g \ h) \ g1 \equiv L2\text{-seq } (L2\text{-catch } g \ (ETA\text{-TUPLED } h)) \ (ETA\text{-TUPLED } g1)$

unfolding *ETA-TUPLED-def* **by** *simp*

lemma *L2-call-ETA-TUPLED2*: $L2\text{-seq } (L2\text{-call } g \ emb \ ns) \ g1 \equiv L2\text{-seq } (L2\text{-call } g \ emb \ ns) \ (ETA\text{-TUPLED } g1)$

unfolding *ETA-TUPLED-def* **by** *simp*

lemma *distinct-sets-consI*: $distinct\text{-sets } ps \implies distinct\text{-sets } (p\#ps) \longleftrightarrow p \cap \bigcup$

(set ps) = {}
 by (simp)

lemma disjoint-subset-simps:

ASSUMPTION $(A \cap B = \{\}) \implies A' \subseteq A \implies B \cap A' = \{\} \longleftrightarrow \text{True}$
 ASSUMPTION $(B \cap A = \{\}) \implies A' \subseteq A \implies B \cap A' = \{\} \longleftrightarrow \text{True}$
 ASSUMPTION $(A \cap B = \{\}) \implies B' \subseteq B \implies B' \cap A = \{\} \longleftrightarrow \text{True}$
 ASSUMPTION $(B \cap A = \{\}) \implies B' \subseteq B \implies B' \cap A = \{\} \longleftrightarrow \text{True}$
 by (auto simp add: ASSUMPTION-def)

lemma disjoint-subset-simps':

$(B \cap A = \{\}) \implies A' \subseteq A \implies B \cap A' = \{\} \longleftrightarrow \text{True}$
 $(A \cap B = \{\}) \implies B' \subseteq B \implies B' \cap A = \{\} \longleftrightarrow \text{True}$
 $(B \cap A = \{\}) \implies B' \subseteq B \implies B' \cap A = \{\} \longleftrightarrow \text{True}$
 $(A \cap B = \{\}) \implies A' \subseteq A \implies B \cap A' = \{\} \longleftrightarrow \text{True}$
 by auto

lemma field-lvalue-ptr-span-trans:

fixes p: 'a::mem-type ptr
 assumes field-lookup (typ-info-t TYPE('a)) f 0 = Some (t, n)
 assumes export-uinfo t = typ-uinfo-t (TYPE('b'))
 assumes ptr-span p \subseteq A
 shows $\{\&(p \rightarrow f) .. + \text{size-of TYPE('b::c-type)}\} \subseteq A$
 using field-tag-sub assms export-size-of by fastforce

lemma field-lvalue-ptr-span-root-contained:

fixes p: 'a::mem-type ptr
 assumes field-lookup (typ-info-t TYPE('a)) f 0 = Some (t, n)
 assumes export-uinfo t = typ-uinfo-t (TYPE('b'))
 shows $\{\&(p \rightarrow f) .. + \text{size-of TYPE('b::c-type)}\} \subseteq \text{ptr-span } p$
 using field-tag-sub assms export-size-of by fastforce

lemma field-lvalue-disjoint-fields-same-root:

fixes p: 'a::mem-type ptr
 assumes f: field-lookup (typ-info-t TYPE('a)) f 0 = Some (T1, n)
 assumes exp1: export-uinfo T1 = typ-uinfo-t (TYPE('b::c-type'))
 assumes g: field-lookup (typ-info-t TYPE('a')) g 0 = Some (T2, m)
 assumes exp2: export-uinfo T2 = typ-uinfo-t (TYPE('c::c-type'))
 assumes prfx1: \neg prefix f g
 assumes prfx2: \neg prefix g f
 shows $\{\&(p \rightarrow f) .. + \text{size-of TYPE('b::c-type)}\} \cap \{\&(p \rightarrow g) .. + \text{size-of TYPE('c::c-type)}\} = \{\}$
 using field-lookup-non-prefix-disj [OF - f g prfx2 prfx1]
 field-lookup-offset-eq [OF f] field-lookup-offset-eq [OF g] exp1 exp2
 export-size-of [OF exp1] export-size-of [OF exp2]
 apply simp
 by (smt (verit, best) add commute add-diff-cancel-left'
 diff-is-0-eq dual-order.trans export-size-of f field-lookup-offset-size
 field-lookup-wf-size-desc-gt field-lvalue-def fold-typ-uinfo-t g intvl-fields-disj)

linorder-not-less max-size nat-less-le unat-of-nat-field-offset wf-size-desc)

lemma *ptr-coerce-index-array-ptr-index-conv*:

ptr-coerce $p +_p$ *uint* $i = \text{array-ptr-index } p \text{ False } (\text{nat } (\text{uint } i))$
by (*auto simp add: array-ptr-index-def*)

lemma *ptr-coerce-index-array-ptr-index-numeral-conv*:

ptr-coerce $p +_p$ (*numeral* i) = *array-ptr-index* p *False* (*numeral* i)
by (*auto simp add: array-ptr-index-def*)

lemma *ptr-coerce-index-array-ptr-index-numeral-conv'*:

fixes p : ($'a :: \text{c-type}$)[$'b :: \text{finite}$] ptr
assumes *sz-eq*: *size-of* $\text{TYPE}('c::\text{c-type}) = \text{size-of } \text{TYPE}('a)$
shows $\text{PTR-COERCE}('a['b] \rightarrow 'c::\text{c-type}) p +_p (\text{numeral } n) = \text{PTR-COERCE}('a$
 $\rightarrow 'c) (\text{array-ptr-index } p \text{ False } (\text{numeral } n))$
using *assms*
by (*simp add: array-ptr-index-simps(1) c-type-class.ptr-add-def*)

lemma *ptr-coerce-index-array-ptr-index-numeral-conv''*:

fixes p : ($'a :: \text{c-type}$)[$'b :: \text{finite}$] ptr
assumes *sz-eq*: *size-of* $\text{TYPE}('c::\text{c-type}) = \text{size-of } \text{TYPE}('a)$
assumes *bound*: *numeral* $n < \text{CARD}('b)$
shows $\text{PTR-COERCE}('a['b] \rightarrow 'c::\text{c-type}) p +_p (\text{numeral } n) = \text{PTR}('c) \ \&(p \rightarrow [\text{replicate}$
 $(\text{numeral } n) \ \text{CHR } "1"])$
proof –
have *eq1*: $\text{PTR}('c) \ \&(p \rightarrow [\text{replicate } (\text{numeral } n) \ \text{CHR } "1"])$ = $\text{PTR-COERCE}('a$
 $\rightarrow 'c) (\text{PTR}('a) \ \&(p \rightarrow [\text{replicate } (\text{numeral } n) \ \text{CHR } "1"])$
by *simp*
note *replicate-numeral* [*simp del*]
show *?thesis*
apply (*subst eq1*)
apply (*subst array-ptr-index-field-lvalue-conv* [*symmetric, OF bound*])
apply (*rule ptr-coerce-index-array-ptr-index-numeral-conv'* [*OF sz-eq*])
done

qed

lemma *ptr-coerce-index-array-ptr-index-0-conv*:

ptr-coerce $p +_p$ $0 = \text{array-ptr-index } p \text{ False } 0$
by (*auto simp add: array-ptr-index-def*)

lemma *ptr-coerce-index-array-ptr-index-1-conv*:

ptr-coerce $p +_p$ $1 = \text{array-ptr-index } p \text{ False } 1$
by (*auto simp add: array-ptr-index-def*)

lemma *ptr-coerce-index-array-ptr-index-sint-conv*:

$0 \leq s \ i \implies \text{ptr-coerce } p +_p \ \text{sint } i = \text{array-ptr-index } p \text{ False } (\text{nat } (\text{sint } i))$
by (*auto simp add: array-ptr-index-def word-sle-def*)

lemma *heap-access-Array-element''*:

fixes $p :: ('a::\text{mem-type}['b::\text{finite}]) \text{ ptr}$
assumes $\text{less}: n < \text{CARD}('b)$
shows
 $\text{index } (h\text{-val } hp \text{ } p) \ n$
 $\quad = h\text{-val } hp \text{ } (\text{array-ptr-index } p \ \text{False } n)$
using $\text{heap-access-Array-element}' \ \text{less}$
by auto

lemma $\text{index-fupdate-split}: i < \text{CARD}('n) \implies j < \text{CARD}('n) \implies$
 $\text{index } (fupdate \ i \ f \ (x::'a['n::\text{finite}])) \ j = (\text{if } i \neq j \ \text{then } (x.[j]) \ \text{else } f \ (\text{index } x \ i))$
by $(\text{cases } i=j) \ (\text{auto simp add: arr-fupdate-same arr-fupdate-other})$

lemma $\text{root-disjoint-field-lvalue-disjoint1}:$
fixes $p::'a::\text{mem-type } \text{ ptr}$
assumes $\text{field-lookup}: \text{field-lookup } (typ\text{-info-t } \text{TYPE}('a)) \ \text{path } 0 = \text{Some } (t, n)$
assumes $\text{match}: \text{export-uinfo } t = \text{typ-uinfo-t } \text{TYPE}('f::\text{c-type})$
assumes $\text{ptr-span } p \cap A = \{\}$
shows $A \cap \{\&(p \rightarrow \text{path})..+\text{size-of } \text{TYPE}('f)\} = \{\}$
using assms
by $(\text{simp add: disjoint-field-lvalue-propagation-right export-size-of inter-commute})$

lemma $\text{root-disjoint-field-lvalue-disjoint2}:$
fixes $p::'a::\text{mem-type } \text{ ptr}$
assumes $\text{field-lookup}: \text{field-lookup } (typ\text{-info-t } \text{TYPE}('a)) \ \text{path } 0 = \text{Some } (t, n)$
assumes $\text{match}: \text{export-uinfo } t = \text{typ-uinfo-t } \text{TYPE}('f::\text{c-type})$
assumes $\text{ptr-span } p \cap A = \{\}$
shows $\{\&(p \rightarrow \text{path})..+\text{size-of } \text{TYPE}('f)\} \cap A = \{\}$
using assms
by $(\text{simp add: disjoint-field-lvalue-propagation-right export-size-of inter-commute})$

context heap-state-global
begin

lemma $\text{global-update-frame-commute}:$

shows $\text{glob-upd } g \ (\text{frame } A \ t_0 \ s) =$
 $\text{frame } A \ t_0 \ (\text{glob-upd } g \ s)$

proof $-$

from $\text{glob-htd-commute } \text{glob-hmem-commute}$

show $?thesis$

apply $(\text{clarsimp simp add: frame-def})$

by $(\text{smt } (\text{verit}) \ \text{heap.upd-cong } \text{typing.get-upd } \text{typing.upd-cong } \text{typing.upd-get})$

qed

lemma $L2\text{-modify-no-heap-update-rel-stack}[\text{synthesize-rule } \text{refines-in-out}]:$

assumes $\text{rel-alloc } \mathcal{S} \ M \ A \ t_0 \ s \ t$

assumes $v \ s = v' \ t$

shows refines

$(L2\text{-modify } (\lambda s. \text{glob-upd } (\lambda-. \ v \ s) \ s))$

$(L2\text{-modify } (\lambda s. \text{glob-upd } (\lambda-. v' s) s))$
 $s \ t$
 $(\text{rel-stack } \mathcal{S} \ \{\} \ A \ s \ t_0 \ (\text{rel-xval-stack } L \ (\lambda-. (=))))$

proof –
from *glob-hmem-commute* **have** *no-heap1*: $\bigwedge g \ s. \text{hmem } (\text{glob-upd } g \ s) = \text{hmem}$
 s
by (*metis heap.get-upd heap.upd-get*)

from *glob-htd-commute* **have** *no-htd1*: $\bigwedge g \ s. \text{htd } (\text{glob-upd } g \ s) = \text{htd } s$
by (*metis typing.get-upd typing.upd-get*)

from *assms no-heap1 no-htd1 global-update-frame-commute*
show *?thesis*
by (*auto simp add: refines-def-old rel-stack-def rel-alloc-def L2-defs*)

qed

lemma *rel-alloc-global-independent-eq* [*rel-alloc-independent-globals*]:
assumes *rel-alloc* $\mathcal{S} \ M \ A \ t_0 \ s \ t$
shows $\text{glob } t = \text{glob } s$
by (*metis assms get-upd global-update-frame-commute rel-alloc-fold-frame upd-get*)

end

lemma *L2-seq-guard-cong-stateless*:
 $(\bigwedge s. P \ s = P' \ s) \implies (\bigwedge s. P' \ s \implies X = X') \implies$
 $L2\text{-seq } (L2\text{-guard } P) \ (\lambda-. X) = L2\text{-seq } (L2\text{-guard } P') \ (\lambda-. X')$
using *L2-seq-guard-cong''*
by *metis*

context *globals-stack-heap-state*
begin

lemma *globals-subset-trans*: $\text{NO-MATCH } \mathcal{G} \ X \implies \text{NO-MATCH } \mathcal{G} \ Y \implies X \subseteq \mathcal{G}$
 $\implies \mathcal{G} \subseteq Y \implies X \subseteq Y$
by *blast*

lemma *globals-disjoint-subset*: $\text{NO-MATCH } \mathcal{G} \ X \implies X \subseteq \mathcal{G} \implies \mathcal{G} \cap A = \{\} \implies$
 $X \cap A = \{\}$
by *blast*

end

lemma *Union-Diff-right-conv'*: $X \cup Y = X \cup (Y - X)$
by *blast*

lemma *Union-Diff-right-conv*: $(Y - X) \equiv Z \implies X \cup Y \equiv X \cup Z$
apply (*subst Union-Diff-right-conv'*)
apply *simp*
done

lemma *Union-assoc*: $X \cup Y \cup Z = X \cup (Y \cup Z)$


```

by (simp add: sup-assoc)

simproc-setup Union-Diff-conv (⟨ptr-span x ∪ Y⟩) = ⟨
let
  val cterm-eq = is-equal o Thm.fast-term-ord
  fun dest-union-right ct = ct |> Match-Cterm.switch [
    @{cterm-match ⟨?X ∪ ?Y⟩} #> (fn {X, Y, ...} => X::dest-union-right Y),
    fn - => [ct]]
in
  fn - => fn ctxt => fn ct =>
    let
      val {X, Y, ...} = @{cterm-match ?X ∪ ?Y} ct
      val ys = dest-union-right Y
      val - = if member cterm-eq ys X then () else raise Match
      val lhs = infer-instantiate ⟨X = X and Y = Y in cterm Y - X⟩ ctxt
      val eq = lhs |> Simplifier.asm-full-rewrite (ctxt addsimps @{thms Un-Diff
Diff-triv Union-assoc})
      val rhs = Thm.rhs-of eq
    in
      if cterm-eq (lhs, rhs) then
        NONE
      else
        let val eq' = @{thm Union-Diff-right-conv} OF [eq]
            val - = Utils.verbose-msg 1 (* FIXME *) ctxt (fn - => Union-Diff-conv:
^ Thm.string-of-thm ctxt eq')
            in SOME eq' end
        end
      handle Match => NONE
    end
  end
  declare [[simproc del: Union-Diff-conv]]

lemma
ptr-span q ∩ ptr-span p = {} ⟹ ptr-span x ∩ ptr-span p = {} ⟹
  (ptr-span p ∪ (ptr-span q ∪ (ptr-span x ∪ ptr-span p))) = ptr-span p ∪ (ptr-span
q ∪ ptr-span x)
supply [[simproc add: Union-Diff-conv]]
apply simp
done

lemma
ptr-span q ∩ ptr-span p = {} ⟹ ptr-span x ∩ ptr-span p = {} ⟹
  ((ptr-span p ∪ ptr-span q) ∪ (ptr-span x ∪ ptr-span p)) = ptr-span p ∪ (ptr-span
q ∪ ptr-span x)
supply [[simproc add: Union-Diff-conv]]
apply (simp add: Union-assoc)
done

lemma
assumes disj: ptr-span q ∩ ptr-span p = {}

```

```

    ptr-span p ∩ ptr-span q = {}
    ptr-span x ∩ ptr-span p = {}
    ptr-span x ∩ ptr-span q = {}
  shows ((ptr-span p ∪ ptr-span q) ∪ ((ptr-span p ∪ ptr-span q) ∪ ptr-span x ∪
ptr-span p)) = ptr-span p ∪ (ptr-span q ∪ ptr-span x)
  supply [[simplproc add: Union-Diff-conv]]
  apply (simp add: Union-assoc disj)
done

```

```

lemma subset-union-left: X ⊆ L ⇒ X ⊆ L ∪ R
  by blast

```

```

lemma subset-union-right: X ⊆ R ⇒ X ⊆ L ∪ R
  by blast

```

```

end

```

Chapter 21

HL phase: Heap Lifting / Split Heap

```
theory Split-Heap
imports
  TypHeapSimple
begin
```

```
ML-file ac-names.ML
```

```
definition array-fields :: nat  $\Rightarrow$  qualified-field-name list where
  array-fields n = map ( $\lambda n.$  [replicate n CHR "1'"]) [0..n]
```

```
lemma Nil-nmem-array-fields[simp]: []  $\notin$  set (array-fields n)
by (auto simp add: array-fields-def)
```

```
lemma distinct-prop-disj-fn-array-fields[simp]: distinct-prop disj-fn (array-fields n)
by (simp add: array-fields-def distinct-prop-iff-nth)
```

```
lemma field-lookup-field-ti':
  field-lookup (typ-info-t TYPE('a :: c-type)) f 0 = Some (a, b)  $\implies$  field-ti TYPE('a)
  f = Some a
unfolding field-ti-def by simp
```

```
lemma field-lookup-append-add:
  wf-desc t  $\implies$ 
  field-lookup t (f @ g) n =
    Option.bind (field-lookup t f n) ( $\lambda(t', m).$ 
      Option.bind (field-lookup t' g 0) ( $\lambda(t'', m').$  Some (t'', m + m')))
by (subst field-lookup-append-eq)
  (auto simp: field-lookup-offset intro: field-lookup-offset-None[THEN iffD2])
  split: bind-split)
```

```
lemma nth-array-fields: i < n  $\implies$  array-fields n ! i = [replicate i CHR "1'"]
by (simp add: array-fields-def)
```

lemma *array-fields-Suc*: $\text{array-fields } (\text{Suc } n) = \text{array-fields } n @ \llbracket \text{replicate } n \text{ CHR } "1" \rrbracket$

by (*simp add: array-fields-def*)

lemma *find-append*: $\text{find } P (xs @ ys) = (\text{case find } P \text{ xs of None } \Rightarrow \text{find } P \text{ ys} \mid \text{Some } x \Rightarrow \text{Some } x)$

by (*induct xs*) *auto*

lemma *length-array-fields*: $\text{length } (\text{array-fields } n) = n$

by (*simp add: array-fields-def*)

lemma *set-array-fields*: $\text{set } (\text{array-fields } n) = (\bigcup i < n. \{\llbracket \text{replicate } i \text{ CHR } "1" \rrbracket\})$

by (*induct n*) (*auto simp add: array-fields-def*)

lemma *find-array-fields-Some*:

$\text{find } P (\text{array-fields } n) = \text{Some } y \longleftrightarrow$

$(\exists i < n. y = \llbracket \text{replicate } i \text{ CHR } "1" \rrbracket \wedge P \ y \wedge (\forall j < i. \neg P (\llbracket \text{replicate } j \text{ CHR } "1" \rrbracket)))$

proof (*induct n arbitrary: y*)

case 0

then show *?case*

by (*simp add: array-fields-def*)

next

case (*Suc n*)

show *?case*

proof (*cases find P (array-fields n)*)

case *None*

then show *?thesis*

apply (*simp add: array-fields-Suc find-append*)

using *less-Suc-eq*

by (*auto dest!: findNoneD simp add: set-array-fields*)

next

case (*Some a*)

then show *?thesis*

apply (*simp add: array-fields-Suc find-append*)

apply (*simp add: Suc.hyps*)

by (*metis less-Suc-eq linorder-neqE-nat*)

qed

qed

lemma *Bex-intvl-conv*: $(\exists x \in \{0..<n::nat\}. P \ x) \longleftrightarrow (\exists i. i < n \wedge P \ i)$

by *auto*

lemma *Bex-union-conv*: $(\exists x \in A \cup B. P \ x) \longleftrightarrow ((\exists x \in A. P \ x) \vee (\exists x \in B. P \ x))$

by *auto*

lemma *ex-intvl-conj-distribR*: $((\exists x \in \{0..<n\}. P \ x) \wedge Q) \longleftrightarrow (\exists x \in \{0..<n\}. P \ x \wedge Q)$

by *blast*

lemma *ex-less-conj-distribR*: $((\exists i < n::nat. P\ i) \wedge Q) \longleftrightarrow (\exists i < n. P\ i \wedge Q)$
by *blast*

lemma *map-filter-map-Some-conv*:
assumes *all-Some*: $\bigwedge x. x \in set\ xs \implies f\ x = Some\ (g\ x)$
shows $List.map-filter\ f\ xs = map\ g\ xs$
using *all-Some*
apply (*induct xs*)
apply (*auto simp add: List.map-filter-def*)
done

lemma *map-filter-map-compose*:
 $List.map-filter\ f\ (map\ g\ xs) = List.map-filter\ (f\ o\ g)\ xs$
by (*induct xs*) (*auto simp add: List.map-filter-def*)

lemma *map-filter-fun-eq-conv*:
assumes *all-same*: $\bigwedge x. x \in set\ xs \implies f\ x = g\ x$
shows $List.map-filter\ f\ xs = List.map-filter\ g\ xs$
using *all-same*
apply (*induct xs*)
apply (*auto simp add: List.map-filter-def*)
done

lemma *map-filter-empty[simp]*: $List.map-filter\ Map.empty\ xs = []$
by (*simp add: List.map-filter-def*)

lemma *map-filter-Some[simp]*: $List.map-filter\ (\lambda x. Some\ (f\ x))\ xs = map\ f\ xs$
by (*simp add: List.map-filter-def*)

lemma *list-all-field-lookup[simp]*:
 $CARD('n) = m \implies$
 $list-all$
 $(\lambda f. \exists a\ b. field-lookup\ (typ-uinfo-t\ TYPE('a::c-type['n::finite]))\ f\ 0 = Some\ (a,$
 $b))$
 $(map\ (\lambda n. [replicate\ n\ CHR\ '1'])\ [0..<m])$
using *field-lookup-array* **where** $'b = 'n$ **and** $'a = 'a$
apply (*simp add: list-all-iff atLeast0LessThan typ-uinfo-t-def flip: All-less-Ball*)
using *field-lookup-export-uinfo-Some* **by** *blast*

lemma *ptr-span-with-stack-byte-type-subset-field[simp]*:
 $\forall a \in ptr-span\ (p::'a::mem-type\ ptr). root-ptr-valid\ h\ (PTR(stack-byte)\ a) \implies$
 $\exists u. field-ti\ TYPE('a::mem-type)\ f = Some\ u \wedge size-td\ u = sz \implies$
 $(\forall a \in \{\&(p \rightarrow f)..+sz\}. root-ptr-valid\ h\ (PTR(stack-byte)\ a)) \longleftrightarrow True$
by (*auto simp: dest!: mem-type-class.field-tag-sub field-tiD*)

lemma (*in pointer-lense*) *pointer-lense-field-lvalue*:
assumes *f*: $field-ti\ TYPE('c::mem-type)\ f = Some\ u$
and *u*: $size-td\ u = size-of\ TYPE('a)$

shows *pointer-lense* ($\lambda h (p::'c \text{ ptr}). r h (PTR('a) \&(p \rightarrow f)) (\lambda p. w (PTR('a) \&(p \rightarrow f)))$)
apply *unfold-locales*
apply (*simp-all add: read-write-same write-same*)
apply (*subst write-other-commute*)
subgoal for $p \ q$
using *field-tag-sub*[*OF field-tiD, OF f, of p*]
using *field-tag-sub*[*OF field-tiD, OF f, of q*]
apply (*simp add: u*)
apply (*rule disjnt-subset1, erule disjnt-subset2*)
by *auto*
apply *simp*
done

lemma *exists-nat-numeral*: $\exists x::\text{nat}. x < \text{numeral } k$
apply (*rule exI*[**where** $x=0$])
apply (*rule Num.rel-simps*)
done

lemma *fun-upd-eq-cases*: $f(p:=x) = g \longleftrightarrow (g \ p = x \wedge (\forall q. p \neq q \longrightarrow f \ q = g \ q))$
by (*auto simp add: fun-upd-def*)

lemma *fun-upd-apply-eq-cases*: $(f(p:=x)) \ q = g \ q \longleftrightarrow ((p = q \longrightarrow g \ q = x) \wedge (p \neq q \longrightarrow f \ q = g \ q))$
by (*auto simp add: fun-upd-def*)

lemma *comp-fun-upd-same-fuse*:
 $(\lambda f. f(p := x)) \ o (\lambda f. f(p := y)) = (\lambda f. f(p:=x))$
by *auto*

lemma *comp-fun-upd-other-commute*:
 $p \neq q \implies (\lambda f. f(q := y)) \ o (\lambda f. f(p := x)) = (\lambda f. f(p := x)) \ o (\lambda f. f(q := y))$
by (*auto simp add: comp-def fun-upd-def fun-eq-iff*)

lemma *map-td-compose*:
fixes $t::('a, 'b) \text{ typ-desc}$
and $st::('a, 'b) \text{ typ-struct}$
and $ts::('a, 'b) \text{ typ-tuple list}$
and $x::('a, 'b) \text{ typ-tuple}$

shows
 $\text{map-td } f1 \ g1 (\text{map-td } f2 \ g2 \ t) = \text{map-td } (\lambda n \text{ algn. } f1 \ n \text{ algn } o \ f2 \ n \text{ algn}) (g1 \ o \ g2) \ t$ **and**
 $\text{map-td-struct } f1 \ g1 (\text{map-td-struct } f2 \ g2 \ st) = \text{map-td-struct } (\lambda n \text{ algn. } f1 \ n \text{ algn } o \ f2 \ n \text{ algn}) (g1 \ o \ g2) \ st$ **and**
 $\text{map-td-list } f1 \ g1 (\text{map-td-list } f2 \ g2 \ ts) = \text{map-td-list } (\lambda n \text{ algn. } f1 \ n \text{ algn } o \ f2 \ n \text{ algn}) (g1 \ o \ g2) \ ts$ **and**
 $\text{map-td-tuple } f1 \ g1 (\text{map-td-tuple } f2 \ g2 \ x) = \text{map-td-tuple } (\lambda n \text{ algn. } f1 \ n \text{ algn } o \ f2 \ n \text{ algn}) (g1 \ o \ g2) \ x$
proof (*induct t and st and ts and x*)

```

    case (TypDesc nat typ-struct list)
  then show ?case by auto
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case by auto
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by auto
qed

```

lemma (in *mem-type*) *distinct-all-field-names*:
distinct (all-field-names (typ-info-t TYPE('a)))
using *distinct-all-field-names wf-desc* **by** *blast*

lemma *field-lookup-same-type-disjoint*:

```

[[field-lookup t f m = Some (d,n);
  field-lookup t f' m = Some (d',n'); f ≠ f'; export-uinfo d = export-uinfo d';
  wf-desc t; size-td t < addr-card]] ⇒
  {(of-nat n)::addr..+size-td (d::('a,'b) typ-info)} ∩ {of-nat n'..+size-td d'} =
{} and

```

```

[[field-lookup-struct st f m = Some (d,n);
  field-lookup-struct st f' m = Some (d',n'); f ≠ f'; export-uinfo d = export-uinfo
d';
  wf-desc-struct st; size-td-struct st < addr-card]] ⇒
  {(of-nat n)::addr..+size-td (d::('a,'b) typ-info)} ∩ {of-nat n'..+size-td d'} =
{} and

```

```

[[field-lookup-list ts f m = Some (d,n);
  field-lookup-list ts f' m = Some (d',n'); f ≠ f'; export-uinfo d = export-uinfo
d';
  wf-desc-list ts; size-td-list ts < addr-card]] ⇒
  {(of-nat n)::addr..+size-td (d::('a,'b) typ-info)} ∩ {of-nat n'..+size-td d'} =
{} and

```

```

[[field-lookup-tuple x f m = Some (d,n) ;
  field-lookup-tuple x f' m = Some (d',n'); f ≠ f'; export-uinfo d = export-uinfo
d';
  wf-desc-tuple x; size-td-tuple x < addr-card]] ⇒
  {(of-nat n)::addr..+size-td (d::('a,'b) typ-info)} ∩ {of-nat n'..+size-td d'} =

```

```

{}
proof (induct t and st and ts and x arbitrary: f m d n f' m d' n' and f m d n
f' m d' n' and
  f m d n f' m d' n' and f m d n f' m d' n')
  case (TypDesc algn st nm)
  then show ?case
  by (metis field-lookup.simps field-lookup-export-uinfo-Some field-lookup-same-type-empty(1)
size-td.simps wf-desc.simps)
next
  case (TypScalar n algn x)
  then show ?case by auto
next
  case (TypAggregate ts)
  then show ?case by simp
next
  case Nil-typ-desc
  then show ?case by simp
next
  case (Cons-typ-desc x xs)
  from Cons-typ-desc.premis obtain
    f: field-lookup-list (x # xs) f m = Some (d, n) and
    f': field-lookup-list (x # xs) f' m = Some (d', n') and
    neq-f-f': f ≠ f' and
    eq-d-d': export-uinfo d = export-uinfo d' and
    sz: size-td-tuple x + size-td-list xs < addr-card and
    wf-x: wf-desc-tuple x and
    distinct: dt-snd x ∉ dt-snd ' set xs and
    wf-xs: wf-desc-list xs by clarsimp
  from sz have sz-xs: size-td-list xs < addr-card by auto
  from sz have sz-x: size-td-tuple x < addr-card by auto
  show ?case
  proof (cases field-lookup-tuple x f m)
    case None
    note f-None = this
    from f None have f-xs: field-lookup-list xs f (m + size-td (dt-fst x)) = Some
(d, n) by simp
    from td-set-list-field-lookup-listD [OF f-xs] have (d, n) ∈ td-set-list xs (m +
size-td (dt-fst x)) .
    from td-set-list-intvl-sub [OF this]
    have contained-f: {word-of-nat n..+size-td d} ⊆ {word-of-nat (m + size-td
(dt-fst x))..+size-td-list xs} .
    show ?thesis
    proof (cases field-lookup-tuple x f' m)
      case None
      from f' None have f'-xs: field-lookup-list xs f' (m + size-td (dt-fst x)) =
Some (d', n') by simp
      from Cons-typ-desc.hyps(2) [OF f-xs f'-xs neq-f-f' eq-d-d' wf-xs sz-xs]
      show ?thesis .
    next

```



```

    case (Some -)
    with f' have f'-x: field-lookup-tuple x f' m = Some (d', n') by simp
    from td-set-tuple-field-lookup-tupleD [OF f'-x] have (d', n') ∈ td-set-tuple x
m .
    from td-set-tuple-intvl-sub [OF this]
    have {word-of-nat n'..+size-td d'} ⊆ {word-of-nat m..+size-td-tuple x} .
    with contained-f sz show ?thesis
    by (smt (verit, ccfv-SIG) disj-subset init-intvl-disj of-nat-add size-td-tuple-dt-fst)
qed
next
case (Some -)
from f Some have f-x: field-lookup-tuple x f m = Some (d, n) by simp
from td-set-tuple-field-lookup-tupleD [OF f-x] have (d, n) ∈ td-set-tuple x m .
from td-set-tuple-intvl-sub [OF this]
have contained-f: {word-of-nat n..+size-td d} ⊆ {word-of-nat m..+size-td-tuple
x} .
show ?thesis
proof (cases field-lookup-tuple x f' m)
  case None
  from f' None have f'-xs: field-lookup-list xs f' (m + size-td (dt-fst x)) =
Some (d', n') by simp
  from td-set-list-field-lookup-listD [OF f'-xs]
  have (d', n') ∈ td-set-list xs (m + size-td (dt-fst x)) .
  from td-set-list-intvl-sub [OF this]
  have {word-of-nat n'..+size-td d'} ⊆ {word-of-nat (m + size-td (dt-fst
x))..+size-td-list xs} .
  with contained-f sz show ?thesis
  by (smt (verit, ccfv-threshold) Int-commute disjoint-subset init-intvl-disj
of-nat-add size-td-tuple-dt-fst)
next
case (Some -)
with f' have f'-x: field-lookup-tuple x f' m = Some (d', n') by simp
from Cons-typ-desc.hyps(1) [OF f-x f'-x neq-f-f' eq-d-d' wf-x sz-x]
show ?thesis .
qed
qed
next
case (DTuple-typ-desc typ-desc list b)
then show ?case
by (metis fl5 list.expand option.simps(3) size-td-tuple.simps wf-desc-tuple.simps)
qed

```

lemma *field-lookup-same-type-u-disjoint:*

```

[[field-lookup t f m = Some (d,n);
  field-lookup t f' m = Some (d,n'); f ≠ f';
  wf-desc t; size-td t < addr-card]] ⇒
{(of-nat n)::addr..+size-td d} ∩ {of-nat n'..+size-td d} = {} and

```

```

[[field-lookup-struct st f m = Some (d,n);
  field-lookup-struct st f' m = Some (d,n'); f ≠ f';
  wf-desc-struct st; size-td-struct st < addr-card ]] ⇒
  {(of-nat n)::addr..+size-td d} ∩ {(of-nat n')::addr..+size-td d} = {} and

[[field-lookup-list ts f m = Some (d,n);
  field-lookup-list ts f' m = Some (d,n'); f ≠ f';
  wf-desc-list ts; size-td-list ts < addr-card]] ⇒
  {(of-nat n)::addr..+size-td d} ∩ {(of-nat n')::addr..+size-td d} = {} and
[[field-lookup-tuple x f m = Some (d,n) ;
  field-lookup-tuple x f' m = Some (d,n'); f ≠ f';
  wf-desc-tuple x; size-td-tuple x < addr-card]] ⇒
  {(of-nat n)::addr..+size-td d} ∩ {(of-nat n')::addr..+size-td d} = {}
proof (induct t and st and ts and x arbitrary: f m d n f' m n' and f m d n f'
m n' and
  f m d n f' m n' and f m d n f' m n')
case (TypDesc algn st nm)
then show ?case
  by (metis field-lookup.simps field-lookup-same-type-empty(1)
size-td.simps wf-desc.simps)
next
  case (TypScalar n algn x)
  then show ?case by auto
next
  case (TypAggregate ts)
  then show ?case by simp
next
  case Nil-typ-desc
  then show ?case by simp
next
  case (Cons-typ-desc x xs)
  from Cons-typ-desc.prem obtain
    f: field-lookup-list (x # xs) f m = Some (d, n) and
    f': field-lookup-list (x # xs) f' m = Some (d, n') and
    neq-f-f': f ≠ f' and
    sz: size-td-tuple x + size-td-list xs < addr-card and
    wf-x: wf-desc-tuple x and
    distinct: dt-snd x ∉ dt-snd ' set xs and
    wf-xs: wf-desc-list xs by clarsimp
  from sz have sz-xs: size-td-list xs < addr-card by auto
  from sz have sz-x: size-td-tuple x < addr-card by auto
  show ?case
  proof (cases field-lookup-tuple x f m)
    case None
    note f-None = this
    from f-None have f-xs: field-lookup-list xs f (m + size-td (dt-fst x)) = Some
(d, n) by simp
    from td-set-list-field-lookup-listD [OF f-xs] have (d, n) ∈ td-set-list xs (m +
size-td (dt-fst x)) .

```

```

from td-set-list-intvl-sub [OF this]
  have contained-f: {word-of-nat n'..+size-td d}  $\subseteq$  {word-of-nat (m + size-td (dt-fst x))..+size-td-list xs} .
  show ?thesis
  proof (cases field-lookup-tuple x f' m)
    case None
      from f' None have f'-xs: field-lookup-list xs f' (m + size-td (dt-fst x)) = Some (d, n') by simp
      from Cons-tyt-desc.hyps(2) [OF f-xs f'-xs neq-f-f' wf-xs sz-xs]
      show ?thesis .
    next
      case (Some -)
        with f' have f'-x: field-lookup-tuple x f' m = Some (d, n') by simp
        from td-set-tuple-field-lookup-tupleD [OF f'-x] have  $(d, n') \in \text{td-set-tuple } x$ 
        from td-set-tuple-intvl-sub [OF this]
        have {word-of-nat n'..+size-td d}  $\subseteq$  {word-of-nat m..+size-td-tuple x} .
        with contained-f sz show ?thesis
        by (smt (verit, ccfv-SIG) disj-subset init-intvl-disj of-nat-add size-td-tuple-dt-fst)
      qed
    next
      case (Some -)
        from f Some have f-x: field-lookup-tuple x f m = Some (d, n) by simp
        from td-set-tuple-field-lookup-tupleD [OF f-x] have  $(d, n) \in \text{td-set-tuple } x$  .
        from td-set-tuple-intvl-sub [OF this]
        have contained-f: {word-of-nat n'..+size-td d}  $\subseteq$  {word-of-nat m..+size-td-tuple x} .
        show ?thesis
        proof (cases field-lookup-tuple x f' m)
          case None
            from f' None have f'-xs: field-lookup-list xs f' (m + size-td (dt-fst x)) = Some (d, n') by simp
            from td-set-list-field-lookup-listD [OF f'-xs]
            have  $(d, n') \in \text{td-set-list } xs (m + size-td (dt-fst x))$  .
            from td-set-list-intvl-sub [OF this]
            have {word-of-nat n'..+size-td d}  $\subseteq$  {word-of-nat (m + size-td (dt-fst x))..+size-td-list xs} .
            with contained-f sz show ?thesis
            by (smt (verit, ccfv-threshold) Int-commute disjoint-subset init-intvl-disj of-nat-add size-td-tuple-dt-fst)
          next
            case (Some -)
              with f' have f'-x: field-lookup-tuple x f' m = Some (d, n') by simp
              from Cons-tyt-desc.hyps(1) [OF f-x f'-x neq-f-f' wf-x sz-x]
              show ?thesis .
            qed
          qed
        next
          case (DTuple-tyt-desc tyt-desc list b)

```

then show *?case*
by (*metis fl5 list.expand option.simps(3) size-td-tuple.simps wf-desc-tuple.simps*)
qed

lemma *td-set-list-intvl-sub-nat*:
 $(d,n) \in \text{td-set-list } t \ m \implies \{n..< n + \text{size-td } d\} \subseteq \{m..< m + \text{size-td-list } t\}$
apply(*frule td-set-list-offset-le*)
apply(*drule td-set-list-offset-size-m*)
apply *auto*
done

lemma *td-set-tuple-intvl-sub-nat*:
 $(d,n) \in \text{td-set-tuple } t \ m \implies \{n..< n + \text{size-td } d\} \subseteq \{m..< m + \text{size-td-tuple } t\}$
apply(*frule td-set-tuple-offset-le*)
apply(*drule td-set-tuple-offset-size-m*)
apply *auto*
done

lemma *field-lookup-same-type-u-disjoint-nat*:
 $\llbracket \text{field-lookup } t \ f \ m = \text{Some } (d,n);$
 $\text{field-lookup } t \ f' \ m = \text{Some } (d,n'); f \neq f';$
 $\text{wf-desc } t \rrbracket \implies$
 $\{n..< n + \text{size-td } d\} \cap \{n'..< n' + \text{size-td } d\} = \{\}$ **and**

$\llbracket \text{field-lookup-struct } st \ f \ m = \text{Some } (d,n);$
 $\text{field-lookup-struct } st \ f' \ m = \text{Some } (d,n'); f \neq f';$
 $\text{wf-desc-struct } st \rrbracket \implies$
 $\{n..< n + \text{size-td } d\} \cap \{n'..< n' + \text{size-td } d\} = \{\}$ **and**

$\llbracket \text{field-lookup-list } ts \ f \ m = \text{Some } (d,n);$
 $\text{field-lookup-list } ts \ f' \ m = \text{Some } (d,n'); f \neq f';$
 $\text{wf-desc-list } ts \rrbracket \implies$
 $\{n..< n + \text{size-td } d\} \cap \{n'..< n' + \text{size-td } d\} = \{\}$ **and**

$\llbracket \text{field-lookup-tuple } x \ f \ m = \text{Some } (d,n);$
 $\text{field-lookup-tuple } x \ f' \ m = \text{Some } (d,n'); f \neq f';$
 $\text{wf-desc-tuple } x \rrbracket \implies$
 $\{n..< n + \text{size-td } d\} \cap \{n'..< n' + \text{size-td } d\} = \{\}$

proof (*induct t and st and ts and x arbitrary: f m d n f' m n' and f m d n f'*
m n' and

f m d n f' m n' and f m d n f' m n')

case (*TypDesc algn st nm*)

then show *?case*

by (*metis field-lookup.simps field-lookup-same-type-empty(1)*
wf-desc.simps)

next

case (*TypScalar n algn x*)

then show *?case by auto*

next

case (*TypAggregate ts*)

```

then show ?case by auto
next
case Nil-typ-desc
then show ?case by simp
next
case (Cons-typ-desc x xs)

from Cons-typ-desc.prem1 obtain
f: field-lookup-list (x # xs) f m = Some (d, n) and
f': field-lookup-list (x # xs) f' m = Some (d, n') and
neq-f-f': f ≠ f' and
wf-x: wf-desc-tuple x and
distinct: dt-snd x ∉ dt-snd `set xs and
wf-xs: wf-desc-list xs by clarsimp

show ?case
proof (cases field-lookup-tuple x f m)
case None
note f-None = this
from f-None have f-xs: field-lookup-list xs f (m + size-td (dt-fst x)) = Some
(d, n) by simp
from td-set-list-field-lookup-listD [OF f-xs] have (d, n) ∈ td-set-list xs (m +
size-td (dt-fst x)) .
from td-set-list-intvl-sub-nat [OF this]
have contained-f: {n.. $n+size-td\ d\} \subseteq \{(m + size-td (dt-fst x))..(m + size-td
(dt-fst x)) + size-td-list\ xs\} .
show ?thesis
proof (cases field-lookup-tuple x f' m)
case None
from f'-None have f'-xs: field-lookup-list xs f' (m + size-td (dt-fst x)) =
Some (d, n') by simp
from Cons-typ-desc.hyps(2) [OF f-xs f'-xs neq-f-f' wf-x]
show ?thesis .
next
case (Some -)
with f' have f'-x: field-lookup-tuple x f' m = Some (d, n') by simp
from td-set-tuple-field-lookup-tupleD [OF f'-x] have (d, n') ∈ td-set-tuple x
m .
from td-set-tuple-intvl-sub-nat[OF this]
have {n'.. $n'+size-td\ d\} \subseteq \{m.. $m+size-td-tuple\ x\} .
with contained-f show ?thesis
by (smt (verit, best) disj-inter-swap disj-subset ivl-disj-int-two(3) size-td-tuple-dt-fst)
qed

next
case (Some -)
from f-Some have f-x: field-lookup-tuple x f m = Some (d, n) by simp
from td-set-tuple-field-lookup-tupleD [OF f-x] have (d, n) ∈ td-set-tuple x m .
from td-set-tuple-intvl-sub-nat [OF this]$$$ 
```

```

have contained-f: {n.. $n + \text{size-td } d$ }  $\subseteq$  {m.. $m + \text{size-td-tuple } x$ } .
show ?thesis
proof (cases field-lookup-tuple x f' m)
  case None
    from f' None have f'-xs: field-lookup-list xs f' (m + size-td (dt-fst x)) =
Some (d, n') by simp
    from td-set-list-field-lookup-listD [OF f'-xs]
    have (d, n')  $\in$  td-set-list xs (m + size-td (dt-fst x)) .
    from td-set-list-intvl-sub-nat [OF this]
    have {n'.. $n' + \text{size-td } d$ }  $\subseteq$  {(m + size-td (dt-fst x)).. $(m + \text{size-td } (dt-fst
x)) + \text{size-td-list } xs$ } .
    with contained-f show ?thesis
    by (metis Int-commute disjoint-subset ivl-disj-int-two(3) size-td-tuple-dt-fst)
  next
    case (Some -)
    with f' have f'-x: field-lookup-tuple x f' m = Some (d, n') by simp
    from Cons-typ-desc.hyps(1) [OF f-x f'-x neq-f-f' wf-x ]
    show ?thesis .
  qed
qed
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
  by (metis fl5 list.expand option.simps(3) wf-desc-tuple.simps)
qed

```

```

lemma (in mem-type) field-lookup-same-type-disjoint:
assumes f: field-lookup (typ-info-t TYPE('a)) f m = Some (t, n)
assumes f': field-lookup (typ-info-t TYPE('a)) f' m = Some (t', n')
assumes neq: f  $\neq$  f'
assumes same: export-uinfo t = export-uinfo t'
shows {(of-nat n):: $\text{addr}.. + \text{size-td } t$ }  $\cap$  {(of-nat n').. $\text{size-td } t'$ } = {}
proof -
  have size-td (typ-info-t TYPE('a)) < addr-card
  by (simp add: size-of-fold)

  from field-lookup-same-type-disjoint(1) [OF f f' neq same wf-desc this]
  show ?thesis
  by simp
qed

```

```

lemma (in mem-type) field-lookup-same-type-ptr-span-disjoint:
fixes p::'a ptr
assumes f: field-lookup (typ-info-t TYPE('a)) f 0 = Some (t, n)
assumes f': field-lookup (typ-info-t TYPE('a)) f' 0 = Some (t', n')
assumes neq: f  $\neq$  f'
assumes t: export-uinfo t = typ-uinfo-t (TYPE('b)::c-type)
assumes t': export-uinfo t' = typ-uinfo-t (TYPE('b)::c-type)
shows ptr-span (PTR('b)  $\&$ (p $\rightarrow$ f))  $\cap$  ptr-span (PTR('b)  $\&$ (p $\rightarrow$ f')) = {}

```

proof –
from $t\ t'$ **have** $\text{export-uinto } t = \text{export-uinto } t'$ **by** *simp*
from *field-lookup-same-type-disjoint* [*OF ff' neq this*]
have $\{\text{word-of-nat } n::\text{addr}..+\text{size-td } t\} \cap \{\text{word-of-nat } n'..+\text{size-td } t'\} = \{\}$.
moreover
from t **have** $\text{size-td } t = \text{size-of } \text{TYPE}('b)$
by (*simp add: export-size-of*)
moreover
from t' **have** $\text{size-td } t' = \text{size-of } \text{TYPE}('b)$
by (*simp add: export-size-of*)
ultimately
show *?thesis*
using ff'
by (*simp add: field-lvalue-def intvl-disj-offset*)
qed

lemma *sub-type-valid-field-lvalue-overlap*:
fixes $p::'a::\text{mem-type ptr}$
fixes $q::'b::\text{mem-type ptr}$
assumes *subtype*: $\text{TYPE}('b) \leq_{\tau} \text{TYPE}('a)$
assumes *valid-p*: $d \models_t p$
assumes *overlap*: $\text{ptr-span } p \cap \text{ptr-span } q \neq \{\}$
shows $d \models_t q \longleftrightarrow$
 $(\exists f\ t\ n. \text{field-lookup } (\text{typ-into-t } \text{TYPE}('a))\ f\ 0 = \text{Some } (t, n) \wedge$
 $\text{export-uinto } t = \text{typ-uinto-t } \text{TYPE}('b) \wedge$
 $q = \text{PTR}('b) (\&(p \rightarrow f)))$

proof
assume *valid-q*: $d \models_t q$
show $\exists f\ t\ n. \text{field-lookup } (\text{typ-into-t } \text{TYPE}('a))\ f\ 0 = \text{Some } (t, n) \wedge$
 $\text{export-uinto } t = \text{typ-uinto-t } \text{TYPE}('b) \wedge q = \text{PTR}('b) (\&(p \rightarrow f))$
proof –

from *subtype* **have** $\text{typ-uinto-t } \text{TYPE}('b) \leq \text{typ-uinto-t } \text{TYPE}('a)$ **by** (*simp add: sub-tyt-def*)

from *this overlap valid-footprint-sub-cases* [*OF h-t-valid-valid-footprint* [*OF valid-q*] *h-t-valid-valid-footprint* [*OF valid-p*]]
have *field-of*: $\text{field-of } (\text{ptr-val } q - \text{ptr-val } p) (\text{typ-uinto-t } \text{TYPE}('b)) (\text{typ-uinto-t } \text{TYPE}('a))$
apply (*simp add: size-of-def*)
by (*metis Int-commute le-less-trans less-irrefl*)
then obtain f **where**
 $fl: \text{field-lookup } (\text{typ-uinto-t } (\text{TYPE}('a)))\ f\ 0 = \text{Some } (\text{typ-uinto-t } \text{TYPE}('b),$
 $\text{unat } (\text{ptr-val } q - \text{ptr-val } p))$
using *field-of-lookup-info* **by** *blast*

then obtain t **where**

fl' : *field-lookup* (*typ-info-t* (*TYPE('a)*)) *f 0* = *Some* (*t*, *unat* (*ptr-val q* – *ptr-val p*)) **and**
 t : *export-uinfo* t = *typ-uinfo-t* *TYPE('b)*
using *field-lookup-uinfo-Some-rev* **by** *blast*

from fl'
have *ptr-val-q*: *ptr-val q* = $\&(p \rightarrow f)$
by (*simp add: field-lvalue-def*)

with $fl' t$ **show** *?thesis*
by (*metis Ptr-ptr-val*)

qed

next

assume $\exists f t n$.
field-lookup (*typ-info-t* *TYPE('a)*) *f 0* = *Some* (*t*, *n*) \wedge *export-uinfo* t =
typ-uinfo-t *TYPE('b)* \wedge
 $q = PTR('b) \ \&(p \rightarrow f)$
then show $d \models_t q$
using *valid-p*
using *c-guard-mono h-t-valid-mono* **by** *blast*

qed

lemma *field-lookup-intvl-contained-left*:

fixes $p::'a::mem\text{-}type\ ptr$
assumes fl : *field-lookup* (*typ-info-t* *TYPE('a)*) *f 0* = *Some* (*t*, *k*)
assumes n : $n = size\text{-}td\ t$
assumes m : $m = size\text{-}of\ TYPE('a)$
shows $\{\&(p \rightarrow f)..\ + n\} \cap \{ptr\text{-}val\ p..\ + m\} \neq \{\}$

proof –

from *field-tag-sub* [*OF fl*] **have** $\{\&(p \rightarrow f)..\ + size\text{-}td\ t\} \subseteq ptr\text{-}span\ p$.
moreover from fl **have** $0 < size\text{-}td\ t$
by (*simp add: field-lookup-wf-size-desc-gt*)
ultimately show *?thesis* **using** $n\ m$
by (*simp add: intvl-non-zero-non-empty le-iff-inf*)

qed

lemma *field-lookup-intvl-contained-right*:

fixes $p::'a::mem\text{-}type\ ptr$
assumes fl : *field-lookup* (*typ-info-t* *TYPE('a)*) *f 0* = *Some* (*t*, *k*)
assumes n : $n = size\text{-}td\ t$
assumes m : $m = size\text{-}of\ TYPE('a)$
shows $\{ptr\text{-}val\ p..\ + m\} \cap \{\&(p \rightarrow f)..\ + n\} \neq \{\}$
using *field-lookup-intvl-contained-left* [*OF fl n m*] **by** *blast*

lemma *field-overlap-right*:

fixes $p::'a::mem\text{-}type\ ptr$
assumes *field-lookup*: *field-lookup* (*typ-info-t* *TYPE('a)*) *path 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo* t = *typ-uinfo-t* *TYPE('f)*
shows $(ptr\text{-}span\ p) \cap ptr\text{-}span\ (PTR('f::c\text{-}type) \ \&(p \rightarrow path)) \neq \{\}$


```

using field-lookup match
by (simp add: export-size-of field-lookup-intvl-contained-right)

locale nested-field' =
  fixes t    :: 'a::mem-type atyp-info
  fixes path :: string list
  fixes sel  :: 'a::mem-type ⇒ 'f::mem-type
  fixes upd  :: 'f ⇒ 'a ⇒ 'a

  assumes field-ti: field-ti TYPE('a) path = Some t
  assumes field-typ-match: export-uinfo t = typ-uinfo-t TYPE('f)

  assumes sel-def: sel ≡ from-bytes o access-ti0 t
  assumes upd-def: upd ≡ update-ti t o to-bytes-p
begin

lemma sub-typ: TYPE('f) ≤τ TYPE('a)
  using field-ti field-typ-match
  using field-ti-sub-typ sub-typ-def by blast

lemma h-val-field:
  shows h-val h (PTR('f) &(p→path)) = sel (h-val h p)
  using TypHeap.h-val-field-from-bytes' field-ti field-typ-match sel-def typ-uinfo-t-def
  by fastforce

lemma clift-field-update:
  assumes typed: hrs-htd h ⊨t p
  shows clift (hrs-mem-update (heap-update (PTR('f) &(p→path)) x) h) =
    (clift h)(p→ upd x (h-val (hrs-mem h) p))
proof –
  from h-t-valid-clift-Some-iff typed obtain v where clift: clift h p = Some v by
blast
  from h-val-clift' [OF this]
  have v: v = h-val (hrs-mem h) p by simp
  from clift-field-update [OF field-ti - clift [simplified v], where val=x, OF field-typ-match
[simplified typ-uinfo-t-def]]
  show ?thesis
  by (simp add: upd-def export-size-of field-typ-match update-ti-t-def)
qed

lemma clift-field-update-padding:
  assumes typed: hrs-htd h ⊨t p
  assumes lbs: length bs = size-of TYPE('f)
  shows clift (hrs-mem-update (heap-update-padding (PTR('f) &(p→path)) x bs)
h) =
    (clift h)(p→ upd x (h-val (hrs-mem h) p))
proof –
  from h-t-valid-clift-Some-iff typed obtain v where clift: clift h p = Some v by
blast

```

```

from h-val-clift' [OF this]
have v: v = h-val (hrs-mem h) p by simp
from clift-field-update-padding [OF field-ti - clift [simplified v] lbs, where val=x, OF
field-typ-match [simplified typ-uinfo-t-def]]
show ?thesis
  by (simp add: upd-def export-size-of field-typ-match update-ti-t-def)
qed

```

```

lemma h-val-field-lvalue-update:  $d \models_t p \implies d \models_t q \implies$ 
h-val (heap-update (PTR('f) &(p→path)) x h) q = ((h-val h)(p := upd x (h-val
h p))) q
by (smt (verit, best) c-type-class.lift-def clift-Some-eq-valid h-val-clift'
hrs-htd-def hrs-mem-def hrs-mem-update lift-heap-update lift-t-heap-update
nested-field'.clift-field-update nested-field'-axioms prod.sel(1) prod.sel(2))

```

```

lemma h-val-field-lvalue-update-padding:  $d \models_t p \implies d \models_t q \implies$  length bs = size-of
TYPE('f)  $\implies$ 
h-val (heap-update-padding (PTR('f) &(p→path)) x bs h) q = ((h-val h)(p := upd
x (h-val h p))) q
by (smt (verit, ccfv-threshold) clift-Some-eq-valid fst-conv h-val-clift hrs-htd-def
hrs-htd-mem-update hrs-mem-def hrs-mem-update clift-field-update-padding
clift-field-update h-val-field-lvalue-update prod.collapse snd-conv)

```

end

```

lemma align-addr-card:
  assumes wf-size-desc: wf-size-desc t
  assumes max-size: size-td t < addr-card
  assumes align-size-of:  $2^{\wedge}$  align-td t dvd size-td t
  shows  $2^{\wedge}$  align-td t dvd addr-card
proof –
  from wf-size-desc have sz-nzero:  $0 < \text{size-td } t$ 
    by (simp add: wf-size-desc-gt(1))
  with align-size-of max-size
  have align-bound: align-td t < addr-bitsize
    by (metis addr-card linorder-not-le nat-dvd-not-less power-le-dvd)
  then have bound:  $2^{\wedge}$  align-td t < addr-card
    by (metis addr-card one-less-numeral-iff power-strict-increasing semiring-norm(76))
  from bound align-bound
  show ?thesis
    apply (clarsimp simp: dvd-def)
    apply (rule exI[where  $x=2^{\wedge}(\text{len-of } \text{TYPE}(\text{addr-bitsize}) - \text{align-td } t)$ ])
    apply clarsimp
    apply (simp add: addr-card flip: power-add)
  done
qed

```

```

lemma ptr-aligned-u-field-lookup:
  assumes wf-size-desc: wf-size-desc t
  assumes wf-align: wf-align t
  assumes align-field: align-field t
  assumes max-size: size-td t < addr-card
  assumes align-size-of: 2 ^ align-td t dvd size-td t
  assumes align-size-of-u: 2 ^ align-td u dvd size-td u — does not follow from a
well-formedness condition on t
  assumes fl: field-lookup t path 0 = Some (u, n)
  assumes ptr-aligned-u: ptr-aligned-u t a
  shows ptr-aligned-u u (a + of-nat n)
proof -

  from wf-size-desc fl have wf-size-desc-u: wf-size-desc u
    using field-lookup-wf-size-desc-pres(1) by blast

  from fl have u-n-bound: size-td u + n ≤ size-td t
    by (simp add: field-lookup-offset-size)
  with max-size have n-bound: n < addr-card
    by simp
  from u-n-bound max-size have sz-u: size-td u < addr-card
    by simp

  from align-td-field-lookup(1) [OF wf-align fl] have align-u-t: align-td u ≤ align-td
t .
  from align-field fl
  have 2 ^ (align-td u) dvd n by (simp add: align-field-def)
  moreover
  from ptr-aligned-u have 2 ^ align-td t dvd unat a by (simp add: ptr-aligned-u-def)

  ultimately have 2 ^ align-td u dvd unat (a + word-of-nat n)
  apply -
  apply(subst unat-word-ariths)
  apply(rule dvd-mod)
  apply(rule dvd-add)
  subgoal
    using align-u-t
    using power-le-dvd by blast
  subgoal
    apply(subst unat-of-nat)
    apply(subst Euclidean-Rings.mod-less)
    apply(subst len-of-addr-card)
    apply (rule n-bound)
    apply assumption
  done
  subgoal

```

```

    apply(subst len-of-addr-card)
    apply (rule align-addr-card)
      apply (rule wf-size-desc-u)
      apply (rule sz-u)
    apply (rule align-size-of-u)
  done
done

then show ?thesis
  by (simp add: ptr-aligned-u-def)
qed

```

```

lemma c-guard-u-field-loop:
  assumes wf-size-desc: wf-size-desc t
  assumes wf-align: wf-align t
  assumes align-field: align-field t
  assumes max-size: size-td t < addr-card
  assumes align-size-of: 2 ^ align-td t dvd size-td t
  assumes align-size-of-u: 2 ^ align-td u dvd size-td u — does not follow from a
well-formedness condition on t
  assumes fl: field-lookup t path 0 = Some (u, n)
  assumes c-guard-u: c-guard-u t a
  shows c-guard-u u (a + of-nat n)
proof —
  from wf-size-desc have sz-t: 0 < size-td t
    using wf-size-desc-gt(1) by blast
  from wf-size-desc fl have sz-u: 0 < size-td u
    by (simp add: field-lookup-wf-size-desc-gt)
  show ?thesis
    using fl c-guard-u
    apply (clarsimp simp add: c-guard-u-def)
    apply safe
    subgoal by (rule ptr-aligned-u-field-lookup [OF wf-size-desc wf-align align-field
max-size align-size-of align-size-of-u fl])
    subgoal by (meson c-null-guard-u-def field-lookup-offset-size' intvl-le subsetD)
  done
qed

```

```

lemma cvalid-u-field-lookup:
  assumes wf-size-desc: wf-size-desc t
  assumes wf-align: wf-align t
  assumes align-field: align-field t
  assumes max-size: size-td t < addr-card
  assumes align-size-of: 2 ^ align-td t dvd size-td t
  assumes align-size-of-u: 2 ^ align-td u dvd size-td u — does not follow from a
well-formedness condition on t
  assumes fl: field-lookup t path 0 = Some (u, n)
  assumes cvalid-u: cvalid-u t d a

```

```

shows cvalid-u u d (a + of-nat n)
proof -
  from wf-size-desc have sz-t: 0 < size-td t
    using wf-size-desc-gt(1) by blast
  from wf-size-desc fl have sz-u: 0 < size-td u
    by (simp add: field-lookup-wf-size-desc-gt)

  from fl have u-n-bound: size-td u + n ≤ size-td t
    by (simp add: field-lookup-offset-size^)
  with max-size have n-bound: n < addr-card
    by simp
  from u-n-bound max-size have sz-u-addr-card: size-td u < addr-card
    by simp

  note c-guard = c-guard-u-field-loopup [OF wf-size-desc wf-align align-field max-size
align-size-of align-size-of-u fl]

  from cvalid-u
  show ?thesis
    apply (clarsimp simp: cvalid-u-def valid-footprint-def Let-def sz-t sz-u c-guard)
    subgoal for y
      apply (drule-tac x=n+y in spec)
      apply (erule impE)
      subgoal using u-n-bound by simp
      subgoal
        by (metis (mono-tags, lifting) ab-semigroup-add-class.add-ac(1)
          fl map-list-map-trans of-nat-add td-set-field-lookupD typ-slice-td-set)
      done
    done
  done
qed

locale mem-type-u =
  fixes t::typ-uinfo
  assumes wf-desc: wf-desc t
  assumes wf-size-desc: wf-size-desc t
  assumes wf-align: wf-align t
  assumes align-field: align-field t
  assumes max-size: size-td t < addr-card
  assumes align-size: 2 ^ align-td t dvd size-td t
begin
lemmas ptr-aligned-u-field-lookup = ptr-aligned-u-field-lookup [OF wf-size-desc wf-align
align-field max-size align-size]
lemmas c-guard-u-field-loopup = c-guard-u-field-loopup [OF wf-size-desc wf-align align-field
max-size align-size]
lemmas cvalid-u-field-lookup = cvalid-u-field-lookup [OF wf-size-desc wf-align align-field
max-size align-size]
end

```

```

lemma wf-size-desc-export-uinfo:
  fixes t::('a, 'b) typ-info
  and st::('a, 'b) typ-info-struct
  and ts::('a, 'b) typ-info-tuple list
  and x::('a, 'b) typ-info-tuple
shows
  wf-size-desc t  $\implies$  wf-size-desc (export-uinfo t) and
  wf-size-desc-struct st  $\implies$  wf-size-desc-struct ( map-td-struct field-norm ( $\lambda$ -. ()))
st) and
  wf-size-desc-list ts  $\implies$  wf-size-desc-list ( map-td-list field-norm ( $\lambda$ -. ())) ts) and
  wf-size-desc-tuple x  $\implies$  wf-size-desc-tuple ( map-td-tuple field-norm ( $\lambda$ -. ())) x)
proof (induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case by auto
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case by auto
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by (auto simp add: export-uinfo-def)
qed

```

```

context mem-type
begin

```

```

lemma typ-uinfo-t-mem-type[simp]: mem-type-u (typ-uinfo-t(TYPE('a)))
  apply (simp only: typ-uinfo-t-def)
  apply (unfold-locales)
  apply (simp add: typ-uinfo-t-def wf-desc-export-uinfo-pres(1))
  apply (simp add: wf-size-desc-export-uinfo)
  apply (metis export-uinfo-tydesc-simp local.wf-align typ-desc.exhaust wf-align-map-td(1))
  using local.align-field local.align-field-uinfo local.typ-uinfo-t-def apply fastforce
  apply (simp add: size-of-fold)
  using align-size-of
  apply (simp add: size-of-def align-of-def)
  done

```

```

sublocale u: mem-type-u typ-uinfo-t(TYPE('a))

```

```

    by (rule typ-winfo-t-mem-type)

end

lemma field-names-u-composeI:
  assumes wf-t: wf-desc t
  assumes path1: path1 ∈ set (field-names-u t u)
  assumes path2: path2 ∈ set (field-names-u u v)
  shows path1 @ path2 ∈ set (field-names-u t v)
  proof -
    from path1 wf-t obtain n where fl1: field-lookup t path1 0 = Some (u, n)
      using field-names-u-filter-all-field-names-conv(1) by fastforce
    with wf-t have wf-u: wf-desc u
      using field-lookup-wf-desc-pres(1) by blast
    from path2 wf-u obtain m where fl2: field-lookup u path2 0 = Some (v, m)
      using field-names-u-filter-all-field-names-conv(1) by fastforce
    hence fl2: field-lookup u path2 n = Some (v, n + m)
      by (simp add: field-lookup-offset)
    from field-lookup-prefix-Some''(1)[rule-format, OF fl1 wf-t, of path2] fl2
    have field-lookup t (path1 @ path2) 0 = Some (v, n + m) by simp
    then
    show ?thesis
      by (simp add: field-lookup-all-field-names(1) field-names-u-filter-all-field-names-conv(1)
        wf-t)
  qed

```

```

lemma field-names-u-composeD:
  assumes wf-t: wf-desc t
  assumes append: path1 @ path2 ∈ set (field-names-u t v)
  shows ∃ u. path1 ∈ set (field-names-u t u) ∧ path2 ∈ set (field-names-u u v)
  proof -
    from append wf-t obtain n where fl1: field-lookup t (path1 @ path2) 0 = Some
      (v, n)
      using field-names-u-filter-all-field-names-conv(1) by fastforce
    from fl1
    obtain u m where
      path1: field-lookup t path1 0 = Some (u, m) and path2: field-lookup u path2 m
    = Some (v, n)
      using field-lookup-append-Some wf-t by blast
    thus ?thesis
      by (smt (verit, ccfv-threshold) all-field-names-exists-field-names-u(1)
        field-lookup-all-field-names(1) field-names-u-composeI field-names-u-filter-all-field-names-conv(1)
        local.fl1 mem-Collect-eq option.inject prod.inject set-filter wf-t)
  qed

```

lemma *field-names-u-field-offset-untyped-append*:
assumes *wf-desc root*
assumes *path1*: $path1 \in set (field-names-u\ root\ outer)$
assumes *path2*: $path2 \in set (field-names-u\ outer\ inner)$
shows $field-offset-untyped\ root\ (path1\ @\ path2) = field-offset-untyped\ root\ path1$
 $+ field-offset-untyped\ outer\ path2$
by (*smt (verit, ccfv-threshold) assms(1) field-lookup-offset field-lookup-prefix-Some''(1)*
field-lookup-wf-desc-pres(1) field-names-u-filter-all-field-names-conv(1) field-offset-untyped-def
mem-Collect-eq option.sel path1 path2 prod.sel(2) set-filter)

lemma *root-ptr-valid-u-cases*:
assumes *root-ptr-valid-u t1 d a1*
assumes *root-ptr-valid-u t2 d a2*
shows $(t1 = t2 \wedge a1 = a2) \vee (\{a1\} ..+ size-td\ t1 \cap \{a2\} ..+ size-td\ t2) = \{\}$
apply (*rule valid-root-footprints-cases*)
using *assms*
by (*auto simp add: root-ptr-valid-u-def*)

lemma *field-offset-untyped-empty[simp]*: $field-offset-untyped\ t\ [] = 0$
by (*simp add: field-offset-untyped-def*)

lemma *field-names-u-refl[simp]*: $field-names-u\ t\ t = []$
by (*cases t*) *auto*

lemma *field-names-u-size-td-bounds*:
assumes *wf-t: wf-desc t*
assumes *path*: $path \in set (field-names-u\ t\ u)$
shows $field-offset-untyped\ t\ path + size-td\ u \leq size-td\ t$
proof –
from *wf-t path* **obtain** *n* **where** *fl*: $field-lookup\ t\ path\ 0 = Some\ (u,\ n)$
using *field-names-u-filter-all-field-names-conv(1)* **by** *auto*

hence n : $field-offset-untyped\ t\ path = n$
by (*simp add: field-offset-untyped-def*)
from *field-lookup-offset-size' [OF fl] n*
show *?thesis* **by** *simp*
qed

lemma *field-lookup-path-cases*:
assumes *fl1*: $field-lookup\ t\ path\ 0 = Some\ (u,\ n)$
shows $(t = u \wedge path = []) \vee ((t \neq u) \wedge path \neq [])$
using *field-lookup-same-type-empty(1) local.fl1* **by** *blast*

lemma *field-lookup-cycle-cases*:
assumes *fl1*: $field-lookup\ t\ path1\ 0 = Some\ (u,\ n1)$
assumes *fl2*: $field-lookup\ u\ path2\ 0 = Some\ (v,\ n2)$
shows $(t = v \wedge u = v \wedge path1 = [] \wedge path2 = [] \wedge n1 = 0 \wedge n2 = 0)$ **by** *blast*

$(t \neq v)$
using *field-lookup-path-cases* [OF fl1] *field-lookup-path-cases* [OF fl2] fl1 fl2
by (*metis add.right-neutral diff-add-0 nat-less-le ordered-cancel-comm-monoid-diff-class.add-diff-inverse td-set-field-lookupD td-set-size-lte'(1)*)

lemma *field-lookup-refl-iff*: $\text{field-lookup } t \ p \ n = \text{Some } (t, m) \longleftrightarrow p = [] \wedge n = m$

proof

assume fl: *field-lookup* t p n = Some (t, m)

note CTypes.*field-lookup-offset2*[OF this]

from *field-lookup-cycle-cases*[OF this this] fl **show** $p = [] \wedge n = m$

by *auto*

qed *auto*

lemma *valid-root-footprint-contained-sub-ty*:

assumes *valid-root-x*: *valid-root-footprint* d x t

assumes *valid-y*: *valid-footprint* d y s

assumes *contained*: $\{y \text{ ..+ size-td } s\} \subseteq \{x \text{ ..+ size-td } t\}$

shows $s \leq t$

proof –

from *valid-y* **have** $0 < \text{size-td } s$ **by** (*simp add: valid-footprint-def Let-def*)

hence $\{y \text{ ..+ size-td } s\} \neq \{\}$

by (*simp add: intvl-non-zero-non-empty*)

with *contained* **have** $\{x \text{ ..+ size-td } t\} \cap \{y \text{ ..+ size-td } s\} \neq \{\}$ **by** *blast*

from *valid-root-footprint-overlap-sub-ty* [OF *valid-root-x valid-y this*]

show *?thesis*.

qed

lemma *c-null-guard-u-no-overflow*: $c\text{-null-guard-u } t \ a \implies \text{unat } a + \text{size-td } t \leq \text{addr-card}$

unfolding *c-null-guard-u-def addr-card-def*

using *zero-not-in-intvl-no-overflow*

by (*metis card-word*)

lemma *c-null-guard-u-size-td-limit*: $c\text{-null-guard-u } t \ a \implies \text{size-td } t < \text{addr-card}$

unfolding *c-null-guard-u-def addr-card-def*

by (*metis (no-types, lifting) Abs-fnat-hom-0 add-diff-cancel-right' add-leE cancel-comm-monoid-add-class.diff-cancel*

card-word first-in-intvl linorder-not-less nat-less-le unat-eq-of-nat zero-less-card-finite zero-not-in-intvl-no-overflow)

lemma *c-null-guard-u-intvl-nat-conv*:

assumes *c-null-guard*: *c-null-guard-u* t a

shows $\{a \text{ ..+ size-td } t\} = \{x. (\text{unat } a \leq \text{unat } x \wedge \text{unat } x < (\text{unat } a + \text{size-td } t))\}$

using *intvl-no-overflow-nat-conv* [OF *c-null-guard-u-no-overflow* [OF *c-null-guard*]]

by *simp*

lemma *valid-footprint-overlap-sub-ty*:

assumes *valid-x*: *valid-footprint* d x t

assumes *valid-y*: *valid-footprint* *d y s*
assumes *overlap*: $\{x \text{ ..+ size-td } t\} \cap \{y \text{ ..+ size-td } s\} \neq \{\}$
shows $s \leq t \vee t \leq s$
by (*meson field-of-sub order-less-imp-le overlap*
valid-footprint-neq-disjoint valid-x valid-y)

lemma *valid-footprint-overlap-sub-typ-cases* [*consumes 3, case-names eq less1 less2*]:

assumes *valid-x*: *valid-footprint* *d x t*
assumes *valid-y*: *valid-footprint* *d y s*
assumes *overlap*: $\{x \text{ ..+ size-td } t\} \cap \{y \text{ ..+ size-td } s\} \neq \{\}$
assumes *eq*: $s = t \implies P$
assumes *less-s-t*: $s < t \implies P$
assumes *less-t-s*: $t < s \implies P$
shows *P*
using *valid-footprint-overlap-sub-typ* [*OF valid-x valid-y overlap*] *eq less-s-t less-t-s*
by *fastforce*

lemma *valid-footprint-contained-sub-typ*:

assumes *valid-x*: *valid-footprint* *d x t*
assumes *valid-y*: *valid-footprint* *d y s*
assumes *contained*: $\{x \text{ ..+ size-td } t\} \subseteq \{y \text{ ..+ size-td } s\}$
shows $s \leq t \vee t \leq s$
by (*metis intvl-non-zero-non-empty le-iff-inf contained valid-footprint-def*
valid-footprint-overlap-sub-typ valid-x valid-y)

lemma *valid-footprint-contained-sub-typ-cases*:

assumes *valid-x*: *valid-footprint* *d x t*
assumes *valid-y*: *valid-footprint* *d y s*
assumes *contained*: $\{x \text{ ..+ size-td } t\} \subseteq \{y \text{ ..+ size-td } s\}$
assumes *eq*: $s = t \implies P$
assumes *less-s-t*: $s < t \implies P$
assumes *less-t-s*: $t < s \implies P$
shows *P*
using *valid-footprint-contained-sub-typ* [*OF valid-x valid-y contained*] *eq less-s-t less-t-s*
by *fastforce*

lemma *all-field-names-field-lookup'*:

fixes *t*::('a, 'b) *typ-desc*
and *st*::('a, 'b) *typ-struct*
and *ts*::('a, 'b) *typ-tuple list*
and *x*::('a, 'b) *typ-tuple*
shows
wf-desc t $\implies f \in \text{set } (all\text{-field-names } t) \implies \exists u n. \text{field-lookup } t f 0 = \text{Some } (u, n)$ **and**
wf-desc-struct st $\implies f \in \text{set } (all\text{-field-names-struct } st) \implies \exists u n. \text{field-lookup-struct } st f 0 = \text{Some } (u, n)$ **and**
wf-desc-list ts $\implies f \in \text{set } (all\text{-field-names-list } ts) \implies \exists u n. \text{field-lookup-list } ts f$

$0 = \text{Some } (u, n)$ **and**
 $\text{wf-desc-tuple } x \implies f \in \text{set } (\text{all-field-names-tuple } x) \implies \exists u n. \text{field-lookup-tuple } x f 0 = \text{Some } (u, n)$
proof (*induct t and st and ts and x arbitrary: f and f and f and f*)
case (*TypDesc nat typ-struct list*)
then show *?case by auto*
next
case (*TypScalar nat1 nat2 a*)
then show *?case by auto*
next
case (*TypAggregate list*)
then show *?case by auto*
next
case *Nil-typ-desc*
then show *?case by auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case apply (cases dt-tuple)*
apply (*clarsimp split: option.splits simp add: all-field-names-list-conv*)
by (*metis field-lookup-list-offset-None not-Some-eq-tuple*)
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case by auto*
qed

lemma *valid-foot-print-intvl-self: valid-footprint d a t $\implies a \in \{a \dots + \text{size-td } t\}$*
by (*simp add: valid-footprint-def Let-def intvl-self*)

lemma *field-lookup-Some-field-names-u:*

fixes $t :: \text{typ-uinfo}$ **and**
 $st :: \text{typ-uinfo-struct}$ **and**
 $ts :: \text{typ-uinfo-tuple list}$ **and**
 $x :: \text{typ-uinfo-tuple}$
shows
 $\text{field-lookup } t f n = \text{Some } (s, m) \implies f \in \text{set } (\text{field-names-u } t s)$
 $\text{field-lookup-struct } st f n = \text{Some } (s, m) \implies f \in \text{set } (\text{field-names-struct-u } st s)$
 $\text{field-lookup-list } ts f n = \text{Some } (s, m) \implies f \in \text{set } (\text{field-names-list-u } ts s)$
 $\text{field-lookup-tuple } x f n = \text{Some } (s, m) \implies f \in \text{set } (\text{field-names-tuple-u } x s)$
proof (*induct t and st and ts and x arbitrary: f n m and f n m and f n m*
and f n m)
case (*TypDesc nat typ-struct list*)
then show *?case*
apply *clarsimp*
by (*metis TypDesc.premis field-lookup-same-type-empty(1)*)
next
case (*TypScalar nat1 nat2 a*)
then show *?case by auto*
next
case (*TypAggregate list*)

```

then show ?case by auto
next
case Nil-typ-desc
then show ?case by auto
next
case (Cons-typ-desc dt-tuple list)
then show ?case
  by (auto split: option.splits)
next
case (DTuple-typ-desc typ-desc list b)
then show ?case apply clarsimp
  by (metis imageI list.exhaust-sel option.distinct(1))
qed

```

lemma *field-lookup-non-prefix-disj'*:

```

assumes wf: mem-type-u t
assumes f: field-lookup t f 0 = Some (tf, n)
assumes g: field-lookup t g 0 = Some (tg, m)
assumes f-g: disj-fn f g
shows disjnt {(of-nat n::addr) ..+ size-td tf} {(of-nat m ..+ size-td tg)}
using mem-type-u.max-size[OF wf] f-g
using field-lookup-offset-size'[OF f]
using field-lookup-offset-size'[OF g]
using field-lookup-non-prefix-disj[OF wf[THEN mem-type-u.wf-desc] f g]
unfolding intvl-eq-of-nat-Ico-add
apply (subst disjnt-of-nat)
unfolding len-of-addr-card
by (auto simp add: disjnt-def disj-fn-def less-eq-list-def)

```

lemma *sub-typ-stack-byte-u*:

```

t ≤ typ-uinfo-t (TYPE(stack-byte)) ⇒ t = typ-uinfo-t TYPE(stack-byte)
by (simp add: sub-typ-def typ-uinfo-t-def typ-info-stack-byte typ-tag-le-def)

```

lemma *root-ptr-valid-not-subtype-disjoint-u*:

```

[[ root-ptr-valid d (p::'a::mem-type ptr);
  cvalid-u t d q;
  ¬ t ≤ typ-uinfo-t TYPE('a) ]] ⇒
ptr-span p ∩ {q ..+ size-td t} = {}
by (metis cvalid-u-def root-ptr-valid-valid-root-footprint size-of-fold
  typ-uinfo-size valid-root-footprint-overlap-sub-typ)

```

lemma *stack-allocs-disjoint-u*:

```

assumes stack-alloc: (p, d') ∈ stack-allocs n S (TYPE('a::mem-type)) d
assumes no-stack: t ≠ typ-uinfo-t (TYPE(stack-byte))
assumes typed: cvalid-u t d q
shows {ptr-val p ..+ n * size-of TYPE('a)} ∩ {q ..+ size-td t} = {}
proof -

```

```

from stack-alloc obtain
  typ-uinfo-t (TYPE('a))  $\neq$  typ-uinfo-t (TYPE(stack-byte)) and
  stack-bytes:  $\forall a \in \{ptr\text{-val } p \text{ ..} + n * \text{size-of } TYPE('a)\}$ . root-ptr-valid d (PTR
(stack-byte) a)
  by (cases rule: stack-allocs-cases) auto
from no-stack have no-sub-typ:  $\neg t \leq \text{typ-uinfo-t } (TYPE(\text{stack-byte}))$  by (metis
sub-typ-stack-byte-u)
  {
    fix a
    assume a:  $a \in \{ptr\text{-val } p \text{ ..} + n * \text{size-of } TYPE('a)\}$ 
    have  $a \notin \{q \text{ ..} + \text{size-td } t\}$ 
    proof -
    from stack-bytes [rule-format, OF a] have root-ptr-valid d (PTR (stack-byte)
a) .
  
```

```

    from root-ptr-valid-not-subtype-disjoint-u [OF this typed no-sub-typ] show
?thesis
    by (simp add: disjoint-iff first-in-intvl)
    qed
  }
then show  $\{ptr\text{-val } p \text{ ..} + n * \text{size-of } TYPE('a)\} \cap \{q \text{ ..} + \text{size-td } t\} = \{\}$ 
by auto
qed

```

```

lemma fold-ptr-retyp-disjoint-u:
  fixes p::'a::mem-type ptr
  shows  $\llbracket \text{cvalid-u } t \text{ } d \text{ } q; \{ptr\text{-val } p \text{ ..} + n * \text{size-of } TYPE('a)\} \cap \{q \text{ ..} + \text{size-td } t\} = \{\} \rrbracket \implies$ 
    cvalid-u t (fold ( $\lambda i. ptr\text{-retyp } (p +_p \text{int } i)$ ) [ $0..<n$ ] d) q
  apply(clarsimp simp: cvalid-u-def)
  apply(erule fold-ptr-retyp-valid-footprint-disjoint)
  apply(fastforce simp: size-of-def)
  done

```

```

lemma fold-ptr-force-type-disjoint-u:
  fixes p::'a::mem-type ptr
  shows  $\llbracket \text{cvalid-u } t \text{ } d \text{ } q; \{ptr\text{-val } p \text{ ..} + n * \text{size-of } TYPE('a)\} \cap \{q \text{ ..} + \text{size-td } t\} = \{\} \rrbracket \implies$ 
    cvalid-u t (fold ( $\lambda i. ptr\text{-force-type } (p +_p \text{int } i)$ ) [ $0..<n$ ] d) q
  apply(clarsimp simp: cvalid-u-def)
  apply(erule fold-ptr-force-type-valid-footprint-disjoint)
  apply(fastforce simp: size-of-def)
  done

```

```

lemma fold-ptr-retyp-disjoint2-u:
  fixes p::'a::mem-type ptr
  assumes no-overflow:  $0 \notin \{ptr\text{-val } p \text{ ..} + n * \text{size-of } TYPE('a)\}$ 
  shows  $\llbracket \text{cvalid-u } t \text{ } (fold } (\lambda i. ptr\text{-retyp } (p +_p \text{int } i)) [0..<n] \text{ } d) \text{ } q; \{ptr\text{-val } p \text{ ..} + n * \text{size-of } TYPE('a)\} \cap \{q \text{ ..} + \text{size-td } t\} = \{\} \rrbracket$ 

```

```

  => cvalid-u t d q
apply(clarsimp simp: cvalid-u-def)
apply(erule fold-ptr-retyp-valid-footprint-disjoint2 [OF no-overflow])
apply(simp add: size-of-def)
apply fast
done

```

```

lemma fold-ptr-force-type-disjoint2-u:
  fixes p::'a::mem-type ptr
  assumes no-overflow: 0 ∉ {ptr-val p..+ n * size-of TYPE('a)}
  shows  $\llbracket$ cvalid-u t (fold (λi. ptr-force-type (p +p int i)) [0..n] d) q;
    {ptr-val p..+ n * size-of TYPE('a)} ∩ {q ..+ size-td t} = {}  $\rrbracket$ 
  => cvalid-u t d q
apply(clarsimp simp: cvalid-u-def)
apply(erule fold-ptr-force-type-valid-footprint-disjoint2 [OF no-overflow])
apply(simp add: size-of-def)
apply fast
done

```

```

lemma fold-ptr-retyp-disjoint-iff-u:
  fixes p::'a::mem-type ptr
  assumes no-overflow: 0 ∉ {ptr-val p..+ n * size-of TYPE('a)}
  shows {ptr-val p..+ n * size-of TYPE('a)} ∩ {q ..+ size-td t} = {}
  => cvalid-u t (fold (λi. ptr-retyp (p +p int i)) [0..n] d) q = cvalid-u t d q
apply standard
  apply (erule (1) fold-ptr-retyp-disjoint2-u [OF no-overflow])
  apply (erule (1) fold-ptr-retyp-disjoint-u)
done

```

```

lemma fold-ptr-force-type-disjoint-iff-u:
  fixes p::'a::mem-type ptr
  assumes no-overflow: 0 ∉ {ptr-val p..+ n * size-of TYPE('a)}
  shows {ptr-val p..+ n * size-of TYPE('a)} ∩ {q ..+ size-td t} = {}
  => cvalid-u t (fold (λi. ptr-force-type (p +p int i)) [0..n] d) q = cvalid-u t d q
apply standard
  apply (erule (1) fold-ptr-force-type-disjoint2-u [OF no-overflow])
  apply (erule (1) fold-ptr-force-type-disjoint-u)
done

```

```

lemma stack-allocs-preserves-typing-u:
  assumes stack-alloc: (p, d') ∈ stack-allocs n S (TYPE('a::mem-type)) d
  assumes no-stack: t ≠ typ-uinfo-t (TYPE(stack-byte))
  assumes typed: cvalid-u t d q
  shows cvalid-u t d' q
proof –
  from stack-alloc obtain
    d': d' = fold (λi. ptr-force-type (p +p int i)) [0..n] d and
    no-overflow: 0 ∉ {ptr-val p ..+ n * size-of TYPE('a)}
  by (cases rule: stack-allocs-cases) auto

```

from *stack-allocs-disjoint-u* [*OF stack-alloc no-stack typed*]
have $\{ptr\text{-val } p..+n * \text{size-of } TYPE('a)\} \cap \{q..+size\text{-td } t\} = \{\}$.
from *fold-ptr-force-type-disjoint-iff-u* [*OF no-overflow this, where $d=d$*] *typed*
show *?thesis*
by (*simp add: d'*)
qed

lemma *stack-allocs-root-ptr-valid-u-new-cases*:

assumes *stack-alloc*: $(p, d') \in \text{stack-allocs } n \mathcal{S} (TYPE('a::\text{mem-type})) d$
assumes *valid*: *root-ptr-valid-u* $t d' q$
shows $(\exists i < n. q = ptr\text{-val } (p +_p \text{int } i) \wedge t = typ\text{-uinfo-t } (TYPE('a))) \vee$
root-ptr-valid-u $t d q$
proof (*cases* $\{ptr\text{-val } p ..+ n * \text{size-of } TYPE('a)\} \cap \{q ..+ size\text{-td } t\} = \{\}$)
case *True*
with *stack-alloc valid* **show** *?thesis*
by (*smt (verit) disjoint-iff fold-ptr-force-type-other intvlI root-ptr-valid-u-def*
stack-allocs-cases valid-root-footprint-def)
next
case *False*
with *stack-alloc array-to-index-span stack-alloc valid* **show** *?thesis*
by (*smt (verit, ccfv-SIG) disjoint-iff root-ptr-valid-root-ptr-valid-u-conv*
root-ptr-valid-u-cases size-of-tag stack-allocs-root-ptr-valid-same-type-cases)
qed

lemma *cvalid-u-c-guard-u*:

cvalid-u $t d a \implies c\text{-guard-u } t a$
by (*simp add: cvalid-u-def*)

lemma *stack-allocs-preserves-root-ptr-valid-u*:

assumes *stack-alloc*: $(p, d') \in \text{stack-allocs } n \mathcal{S} (TYPE('a::\text{mem-type})) d$
assumes *no-stack*: $t \neq typ\text{-uinfo-t } (TYPE(\text{stack-byte}))$
assumes *typed*: *root-ptr-valid-u* $t d q$
shows *root-ptr-valid-u* $t d' q$

proof –

from *stack-alloc* **obtain**

d' : $d' = fold (\lambda i. ptr\text{-force-type } (p +_p \text{int } i)) [0..<n] d$ **and**
no-overflow: $0 \notin \{ptr\text{-val } p ..+ n * \text{size-of } TYPE('a)\}$
by (*cases rule: stack-allocs-cases*) *auto*

from *typed* **have** *typed-q*: *cvalid-u* $t d q$

by (*simp add: root-ptr-valid-u-cvalid-u*)

from *stack-allocs-disjoint-u* [*OF stack-alloc no-stack this*]

have *disj*: $\{ptr\text{-val } p..+n * \text{size-of } TYPE('a)\} \cap \{q..+size\text{-td } t\} = \{\}$.

from *stack-allocs-preserves-typing-u* [*OF stack-alloc no-stack typed-q*]

have *typed'*: *cvalid-u* $t d' q$.

hence *valid-fp*: *valid-footprint* $d' q t$

by (*simp add: cvalid-u-def*)

```

show ?thesis
apply (simp add: root-ptr-valid-u-def valid-root-footprint-valid-footprint-dom-conv
valid-fp
cvalid-u-c-guard-u [OF typed'])
apply (simp add: d')
using disj fold-ptr-force-type-other
by (smt (verit) d' disjoint-iff intvll root-ptr-valid-u-def stack-alloc
stack-allocs-cases typed valid-root-footprint-dom-typing)

```

qed

lemma *stack-allocs-root-ptr-valid-u-other:*

```

assumes stack-alloc: (p, d') ∈ stack-allocs n S (TYPE('a::mem-type)) d
assumes valid-d: root-ptr-valid-u t d q
assumes non-stack: t ≠ typ-uinfo-t (TYPE(stack-byte))
shows root-ptr-valid-u t d' q
proof (cases root-ptr-valid-u t d' q)
case True
then show ?thesis by simp
next
case False
from stack-alloc
show ?thesis
apply (rule stack-allocs-cases)
using False valid-d non-stack
using stack-alloc stack-allocs-preserves-root-ptr-valid-u by blast

```

qed

lemma *stack-allocs-root-ptr-valid-u-same:*

```

assumes stack-alloc: (p, d') ∈ stack-allocs n S (TYPE('a::mem-type)) d
assumes i: i < n
assumes addr-eq: q = ptr-val (p +p int i)
assumes match: t = typ-uinfo-t (TYPE('a))
shows root-ptr-valid-u t d' q
proof (cases root-ptr-valid-u t d' q)
case True
then show ?thesis by simp
next
case False
from stack-alloc have root-ptr-valid d' (p +p int i)
apply (rule stack-allocs-cases)
using i
by auto

```

with addr-eq match **show** ?thesis

```

apply (clarsimp simp add: root-ptr-valid-def root-ptr-valid-u-def)
apply (simp add: c-guard-def c-guard-u-def c-null-guard-def c-null-guard-u-def
ptr-aligned-def ptr-aligned-u-def)
apply (auto simp add: align-of-def align-td-uinfo[symmetric] size-of-def )

```


done
qed

lemma *stack-allocs-root-ptr-valid-u-cases*:

assumes *stack-alloc*: $(p, d') \in \text{stack-allocs } n \ \mathcal{S} \ (\text{TYPE}('a::\text{mem-type})) \ d$
assumes *non-stack-byte*: $t \neq \text{typ-uinfo-t} \ (\text{TYPE}(\text{stack-byte}))$
shows *root-ptr-valid-u* $t \ d' \ q \longleftrightarrow$
 $(\exists i < n. \ q = \text{ptr-val} \ (p \ +_p \ \text{int } i) \wedge t = \text{typ-uinfo-t} \ (\text{TYPE}('a))) \vee$
root-ptr-valid-u $t \ d \ q$

using *stack-alloc non-stack-byte*
stack-allocs-root-ptr-valid-u-new-cases stack-allocs-root-ptr-valid-u-other
stack-allocs-root-ptr-valid-u-same
by *metis*

lemma *stack-releases-root-ptr-valid-u1*:

fixes $p::'a::\text{mem-type} \ \text{ptr}$
assumes *non-stack-p*: $\text{typ-uinfo-t} \ \text{TYPE}('a) \neq \text{typ-uinfo-t} \ \text{TYPE}(\text{stack-byte})$
assumes *non-stack-q*: $t \neq \text{typ-uinfo-t} \ \text{TYPE}(\text{stack-byte})$
assumes *root-q*: *root-ptr-valid-u* $t \ (\text{stack-releases } n \ p \ d) \ q$
shows $\{\text{ptr-val } p \ ..+ \ n * \text{size-of } \text{TYPE}('a::\text{mem-type})\} \cap \{q \ ..+ \ \text{size-td } t\} = \{\}$
 \wedge *root-ptr-valid-u* $t \ d \ q$
apply (*rule context-conjI*)
subgoal
using *assms*
by (*smt (verit, best) cvalid-u-def disjoint-iff in-ptr-span-itself ptr-val.ptr-val-def*
root-ptr-valid-u-cvalid-u size-of-tag stack-releases-valid-root-footprint sub-typ-stack-byte-u
valid-root-footprint-overlap-sub-typ)
subgoal
using *assms*
by (*smt (verit, best) disjoint-iff intvlI root-ptr-valid-u-def stack-releases-other*
valid-root-footprint-def)
done

lemma *stack-releases-root-ptr-valid-u2*:

fixes $p::'a::\text{mem-type} \ \text{ptr}$
assumes *disj*: $\{\text{ptr-val } p \ ..+ \ n * \text{size-of } \text{TYPE}('a::\text{mem-type})\} \cap \{q \ ..+ \ \text{size-td}$
 $t\} = \{\}$
assumes *valid-q*: *root-ptr-valid-u* $t \ d \ q$
shows *root-ptr-valid-u* $t \ (\text{stack-releases } n \ p \ d) \ q$
using *assms*
by (*simp add: intvlI orthD2 root-ptr-valid-u-def stack-releases-other valid-root-footprint-def*)

lemma *stack-release-root-ptr-valid-u2*:

fixes $p::'a::\text{mem-type} \ \text{ptr}$
assumes *disj*: $\text{ptr-span } p \cap \{q \ ..+ \ \text{size-td } t\} = \{\}$
assumes *valid-q*: *root-ptr-valid-u* $t \ d \ q$
shows *root-ptr-valid-u* $t \ (\text{stack-release } p \ d) \ q$
using *assms*

by (*smt* (*verit*, *best*) *disjoint-iff* *intvlI* *root-ptr-valid-u-def* *stack-release-other* *valid-root-footprint-def*)

lemma *stack-releases-root-ptr-valid-u-cases*:

fixes *p*:*'a*::*mem-type ptr*
assumes *non-stack-p*: *typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE(stack-byte)*
assumes *non-stack-q*: *t ≠ typ-uinfo-t TYPE(stack-byte)*
shows *root-ptr-valid-u t (stack-releases n p d) q* \longleftrightarrow
 $\{ptr-val\ p \ ..+ \ n * \ size-of \ TYPE('a::mem-type)\} \cap \{q \ ..+ \ size-td \ t\} = \{\}$ \wedge
root-ptr-valid-u t d q
using *assms stack-releases-root-ptr-valid-u1 stack-releases-root-ptr-valid-u2* **by**
blast

lemma *valid-root-footprint-is-root*:

assumes *wf*: *wf-desc t*
assumes *wf-size*: *wf-size-desc t*
assumes *f*: *field-lookup t path 0 = Some (s, n)*
assumes *footprint-t*: *valid-footprint d a t*
assumes *root-s*: *valid-root-footprint d (a + of-nat n) s*
shows (*t = s* \wedge *path = []*)
proof (*cases path*)
case *Nil*
with *f* **have** *t = s*
by *simp*
with *Nil* **show** *?thesis* **by** *simp*
next
case (*Cons fld path'*)
from *f* **have** *neq*: *t ≠ s*
using *field-lookup-same-type-empty(1) local.Cons* **by** *blast*
from *f* **have** *s-t*: *s ≤ t*
using *td-set-field-lookupD typ-tag-le-def* **by** *blast*
from *neq s-t* **have** $\neg t \leq s$
using *sub-tag-antisym* **by** *blast*
with *valid-root-footprint-overlap-sub-typ [OF root-s footprint-t]*
have $\{a + \ word-of-nat \ n..+size-td \ s\} \cap \{a..+size-td \ t\} = \{\}$
by *blast*
moreover
from *f* **have** *td-set*: $(s,n) \in td-set \ t \ 0$
using *local.wf td-set-field-lookup* **by** *blast*
from *td-set-offset-size [OF td-set]* **have** $\{a + \ word-of-nat \ n..+size-td \ s\} \subseteq$
 $\{a..+size-td \ t\}$
by (*simp add: intvl-le*)
moreover
from *field-lookup-wf-size-desc-gt [OF f wf-size]* **have** $0 < size-td \ s$.
hence $\{a + \ word-of-nat \ n..+size-td \ s\} \neq \{\}$
by (*simp add: intvl-non-zero-non-empty*)
ultimately **have** *False*
by *blast*
then **show** *?thesis* **by** *simp*

qed

corollary *root-ptr-valid-u-is-root*:

assumes *wf*: *wf-desc t*
assumes *wf-size*: *wf-size-desc t*
assumes *f*: *field-lookup t path 0 = Some (s, n)*
assumes *cvalid-u-t*: *cvalid-u t d a*
assumes *root-s*: *root-ptr-valid-u s d (a + of-nat n)*
shows ($t = s \wedge \text{path} = []$)

proof –

from *cvalid-u-t* **have** *footprint-t*: *valid-footprint d a t* **by** (*simp add: cvalid-u-def*)
from *root-s* **have** *valid-root-footprint d (a + of-nat n) s* **by** (*simp add: root-ptr-valid-u-def*)
from *valid-root-footprint-is-root [OF wf wf-size f footprint-t this]* **show** *?thesis* .

qed

lemma *valid-footprint-field-lookup*:

assumes *wf*: *wf-desc t*
assumes *wf-size-desc*: *wf-size-desc t*
assumes *valid-t*: *valid-footprint d a t*
assumes *fl*: *field-lookup t path1 0 = Some (s, n)*
shows *valid-footprint d (a + word-of-nat n) s*
using *valid-t fl wf-size-desc*
apply (*clarsimp simp add: valid-footprint-def Let-def, intro conjI*)
subgoal using *field-lookup-wf-size-desc-gt* **by** *blast*
subgoal by (*smt (verit, ccfv-SIG) Abs-fnat-hom-add add commute add-less-mono1 field-lookup-offset-size' group-cancel.add2 map-list-map-trans order-less-le-trans td-set-field-lookupD typ-slice-td-set*)
done

lemma *in-set-mapD*: $x \in \text{set} (\text{map } f \text{ } xs) \implies \exists y \in \text{set } xs. x = f y$

by (*induct xs*) *auto*

lemma *findSomeI*: $x \in \text{set } xs \implies P x \implies \exists x. \text{find } P \text{ } xs = \text{Some } x$

by (*induct xs*) *auto*

lemma *find-in-set-inj-distinct*:

$\text{inj-on } P (\text{set} (\text{map } \text{fst } xs)) \implies (x, v) \in \text{set } xs \implies P x \implies \text{distinct} (\text{map } \text{fst } xs) \implies$
 $\text{find } (\lambda(x, -). P x) \text{ } xs = \text{Some } (x, v)$
by (*induct xs*) (*auto simp add: injD rev-image-eqI*)

definition *inj-on-true* $P A = (\forall x y. x \in A \longrightarrow y \in A \longrightarrow P x \longrightarrow P y \longrightarrow x = y)$

lemma *inj-on* $P A \implies \text{inj-on-true } P A$

by (*auto simp add: inj-on-def inj-on-true-def*)

lemma *inj-on-trueD*: $\text{inj-on-true } P \ A \implies x \in A \implies y \in A \implies P \ x \implies P \ y \implies x = y$

by (*simp add: inj-on-true-def*)

lemma *inj-on-trueI*: $(\bigwedge x \ y. x \in A \implies y \in A \implies P \ x \implies P \ y \implies x = y) \implies \text{inj-on-true } P \ A$

by (*simp add: inj-on-true-def*)

lemma *inj-on-true-mono*: $A \subseteq B \implies \text{inj-on-true } P \ B \implies \text{inj-on-true } P \ A$

by (*auto simp add: inj-on-true-def*)

lemma *find-in-set-inj-on-true-distinct*:

$\text{inj-on-true } P \ (\text{set } (\text{map } \text{fst } xs)) \implies (x, v) \in \text{set } xs \implies P \ x \implies \text{distinct } (\text{map } \text{fst } xs) \implies$

$\text{find } (\lambda(x, -). P \ x) \ xs = \text{Some } (x, v)$

apply (*induct xs*)

apply *simp*

apply *simp*

by (*smt (verit, best) case-prod-beta' fst-conv image-iff inj-on-true-def insertCI*)

lemma *find-in-set-inj-on-true-distinct'*:

$\text{inj-on-true } P \ (\text{set } xs) \implies x \in \text{set } xs \implies P \ x \implies \text{distinct } xs \implies$

$\text{find } (\lambda x. P \ x) \ xs = \text{Some } x$

apply (*induct xs*)

apply *simp*

apply *simp*

by (*metis inj-on-true-def insertCI*)

lemma *typ-tag-le-size-le*: $t < (u::\text{typ-uinfo}) \implies \text{size } t < \text{size } u$

using *td-set-size-lte*

by (*metis typ-tag-le-def typ-tag-lt-def*)

lemma *c-null-guard-intvl-nat-conv*:

fixes $p::'a::\text{mem-type } ptr$

assumes *c-null-guard*: *c-null-guard* p

shows $\text{ptr-span } p = \{x. (\text{unat } (\text{ptr-val } p) \leq \text{unat } x \wedge \text{unat } x < (\text{unat } (\text{ptr-val } p) + \text{size-of } \text{TYPE}('a)))\}$

proof –

from *c-null-guard*

have *c-null-guard-u* (*typ-uinfo-t* *TYPE('a)*) (*ptr-val* p)

by (*simp add: c-null-guard-c-null-guard-u-conv*)

from *c-null-guard-u-intvl-nat-conv* [*OF this*]

show *?thesis*

by (*simp add: size-of-def*)

qed

lemma *c-null-guard-ptr-add*:

fixes $p::'a::\text{mem-type } ptr$

assumes *bound*: $\text{unat } (\text{ptr-val } p) + \text{Suc } n * \text{size-of } \text{TYPE}('a) \leq \text{addr-card}$
assumes *not-null*: $\text{ptr-val } p \neq 0$
assumes *le*: $i \leq n$
shows *c-null-guard* $(p +_p \text{int } i)$
proof –

from *bound le* **have** *bound1*: $\text{unat } (\text{ptr-val } p) + i * \text{size-of } \text{TYPE}('a) \leq \text{addr-card}$
by (*meson add-le-mono dual-order.eq-iff dual-order.trans le-SucI mult-le-cancel2*)
from *bound le* **have** *bound2*: $\text{unat } (\text{ptr-val } p) + i * \text{size-of } \text{TYPE}('a) + \text{size-of } \text{TYPE}('a) \leq \text{addr-card}$
by (*smt (verit, ccfv-SIG) bound1 less-le-mult mult-strict-right-mono not-less-eq-eq of-nat-add of-nat-le-iff of-nat-mult*)

show *?thesis*
apply (*clarsimp simp add: ptr-add-def c-null-guard-def intvl-def*)
using *bound1 bound2 not-null*
by (*smt (verit, del-insts) Abs-fnat-hom-add Abs-fnat-hom-mult add-eq-0-iff-both-eq-0 add-mono-thms-linordered-field(4) eq-imp-le le-Suc-ex len-of-addr-card trans-less-add1 unat-of-nat-len unsigned-0 word-unat.Rep-inverse'*)
qed

lemma *ptr-add-disjoint-index*:
fixes *p::'a::mem-type ptr*
assumes *bound*: $\text{unat } (\text{ptr-val } p) + \text{Suc } n * \text{size-of } \text{TYPE}('a) \leq \text{addr-card}$
assumes *le*: $i < n$
assumes *n*: *c-null-guard* $(p +_p \text{int } n)$
assumes *i*: *c-null-guard* $(p +_p \text{int } i)$
shows *ptr-span* $(p +_p \text{int } n) \cap \text{ptr-span } (p +_p \text{int } i) = \{\}$
proof –

from *bound le* **have** *bound-i*: $\text{unat } (\text{ptr-val } p) + \text{Suc } i * \text{size-of } \text{TYPE}('a) \leq \text{addr-card}$
by *simp*
by (*smt (verit, best) Suc-less-SucD add-le-cancel-left dual-order.strict-iff-not le-trans mult-less-cancel2 nat-le-linear*)

from *bound* **have** *bound1*: $\text{unat } (\text{ptr-val } p) + n * \text{size-of } \text{TYPE}('a) \leq \text{addr-card}$
by *simp*
from *bound* **have** *bound2*: $\text{unat } (\text{ptr-val } p) + n * \text{size-of } \text{TYPE}('a) + \text{size-of } \text{TYPE}('a) \leq \text{addr-card}$
by *simp*

from *bound-i* **have** *bound-i1*: $\text{unat } (\text{ptr-val } p) + i * \text{size-of } \text{TYPE}('a) \leq \text{addr-card}$
by *simp*
from *bound-i* **have** *bound-i2*: $\text{unat } (\text{ptr-val } p) + i * \text{size-of } \text{TYPE}('a) + \text{size-of } \text{TYPE}('a) \leq \text{addr-card}$
by *simp*

show *?thesis*
apply (*simp add: c-null-guard-intvl-nat-conv n i*)

```

    using le bound n i bound1 bound2 bound-i1 bound-i2
    by (auto simp add: ptr-add-def)
      (smt (verit) len-of-addr-card less-le-mult mult-strict-right-mono of-nat-add
of-nat-inverse of-nat-le-iff of-nat-less-iff of-nat-mult word-unat.Rep-inverse)
qed

```

lemma *ptr-add-disjoint-last*:

```

fixes p::'a::mem-type ptr
assumes bound: unat (ptr-val p) + Suc n * size-of TYPE('a) ≤ addr-card
assumes not-null: ptr-val p ≠ 0
shows {ptr-val p..+n * size-of TYPE('a)} ∩ ptr-span (p +p int n) = {}
proof -
  {
    fix i assume i-bound: i < n
    have ptr-span (p +p int i) ∩ ptr-span (p +p int n) = {}
    proof -
      from c-null-guard-ptr-add [OF bound not-null] i-bound obtain
        c-null-guard (p +p int n) and c-null-guard (p +p int i) by auto
      from ptr-add-disjoint-index [OF bound i-bound this]
      show ?thesis by blast
    qed
  }
}

```

then show ?thesis

```

by (auto simp add: array-index-span-conv)
qed

```

lemma *h-val-fold*:

```

fixes p::'a::mem-type ptr
assumes bound: unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card
assumes not-null: ptr-val p ≠ 0
assumes i-bound: i < n
showsh-val ((fold (λi. heap-update (p +p int i) v) [0..<n]) h) (p +p int i) = v
using bound i-bound
proof (induct n arbitrary: h i)
  case 0
    then show ?case by simp
  next
    case (Suc n)
      from Suc.premis obtain i-bound: i < Suc n and bound: unat (ptr-val p) + Suc
n * size-of TYPE('a) ≤ addr-card by blast
      hence bound': unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card by simp
      show ?case
      proof (cases i = n)
        case True
          show ?thesis by (simp add: True)
        next
          case False
            with i-bound have i-bound': i < n by simp

```

```

note hyp = Suc.hyps [OF bound' i-bound']
from c-null-guard-ptr-add [OF bound not-null, of n]
have null-guard-n: c-null-guard (p +p int n) by simp
from c-null-guard-ptr-add [OF bound not-null, of i] i-bound'
have null-guard-i: c-null-guard (p +p int i) by simp
from ptr-add-disjoint-index [OF bound i-bound' null-guard-n null-guard-i]
have disj: ptr-span (p +p int n) ∩ ptr-span (p +p int i) = {} .
show ?thesis
  by (simp add: h-val-update-regions-disjoint [OF disj] hyp)
qed
qed

lemma h-val-fold-disjoint:
  fixes p:: 'a::mem-type ptr
  fixes q:: 'b::mem-type ptr
  assumes disj: {ptr-val p..+n * size-of TYPE('a)} ∩ ptr-span q = {}
  shows h-val (fold (λi. heap-update (p +p int i) (v i)) [0..<n] h) q =
    h-val h q
  using disj
proof (induct n arbitrary: h)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prem1 have disj: {ptr-val p..+Suc n * size-of TYPE('a)} ∩ ptr-span
q = {} .
  from disj have disj': {ptr-val p..+n * size-of TYPE('a)} ∩ ptr-span q = {}
  by (smt (verit, best) first-in-intvl intvlD intvlI intvl-inter le-add2 mult-Suc
order-less-le-trans orthD2 sz-nzero)
  have ptr-span (p +p int n) ⊆ {ptr-val p..+Suc n * size-of TYPE('a)}
  by (metis (no-types, lifting) CTypesDefs.ptr-add-def intvl-le mult-Suc nat-le-linear
of-int-of-nat-eq of-nat-mult ptr-val.simps)

  with disj have disj-n: ptr-span (p +p int n) ∩ ptr-span q = {}
  using intvl-le by fastforce

  note hyp = Suc.hyps [OF disj']
  show ?case by (simp add: h-val-update-regions-disjoint [OF disj-n] hyp)
qed

lemma h-val-fold-zero-disjoint:
  fixes p:: 'a::mem-type ptr
  fixes q:: 'b::mem-type ptr
  assumes disj: {ptr-val p..+n * size-of TYPE('a)} ∩ ptr-span q = {}
  shows h-val (fold (λi. heap-update (p +p int i) c-type-class.zero) [0..<n] h) q =
    h-val h q
  using disj
  by (rule h-val-fold-disjoint)

```

lemma *fold-heap-update-commute*:

fixes $p::'a:: \text{mem-type ptr}$

fixes $q::'b:: \text{mem-type ptr}$

assumes $\text{disjoint: } \{\text{ptr-val } p \dots + n * \text{size-of TYPE('a)}\} \cap \text{ptr-span } q = \{\}$

shows $\text{fold } (\lambda i. \text{heap-update } (p +_p \text{int } i) (v \ i)) [0..<n] (\text{heap-update } q \ x \ h) =$
 $(\text{heap-update } q \ x) (\text{fold } (\lambda i. \text{heap-update } (p +_p \text{int } i) (v \ i)) [0..<n] h)$

using *disjoint*

proof (*induct n arbitrary: h*)

case 0

then show *?case by simp*

next

case (*Suc n*)

from *Suc.prem*s **have** $\text{disj: } \{\text{ptr-val } p \dots + \text{Suc } n * \text{size-of TYPE('a)}\} \cap \text{ptr-span}$
 $q = \{\}$.

from *disj* **have** $\text{disj': } \{\text{ptr-val } p \dots + n * \text{size-of TYPE('a)}\} \cap \text{ptr-span } q = \{\}$

by (*smt (verit, best) first-in-intvl intvlD intvlI intvl-inter le-add2 mult-Suc*
order-less-le-trans orthD2 sz-nzero)

have $\text{ptr-span } (p +_p \text{int } n) \subseteq \{\text{ptr-val } p \dots + \text{Suc } n * \text{size-of TYPE('a)}\}$

by (*metis (no-types, lifting) CTypesDefs.ptr-add-def intvl-le mult-Suc nat-le-linear*
of-int-of-nat-eq of-nat-mult ptr-val.simps)

with *disj* **have** $\text{disj-n: } \text{ptr-span } (p +_p \text{int } n) \cap \text{ptr-span } q = \{\}$

using *intvl-le* **by** *fastforce*

note $\text{hyp} = \text{Suc.hyps } [OF \ \text{disj}]$

note $\text{commute} = \text{heap-update-commute } [OF \ \text{disj-n, simplified}]$

show *?case* **apply** *simp*

by (*simp add: hyp commute*)

qed

lemma *fold-heap-update-padding-commute*:

fixes $p::'a:: \text{mem-type ptr}$

fixes $q::'b:: \text{mem-type ptr}$

assumes $\text{disjoint: } \{\text{ptr-val } p \dots + n * \text{size-of TYPE('a)}\} \cap \text{ptr-span } q = \{\}$

assumes $\text{lbs: } \bigwedge i. i < n \implies \text{length } (bs \ i) = \text{size-of TYPE('a)}$

assumes $\text{lbs': } \text{length } bs' = \text{size-of TYPE('b)}$

shows $\text{fold } (\lambda i. \text{heap-update-padding } (p +_p \text{int } i) (v \ i) (bs \ i)) [0..<n] (\text{heap-update-padding}$
 $q \ x \ bs' \ h) =$
 $(\text{heap-update-padding } q \ x \ bs') (\text{fold } (\lambda i. \text{heap-update-padding } (p +_p \text{int } i)$
 $(v \ i) (bs \ i)) [0..<n] h)$

using *disjoint lbs*

proof (*induct n arbitrary: h bs*)

case 0

then show *?case by simp*

next

case (*Suc n*)

from *Suc.prem*s **obtain** $\text{disj: } \{\text{ptr-val } p \dots + \text{Suc } n * \text{size-of TYPE('a)}\} \cap \text{ptr-span}$
 $q = \{\}$ **and**

$\text{lbs-n: } \bigwedge i. i < n \implies \text{length } (bs \ i) = \text{size-of TYPE('a)}$ **and** $\text{lbs-n': } \text{length } (bs \ n)$
 $= \text{size-of TYPE('a)}$


```

    by auto
  from disj have disj': {ptr-val p..+n * size-of TYPE('a)} ∩ ptr-span q = {}
    by (smt (verit, best) first-in-intvl intvlD intvlI intvl-inter le-add2 mult-Suc
order-less-le-trans orthD2 sz-nzero)
  have ptr-span (p +p int n) ⊆ {ptr-val p..+Suc n * size-of TYPE('a)}
    by (metis (no-types, lifting) CTypesDefs.ptr-add-def intvl-le mult-Suc nat-le-linear
of-int-of-nat-eq of-nat-mult ptr-val.simps)

  with disj have disj-n: ptr-span (p +p int n) ∩ ptr-span q = {}
    using intvl-le by fastforce
  note hyp = Suc.hyps [OF disj' lbs-n]
  note commute = heap-update-padding-commute [OF disj-n lbs-n' lbs', simplified]
  show ?case
    apply simp
    apply (simp add: hyp commute)
  done
qed

```

```

lemma fold-heap-update-padding-heap-update-commute:
  fixes p::'a:: mem-type ptr
  fixes q::'b:: mem-type ptr
  assumes disjoint: {ptr-val p ..+ n * size-of TYPE('a)} ∩ ptr-span q = {}
  assumes lbs: ∧i. i < n ⇒ length (bs i) = size-of TYPE('a)
  shows fold (λi. heap-update-padding (p +p int i) (v i) (bs i)) [0..<n] (heap-update
q x h) =
    (heap-update q x) (fold (λi. heap-update-padding (p +p int i) (v i) (bs i))
[0..<n] h)
  using disjoint lbs
proof (induct n arbitrary: h bs)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prem1 obtain disj: {ptr-val p..+Suc n * size-of TYPE('a)} ∩ ptr-span
q = {} and
    lbs-n: ∧i. i < n ⇒ length (bs i) = size-of TYPE('a) and lbs-n': length (bs n)
= size-of TYPE('a)
  by auto
  from disj have disj': {ptr-val p..+n * size-of TYPE('a)} ∩ ptr-span q = {}
    by (smt (verit, best) first-in-intvl intvlD intvlI intvl-inter le-add2 mult-Suc
order-less-le-trans orthD2 sz-nzero)
  have ptr-span (p +p int n) ⊆ {ptr-val p..+Suc n * size-of TYPE('a)}
    by (metis (no-types, lifting) CTypesDefs.ptr-add-def intvl-le mult-Suc nat-le-linear
of-int-of-nat-eq of-nat-mult ptr-val.simps)

```

```

  with disj have disj-n: ptr-span (p +p int n) ∩ ptr-span q = {}
    using intvl-le by fastforce
  note hyp = Suc.hyps [OF disj' lbs-n]
  note commute = heap-update-padding-heap-update-commute [OF disj-n lbs-n' ,

```

```

simplified]
show ?case
  apply simp
  apply (simp add: hyp commute )
done
qed

```

lemma *fold-heap-update-collapse:*

```

fixes p::'a::xmem-type ptr
assumes bound: unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card
assumes not-null: ptr-val p ≠ 0
shows fold (λi. heap-update (p +p int i) (v i)) [0..n]
      ((fold (λi. heap-update (p +p int i) (w i)) [0..n] h) =
       fold (λi. heap-update (p +p int i) (v i)) [0..n] h
using bound
proof (induct n arbitrary: h)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prem1 obtain bound: unat (ptr-val p) + Suc n * size-of TYPE('a) ≤
  addr-card by blast
  hence bound': unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card by simp
  from ptr-add-disjoint-last [OF bound not-null]
  have disj: {ptr-val p..n * size-of TYPE('a)} ∩ ptr-span (p +p int n) = {} .
  show ?case
    apply simp
    apply (subst (2) Suc.hyps [OF bound', symmetric])
    apply (subst fold-heap-update-commute [OF disj])
    apply (simp add: heap-update-collapse)
done
qed

```

lemma *fold-heap-update-padding-collapse:*

```

fixes p::'a::xmem-type ptr
assumes bound: unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card
assumes not-null: ptr-val p ≠ 0
assumes lbs: ∧i. i < n ⇒ length (bs i) = size-of TYPE('a)
assumes lbs': ∧i. i < n ⇒ length (bs' i) = size-of TYPE('a)
shows fold (λi. heap-update-padding (p +p int i) (v i) (bs i)) [0..n]
      ((fold (λi. heap-update-padding (p +p int i) (w i) (bs' i)) [0..n] h) =
       fold (λi. heap-update-padding (p +p int i) (v i) (bs i)) [0..n] h
using bound lbs lbs'
proof (induct n arbitrary: h bs bs')
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prem1 obtain

```

bound: $\text{unat } (\text{ptr-val } p) + \text{Suc } n * \text{size-of } \text{TYPE}'(a) \leq \text{addr-card}$ **and**
lbs-n: $\bigwedge i. i < n \implies \text{length } (bs \ i) = \text{size-of } \text{TYPE}'(a)$ **and**
lbs'-n: $\bigwedge i. i < n \implies \text{length } (bs' \ i) = \text{size-of } \text{TYPE}'(a)$ **and**
lbs'-n': $\text{length } (bs' \ n) = \text{size-of } \text{TYPE}'(a)$ **and**
lbs-n': $\text{length } (bs \ n) = \text{size-of } \text{TYPE}'(a)$
by auto
hence bound': $\text{unat } (\text{ptr-val } p) + n * \text{size-of } \text{TYPE}'(a) \leq \text{addr-card}$ **by simp**
from ptr-add-disjoint-last [OF bound not-null]
have disj: $\{\text{ptr-val } p..+n * \text{size-of } \text{TYPE}'(a)\} \cap \text{ptr-span } (p +_p \text{ int } n) = \{\}$.
show ?case
apply simp
apply (subst (2) Suc.hyps [OF bound' lbs-n lbs'-n, symmetric])
apply (assumption)
apply assumption
apply (subst fold-heap-update-padding-commute [OF disj lbs-n lbs'-n'])
apply (assumption)
apply (simp add: heap-update-padding-collapse [OF lbs-n' lbs'-n'])
done
qed

lemma fold-heap-update-padding-heap-update-collapse:
fixes $p::'a::\text{xmem-type ptr}$
assumes bound: $\text{unat } (\text{ptr-val } p) + n * \text{size-of } \text{TYPE}'(a) \leq \text{addr-card}$
assumes not-null: $\text{ptr-val } p \neq 0$
assumes lbs: $\bigwedge i. i < n \implies \text{length } (bs \ i) = \text{size-of } \text{TYPE}'(a)$
shows $\text{fold } (\lambda i. \text{heap-update-padding } (p +_p \text{ int } i) \ (v \ i) \ (bs \ i)) \ [0..<n]$
 $\quad ((\text{fold } (\lambda i. \text{heap-update } (p +_p \text{ int } i) \ (w \ i)) \ [0..<n]) \ h) =$
 $\quad \text{fold } (\lambda i. \text{heap-update-padding } (p +_p \text{ int } i) \ (v \ i) \ (bs \ i)) \ [0..<n] \ h$
using bound lbs
proof (induct n arbitrary: h bs)
case 0
then show ?case by simp
next
case (Suc n)
from Suc.premis obtain
bound: $\text{unat } (\text{ptr-val } p) + \text{Suc } n * \text{size-of } \text{TYPE}'(a) \leq \text{addr-card}$ **and**
lbs-n: $\bigwedge i. i < n \implies \text{length } (bs \ i) = \text{size-of } \text{TYPE}'(a)$ **and**
lbs-n': $\text{length } (bs \ n) = \text{size-of } \text{TYPE}'(a)$
by auto
hence bound': $\text{unat } (\text{ptr-val } p) + n * \text{size-of } \text{TYPE}'(a) \leq \text{addr-card}$ **by simp**
from ptr-add-disjoint-last [OF bound not-null]
have disj: $\{\text{ptr-val } p..+n * \text{size-of } \text{TYPE}'(a)\} \cap \text{ptr-span } (p +_p \text{ int } n) = \{\}$.
show ?case
apply simp
apply (subst (2) Suc.hyps [OF bound' lbs-n , symmetric])
apply (assumption)
apply (subst fold-heap-update-padding-heap-update-commute [OF disj lbs-n])
apply (assumption)
thm heap-update-padding-heap-update-collapse

apply (*simp add: heap-update-padding-heap-update-collapse [OF lbs-n']*)
done
qed

21.1 Open Types

named-theorems \mathcal{T} -def and

ptr-valid-definition and *fold-ptr-valid* and *ptr-valid* and *ptr-valid-u-recursion*
and

addressable-ptr-valid-field and *plift-simps* and *stack-the-default-plift* and
plift-heap-update-padding-heap-update-conv and
plift-heap-update-list-stack-byte

lemma *list-ex-array-fields*[*simp*]:

list-ex P (array-fields m) \longleftrightarrow ($\exists n < m. P [\text{replicate } n \text{ CHR } "1"]$)

by (*simp add: array-fields-def list-ex-iff Bex-def*)

lemma *list-all-field-lookup-array-fields*[*simp*]:

CARD('n) = m \implies

list-all

($\lambda f. \exists a b. \text{field-lookup (typ-uinfo-t TYPE('a::c-type['n::finite])} f 0 = \text{Some (a, b)}$)

(array-fields m)

by (*simp add: array-fields-def*)

lemma *list-ex-subst-def*:

c \equiv xs \implies list-ex P c = list-ex P xs

by *simp*

lemma *ex-field-lookup-append*:

wf-desc t \implies

field-lookup t p1 n1 = Some (u, m1) \implies

field-lookup u p2 n2 = Some (v, m2) \implies

$\exists m. \text{field-lookup t (p1 @ p2) n3 = Some (v, m)}$

by (*metis field-lookup-offset-shift' field-lookup-prefix-Some'*)

named-theorems *addressable-field-exec*

locale *open-types* =

fixes $\mathcal{T}:: (\text{typ-uinfo} * \text{qualified-field-name list}) \text{list}$ — toplevel addressable fields
of a type

assumes *mem-type-u: $\bigwedge t fs. (t, fs) \in \text{set } \mathcal{T} \implies \text{mem-type-u } t$*

assumes *wf- \mathcal{T}' :*

list-all ($\lambda(t, fs).$

$\square \notin \text{set } fs \wedge$

list-all ($\lambda f. \text{field-lookup t f 0} \neq \text{None}$) fs \wedge

distinct-prop disj-fn fs) \mathcal{T}

assumes *distinct- \mathcal{T} : distinct (map fst \mathcal{T})*

begin

inductive *addressable-field* :: *typ-uinfo* \Rightarrow *qualified-field-name* \Rightarrow *typ-uinfo* \Rightarrow *bool*
where

addressable-field-refl[*intro!*, *simp*]: $\bigwedge t.$ *addressable-field* *t* [] *t*
| *addressable-field-step*: $\bigwedge t$ *fs* *p* *p'* *u* *v* *n*.
map-of \mathcal{T} *t* = *Some fs* \Longrightarrow *p* \in *set fs* \Longrightarrow *field-lookup* *t* *p* 0 = *Some (u, n)* \Longrightarrow
addressable-field *u* *p'* *v* \Longrightarrow *addressable-field* *t* (*p* @ *p'*) *v*

inductive *ptr-valid-u* :: *typ-uinfo* \Rightarrow *heap-typ-desc* \Rightarrow *addr* \Rightarrow *bool* **for** *t d* **where**
 $\bigwedge r$ *p a*. *root-ptr-valid-u* *r* *d* *a* \Longrightarrow *addressable-field* *r* *p* *t* \Longrightarrow
ptr-valid-u *t* *d* (*a* + *of-nat* (*field-offset-untyped* *r* *p*))

definition *ptr-valid* :: *heap-typ-desc* \Rightarrow '*a*::*mem-type* *ptr* \Rightarrow *bool* **where**
[*ptr-valid-definition*]:
ptr-valid *d* *p* \equiv *ptr-valid-u* (*typ-uinfo-t* *TYPE*('a)) *d* (*ptr-val* *p*)

lemma *addressable-field-single*:

map-of \mathcal{T} *t* = *Some fs* \Longrightarrow *p* \in *set fs* \Longrightarrow *field-lookup* *t* *p* 0 = *Some (v, n)* \Longrightarrow
addressable-field *t* *p* *v*
using *addressable-field-step*[*of t fs p v n* []] **by** *auto*

lemma *addressable-field-append*:

assumes *: *addressable-field* *t* *p1* *u* *addressable-field* *u* *p2* *v*
shows *addressable-field* *t* (*p1* @ *p2*) *v*
proof (*use* * **in** \langle *induction arbitrary: p2 v* \rangle)
case *: (*addressable-field-step* *t* *fs* *p* *p'* *u* *v* *n* *w* *p1*)

show ?*case*
unfolding *append-assoc*
apply (*rule* *addressable-field-step*[*OF* *(1,2,3)])
apply (*rule* *)+
done

qed *simp*

lemma *addressable-field-rev-induct*[*consumes 1*, *case-names refl step*]:

assumes *: *addressable-field* *t* *p* *u*
assumes 1: $\bigwedge t.$ *P* *t* [] *t*
assumes 2: $\bigwedge t$ *p'* *u* *fs* *f* *v* *n*.
addressable-field *t* *p'* *u* \Longrightarrow *map-of* \mathcal{T} *u* = *Some fs* \Longrightarrow *f* \in *set fs* \Longrightarrow
field-lookup *u* *f* 0 = *Some (v, n)* \Longrightarrow *P* *t* *p'* *u* \Longrightarrow *P* *t* (*p'* @ *f*) *v*
shows *P* *t* *p* *u*

proof –

have *addressable-field* *t* *p1* *u* \Longrightarrow *P* *t* *p1* *u* \Longrightarrow *P* *t* (*p1* @ *p2*) *v*
if *: *addressable-field* *u* *p2* *v* **for** *t* *p1* *u* *p2* *v*
proof (*use* * **in** \langle *induction arbitrary: t p1* \rangle)
case *: (*addressable-field-step* *t* *fs* *f* *q* *u* *v* *n* *w*)
show ?*case*
unfolding *append-assoc*[*symmetric*]
by (*intro* *.*IH* 2[*OF* *.*prems*(1) *(1,2,3)] *)

```

      (intro addressable-field-append[OF *.prems(1)]
        addressable-field-single[OF *(1,2,3)])
    qed simp
    from this[OF * addressable-field-refl 1] show ?thesis by simp
  qed

lemma addressable-field-rev-cases[consumes 1, case-names refl step]:
  assumes *: addressable-field t p u
  assumes 1: t = u  $\implies$  p = []  $\implies$  P
  assumes 2:  $\bigwedge p' fs f v n.$ 
    addressable-field t p' v  $\implies$  map-of  $\mathcal{T}$  v = Some fs  $\implies$  f  $\in$  set fs  $\implies$ 
    field-lookup v f 0 = Some (u, n)  $\implies$  p = p' @ f  $\implies$  P
  shows P
  using *
  by (induction t==t p==p u==u rule: addressable-field-rev-induct)
    (use 1 2 in auto)

lemma wf- $\mathcal{T}$ : map-of  $\mathcal{T}$  t = Some fs  $\implies$  list-all ( $\lambda f.$  field-lookup t f 0  $\neq$  None)
  fs
  using wf- $\mathcal{T}'$  by (auto simp: list-all-iff dest!: map-of-SomeD)

lemma  $\mathcal{T}$ -not-nil: map-of  $\mathcal{T}$  t = Some fs  $\implies$  []  $\notin$  set fs
  using wf- $\mathcal{T}'$  by (auto simp: list-all-iff dest!: map-of-SomeD)

lemma addressable-field-refl-iff[simp]:
  addressable-field t [] u  $\longleftrightarrow$  t = u
  by (subst addressable-field.simps) (auto simp add:  $\mathcal{T}$ -not-nil cong: HOL.conj-cong)

lemma addressable-field-domain:
  addressable-field t p u  $\implies$  t  $\in$  fst `set  $\mathcal{T} \vee (t = u \wedge p = [])$ 
  apply (subst (asm) addressable-field.simps)
  by (auto simp: image-iff split-paired-Ball dest!: map-of-SomeD) blast+

lemma addressable-fieldD-mem-type-u:
  addressable-field t p u  $\implies$  mem-type-u t  $\vee (t = u \wedge p = [])$ 
  by (auto dest!: addressable-field-domain mem-type-u)

lemma addressable-fieldD-field-lookup:
  assumes p: addressable-field t p u shows  $\exists n.$  field-lookup t p 0 = Some (u, n)
  by (use p in induction)
    (auto intro: ex-field-lookup-append mem-type-u mem-type-u.wf-desc dest: map-of-SomeD)

lemma addressable-field-same-iff[simp]:
  addressable-field t p t  $\longleftrightarrow$  p = []
  using addressable-fieldD-field-lookup[of t p t] field-lookup-same-type-empty(1)[of
  t p 0 t]
  by auto

lemma addressable-fieldD-field-lookup':

```

addressable-field $t p u \implies \text{field-lookup } t p 0 = \text{Some } (u, \text{field-offset-untyped } t p)$
using *addressable-fieldD-field-lookup*[*of* $t p u$] **by** (*auto simp add: field-offset-untyped-def*)

lemma *addressable-fieldD-field-ti*:

assumes p : *addressable-field* (*typ-uinfo-t* $\text{TYPE}('a::\text{c-type})$) $p t$
shows $\exists u. \text{field-ti } \text{TYPE}('a) p = \text{Some } u \wedge \text{export-uinfo } u = t$
using *field-lookup-uinfo-Some-rev*[*OF* *addressable-fieldD-field-lookup'*[*OF* p]]
by (*clarsimp simp: field-ti-def simp del: field-lookup-offset-untyped-eq*)

lemma *addressable-fieldD-field-names*:

addressable-field $t p u \implies p \in \text{set } (\text{field-names-}u t u)$
by (*auto dest: addressable-fieldD-field-lookup intro: field-lookup-Some-field-names-u(1)*)

lemma *mem-type-u-addressable-field*:

assumes $*$: *addressable-field* $t fs u$
shows *mem-type-u* $u \implies \text{mem-type-u } t$
by (*use * in induction*) (*auto intro: mem-type-u dest!: map-of-SomeD*)

lemma *wf-desc-addressable-field*:

assumes $*$: *addressable-field* $t fs u$
shows *wf-desc* $t \implies \text{wf-desc } u$
using *addressable-fieldD-field-lookup'*[*OF* $*$] **by** (*auto intro: field-lookup-wf-desc-pres*)

lemma *wf-size-desc-addressable-field*:

assumes $*$: *addressable-field* $t fs u$
shows *wf-size-desc* $t \implies \text{wf-size-desc } u$
using *addressable-fieldD-field-lookup'*[*OF* $*$] **by** (*auto intro: field-lookup-wf-size-desc-pres*)

lemma *field-offset-append-of-addressable-field*:

assumes p : *addressable-field* $t p u$ **and** q : *addressable-field* $u q v$
shows *field-offset-untyped* $t (p @ q) = \text{field-offset-untyped } t p + \text{field-offset-untyped } u q$

proof –

have $p = [] \wedge q = [] \vee \text{mem-type-u } t$
using *addressable-fieldD-mem-type-u*[*OF* p] *addressable-fieldD-mem-type-u*[*OF* q] **by** *auto*

then show *?thesis*

proof (*elim disjE conjE*)

assume *mem-type-u* t **then show** *?thesis*

by (*intro field-names-u-field-offset-untyped-append*[*OF* *mem-type-u.wf-desc* *addressable-fieldD-field-names*[*OF* p] *addressable-fieldD-field-names*[*OF*

q]])

qed *simp*

qed

lemma *intvl-subset-of-addressable-field*:

assumes p : *addressable-field* $t p u$
shows $\{a + \text{of-nat } (\text{field-offset-untyped } t p) ..+ \text{size-td } u\} \subseteq \{a ..+ \text{size-td } t\}$
using *addressable-fieldD-field-lookup*[*OF* p]

using *intvl-subset-of-field-lookup*[of *t p u field-offset-untyped t p a*]
by (*auto simp: field-offset-untyped-def*)

lemma *root-ptr-valid-u-ptr-valid-u*:
assumes *a: root-ptr-valid-u t d a* **shows** *ptr-valid-u t d a*
using *ptr-valid-u.intros[OF a addressable-field-refl]* **by** *simp*

lemma *root-ptr-valid-ptr-valid*: *root-ptr-valid d p \implies ptr-valid d p*
unfolding *root-ptr-valid-root-ptr-valid-u-conv ptr-valid-def*
by (*rule root-ptr-valid-u-ptr-valid-u*)

lemma *fold-ptr-valid'[fold-ptr-valid]*:
ptr-valid-u (typ-uinfo-t TYPE('a::mem-type)) d x = ptr-valid d (PTR('a) x)
by (*simp add: ptr-valid-def*)

lemma *ptr-valid-u-trans*:
assumes **: ptr-valid-u t d a*
shows *addressable-field t p u \implies ptr-valid-u u d (a + of-nat (field-offset-untyped t p))*
proof (*use * in induction*)
case (*1 r q a*)
have *eq*:
 $a + \text{word-of-nat (field-offset-untyped r q)} + \text{word-of-nat (field-offset-untyped t p)} =$
 $a + \text{word-of-nat (field-offset-untyped r (q @ p))}$
using *field-offset-append-of-addressable-field[OF 1(2,3)]* **by** *simp*
show *?case unfolding eq*
by (*intro ptr-valid-u.intros 1 addressable-field-append[OF 1(2,3)]*)
qed

lemma *ptr-valid-u-step*:
assumes **: map-of \mathcal{T} t = Some F f \in set F field-lookup t f 0 = Some (u, n)*
and *a: ptr-valid-u t d a*
shows *ptr-valid-u u d (a + of-nat (field-offset-untyped t f))*
by (*rule ptr-valid-u-trans[OF a]*) (*rule addressable-field-single[OF *]*)

lemma *ptr-valid-u-recursion'[ptr-valid-u-recursion]*:
ptr-valid-u t d a \iff
root-ptr-valid-u t d a \vee
list-ex ($\lambda(u, fs). \text{list-ex } (\lambda f. \exists a' n. \text{field-lookup } u f 0 = \text{Some } (t, n) \wedge$
ptr-valid-u u d a' \wedge a = a' + of-nat (field-offset-untyped u f)) fs) \mathcal{T}

proof –
have
 $\text{root-ptr-valid-u t d}$
 $(a + \text{word-of-nat (field-offset-untyped r p)}) \vee$
 $(\exists x \in \text{set } \mathcal{T}. \text{case } x \text{ of } (u, fs) \Rightarrow \exists f \in \text{set } fs.$
 $(\exists n. \text{field-lookup } u f 0 = \text{Some } (t, n)) \wedge$
 $(\exists a'. \text{ptr-valid-u u d a'} \wedge$
 $a + \text{word-of-nat (field-offset-untyped r p)} =$


```

      a' + word-of-nat (field-offset-untyped u f)))
    if p: addressable-field r p t and r: root-ptr-valid-u r d a
    for a r p
  proof (use p in ⟨cases rule: addressable-field-rev-cases⟩)
    case refl with r show ?thesis by simp
  next
    case *: (step p' fs p'' u m)
    then have **: (u, fs) ∈ set T
      by (auto simp: map-of-SomeD)
    have mem-type-u u
      using mem-type-u[of u fs] *(2) by (auto dest: map-of-SomeD)
    with * have mem-r: mem-type-u r
      by (auto dest: mem-type-u-addressable-field)

    have u-t: addressable-field u p'' t
      using *(2,3,4) by (rule addressable-field-single)

    show ?thesis
      apply (intro disjI2 beXI[of - (u, fs)] **)
      apply simp
      apply (intro conjI beXI[of - p''] * exI[of - a + of-nat (field-offset-untyped r
p')])
      subgoal using *(4) by (rule exI)
      subgoal using r *(1) by (rule ptr-valid-u.intros)
      apply (simp add: field-offset-append-of-addressable-field[OF *(1) u-t] ⟨p = p'
@ p''⟩)
      done
    qed
  moreover have
    ptr-valid-u t d
      (a' + word-of-nat (field-offset-untyped r p) + word-of-nat (field-offset-untyped
u f))
    if 1: (u, fs) ∈ set T and 2: f ∈ set fs field-lookup u f 0 = Some (t, n)
      and r: root-ptr-valid-u r d a' addressable-field r p u
      for u fs f n r p a'
  proof -
    have 1: map-of T u = Some fs
      using 1 by (simp add: map-of-eq-Some-iff[OF distinct-T])
    show ?thesis
      using ptr-valid-u.intros[OF r] addressable-field-single[OF 1 2]
      by (rule ptr-valid-u-trans)
    qed
  ultimately show ?thesis
    by (auto simp: list-ex-iff del: disjCI
      elim!: ptr-valid-u.cases intro: root-ptr-valid-u-ptr-valid-u)
  qed

lemma T-distinct: map-of T t = Some fs ⟹ distinct-prop disj-fn fs
  using wf-T' by (auto simp: list-all-iff dest!: map-of-SomeD)

```

lemma *addressable-field-unique*:
assumes t : *addressable-field* r p t **shows** *addressable-field* r p $t' \implies t = t'$
proof (*use* t **in** *induction*)
case 1: (*addressable-field-step* t fs p p' u v n)
from \langle *addressable-field* t (p @ p') t' \rangle **show** ?*case*
proof *cases*
case 2: (*addressable-field-step* fs' q q' w m)
with 1 **have** *map-of* \mathcal{T} $t = \text{Some } fs$ $p \in \text{set } fs$ $q \in \text{set } fs$ **by** *auto*
with 1 \mathcal{T} -*distinct*[*of* t fs] **have** $p = q \vee \text{disj-fn } p$ q
using *pairwise-set-of-distinct-prop*[*of* disj-fn , *OF disj-fn-comm*, *of fs*]
by (*auto simp: pairwise-def*)
with $\langle p$ @ $p' = q$ @ $q' \rangle$ **have** [*simp*]: $q = p \wedge q' = p'$
by (*auto simp: disj-fn-def less-eq-list-def prefix-def append-eq-append-conv2*)
with 1(3) 2(4) **have** $u = w$
by *simp*
with 1(5) \langle *addressable-field* w q' t' \rangle **show** ?*thesis*
by *simp*
qed (*use* 1 **in** *simp*)
qed *simp*

lemma *addressable-fields-split*:
assumes r : *addressable-field* r p t *addressable-field* r (p @ p') u
shows *addressable-field* t p' u
proof (*use* r **in** *induction*)
case 1: (*addressable-field-step* t fs $p0$ $p1$ u v n)
from 1(6, 1–5) **show** ?*case*
proof *cases*
case 2: (*addressable-field-step* fs' q q' t' m)
with 1 **have** *map-of* \mathcal{T} $t = \text{Some } fs$ $p0 \in \text{set } fs$ $q \in \text{set } fs$ **by** *auto*
with 1 \mathcal{T} -*distinct*[*of* t fs] **have** $p0 = q \vee \text{disj-fn } p0$ q
using *pairwise-set-of-distinct-prop*[*of* disj-fn , *OF disj-fn-comm*, *of fs*]
by (*auto simp: pairwise-def*)
with 2(1) **have** $p0 = q \wedge p1$ @ $p' = q'$
by (*auto simp: disj-fn-def less-eq-list-def prefix-def append-eq-append-conv2*)
with 1 2 **show** ?*thesis*
by *simp*
qed *simp*
qed *simp*

lemma *addressable-field-antisymm*:
addressable-field t p $u \implies$ *addressable-field* u q $t \implies u = t$
using *addressable-field-append*[*of* t p u q t] **by** *simp*

lemma *addressable-field-sub-typ*: *addressable-field* t $path$ $u \implies u \leq t$
using *addressable-fieldD-field-lookup*[*of* t $path$ u] *typ-tag-le-def*
by (*blast dest: td-set-field-lookupD*)

lemma *sub-typ-of-field-lookup*:

$field_lookup (typ_uinfo_t \text{TYPE}('a)) p 0 = \text{Some} (typ_uinfo_t \text{TYPE}('b), n) \implies$
 $\text{TYPE}('b::mem_type) \leq_{\tau} \text{TYPE}('a::mem_type)$
apply (clarsimp simp add: sub-typ-def)
apply (rule field-of-sub[of of-nat n])
using field-lookup-offset-size'[of typ-uinfo-t TYPE('a) p]
 $mem_type_class.u.max_size[\mathbf{where} 'a='a]$
by (auto simp add: field-of-def field-offset-def field-offset-untyped-def addr-card-len-of-conv
intro!: td-set-field-lookupD[of - p] unat-of-nat-eq[symmetric])

lemma addressable-fields-imp-sub-typ:
 $addressable_field (typ_uinfo_t \text{TYPE}('a)) p (typ_uinfo_t \text{TYPE}('b)) \implies$
 $\text{TYPE}('b::mem_type) \leq_{\tau} \text{TYPE}('a::mem_type)$
by (simp add: addressable-field-sub-typ sub-typ-def)

lemma ptr-valid-u-cases:
assumes t: mem-type-u t
assumes t-a: ptr-valid-u t d a
assumes u-b: ptr-valid-u u d b
shows $disjnt \{a..+size-td\} \{b..+size-td\} \vee$
 $(t = u \wedge a = b) \vee$
 $(\exists p. addressable_field\ t\ p\ u \wedge t \neq u \wedge b = a + \text{word-of-nat} (field_offset_untyped$
 $t\ p)) \vee$
 $(\exists p. addressable_field\ u\ p\ t \wedge t \neq u \wedge a = b + \text{word-of-nat} (field_offset_untyped$
 $u\ p))$
(is ?disj \vee ?addr)

proof (rule disjCI2)
assume a-b: $\neg ?disj$
from t-a **obtain** r f0 x **where** x0: root-ptr-valid-u r d x
and a: $a = x + \text{of-nat} (field_offset_untyped\ r\ f0)$
and r-t: addressable-field r f0 t
by (auto elim!: ptr-valid-u.cases)
from u-b **obtain** r' f1 x' **where** x1: root-ptr-valid-u r' d x'
and b: $b = x' + \text{of-nat} (field_offset_untyped\ r'\ f1)$
and r-u: addressable-field r' f1 u
by (auto elim!: ptr-valid-u.cases)
from a-b **have** $\neg disjnt \{x..+size-td\} \{x'..+size-td\}$
using intvl-subset-of-addressable-field[OF r-t, of x]
using intvl-subset-of-addressable-field[OF r-u, of x']
by (auto intro: disjnt-subset1 disjnt-subset2 simp: a b)
with root-ptr-valid-u-cases[OF x0 x1] **have** [simp]: $r' = r\ x' = x$
by (auto simp: disjnt-def)
have r: mem-type-u r
using r-t t **by** (rule mem-type-u-addressable-field)
then have wf-r: wf-desc r
by (rule mem-type-u.wf-desc)
have $\neg disjnt$
 $\{(word_of_nat (field_offset_untyped\ r\ f0)::addr)..+size-td\} \{$
 $word_of_nat (field_offset_untyped\ r\ f1)..+size-td\}$
using a-b **by** (simp add: a b)

```

with field-lookup-non-prefix-disj'[OF r
  addressable-fieldD-field-lookup'[OF r-t]
  addressable-fieldD-field-lookup'[of r f1 u]] r-u
obtain f' where f0 = f1 @ f' ∨ f1 = f0 @ f'
  by (auto simp: disj-fn-def less-eq-list-def prefix-def)
then show ?addr
proof (elim disjE)
  assume f0 = f1 @ f'
  with r-t r-u addressable-fields-split[of r f1 u f' t]
  show ?thesis
    by (cases u = t) (auto simp add: a b field-offset-append-of-addressable-field)
next
  assume f1 = f0 @ f'
  with r-t r-u addressable-fields-split[of r f0 t f' u]
  show ?thesis
    by (cases u = t) (auto simp add: a b field-offset-append-of-addressable-field)
qed
qed

```

lemma *ptr-valid-cases'*:

```

fixes p :: 'a::mem-type ptr
fixes q :: 'b::mem-type ptr
assumes p: ptr-valid d p
assumes q: ptr-valid d q
shows disjnt (ptr-span p) (ptr-span q) ∨
  (typ-uinfo-t TYPE('a) = typ-uinfo-t TYPE('b) ∧ ptr-val p = ptr-val q) ∨
  (∃f. addressable-field (typ-uinfo-t TYPE('a)) f (typ-uinfo-t TYPE('b)) ∧
    typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('b) ∧
    q = PTR('b) &( p → f )) ∨
  (∃f. addressable-field (typ-uinfo-t TYPE('b)) f (typ-uinfo-t TYPE('a)) ∧
    typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('b) ∧
    p = PTR('a) &( q → f ))
using ptr-valid-u-cases[OF - p[unfolded ptr-valid-def] q[unfolded ptr-valid-def]]
by (simp add: size-of-def ptr-val-conv field-lvalue-def field-offset-def)

```

lemma *ptr-valid-u-cases-weak*:

```

assumes mem-type-u t
assumes ptr-valid-u t d a
assumes ptr-valid-u u d b
shows disjnt {a..+size-td t} {b..+size-td u} ∨
  (∃p. addressable-field t p u ∧ b = a + word-of-nat (field-offset-untyped t p)) ∨
  (∃p. addressable-field u p t ∧ a = b + word-of-nat (field-offset-untyped u p))
using ptr-valid-u-cases[OF assms] by (cases t = u) auto

```

lemma *ptr-valid-cases-weak*:

```

fixes p :: 'a::mem-type ptr
fixes q :: 'b::mem-type ptr
assumes p: ptr-valid d p
assumes q: ptr-valid d q

```

shows $\text{disjnt } (\text{ptr-span } p) (\text{ptr-span } q) \vee$
 $(\exists f. \text{addressable-field } (\text{typ-uinfo-t } \text{TYPE}'a) f (\text{typ-uinfo-t } \text{TYPE}'b) \wedge$
 $q = \text{PTR}'b \ \&(p \rightarrow f)) \vee$
 $(\exists f. \text{addressable-field } (\text{typ-uinfo-t } \text{TYPE}'b) f (\text{typ-uinfo-t } \text{TYPE}'a) \wedge$
 $p = \text{PTR}'a \ \&(q \rightarrow f))$
using $\text{ptr-valid-cases}'[\text{OF } \text{assms}]$
by $(\text{cases } \text{typ-uinfo-t } \text{TYPE}'a = \text{typ-uinfo-t } \text{TYPE}'b)$
 $(\text{auto simp: ptr-val-conv field-lvalue-def})$

lemma $\text{ptr-valid-u-cvalid-u}$:

assumes $t: \text{mem-type-u } t$
assumes $\text{ptr-valid: ptr-valid-u } t \ d \ a$
shows $\text{cvalid-u } t \ d \ a$
using ptr-valid

proof

fix $r \ fs \ a'$ **assume** $r\text{-t: addressable-field } r \ fs \ t$
and $a\text{-eq: } a = a' + \text{word-of-nat } (\text{field-offset-untyped } r \ fs)$
and $a': \text{root-ptr-valid-u } r \ d \ a'$
have $r: \text{mem-type-u } r$
using $\text{mem-type-u-addressable-field}[\text{OF } r\text{-t } t]$.
have $\text{align-t: } 2 \wedge \text{align-td } t \ \text{dvd } \text{size-td } t$
using t **by** $(\text{rule } \text{mem-type-u.align-size})$

show $\text{cvalid-u } t \ d \ a$ **unfolding** $a\text{-eq}$

using $r \ \text{align-t} \ \text{addressable-fieldD-field-lookup}'[\text{OF } r\text{-t}] \ \text{root-ptr-valid-u-cvalid-u}[\text{OF}$
 $a']$
by $(\text{rule } \text{mem-type-u.cvalid-u-field-lookup})$

qed

lemma $\text{ptr-valid-h-t-valid: ptr-valid } d \ p \implies d \models_t (p::'a::\text{mem-type } \text{ptr})$

unfolding $\text{ptr-valid-def cvalid-cvalid-u-conv}$
by $(\text{rule } \text{ptr-valid-u-cvalid-u} \ \text{simp-all})$

sublocale $\text{ptr-valid?}: \text{wf-ptr-valid } \text{ptr-valid}$

by $\text{unfold-locales } (\text{rule } \text{ptr-valid-h-t-valid})$

lemma $\text{h-val-heap-update}' [\text{plift-simps}]$:

fixes $p::'a::\text{mem-type } \text{ptr}$
fixes $q::'a::\text{mem-type } \text{ptr}$
assumes $\text{ptr-valid-p: ptr-valid } d \ p$
assumes $\text{ptr-valid-q: ptr-valid } d \ q$
shows $\text{h-val } (\text{heap-update } p \ v \ h) \ q = ((\text{h-val } h)(p:=v)) \ q$
by $(\text{metis } \text{c-type-class.lift-def lift-heap-update ptr-valid-h-t-valid ptr-valid-p ptr-valid-q})$

lemma $\text{h-val-heap-update-padding}' [\text{plift-simps}]$:

fixes $p::'a::\text{mem-type } \text{ptr}$
fixes $q::'a::\text{mem-type } \text{ptr}$
assumes $\text{ptr-valid-p: ptr-valid } d \ p$

assumes *ptr-valid-q*: *ptr-valid d q*
assumes *lbs*: *length bs = size-of TYPE('a)*
shows *h-val (heap-update-padding p v bs h) q = ((h-val h)(p:=v)) q*
by (*smt (verit, best) fun-upd-apply-eq-cases h-val-heap-update-padding*
h-val-heap-update-padding-disjoint lbs local.ptr-valid-h-t-valid ptr-valid-p ptr-valid-q
ptr-valid-same-type-cases)

lemmas *ptr-valid-pltft-Some-hval [pltft-simps]*

theorem *disjoint-type-pltft*:

fixes *p::'a::mem-type ptr*

assumes *unrelated1*: $\neg \text{TYPE}('a) \leq_{\tau} \text{TYPE}('b::\text{mem-type})$

assumes *unrelated2*: $\neg \text{TYPE}('b) \leq_{\tau} \text{TYPE}('a)$

assumes *ptr-valid-p*: *ptr-valid (hrs-htd h) p*

shows *pltft ((hrs-mem-update (heap-update p v) h)) = (pltft h::'b ptr \Rightarrow 'b option)*

proof –

from *unrelated1 unrelated2* **have** *disj*: *typ-uinfo-t TYPE('a) \perp_t typ-uinfo-t TYPE('b)*

using *tag-disj-def unfolding sub-typ-def* **by** *blast*

from *ptr-valid-p* **have** *hrs-htd h \models_t p*

by (*simp add: ptr-valid-h-t-valid*)

from *clift-heap-update-same [OF this disj]*

have *clift (hrs-mem-update (heap-update p v) h) = (clift h::'b ptr \Rightarrow 'b option)*

by *simp*

then show *?thesis*

apply –

apply (*rule ext*)

by (*metis hrs-htd-mem-update pltft-None pltft-clift-conv*)

qed

theorem *disjoint-type-update-h-val*:

fixes *p::'a::mem-type ptr*

fixes *q::'b::mem-type ptr*

assumes *unrelated1*: $\neg \text{TYPE}('a) \leq_{\tau} \text{TYPE}('b)$

assumes *unrelated2*: $\neg \text{TYPE}('b) \leq_{\tau} \text{TYPE}('a)$

assumes *typed-p*: *hrs-htd h \models_t p*

assumes *typed-q*: *hrs-htd h \models_t q*

shows *h-val (hrs-mem (hrs-mem-update (heap-update p v) h)) q = h-val (hrs-mem h) q*

proof –

from *unrelated1 unrelated2* **have** *disj*: *typ-uinfo-t TYPE('a) \perp_t typ-uinfo-t TYPE('b)*

using *tag-disj-def unfolding sub-typ-def* **by** *blast*

from *clift-heap-update-same [OF typed-p disj]*

have *clift (hrs-mem-update (heap-update p v) h) q = clift h q*

by *simp*

with *typed-p typed-q*

show *?thesis*

by (*metis clift-Some-eq-valid h-val-clift hrs-mem-def prod.collapse*)

qed

lemma *intvl-Suc-0*: $\{x \text{ ..+ Suc } 0\} = \{x\}$
by (*auto simp: intvl-def*)

lemma *ptr-span-subset-of-addressable-field*:
fixes $q :: 'a :: \text{mem-type ptr}$
shows *addressable-field* $t\ p$ (*typ-uinfo-t* $\text{TYPE}('a)$) \implies
 $\text{ptr-val } q = x + \text{word-of-nat } (\text{field-offset-untyped } t\ p) \implies$
 $\text{ptr-span } q \subseteq \{x \text{ ..+ size-td } t\}$
using *intvl-subset-of-addressable-field*[*of* $t\ p$ *typ-uinfo-t* $\text{TYPE}('a)$ x]
by (*simp flip: size-of-tag*)

lemma *addressable-field-wf-descD*:
assumes *path*: *addressable-field* t *path* u
shows $t = u \vee (\text{wf-desc } t \wedge \text{wf-desc } u)$
using *addressable-fieldD-mem-type-u*[*OF* *path*] *wf-desc-addressable-field*[*OF* *path*]
by (*blast intro: mem-type-u.wf-desc*)

lemma *addressable-field-wf-size-descD*:
assumes *path*: *addressable-field* t *path* u
shows $t = u \vee (\text{wf-size-desc } t \wedge \text{wf-size-desc } u)$
using *addressable-fieldD-mem-type-u*[*OF* *path*] *wf-size-desc-addressable-field*[*OF* *path*]
by (*blast intro: mem-type-u.wf-size-desc*)

lemma *root-ptr-valid-u-ptr-valid-u-cases*:
fixes $p\ q :: \text{addr}$
assumes *root-valid-p*: *root-ptr-valid-u* $t\ d\ p$
assumes *valid-q*: *ptr-valid-u* $s\ d\ q$
shows $\{p \text{ ..+ size-td } t\} \cap \{q \text{ ..+ size-td } s\} = \{\}$ \vee
 $(\exists \text{path. } \text{addressable-field } t\ \text{path } s \wedge q = p + \text{of-nat } (\text{field-offset-untyped } t\ \text{path}))$
proof (*cases* $\{p \text{ ..+ size-td } t\} \cap \{q \text{ ..+ size-td } s\} = \{\}$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
from *valid-q*
obtain *root path x* **where**
 $\text{path}: \text{addressable-field } \text{root } \text{path } s$ **and**
 $\text{root-ptr-valid-u}: \text{root-ptr-valid-u } \text{root } d\ x$ **and**
 $q: q = x + \text{word-of-nat } (\text{field-offset-untyped } \text{root } \text{path})$
by (*clarsimp simp add: ptr-valid-def elim!: ptr-valid-u.cases*)

then have *contained*: $\{q \text{ ..+ size-td } s\} \subseteq \{x \text{ ..+ size-td } \text{root}\}$
using *intvl-subset-of-addressable-field*[*OF* *path*] **by** (*simp add: size-of-def*)

from *root-valid-p* **have** *root-ptr-valid-u-p*: *root-ptr-valid-u* $t\ d\ p$
by (*simp add: root-ptr-valid-root-ptr-valid-u-conv*)
from *root-ptr-valid-u-cases* [*OF* *root-ptr-valid-u-p* *root-ptr-valid-u*] *False* *contained*

```

obtain  $t = \text{root } p = x$ 
  by (metis (no-types, lifting) disj-subset preorder-class.dual-order.refl)
with path q False show ?thesis
  by auto
qed

lemma root-ptr-valid-u-ptr-valid-cases:
  fixes  $p :: \text{addr}$  and  $q :: 'b::\text{mem-type ptr}$ 
  assumes root-valid-p: root-ptr-valid-u t d p
  assumes valid-q: ptr-valid d q
  shows  $\{p ..+ \text{size-td } t\} \cap \text{ptr-span } q = \{\}$   $\vee$ 
     $(\exists \text{path. addressable-field } t \text{ path } (\text{typ-uinfo-t TYPE('b)}) \wedge$ 
       $\text{ptr-val } q = p + \text{of-nat } (\text{field-offset-untyped } t \text{ path}))$ 
  using root-ptr-valid-u-ptr-valid-u-cases[OF root-valid-p valid-q[unfolded ptr-valid-def]]
  by (simp add: size-of-def)

lemma root-ptr-valid-ptr-valid-cases:
  fixes  $p:: 'a::\text{mem-type ptr}$ 
  fixes  $q:: 'b::\text{mem-type ptr}$ 
  assumes root-valid-p: root-ptr-valid d p
  assumes valid-q: ptr-valid d q
  shows  $\text{ptr-span } p \cap \text{ptr-span } q = \{\}$   $\vee$ 
     $(\exists \text{path. addressable-field } (\text{typ-uinfo-t TYPE('a)}) \text{ path } (\text{typ-uinfo-t TYPE('b)})$ 
 $\wedge$ 
     $q = \text{PTR } ('b) \ \&(p \rightarrow \text{path}))$ 
  using root-ptr-valid-u-ptr-valid-cases[OF
    root-valid-p[unfolded root-ptr-valid-root-ptr-valid-u-conv] valid-q]
  by (simp add: size-of-def field-lvalue-def ptr-val-conv field-offset-def)

lemma stack-allocs-ptr-valid-cases1:
  fixes  $p:: 'a::\text{mem-type ptr}$ 
  fixes  $q:: 'b::\text{mem-type ptr}$ 
  assumes alloc: (p, d') ∈ stack-allocs n S TYPE('a) d
  assumes q-not-stack-byte: typ-uinfo-t TYPE('b) ≠ typ-uinfo-t TYPE(stack-byte)
  assumes valid-q: ptr-valid d' q
  shows  $(\exists i \text{ path. } i < n \wedge$ 
     $\text{addressable-field } (\text{typ-uinfo-t TYPE('a)}) \text{ path } (\text{typ-uinfo-t TYPE('b)}) \wedge$ 
     $q = \text{PTR } ('b) \ \&((p +_p \text{int } i) \rightarrow \text{path})) \wedge \neg \text{ptr-valid } d \ q) \vee$ 
     $(\text{ptr-valid } d \ q \wedge \{\text{ptr-val } p ..+ n * \text{size-of TYPE('a)}\} \cap \text{ptr-span } q = \{\})$ 
proof –
  from valid-q
  obtain root path x where
    path: addressable-field root path (typ-uinfo-t TYPE('b)) and
    root-ptr-valid-u: root-ptr-valid-u root d' x and
    q: ptr-val q = x + word-of-nat (field-offset-untyped root path)
  by (clarsimp simp add: ptr-valid-def elim!: ptr-valid-u.cases)
from path q-not-stack-byte have root-not-stack-byte: root ≠ typ-uinfo-t TYPE(stack-byte)
  using addressable-field-sub-typ sub-typ-stack-byte-u by blast

```



```

from stack-allocs-root-ptr-valid-u-cases [OF alloc root-not-stack-byte] root-ptr-valid-u
consider i where  $i < n$   $x = \text{ptr-val } (p +_p \text{ int } i)$   $\text{root} = \text{typ-uinfo-t } \text{TYPE}('a)$ 
  | root-ptr-valid-u root d x by auto
then show ?thesis
proof cases
  case 1

    have  $\text{ptr-span } q \subseteq \text{ptr-span } (p +_p \text{ int } i)$ 
      using ptr-span-subset-of-addressable-field[OF path q] by (simp add: size-of-def
1)
    also have  $\dots \subseteq \{\text{ptr-val } p..+n * \text{size-of } \text{TYPE}('a)\}$ 
      using alloc 1(1) by (rule stack-allocs-contained)
    finally have  $\text{ptr-val } q \in \{\text{ptr-val } p..+n * \text{size-of } \text{TYPE}('a)\}$ 
      using mem-type-self by auto
    then have  $\text{root-q: root-ptr-valid } d$  (PTR-COERCE('b  $\rightarrow$  stack-byte) q)
      using stack-allocs-cases[OF alloc] by (auto simp add: q 1 simp flip: Ptr-ptr-coerce)
    have False if  $q': \text{ptr-valid } d q$ 
      using root-ptr-valid-ptr-valid-cases[OF root-q q'] q-not-stack-byte
    by (auto simp add: intvl-Suc-0 dest!: addressable-field-sub-typ sub-typ-stack-byte-u)
    then have  $\neg \text{ptr-valid } d q$  by auto
    with 1 path show ?thesis
      by (auto simp: ptr-val-conv q field-lvalue-def field-offset-def
        intro!: exI[of - i] exI[of - path])
  next
  case 2
  then have  $\text{ptr-valid } d q$ 
    by (auto simp add: ptr-valid-def q intro!: ptr-valid-u.intros path)
  with stack-allocs-disjoint [OF alloc q-not-stack-byte ptr-valid-h-t-valid, of q]
  show ?thesis by simp
qed
qed

```

lemma *stack-allocs-ptr-valid-cases1'* [*consumes 3, case-names addressable-field disjoint*]:

```

fixes  $p:: 'a::\text{mem-type } \text{ptr}$ 
fixes  $q:: 'b::\text{mem-type } \text{ptr}$ 
assumes  $\text{alloc: } (p, d') \in \text{stack-allocs } n \mathcal{S} \text{TYPE}('a) d$ 
assumes  $q\text{-not-stack-byte: typ-uinfo-t } \text{TYPE}('b) \neq \text{typ-uinfo-t } \text{TYPE}(\text{stack-byte})$ 
assumes  $\text{valid-q: ptr-valid } d' q$ 
assumes  $\text{addressable-field: } \bigwedge i \text{ path. } i < n \implies$ 
   $\text{addressable-field } (\text{typ-uinfo-t } \text{TYPE}('a)) \text{ path } (\text{typ-uinfo-t } \text{TYPE}('b)) \implies$ 
   $q = \text{PTR } ('b) \ \&\&((p +_p \text{ int } i) \rightarrow \text{path}) \implies \neg \text{ptr-valid } d q \implies P$ 
assumes  $\text{disjoint: ptr-valid } d q \implies \{\text{ptr-val } p ..+ n * \text{size-of } \text{TYPE}('a)\} \cap$ 
 $\text{ptr-span } q = \{\} \implies P$ 
shows  $P$ 
using stack-allocs-ptr-valid-cases1 [OF alloc q-not-stack-byte valid-q] addressable-field disjoint by blast

```

lemma *stack-allocs-ptr-valid-cases2*:

```

fixes p: 'a::mem-type ptr
fixes q: 'b::mem-type ptr
assumes alloc: (p, d') ∈ stack-allocs n S TYPE('a) d
assumes q-not-stack-byte: typ-uinto-t TYPE('b) ≠ typ-uinto-t TYPE(stack-byte)
assumes valid-q: ptr-valid d q
shows ptr-valid d' q
proof –
  from valid-q
  obtain root path x where
    path: addressable-field root path (typ-uinto-t TYPE('b)) and
    root-ptr-valid-u: root-ptr-valid-u root d x and
    x: ptr-val q = x + word-of-nat (field-offset-untyped root path)
    by (clarsimp simp add: ptr-valid-def ptr-valid-u.simps)

from path q-not-stack-byte have root-not-stack-byte: root ≠ typ-uinto-t TYPE(stack-byte)
  using addressable-field-sub-typ sub-typ-stack-byte-u by blast

from stack-allocs-root-ptr-valid-u-other [OF alloc root-ptr-valid-u root-not-stack-byte]
have root-ptr-valid-u root d' x .
with path x
show ptr-valid d' q
  using ptr-valid-def ptr-valid-u.simps by blast
qed

```

```

lemma stack-allocs-ptr-valid-cases3:
  fixes p: 'a::mem-type ptr
  fixes q: 'b::mem-type ptr
  assumes alloc: (p, d') ∈ stack-allocs n S TYPE('a) d
  assumes q-not-stack-byte: typ-uinto-t TYPE('b) ≠ typ-uinto-t TYPE(stack-byte)
  assumes i: i < n
  assumes path: addressable-field (typ-uinto-t TYPE('a)) path (typ-uinto-t TYPE('b))
  assumes q: q = PTR ('b) &((p +p int i)→path)
  shows ptr-valid d' q
  apply (simp add: q ptr-valid-def field-lvalue-def field-offset-def)
  apply (intro ptr-valid-u.intros[OF - path])
  apply (subst stack-allocs-root-ptr-valid-u-cases[OF alloc stack-allocs-no-stack-byte[OF alloc]])
  apply (auto intro: i)
  done

```

```

theorem stack-allocs-ptr-valid-cases:
  fixes p: 'a::mem-type ptr
  fixes q: 'b::mem-type ptr
  assumes alloc: (p, d') ∈ stack-allocs n S TYPE('a) d
  assumes q-not-stack-byte: typ-uinto-t TYPE('b) ≠ typ-uinto-t TYPE(stack-byte)
  shows ptr-valid d' q ↔
    (((∃ i path. i < n ∧ addressable-field (typ-uinto-t TYPE('a)) path (typ-uinto-t
    TYPE('b)) ∧
      q = PTR ('b) &((p +p int i)→path)) ∧ ¬ ptr-valid d q) ∨

```

```

    (ptr-valid d q  $\wedge$  {ptr-val p ..+ n * size-of TYPE('a)}  $\cap$  ptr-span q = {}))
using stack-allocs-ptr-valid-cases1 [OF alloc q-not-stack-byte]
    stack-allocs-ptr-valid-cases2 [OF alloc q-not-stack-byte]
    stack-allocs-ptr-valid-cases3 [OF alloc q-not-stack-byte]
by blast

lemma stack-releases-ptr-valid-cases1 :
  fixes p: 'a::mem-type ptr
  fixes q: 'b::mem-type ptr
  assumes p-not-stack-byte: typ-uinto-t TYPE('a)  $\neq$  typ-uinto-t TYPE(stack-byte)
  assumes q-not-stack-byte: typ-uinto-t TYPE('b)  $\neq$  typ-uinto-t TYPE(stack-byte)
  assumes valid-q: ptr-valid (stack-releases n p d) q
  shows {ptr-val p ..+ n * size-of TYPE('a::mem-type)}  $\cap$  ptr-span q = {}  $\wedge$ 
    ptr-valid d q
proof -
  from valid-q
  obtain root path x where
    path: addressable-field root path (typ-uinto-t TYPE('b)) and
    root-ptr-valid-u: root-ptr-valid-u root (stack-releases n p d) x and
    x: ptr-val q = x + word-of-nat (field-offset-untyped root path)
  by (clarsimp simp add: ptr-valid-def ptr-valid-u.simps)
  from path q-not-stack-byte have root-not-stack-byte: root  $\neq$  typ-uinto-t TYPE(stack-byte)
  using addressable-field-sub-typ sub-typ-stack-byte-u by blast

  from stack-releases-root-ptr-valid-u-cases [OF p-not-stack-byte root-not-stack-byte,
of n p d x ] root-ptr-valid-u
  obtain root-cases:
    {ptr-val p..+n * size-of TYPE('a)}  $\cap$  {x..+size-td root} = {}
    root-ptr-valid-u root d x by auto
  moreover
  have ptr-span q  $\subseteq$  {x..+size-td root}
  using path x by (rule ptr-span-subset-of-addressable-field)
  moreover
  from root-cases(2) path x have ptr-valid d q
  using ptr-valid-def ptr-valid-u.simps by blast
  ultimately
  show ?thesis by blast
qed

lemma stack-releases-ptr-valid-cases2 :
  fixes p: 'a::mem-type ptr
  fixes q: 'b::mem-type ptr
  assumes p-not-stack-byte: typ-uinto-t TYPE('a)  $\neq$  typ-uinto-t TYPE(stack-byte)
  assumes q-not-stack-byte: typ-uinto-t TYPE('b)  $\neq$  typ-uinto-t TYPE(stack-byte)
  assumes root-p:  $\bigwedge i. i < n \implies$  root-ptr-valid d (p +p int i)
  assumes disj: {ptr-val p ..+ n * size-of TYPE('a::mem-type)}  $\cap$  ptr-span q =
  {}
  assumes valid-q: ptr-valid d q
  shows ptr-valid (stack-releases n p d) q

```

```

proof –
  from valid-q
  obtain root path x where
    path: addressable-field root path (typ-uinfo-t TYPE('b)) and
    root-ptr-valid-u: root-ptr-valid-u root d x and
    x: ptr-val q = x + word-of-nat (field-offset-untyped root path)
    by (clarsimp simp add: ptr-valid-def ptr-valid-u.simps)

from path q-not-stack-byte have root-not-stack-byte: root ≠ typ-uinfo-t TYPE(stack-byte)
  using addressable-field-sub-typ sub-typ-stack-byte-u by blast

have contained: ptr-span q ⊆ {x..+size-td root}
  using path x by (rule ptr-span-subset-of-addressable-field)

have  $\{ptr-val p..+n * size-of TYPE('a)\} \cap \{x..+size-td root\} = \{\}$ 
proof (rule ccontr)
  assume  $\{ptr-val p..+n * size-of TYPE('a)\} \cap \{x..+size-td root\} \neq \{\}$ 
  then obtain i where i: i < n and overlap: ptr-span (p +p int i) ∩ {x..+size-td
root} ≠ {}
  by (meson array-to-index-span disjoint-iff)
  from root-p [OF i] have root-ptr-valid-u (typ-uinfo-t TYPE('a)) d (ptr-val (p
+p int i))
  by (simp add: root-ptr-valid-root-ptr-valid-u-conv)
  from root-ptr-valid-u-cases [OF root-ptr-valid-u this] overlap
  obtain root = typ-uinfo-t TYPE('a) x = ptr-val (p +p int i)
  by (simp add: inf-commute size-of-tag)
  then have ptr-span (p +p int i) = {x..+size-td root}
  by (simp add: size-of-def)
  with contained have ptr-span q ⊆ ptr-span (p +p int i) by simp
  with disj i show False
  apply (simp add: array-index-span-conv)
  by (smt (verit, ccfv-SIG) UN-I disjoint-iff mem-type-self ord-class.lessThan-iff
subsetD)
  qed

with stack-releases-root-ptr-valid-u-cases [OF p-not-stack-byte root-not-stack-byte]
root-ptr-valid-u

  have root-ptr-valid-u root (stack-releases n p d) x by auto
  with path x
  show ptr-valid (stack-releases n p d) q
  using ptr-valid-def ptr-valid-u.simps by blast
qed

lemma stack-releases-ptr-valid-cases3:
  fixes p:: 'a::mem-type ptr
  fixes q:: 'b::mem-type ptr
  assumes p-not-stack-byte: typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE(stack-byte)

```

assumes *q-not-stack-byte*: $\text{typ-ufinfo-t } \text{TYPE}('b) \neq \text{typ-ufinfo-t } \text{TYPE}(\text{stack-byte})$
assumes *root-p*: $\bigwedge i. i < n \implies \text{root-ptr-valid } d (p +_p \text{int } i)$
assumes *valid-q*: $\text{ptr-valid } d q$
assumes *invalid-q*: $\neg \text{ptr-valid } (\text{stack-releases } n p d) q$
shows $\exists \text{path } i.$
 $\text{addressable-field } (\text{typ-ufinfo-t } \text{TYPE}('a)) \text{ path } (\text{typ-ufinfo-t } \text{TYPE}('b)) \wedge i < n$
 \wedge
 $q = \text{PTR}('b) \ \&(p +_p \text{int } i \rightarrow \text{path})$
proof –
from *valid-q invalid-q stack-releases-ptr-valid-cases2* [*OF p-not-stack-byte q-not-stack-byte root-p - valid-q, of n*]
have $\{\text{ptr-val } p..+n * \text{size-of } \text{TYPE}('a)\} \cap \text{ptr-span } q \neq \{\}$ **by** *auto*
then obtain *i* **where** *i-bound*: $i < n$ **and** *overlap*: $\text{ptr-span } (p +_p \text{int } i) \cap \text{ptr-span } q \neq \{\}$
by (*auto simp add: array-index-span-conv*)

from *root-ptr-valid-ptr-valid-cases* [*OF root-p [OF i-bound] valid-q, simplified overlap, simplified*] *i-bound*
show *?thesis* **by** *blast*
qed

lemma *stack-releases-ptr-valid-cases*:

fixes *p*: $'a::\text{mem-type } \text{ptr}$
fixes *q*: $'b::\text{mem-type } \text{ptr}$
assumes *p-not-stack-byte*: $\text{typ-ufinfo-t } \text{TYPE}('a) \neq \text{typ-ufinfo-t } \text{TYPE}(\text{stack-byte})$
assumes *q-not-stack-byte*: $\text{typ-ufinfo-t } \text{TYPE}('b) \neq \text{typ-ufinfo-t } \text{TYPE}(\text{stack-byte})$
assumes *root-p*: $\bigwedge i. i < n \implies \text{root-ptr-valid } d (p +_p \text{int } i)$
shows $\text{ptr-valid } (\text{stack-releases } n p d) q \longleftrightarrow$
 $\text{ptr-valid } d q \wedge \{\text{ptr-val } p ..+ n * \text{size-of } \text{TYPE}('a::\text{mem-type})\} \cap \text{ptr-span } q$
 $= \{\}$
using *stack-releases-ptr-valid-cases1* [*OF p-not-stack-byte q-not-stack-byte*]
stack-releases-ptr-valid-cases2 [*OF p-not-stack-byte q-not-stack-byte root-p*] **by**
blast

lemma *ptr-valid-same-type-neq-no-overlap-conv*:

fixes *p*: $'a::\text{mem-type } \text{ptr}$
fixes *q*: $'a::\text{mem-type } \text{ptr}$
assumes *valid-p*: $\text{ptr-valid } d p$
assumes *valid-q*: $\text{ptr-valid } d q$
shows $p \neq q \longleftrightarrow \text{ptr-span } p \cap \text{ptr-span } q = \{\}$
using *cvalid-same-type-neq-disjoint assms ptr-valid-h-t-valid*
by *blast*

theorem *stack-allocs-the-default-plift* [*stack-the-default-plift*]:

fixes *p*: $'a::\text{xmem-type } \text{ptr}$
fixes *q*: $'b::\text{mem-type } \text{ptr}$
assumes *alloc*: $(p, d') \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a) \ (\text{hrs-htd } h)$
assumes *q-not-stack-byte*: $\text{typ-ufinfo-t } \text{TYPE}('b) \neq \text{typ-ufinfo-t } \text{TYPE}(\text{stack-byte})$
shows *the-default c-type-class.zero*

```

      (plift (hrs-mem-update (fold (λi. heap-update (p +p int i) c-type-class.zero)
[0..n]) (hrs-htd-update (λ-. d') h)) q) =
      the-default c-type-class.zero (plift h q)
proof (cases ptr-valid d' q)
  case True
  from alloc q-not-stack-byte True
  show ?thesis
  proof (cases rule: stack-allocs-ptr-valid-cases1')
    case (addressable-field i path)
    from addressable-field obtain
      i: i < n and
      path: addressable-field (typ-uinfo-t TYPE('a)) path (typ-uinfo-t TYPE('b))
and
      q: q = PTR('b) &(p +p int i→path) and
      invalid: ¬ ptr-valid (hrs-htd h) q
      by blast
      from invalid have (plift h q) = None
      by (simp add: plift-None)
      moreover
      from stack-allocs-cases [OF alloc]
      obtain bound: unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card and
      not-null: ptr-val p ≠ 0
      by (metis Ptr-ptr-val c-guard-NULL-simp)

      from path obtain m where
        field-lookup (typ-uinfo-t TYPE('a)) path 0 = Some ((typ-uinfo-t TYPE('b)),
m)
        by (auto dest: addressable-fieldD-field-lookup)

      then obtain t where fl: field-lookup (typ-uinfo-t TYPE('a)) path 0 = Some (t,
m) and
        match: export-uinfo t = typ-uinfo-t TYPE('b)
        using field-lookup-uinfo-Some-rev by blast
        from fl have field-ti: field-ti TYPE('a) path = Some t
        using field-lookup-field-ti by blast
        from field-lookup-zero [OF fl match] have from-bytes-zero: from-bytes (access-ti0
t c-type-class.zero) = (c-type-class.zero::'b) .
        note h-val-field = h-val-field-from-bytes' [OF field-ti match [simplified typ-uinfo-t-def]]
        from h-val-fold [OF bound not-null i]
        have h-val-zero:
          h-val (fold (λi. heap-update (p +p int i) c-type-class.zero) [0..n] (hrs-mem
h)) (p +p int i) = c-type-class.zero .
        have path-zero:
          h-val (fold (λi. heap-update (p +p int i) c-type-class.zero) [0..n] (hrs-mem
h)) (PTR('b) &(p +p int i→path)) = c-type-class.zero
        by (simp add: h-val-field h-val-zero from-bytes-zero)

      from True i path q
      have (plift (hrs-mem-update (fold (λi. heap-update (p +p int i) c-type-class.zero)

```

```

[0..<n]) (hrs-htd-update (λ-. d') h)) q) = Some c-type-class.zero
  apply –
  apply (subst ptr-valid-plift-Some-hval)
  apply (simp add: hrs-htd-update)
  apply (simp add: path-zero hrs-mem-update)
  done
ultimately
show ?thesis by simp
next
case disjoint
then obtain valid-q: ptr-valid (hrs-htd h) q and
  disj: {ptr-val p..+n * size-of TYPE('a)} ∩ ptr-span q = {} by blast

from valid-q have (plift h q) = Some (h-val (hrs-mem h) q)
  by (simp add: ptr-valid-plift-Some-hval valid-q)
moreover
have h-val-eq:
  h-val (fold (λi. heap-update (p +p int i) c-type-class.zero) [0..<n] (hrs-mem
h)) q =
  h-val (hrs-mem h) q by (simp add: h-val-fold-zero-disjoint [OF disj])
from True have valid-q':
  ptr-valid (hrs-htd ((hrs-mem-update (fold (λi. heap-update (p +p int i)
c-type-class.zero) [0..<n])
  (hrs-htd-update (λ-. d') h)))) q
  by (simp add: hrs-htd-update)
have plift (hrs-mem-update (fold (λi. heap-update (p +p int i) c-type-class.zero)
[0..<n])
  (hrs-htd-update (λ-. d') h)) q = Some (h-val (hrs-mem h) q)
  using True by (simp add: ptr-valid-plift-Some-hval [OF valid-q'] hrs-mem-update
h-val-eq)
ultimately
show ?thesis
  by simp
qed
next
case False
from False
have invalid: ¬ ptr-valid (hrs-htd h) q
  using alloc q-not-stack-byte stack-allocs-ptr-valid-cases2 by blast
then
have plift (hrs-mem-update (fold (λi. heap-update (p +p int i) c-type-class.zero)
[0..<n]) (hrs-htd-update (λ-. d') h)) q = None
  using plift-None
  by (metis (no-types, lifting) False hrs-htd-mem-update hrs-htd-update)
moreover
have (plift h q) = None
  using invalid by (simp add: plift-None)
ultimately show ?thesis by simp
qed

```

theorem *stack-releases-the-default-plift* [*stack-the-default-plift*]:

fixes p : 'a::xmem-type ptr
fixes q : 'b::mem-type ptr
assumes $roots$: $\bigwedge i. i < n \implies \text{root-ptr-valid } (\text{hrs-htd } h) (p +_p \text{ int } i)$
assumes $p\text{-not-stack-byte}$: $\text{typ-winfo-t } \text{TYPE}('a) \neq \text{typ-winfo-t } \text{TYPE}(\text{stack-byte})$
assumes $q\text{-not-stack-byte}$: $\text{typ-winfo-t } \text{TYPE}('b) \neq \text{typ-winfo-t } \text{TYPE}(\text{stack-byte})$
shows $\text{the-default } c\text{-type-class.zero}$
 $(\text{plift } (\text{hrs-mem-update } (\text{fold } (\lambda i. \text{heap-update } (p +_p \text{ int } i) c\text{-type-class.zero}) [0..<n]) h)$
 $q) =$
 $\text{the-default } c\text{-type-class.zero } (\text{plift } (\text{hrs-htd-update } (\text{stack-releases } n p) h) q)$

proof ($\text{cases } \text{ptr-valid } (\text{stack-releases } n p (\text{hrs-htd } h)) q$)
case *True*
from *stack-releases-ptr-valid-cases1* [*OF p-not-stack-byte q-not-stack-byte True*]
obtain $\text{disjoint: } \{\text{ptr-val } p..+n * \text{size-of } \text{TYPE}('a)\} \cap \text{ptr-span } q = \{\}$ **and**
 $\text{valid-q: ptr-valid } (\text{hrs-htd } h) q$ **by** *blast*

from *True valid-q* **have** $\text{plift } (\text{hrs-htd-update } (\text{stack-releases } n p) h) q = \text{Some}$
 $(h\text{-val } (\text{hrs-mem } h) q)$
using *ptr-valid-plift-Some-hval*
by (*metis hrs-htd-update hrs-mem-htd-update*)

moreover
from *h-val-fold-zero-disjoint* [*OF disjoint*]
have $h\text{-val } (\text{fold } (\lambda i. \text{heap-update } (p +_p \text{ int } i) c\text{-type-class.zero}) [0..<n]) (\text{hrs-mem } h)$
 $q = h\text{-val } (\text{hrs-mem } h) q$.
with *valid-q*
have $(\text{plift } (\text{hrs-mem-update } (\text{fold } (\lambda i. \text{heap-update } (p +_p \text{ int } i) c\text{-type-class.zero}) [0..<n]) h) q) = \text{Some } (h\text{-val } (\text{hrs-mem } h) q)$
by (*simp add: hrs-mem-update ptr-valid-plift-Some-hval*)
ultimately show *?thesis* **by** *simp*

next
case *False*
from *False*
have $\text{eq1: plift } (\text{hrs-htd-update } (\text{stack-releases } n p) h) q = \text{None}$
by (*simp add: plift-None hrs-htd-update*)
show *?thesis*
proof ($\text{cases } \text{ptr-valid } (\text{hrs-htd } h) q$)
case *True*
show *?thesis*
proof ($\text{cases } n$)
case *0*
then show *?thesis*
using *False True* **by** (*auto simp add: stack-releases-def*)
next
case (*Suc n'*)


```

from roots [of 0] Suc
have not-null: ptr-val p ≠ 0
  by (metis CTypesDefs.ptr-add-def Ptr-ptr-val Zero-not-Suc add.right-neutral
    bot-nat-0.not-eq-extremum mem-type-self mult-zero-left of-int-0 root-ptr-valid-range-not-NULL
    roots semiring-1-class.of-nat-0)

from roots Suc
have bound: unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card
  by (metis array-to-index-span len-of-addr-card root-ptr-valid-range-not-NULL
    zero-not-in-intvl-no-overflow)

  from stack-releases-ptr-valid-cases3 [OF p-not-stack-byte q-not-stack-byte
    roots True False]
  obtain path i where
    path: addressable-field (typ-uinfo-t TYPE('a)) path (typ-uinfo-t TYPE('b))
and
  q: q = PTR('b) &(p +p int i → path) and
  i-bound: i < n
  by blast
from path obtain m where
  field-lookup (typ-uinfo-t TYPE('a)) path 0 = Some ((typ-uinfo-t TYPE('b)),
m)
  by (auto dest: addressable-fieldD-field-lookup)

  then obtain t where fl: field-lookup (typ-uinfo-t TYPE('a)) path 0 = Some
(t, m) and
  match: export-uinfo t = typ-uinfo-t TYPE('b)
  using field-lookup-uinfo-Some-rev by blast
  from fl have field-ti: field-ti TYPE('a) path = Some t
  using field-lookup-field-ti by blast
  from field-lookup-zero [OF fl match] have from-bytes-zero: from-bytes (access-ti0
t c-type-class.zero) = (c-type-class.zero::'b) .
  note h-val-field = h-val-field-from-bytes' [OF field-ti match [simplified typ-uinfo-t-def]]
  from h-val-fold [OF bound not-null i-bound]
  have h-val-zero:
    h-val (fold (λi. heap-update (p +p int i) c-type-class.zero) [0..n] (hrs-mem
h)) (p +p int i) = c-type-class.zero .
  have path-zero:
    h-val (fold (λi. heap-update (p +p int i) c-type-class.zero) [0..n] (hrs-mem
h)) (PTR('b) &(p +p int i → path)) = c-type-class.zero
  by (simp add: h-val-field h-val-zero from-bytes-zero)
  from True i-bound path q
  have plift (hrs-mem-update (fold (λi. heap-update (p +p int i) c-type-class.zero)
[0..n] h) q = Some c-type-class.zero
  apply -
  apply (subst ptr-valid-plift-Some-hval)
  apply (simp add: hrs-htd-update)
  apply (simp add: path-zero hrs-mem-update)
  done

```

```

    then show ?thesis by (simp add: eq1)
  qed
next
  case False
  from False
  have plift (hrs-mem-update (fold (λi. heap-update (p +p int i) c-type-class.zero)
[0..<n]) h) q = None
    using plift-None
    by (metis False hrs-htd-mem-update )
  with eq1 show ?thesis by simp
  qed
qed

```

lemma *h-val-heap-update-padding-heap-update-conv*:

```

  fixes p::'a::mem-type ptr
  fixes q::'b::mem-type ptr
  assumes valid-p: ptr-valid d p
  assumes valid-q: ptr-valid d q
  assumes lbs: length bs = size-of TYPE('a)
  shows h-val ((heap-update-padding p v bs h)) q = h-val (heap-update p v h) q
  using ptr-valid-cases-weak[OF valid-p valid-q]
proof (elim disjE exE conjE)
  assume disjnt (ptr-span p) (ptr-span q) then show ?thesis
    by (simp add: h-val-heap-update-padding-disjoint h-val-update-regions-disjoint
lbs disjnt-def)
next
  fix f assume f: addressable-field (typ-uinfo-t TYPE('a)) f (typ-uinfo-t TYPE('b))
    and q: q = PTR('b) &(p→f)
  from addressable-fieldD-field-ti[OF f]
  obtain t where t: field-ti TYPE('a) f = Some t export-uinfo t = typ-uinfo-t
TYPE('b)
    by auto

  let ?s = from-bytes o access-ti0 t :: 'a ⇒ 'b
  interpret f: nested-field' t f ?s update-ti t o to-bytes-p
    using t by unfold-locales simp-all

```

show ?thesis **unfolding** q **using** lbs

by (subst (1 2) f.h-val-field) (simp add: h-val-heap-update-padding)

next

```

  fix f assume f: addressable-field (typ-uinfo-t TYPE('b)) f (typ-uinfo-t TYPE('a))
    and p: p = PTR('a) &(q→f)
  from addressable-fieldD-field-ti[OF f]
  obtain t where t: field-ti TYPE('b) f = Some t export-uinfo t = typ-uinfo-t
TYPE('a)
    by auto

```

let ?s = from-bytes o access-ti₀ t :: 'b ⇒ 'a

interpret f: nested-field' t f ?s update-ti t o to-bytes-p

```

using t by unfold-locales simp-all

note q = ptr-valid-h-t-valid[OF valid-q]
show ?thesis unfolding p
  apply (subst f.h-val-field-lvalue-update-padding[OF q q lbs])
  apply (subst f.h-val-field-lvalue-update[OF q q])
  apply simp
  done
qed

lemma plift-heap-update-padding-heap-update-conv:
  fixes p::'a::mem-type ptr
  fixes q::'b::mem-type ptr
  assumes valid-p: ptr-valid (hrs-htd h) p
  assumes lbs: length bs = size-of TYPE('a)
  shows plift (hrs-mem-update (heap-update-padding p v bs) h) q = plift (hrs-mem-update
(heap-update p v) h) q
  using h-val-heap-update-padding-heap-update-conv
  by (metis hrs-htd-mem-update hrs-mem-update lbs valid-p wf-ptr-valid.plift-None
wf-ptr-valid.ptr-valid-plift-Some-hval wf-ptr-valid-axioms)

theorem plift-heap-update-padding-heap-update-pointless-conv [plift-heap-update-padding-heap-update-conv]:
  fixes p::'a::mem-type ptr
  assumes valid-p: ptr-valid (hrs-htd h) p
  assumes lbs: length bs = size-of TYPE('a)
  shows plift (hrs-mem-update (heap-update-padding p v bs) h) = plift (hrs-mem-update
(heap-update p v) h)
  apply (rule ext)
  using plift-heap-update-padding-heap-update-conv assms
  by blast

lemma ptr-valid-stack-byte-disjoint:
  fixes q::'a::{mem-type, stack-type} ptr
  fixes p::stack-byte ptr
  assumes root-ptr-valid h p
  assumes ptr-valid h q
  shows ptr-span p ∩ ptr-span q = {}
  using assms local.ptr-valid-h-t-valid no-stack-byte root-ptr-valid-not-subtype-disjoint
sub-typ-stack-byte
  by blast

lemma h-val-heap-update-list-stack-byte:
  fixes q::'a::{mem-type, stack-type} ptr
  assumes valid-p: root-ptr-valid d (p::stack-byte ptr)
  assumes valid-q: ptr-valid d q
  assumes lbs: length bs = size-of TYPE(stack-byte)
  shows h-val (heap-update-list (ptr-val p) bs h) q = h-val h q
  using ptr-valid-stack-byte-disjoint [OF valid-p valid-q] lbs
  by (simp add: h-val-def heap-list-update-disjoint-same)

```

lemma *h-val-heap-update-list-stack-byte'*:
fixes $q::'a::\{\text{mem-type}, \text{stack-type}\}$ *ptr*
fixes $p::\text{stack-byte}$ *ptr*
assumes $\text{valid-p}: \bigwedge i. i < \text{length } bs \implies \text{root-ptr-valid } d (p +_p \text{int } i)$
assumes $\text{valid-q}: \text{ptr-valid } d q$
shows $h\text{-val } (\text{heap-update-list } (\text{ptr-val } p) bs h) q = h\text{-val } h q$
using *valid-p*
proof (*induct bs arbitrary: p h*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons b bs*)
from *Cons.prem*s *Cons.prem*s [*of 0*] **obtain**
 $\text{valid-bs}: \bigwedge i. i < \text{Suc } (\text{length } bs) \implies \text{root-ptr-valid } d (p +_p \text{int } i)$ **and**
 $\text{valid-p}: \text{root-ptr-valid } d p$ **by** *auto*
from valid-bs **have** $\text{valid-bs}' : \bigwedge i. i < \text{length } bs \implies \text{root-ptr-valid } d ((p +_p 1) +_p \text{int } i)$
apply (*clarsimp simp add: ptr-add-def*)
by (*metis Ex-less-Suc group-cancel.add1 less-trans-Suc of-nat-Suc*)

have $\text{length } [b] = \text{size-of TYPE}(\text{stack-byte})$ **by** *auto*
note $b\text{-eq} = h\text{-val-heap-update-list-stack-byte}$ [*OF valid-p valid-q this*]
have $\text{val-eq}: \text{ptr-val } (p +_p 1) = \text{ptr-val } p + 1$
by (*auto simp add: ptr-add-def*)

note $\text{hyp} = \text{Cons.hyps}$ [*OF valid-bs', simplified val-eq*]
show *?case*
apply (*simp add: hyp*)
using $b\text{-eq}$
apply (*auto simp add: fun-upd-def*)
done
qed

lemma *plift-heap-update-list-stack-byte*:
fixes $q::'a::\{\text{mem-type}, \text{stack-type}\}$ *ptr*
fixes $p::\text{stack-byte}$ *ptr*
assumes $\text{valid-p}: \bigwedge i. i < \text{length } bs \implies \text{root-ptr-valid } (h\text{rs-htd } h) (p +_p \text{int } i)$
shows $\text{plift } (h\text{rs-mem-update } (\text{heap-update-list } (\text{ptr-val } p) bs) h) q = \text{plift } h q$
using *h-val-heap-update-list-stack-byte'*
by (*metis hrs-htd-mem-update hrs-mem-update valid-p wf-ptr-valid.plift-None wf-ptr-valid.ptr-valid-plift-Some-hval wf-ptr-valid-axioms*)

lemma *plift-heap-update-list-stack-byte-pointless*[*plift-heap-update-list-stack-byte*]:
fixes $p::\text{stack-byte}$ *ptr*
assumes $\text{valid-p}: \bigwedge i. i < \text{length } bs \implies \text{root-ptr-valid } (h\text{rs-htd } h) (p +_p \text{int } i)$
shows $\text{plift } (h\text{rs-mem-update } (\text{heap-update-list } (\text{ptr-val } p) bs) h) =$
 $(\text{plift } h::'a::\{\text{mem-type}, \text{stack-type}\} \text{ptr} \Rightarrow 'a \text{ option})$

using *assms plift-heap-update-list-stack-byte* **by** *blast*

lemma *plift-eq-plift*:

hrs-htd h = hrs-htd h' \implies

*($\bigwedge p::'a$ ptr. ptr-valid (hrs-htd h') p \implies
h-val (hrs-mem h) p = h-val (hrs-mem h') p) \implies
(plift h::'a::mem-type ptr \Rightarrow 'a option) = plift h'*

apply (*rule ext*)

apply (*cases h; cases h'*)

apply (*simp add: plift-def lift-t-if h-t-valid-def root-ptr-valid-def hrs-htd-def hrs-mem-def*)

done

definition *addressable-fields* :: 'a::mem-type itself \Rightarrow (field-name list \times 'a xtyp-info) list **where**

*addressable-fields TYPE('a) = (case map-of \mathcal{T} (typ-uinfo-t TYPE('a)) of
None \Rightarrow []
| Some fs \Rightarrow map (λf . (f, the (field-ti TYPE('a) f))) fs)*

abbreviation *merge-addressable-fields* :: 'a::mem-type \Rightarrow 'a \Rightarrow 'a **where**

merge-addressable-fields \equiv merge-ti-list (map snd (addressable-fields TYPE('a)))

definition *read-dedicated-heap* :: heap-raw-state \Rightarrow 'a::mem-type typ-heap-g **where**

read-dedicated-heap h p = merge-addressable-fields ZERO('a) (the-default ZERO('a) (plift h p))

definition *write-dedicated-heap* :: 'a::mem-type ptr \Rightarrow 'a \Rightarrow heap-raw-state \Rightarrow heap-raw-state **where**

write-dedicated-heap p v = hrs-mem-update (heap-upd p (λ old. merge-addressable-fields old v))

lemma *mem-addressable-fields*:

assumes *F: map-of \mathcal{T} (typ-uinfo-t TYPE('a::mem-type)) = Some F*

shows *f \in set F \implies field-ti TYPE('a) f = Some u \implies (f, u) \in set (addressable-fields TYPE('a))*

using *wf- \mathcal{T} [of typ-uinfo-t TYPE('a)]*

by (*force simp add: addressable-fields-def F image-iff list-all-iff field-ti-def*)

lemma *mem-snd-addressable-fields*:

assumes *F: map-of \mathcal{T} (typ-uinfo-t TYPE('a::mem-type)) = Some F f \in set F*

field-ti TYPE('a) f = Some u

shows *u \in snd ' set (addressable-fields TYPE('a))*

using *mem-addressable-fields[OF assms]* **by** (*force simp: image-iff*)

lemma *list-all-field-ti-addressable-fields*:

list-all (λ (f, u). field-ti TYPE('a) f = Some u) (addressable-fields TYPE('a::mem-type))

apply (*clarsimp simp add: addressable-fields-def split: option.splits*)

subgoal premises *prems*

using *prems[THEN wf- \mathcal{T}]*

```

unfolding list.pred-map
by (rule list.pred-mono-strong)
  (auto simp: list-all-iff field-ti-def typ-uinfo-t-def
    dest!: field-lookup-export-uinfo-Some-rev field-lookup-field-ti')
done

lemma distinct-disj-fn-addressable-fields:
  distinct-prop disj-fn (map fst (addressable-fields TYPE('a::mem-type)))
using wf- $\mathcal{T}'$ 
by (auto simp: list-all-iff addressable-fields-def comp-def dest!: map-of-SomeD
  split: option.splits)

lemma list-all-field-ti-snd-addressable-fields:
  list-all ( $\lambda u. \exists f. \text{field-ti } TYPE('a) f = \text{Some } u$ ) (map snd (addressable-fields
  TYPE('a::mem-type)))
unfolding list.pred-map
using list-all-field-ti-addressable-fields
by (rule list.pred-mono-strong) auto

sublocale merge-addressable-fields: is-scene merge-addressable-fields :: 'a::xmem-type
 $\Rightarrow$  'a  $\Rightarrow$  'a
using list-all-field-ti-snd-addressable-fields by (rule is-scene-merge-ti-list)

lemma ptr-valid-u-cases-same-type:
assumes t: mem-type-u t and a: ptr-valid-u t d a and b: ptr-valid-u t d b
shows a = b  $\vee$  disjnt {a..+size-td t} {b..+size-td t}
using ptr-valid-u-cases-weak[OF t a b] by auto

lemma read-dedicated-heap-heap-update-list-stack-byte-pointless[simp]:
fixes p::stack-byte ptr
assumes valid-p:  $\bigwedge i. i < \text{length } bs \implies \text{root-ptr-valid } (\text{hrs-htd } h) (p +_p \text{int } i)$ 
shows read-dedicated-heap (hrs-mem-update (heap-update-list (ptr-val p) bs) h)
=
  (read-dedicated-heap h::'a::{mem-type, stack-type} ptr  $\Rightarrow$  'a)
unfolding read-dedicated-heap-def
by (simp add: plift-heap-update-list-stack-byte-pointless[OF valid-p])

lemma read-dedicated-heap-write-dedicated-heap-ne[simp]:
fixes p :: 'a::xmem-type ptr and q :: 'b::xmem-type ptr
assumes ne: typ-uinfo-t TYPE('a)  $\neq$  typ-uinfo-t TYPE('b)
assumes p: ptr-valid (hrs-htd h) p
shows read-dedicated-heap (write-dedicated-heap p x h) q = read-dedicated-heap
h q
proof cases
let ?A = typ-uinfo-t TYPE('a) and ?B = typ-uinfo-t TYPE('b)
assume q: ptr-valid (hrs-htd h) q
from ptr-valid-cases-weak[OF p q]
show ?thesis
proof (elim disjE exE conjE)

```

```

assume disjnt (ptr-span p) (ptr-span q) with p q show ?thesis
  by (simp add: read-dedicated-heap-def write-dedicated-heap-def disjnt-def
      hrs-mem-update-heap-upd plift-disjoint-region)
next
fix f' assume f'-n: addressable-field ?A f' ?B and q-eq: q = PTR('b) &(p→f')

obtain F f1 f2 u where f'-eq: f' = f1 @ f2
  and F: f1 ∈ set F map-of T ?A = Some F
  and f-u: field-ti TYPE('a) f1 = Some u
  and addressable-field (export-uinfo u) f2 ?B
  using f'-n ne by (cases rule: addressable-field.cases) (auto simp: field-ti-def
dest!: field-lookup-uinfo-Some-rev)

obtain v n where v: export-uinfo v = ?B
  and f-fs: field-lookup (typ-info-t TYPE('a)) (f1 @ f2) 0 = Some (v, n)
  using addressable-fieldD-field-lookup[OF f'-n] unfolding f'-eq
  by (auto dest: field-lookup-uinfo-Some-rev)
obtain m where u-v: field-lookup u f2 0 = Some (v, m)
  using f-fs f-u
  apply (subst (asm) field-lookup-append-eq)
  apply (auto simp add: f-u field-ti-def split: option.splits
      dest: CTypes.field-lookup-offset2)
  done
have p': hrs-htd h ⊨t p
  using p by (rule ptr-valid-h-t-valid)

have u-mem: u ∈ snd ' set (addressable-fields TYPE('a))
  using F(2,1) f-u(1) by (rule mem-snd-addressable-fields)
have f1u-mem: (f1, u) ∈ set (addressable-fields TYPE('a))
  using F(2,1) f-u(1) by (rule mem-addressable-fields)
have sz: size-of TYPE('b) = size-td v
  using v unfolding export-uinfo-size[symmetric] by (simp add: size-of-def)

have length bs = size-of TYPE('b) ⇒
  access-ti v (merge-addressable-fields y x) bs = access-ti v y bs for y bs
  by (intro access-ti-merge-ti-list[OF list-all-field-ti-addressable-fields
      distinct-disj-fn-addressable-fields f1u-mem u-v])
  (simp add: sz)
then show ?thesis using q unfolding f'-eq q-eq
apply (simp add: read-dedicated-heap-def write-dedicated-heap-def ptr-valid-plift-Some-hval
      hrs-mem-update)
  apply (simp add: heap-upd-def
      ptr-valid-heap-update-subtype-field-access-ti-indep[OF p' p' f-fs v])
  done
next
fix f' n assume f'-n: addressable-field ?B f' ?A and p-eq: p = PTR('a) &(q
→ f')
from f'-n ne obtain F f0 f1 t0 n where *: map-of T ?B = Some F f0 ∈ set
F f' = f0 @ f1

```

and $f0$: *field-lookup* $?B$ $f0$ $0 = \text{Some } (t0, n)$ **and** $f1$: *addressable-field* $t0$ $f1$
 $?A$
by *cases auto*
from *addressable-fieldD-field-ti*[*OF* $f'-n$]
obtain tA **where** $f'-u$: *field-ti* *TYPE*('b) $f' = \text{Some } tA$ **and** tA : *export-uinfo*
 $tA = ?A$
by *auto*
have [*simp*]: *size-td* $tA = \text{size-of } \text{TYPE}'(a)$
unfolding *export-uinfo-size*[*symmetric*] tA **by** (*simp add: size-of-def*)

from $f0$ [*unfolded typ-uinfo-t-def, THEN field-lookup-export-uinfo-Some-rev*]
obtain $t0'$ **where** $f0-t0'$: *field-ti* *TYPE*('b) $f0 = \text{Some } t0'$ **and** $t0$: $t0 =$
export-uinfo $t0'$
by (*auto simp: field-ti-def typ-uinfo-t-def*
split: option.splits)

obtain tA' n **where** $t0'-f1$: *field-lookup* $t0'$ $f1$ $0 = \text{Some } (tA', n)$
using *addressable-fieldD-field-lookup*'[*OF* $f1$, *unfolded* $t0$,
THEN field-lookup-export-uinfo-Some-rev]
by *auto*

with *field-ti-append-field-lookup*[*OF* $f0-t0'$ $t0'-f1$] $f'-u$
have $f0'-f1$: *field-lookup* $t0'$ $f1$ $0 = \text{Some } (tA, n)$
by (*simp add: **)

note $q' = \text{ptr-valid.ptr-valid-h-t-valid}$ [*OF* q]

have $f0t'$ -*mem*: $(f0, t0') \in \text{set } (\text{addressable-fields } \text{TYPE}'(b))$
using $*(1,2)$ $f0-t0'$ **by** (*rule mem-addressable-fields*)

from q $f'-n$ *merge-ti-list-update-ti*[*OF*
list-all-field-ti-addressable-fields f0t'-mem distinct-disj-fn-addressable-fields
merge-ti-list-update-ti[*OF*
list-all-field-ti-addressable-fields f0t'-mem distinct-disj-fn-addressable-fields
 $f0'-f1$]
show *?thesis* **unfolding** *p-eq*
by (*simp add: read-dedicated-heap-def write-dedicated-heap-def hrs-mem-update*
heap-upd-def
ptr-valid-plift-Some-hval h-val-field-update[*OF* $f'-u$ tA q' q']
update-ti-t-def)

qed
qed (*simp add: read-dedicated-heap-def write-dedicated-heap-def plift-None*)

lemma *read-dedicated-heap-write-dedicated-heap*[*simp*]:
 $\text{ptr-valid } (\text{hrs-htd } h) (p::'a::\text{xmem-type } \text{ptr}) \implies$
 $\text{read-dedicated-heap } (\text{write-dedicated-heap } p \ x \ h) =$
 $\text{upd-fun } p \ (\lambda \text{old. merge-addressable-fields } \text{old } x) \ (\text{read-dedicated-heap } h)$
by (*simp add: fun-eq-iff the-plift-hval-fun-upd-eqI read-dedicated-heap-def*[*abs-def*]
upd-fun-def)

write-dedicated-heap-def hrs-mem-update-heap-upd fun-upd-def)

lemma *hrs-mem-update-heap-update*:

fixes $p :: 'a::xmem\text{-type}\ ptr$

shows $hrs\text{-mem}\text{-update}\ (heap\text{-update}\ p\ x)\ h =$

$fold\ (\lambda(f, u). hrs\text{-mem}\text{-update}\ (heap\text{-upd}\text{-list}\ (size\text{-td}\ u)\ \&(p \rightarrow f)\ (access\text{-ti}\ u\ x)))$

$(addressable\text{-fields}\ TYPE('a))$

$(write\text{-dedicated}\text{-heap}\ p\ x\ h)$

proof –

show *?thesis*

apply (*subst heap-update-eq-fold-subfields[OF list-all-field-ti-addressable-fields,*

where y=h-val (hrs-mem h) p])

unfolding *hrs-mem-update-comp'[symmetric] write-dedicated-heap-def hrs-mem-update-heap-upd*

apply (*subst fold-functor[of hrs-mem-update]*)

apply (*simp add: hrs-mem-update-def*)

apply (*simp add: hrs-mem-update-def fun-eq-iff*)

apply (*simp add: split-beta'*)

done

qed

lemma *hrs-mem-update-heap-update'*:

fixes $p :: 'a::xmem\text{-type}\ ptr$

shows $hrs\text{-mem}\text{-update}\ (heap\text{-update}\ p\ x) =$

$fold\ (\lambda(f, u). hrs\text{-mem}\text{-update}\ (heap\text{-upd}\text{-list}\ (size\text{-td}\ u)\ \&(p \rightarrow f)\ (access\text{-ti}\ u\ x)))$

$(addressable\text{-fields}\ TYPE('a)) \circ$

$write\text{-dedicated}\text{-heap}\ p\ x$

apply (*rule ext*)

unfolding *comp-apply*

apply (*rule hrs-mem-update-heap-update*)

done

lemma *hrs-htd-write-dedicated-heap[simp]*:

$hrs\text{-htd}\ (write\text{-dedicated}\text{-heap}\ p\ x\ h) = hrs\text{-htd}\ h$

by (*simp add: write-dedicated-heap-def*)

lemma *partial-pointer-lense-merge-addressable-fields*:

fixes $r :: 'h \Rightarrow 'a::xmem\text{-type}\ ptr \Rightarrow 'a$

assumes *lense r w*

shows

$partial\text{-pointer}\text{-lense}\ (\lambda x\ y. merge\text{-addressable}\text{-fields}\ y\ x)$

$(\lambda h\ p\ x. merge\text{-addressable}\text{-fields}\ x\ (r\ h\ p))\ (\lambda p\ x. w\ (upd\text{-fun}\ p\ (\lambda old. merge\text{-addressable}\text{-fields}\ old\ x)))$

proof –

interpret *lense r w by fact*

show *?thesis*

apply *unfold-locales*

apply (*simp-all add: upd-fun.upd-same upd-same*)

```

subgoal by (simp add: comp-def)
subgoal for p q x y h
  by (simp add: upd-fun-commute disj-ptr-span-ptr-neq disjnt-def Int-commute)
done
qed

lemma pointer-lense-of-partials-cover:
fixes rs :: ('h ⇒ 'a::xmem-type ptr ⇒ 'a ⇒ 'a) list
assumes g-u: lense g u
assumes *:
  length ms = length rs length ms = length ws
  list-all (λ(m, r, w). partial-pointer-lense m r w) (zip ms (zip rs ws))
and **:
  ∧ a b c. distinct-prop (λm1 m2. m1 a (m2 b c) = m2 b (m1 a c)) ms
  ∧ p a q b h. distinct-prop (λw1 w2. p = q ∨ disjnt (ptr-span p) (ptr-span q) →
    w1 p a (w2 q b h) = w2 q b (w1 p a h)) ((λp x. u (upd-fun p (λold.
merge-addressable-fields old x))) # ws)
assumes ms-cover: ∧ a b. fold (λm. m a) ms b = merge-addressable-fields a b
assumes R: ∧ h p. R h p = fold (λr. r h p) rs (g h p)
assumes W: ∧ p f h. W p f h =
  fold (λw. w p (f (R h p))) ws (u (upd-fun p (λold. merge-addressable-fields old
(f (R h p)))) h)
shows pointer-lense R W
apply (rule pointer-lense-of-partials[where
  ms=(λa b. merge-addressable-fields b a) # ms and
  rs = (λh p x. merge-addressable-fields x (g h p)) # rs and
  ws = (λp x. u (upd-fun p (λold. merge-addressable-fields old x)))#ws])
subgoal by (simp add: *(1))
subgoal by (simp add: *(2))
subgoal
  apply (rule list-all-zip-zip-cons)
  apply (rule partial-pointer-lense-merge-addressable-fields[OF g-u])
  apply (rule *)
done
subgoal for a b c
proof –
  have scenes-ms: list-all is-scene ms
    using * by (simp add: list-all-length partial-pointer-lense-def)
  have dis-ms: distinct-prop disjnt-scene ms
    using *(1) by (simp add: disjnt-scene-def[abs-def] distinct-prop-distrib-all)
  have comm-ms: pairwise comm-scene (set ms)
    apply (rule pairwise-set-of-distinct-prop)
    apply (simp add: comm-scene-comm)
    apply (use dis-ms in ⟨auto simp: distinct-prop-iff-nth⟩)
  done
  have ms-y: list-all (λm. disjnt-scene m y ∨ m = y) ms if y: y ∈ set ms for y
    using in-set-conv-decomp[THEN iffD1, OF y] dis-ms
    by (auto simp: distinct-prop-append list-all-iff disjnt-scene-comm)

```

```

show ?thesis
  apply (simp add: **)
  apply (simp flip: ms-cover)
  apply (intro ballI)
  apply (rule fold-disjnt-scene[OF scenes-ms comm-ms ms-y])
  apply auto
  done
qed
subgoal by (rule **)
subgoal by (simp add: ms-cover)
subgoal for h p
proof -
  interpret fold: partial-pointer-lense
   $\lambda a.$  fold ( $\lambda m.$  m a) ms  $\lambda h p.$  fold ( $\lambda r.$  r h p) rs  $\lambda p x.$  fold ( $\lambda w.$  w p x) ws
  apply (rule partial-pointer-lense-fold[OF *])
  subgoal using *(1) by auto
  subgoal using *(2) by auto
  done
show R h p = fold ( $\lambda r.$  r h p) (( $\lambda h p x.$  merge-addressable-fields x (g h p)) #
rs) x
  using fold.r-m by (simp add: R flip: ms-cover)
qed
subgoal by (simp add: W)
done

```

lemma cover-read-dedicated-heap[simp]:
merge-addressable-fields (read-dedicated-heap h p) = merge-addressable-fields ZERO('a::xmem-type)
by (simp add: fun-eq-iff read-dedicated-heap-def)

lemma read-dedicated-heap-fun-upd-cover-zero-eq-upd-fun[simp]:
((read-dedicated-heap h)(p := merge-addressable-fields ZERO('a::xmem-type) x))
q =
upd-fun p ($\lambda old.$ merge-addressable-fields old x) (read-dedicated-heap h) q
by (simp add: upd-fun-def fun-upd-def fun-eq-iff)

theorem stack-allocs-read-dedicated-heap [stack-the-default-plift]:
fixes p:: 'a::xmem-type ptr
assumes alloc: (p, d') \in stack-allocs n S TYPE('a) (hrs-htd h)
assumes q-not-stack-byte: typ-uinfo-t TYPE('b::xmem-type) \neq typ-uinfo-t TYPE(stack-byte)
shows read-dedicated-heap (hrs-mem-update (fold ($\lambda i.$ heap-update (p +_p int i)
c-type-class.zero) [0.. n]) (hrs-htd-update ($\lambda.$ d') h)) =
(read-dedicated-heap h :: 'b ptr \Rightarrow -)
apply (rule ext)
unfolding read-dedicated-heap-def stack-allocs-the-default-plift[OF assms] ..

theorem stack-releases-read-dedicated-heap [stack-the-default-plift]:
fixes p:: 'a::xmem-type ptr
assumes roots: $\bigwedge i. i < n \implies$ root-ptr-valid (hrs-htd h) (p +_p int i)
assumes p-not-stack-byte: typ-uinfo-t TYPE('a) \neq typ-uinfo-t TYPE(stack-byte)

assumes *q-not-stack-byte*: $\text{typ-uinto-t } \text{TYPE}('b) \neq \text{typ-uinto-t } \text{TYPE}(\text{stack-byte})$
shows *read-dedicated-heap* (*hrs-mem-update* (*fold* ($\lambda i. \text{heap-update } (p +_p \text{int } i)$) *c-type-class.zero*) [$0..<n$] *h*) =
(*read-dedicated-heap* (*hrs-htd-update* (*stack-releases* *n p*) *h*) :: *'b::xmem-type ptr*
 \Rightarrow -)
unfolding *read-dedicated-heap-def stack-releases-the-default-plit*[*OF assms, of n, simplified*] ..

lemma *ptr-valid-addressable-field-field-lvalue*[*addressable-field-exec*]:
fixes *p::'a::mem-type ptr*
assumes *ptr-valid h p*
assumes *addressable-field* ($\text{typ-uinto-t } (\text{TYPE}('a))$) *fs* ($\text{typ-uinto-t } (\text{TYPE}('b::\text{mem-type}))$)
shows *ptr-valid h* ($\text{PTR}('b) (\&(p \rightarrow fs))$)
using *assms*
by (*metis field-lvalue-def field-offset-def local.fold-ptr-valid'*
local.open-types-axioms local.ptr-valid-u-trans open-types.ptr-valid-def)

lemma *addressable-field-exec*[*addressable-field-exec*]:
shows
addressable-field t ps v =
(*case ps of*
 $\square \Rightarrow t = v$
 $| - \Rightarrow (\exists fs p. \text{map-of } \mathcal{T} t = \text{Some } fs \wedge$
 $\text{List.find } (\lambda p. \exists u n. \text{field-lookup } t p 0 = \text{Some } (u, n) \wedge p @ \text{drop } (\text{length}$
 $p) ps = ps \wedge$
 $\text{addressable-field } u (\text{drop } (\text{length } p) ps) v) fs = \text{Some } p)$)

proof (*cases ps*)
case *Nil*
then show *?thesis* **by** *simp*
next
case (*Cons f ps'*)
show *?thesis*
proof (*cases map-of T t*)
case *None*
then show *?thesis*
apply (*simp add: Cons*)
by (*metis list.simps(2) local.addressable-field.cases option.simps(3)*)
next
case (*Some fs*)
show *?thesis*
proof (*cases* $\exists p. \text{List.find } (\lambda p. \exists u n. \text{field-lookup } t p 0 = \text{Some } (u, n) \wedge$
 $p @ \text{drop } (\text{length } p) ps = ps \wedge$
 $\text{addressable-field } u (\text{drop } (\text{length } p) ps) v) fs = \text{Some } p)$)
case *True*
then obtain *p* **where**
 $\text{find: List.find } (\lambda p. \exists u n. \text{field-lookup } t p 0 = \text{Some } (u, n) \wedge$
 $p @ \text{drop } (\text{length } p) ps = ps \wedge$
 $\text{addressable-field } u (\text{drop } (\text{length } p) ps) v) fs = \text{Some } p$
by *auto*

```

then obtain  $u\ n$  where
   $p: p \in \text{set } fs$  and
   $ps: p @ \text{drop } (\text{length } p) \ ps = ps$  and
   $u: \text{field-lookup } t\ p\ 0 = \text{Some } (u, n)$  and
   $\text{addr}: \text{addressable-field } u\ (\text{drop } (\text{length } p) \ ps) \ v$ 
  using  $\text{findSomeD}$ 
  by ( $\text{smt } (\text{verit}, \text{best})$ )
from  $\text{addressable-field-step } [OF\ \text{Some } p\ u\ \text{addr}]$ 
have  $\text{addr}': \text{addressable-field } t\ (p @ \text{drop } (\text{length } p) \ ps) \ v .$ 
from  $\text{addressable-fieldD-field-lookup } [OF\ \text{addr}]$  obtain  $m$ 
  where  $v: \text{field-lookup } u\ (\text{drop } (\text{length } p) \ ps) \ 0 = \text{Some } (v, m)$ 
  by  $\text{blast}$ 
from  $\text{addressable-fieldD-field-lookup } [OF\ \text{addr}']$  obtain  $k$  where
   $\text{field-lookup } t\ (p @ \text{drop } (\text{length } p) \ ps) \ 0 = \text{Some } (v, k)$ 
  by  $\text{blast}$ 
have  $ps\text{-eq}: (p @ \text{drop } (\text{length } p) \ (f \# ps')) = ps$ 
  using  $ps$ 
  by ( $\text{simp } \text{add}: \text{Cons } ps$ )
show  $?thesis$ 
  using  $\text{find } \text{addr}'$ 
  by ( $\text{simp } \text{add}: \text{Cons } \text{Some } ps\text{-eq}$ )
next
  case  $\text{False}$ 
  with  $\text{Some}$  show  $?thesis$ 
  apply ( $\text{simp } \text{add}: \text{Cons}$ )
  apply ( $\text{force } \text{simp } \text{add}: \text{find-None-iff } \text{elim}: \text{addressable-field.cases}$ )
  done
qed
qed
qed

local-setup  $\langle$ 
  let
     $\text{fun } \text{unfold-ss } \text{ctxt} = \text{ctxt } \text{addsimps}$ 
     $\text{Named-Theorems.get } \text{ctxt} \ @\{\text{named-theorems } \mathcal{T}\text{-def}\} \ @$ 
     $\ @\{\text{thms}$ 
       $\ \text{list-ex-iff}$ 
       $\ \text{fun-upd-other}$ 
       $\ \text{field-lookup-typ-uinfo-t-Some}$ 
       $\ \text{fold-typ-uinfo-t}$ 
       $\ \text{exists-nat-numeral}\}$ 
  in
     $\text{Cached-Theory-Simproc.declare-init-thy-simpset } \ @\{\text{named-theorems } \mathcal{T}\text{-def}\} \ \text{un-}$ 
     $\ \text{fold-ss}$ 
  end
   $\rangle$ 

simproc-setup  $\mathcal{T}$  ( $\langle \text{map-of } \mathcal{T} \ (\text{typ-uinfo-t } \text{TYPE}('a::\text{mem-type})) \rangle \mid \langle \text{map } \text{fst } \mathcal{T} \rangle$ 
 $\mid \langle \text{set } \mathcal{T} \rangle = \langle$ 

```

```

let
  exception Abort;
  fun cache-simproc augment ctxt =
    let val umm-ctxt = Cached-Theory-Simproc.put-time-warp-simpset' @ {named-theorems
T-def} ctxt
        in Cached-Theory-Simproc.gen-simproc (umm-ctxt, Cached-Theory-Simproc.recert,
Cached-Theory-Simproc.add-cache) augment end
    in
      fn phi => fn ctxt => fn ct =>
        let
          val check =
            Match-Cterm.switch [
              (@{cterm-morph-match map-of T (typ-uinfo-t ?T)} phi) #> (fn {T, ...}
=>
                if UMM-Proofs.is-ctype (Thm.term-of T) then () else raise Abort)
            , fn - => ()];
          val - = check ct
          val - = Utils.verbose-msg 3 ctxt (fn - =>
            T invoked: ^ Syntax.string-of-term ctxt (Thm.term-of ct))
        in
          cache-simproc (K single) ctxt ct
        end handle Match => NONE | Abort => NONE
      end
    end
  >

```

end

lemma *fold-field-lvalue*:

```

x + word-of-nat (field-offset-untyped (typ-uinfo-t TYPE('a::c-type)) f) = &(PTR('a)
x→f)
by (simp add: field-lvalue-def field-offset-def)

```

lemma *field-lvalue-same-root-ptr-conv*:

```

&(p::'a:: c-type ptr→f) = &(q::'a:: c-type ptr→f) ⟷ p = q
by (auto simp add: field-lvalue-same-conv)

```

lemma *ptr-exhaust-eq*: **fixes** $p::'a::c\text{-type } ptr$ **shows** $PTR('a) (ptr\text{-val } p) = p$

by (cases p) simp

lemma *fold-exists-ptr*: $(\exists x. P (PTR('a::c\text{-type}) x)) \longleftrightarrow (\exists q. P q)$

by (metis ptr.exhaust)

21.1.1 Syntax $PTR\text{-VALID}('a)$

context *open-types*

begin

We want to provide syntax that makes the type of *ptr-valid* visible on the term level for interpretations of the *open-types*. This is a bit tricky as Isabelle does not (yet) provide local syntax and local print / parse translations. We

implement it by a combination of theory level syntax and translations and a local declarations for the interpretations.

end

syntax

-ptr-valid :: *type* \Rightarrow *logic* ((1PTR'-VALID)/(1'(-')))

ML \langle

structure *Ptr-Valid-Translation* =

struct

val *show-ptr-valid* = *Attrib.setup-config-bool* @{*binding* *show-ptr-valid*} (*K true*)

val *ptr-validN* = @{*const-name* *open-types.ptr-valid*} |> *Long-Name.base-name*

val *local-ptr-validN* = *Long-Name.qualify* *Long-Name.localN* *ptr-validN*

type *options* = {*ptr-valid:term*, *ptr-valid-name: string*} *option* — NONE means we are within the (abstract) locale, SOME provides the relevant parameters from the global interpretations

fun *ptr-valid-term* *NONE* *ctxt* = *Syntax.read-term* *ctxt* *local-ptr-validN*

| *ptr-valid-term* (*SOME* {*ptr-valid*, ...}) = *ptr-valid*

fun *instantiate-ptr-valid* *opt* *ctxt* *typ* =

let

val **Const**- \langle *open-types.ptr-valid* - for *T* \rangle = *ptr-valid-term* *opt* *ctxt*

in

Const- \langle *open-types.ptr-valid* *typ* for *T* \rangle

end

fun *const-name* *check* *ctxt* *s* =

let

val *Const* (*c*, -) = *Proof-Context.read-const* {*proper* = *true*, *strict* = *false*} *ctxt*

s

val *res* = *check* (*Proof-Context.consts-of* *ctxt*, *c*)

handle *TYPE* (*msg*, -, -) \Rightarrow *error* (*msg* ^ *Position.here* **here**);

in

res

end

fun *const-syntax* (*SOME* {*ptr-valid-name*, ...}) *ctxt* =

const-name (*fn* (-, *c*) \Rightarrow *Lexicon.mark-const* *c*) *ctxt* *ptr-valid-name*

| *const-syntax* *NONE* *ctxt* =

Lexicon.mark-const *local-ptr-validN*

fun *gen-ptr-valid-tr'* *ptr-valid* *ctxt* *T* *ts* = *if* *Config.get* *ctxt* *show-ptr-valid* *then*

let

val *ptr-valid'* as **Const**- \langle *open-types.ptr-valid* *typ* for - \rangle =

Syntax.read-term *ctxt* *ptr-validN*

```

    val [-, Type ⟨ptr typ⟩] = T |> binder-types
    val head = Syntax.const syntax-const ⟨-ptr-valid⟩ $ Syntax-Phases.term-of-typ
  ctxt typ
  in
    Term.betapplys (head, ts)
  end
  handle Bind => raise Match
  else raise Match

```

```

fun add-ptr-valid-print-translation (opt:options) thy =
  let
    val ctxt = Proof-Context.init-global thy
    val const-syntax = const-syntax opt ctxt
    val ptr-valid = instantiate-ptr-valid opt
  in
    thy |> Sign.typed-print-translation [(const-syntax, gen-ptr-valid-tr' ptr-valid)]
  end
end

```

```

parse-translation ⟨
  let
    fun ptr-valid-tr ctxt [t] =
      let
        val Const ⟨open-types.ptr-valid - for T ⟩ =
          Syntax.read-term ctxt Ptr-Valid-Translation.ptr-validN
        val typ = Syntax-Phases.decode-typ t
      in
        Const ⟨open-types.ptr-valid typ⟩ $ T
      end
    handle Bind => error (PTR-VALID: no instance of ptr-valid in context)
  in
    [(syntax-const ⟨-ptr-valid⟩, ptr-valid-tr)]
  end

```

```

setup ⟨Ptr-Valid-Translation.add-ptr-valid-print-translation NONE⟩

```

This theory level setup actually provides the local syntax when working within a locale like *open-types*. The *print-translation* is triggered on constant (abbreviation) *local.ptr-valid*

```

context open-types

```

```

begin

```

```

term PTR-VALID(32 word) — Works in both directions by theory level setup above.

```

```

declaration ⟨fn phi => fn context =>

```

```

  let
    val ptr-valid = Morphism.term phi term ⟨ptr-valid⟩
    val ptr-valid-name = Morphism.binding phi (Binding.make (Ptr-Valid-Translation.ptr-validN,
here))
    |> Binding.name-of

```



```

    val opt = SOME {ptr-valid = ptr-valid, ptr-valid-name = ptr-valid-name}
  in
    context |> Context.mapping (Ptr-Valid-Translation.add-ptr-valid-print-translation
opt) I
  end

```

This declaration provides the setup for global interpretations. The print translation is triggered on constant (abbreviation) $\langle instance-name \rangle.ptr\text{-}valid$ introduced by the interpretation. Keep in mind, that when printing a term on the internal $PTR\text{-}VALID('a)$ first the notations / abbreviations are applied. These introduce the syntactic constant $PTR\text{-}VALID('a)$ on which the translation then triggers.

end

21.1.2 Syntax $IS\text{-}VALID('a)$

context *open-types*
begin

Building on top of $PTR\text{-}VALID$ we provide syntax that also takes the *heap-typing* of lifted globals into account: $IS\text{-}VALID(32\text{ word})\ s\ p$ is translated to $PTR\text{-}VALID('a)\ (heap\text{-}typing\ s)\ p$ Note that *heap-typing* is a field of the lifted globals record that is defined later.

end

syntax

```
-is-valid :: type => logic => logic ((1IS'-VALID/(1'(-))) - [0, 1000] 1000)
```

parse-translation <

let

```
fun is-valid-tr ctxt [t, s] =
```

```
let
```

```
val Const-<open-types.ptr-valid - for T > =
```

```
  Syntax.read-term ctxt Ptr-Valid-Translation.ptr-validN
```

```
val typ = Syntax-Phases.decode-typ t
```

```
in ( Const<open-types.ptr-valid typ> $ T ) $ ((Const (AC-Names.heap-typingN,
dummyT)) $ s) end
```

```
  handle Bind => error (IS-VALID: no instance of ptr-valid in context )
```

```
in
```

```
  [(syntax-const <-is-valid>, is-valid-tr)]
```

```
end
```

```
>
```

print-translation <

let

```
val show-is-valid = Attrib.setup-config-bool @{binding show-is-valid} (K true)
```

```
fun is-valid-tr' ctxt (typ::Const (marked-heap-typing-name, heap-typingT))$s::ts
```

```
=
```

```

    if Config.get ctxt show-is-valid then
      let
        val heap-typing-name = try ⟨Lexicon.unmark-const marked-heap-typing-name
          catch - => raise Match⟩
        val - = if Long-Name.base-name heap-typing-name = AC-Names.heap-typingN
      then () else raise Match
      in
        Term.betapplys (Syntax.const syntax-const ⟨-is-valid⟩ $ typ $ s, ts)
      end
    else raise Match
  in
    [(syntax-const ⟨-ptr-valid⟩, is-valid-tr^)]
  end
end

```

end

theory *AbstractArrays*

imports

TypHeapLib

WordSetup

begin

primrec

*array-addr*s :: ('a::mem-type) ptr ⇒ nat ⇒ 'a ptr list

where

*array-addr*s - 0 = []

| *array-addr*s p (Suc n) = p # (*array-addr*s (p +_p 1) n)

declare *array-addr*s.simps(2) [*simp del*]

lemma *hd-in-array-addr*s [*simp*]:

($x \in \text{set } (\text{array-addr}s \ x \ n)$) = ($n > 0$)

by (*cases n*, *auto simp: array-addr*s.simps(2))

lemma *array-addr*s-1 [*simp*]:

*array-addr*s p (Suc 0) = [p]

*array-addr*s p 1 = [p]

by (*auto simp: array-addr*s.simps(2))

lemma *array-addr*s-ptr-aligned:

$\llbracket x \in \text{set } (\text{array-addr}s \ p \ n); \text{ptr-aligned } p \rrbracket \implies \text{ptr-aligned } x$

apply (*induct n arbitrary: x p*)

subgoal by *clarsimp*

subgoal for $n \ x \ p$

```

apply (clarsimp simp: array-addr.simps(2))
apply (erule disjE)
apply clarsimp
apply atomize
apply (drule-tac x=x in spec)
apply (drule-tac x=p +p 1 in spec)
apply (clarsimp simp: ptr-aligned-plus)
done
done

```

lemma *set-array-addr-unfold-last*:

```

shows set (array-addr a (Suc n)) = set (array-addr a n) ∪ {(a :: ('a::mem-type) ptr) +p int n}

```

```

  (is ?LHS a n = ?RHS a n)

```

```

proof (induct n arbitrary: a)

```

```

  fix a

```

```

  show ?LHS a 0 = ?RHS a 0

```

```

    by clarsimp

```

```

next

```

```

  fix a n

```

```

  assume induct: ∧a. ?LHS a n = ?RHS a n

```

```

  show ?LHS a (Suc n) = ?RHS a (Suc n)

```

```

  apply (subst array-addr.simps(2))

```

```

  apply (subst set-simps)

```

```

  apply (subst induct [where a=a +p 1])

```

```

  apply (subst array-addr.simps(2))

```

```

  apply (subst set-simps)

```

```

  apply (clarsimp simp: CTypesDefs.ptr-add-def field-simps insert-commute)

```

```

  done

```

```

qed

```

lemma *set-array-addr*:

```

set (array-addr (p :: ('a::mem-type) ptr) n)
  = {x. ∃k. x = p +p int k ∧ k < n }

```

```

apply (induct n arbitrary: p)

```

```

subgoal by (clarsimp simp: not-less)

```

```

subgoal for n p

```

```

  apply (subst set-array-addr-unfold-last)

```

```

  apply atomize

```

```

  apply (drule-tac x=p in spec)

```

```

  apply (erule ssubst)

```

```

  apply (rule set-eqI)

```

```

  apply (rule iffI)

```

```

  apply clarsimp

```

```

  apply (erule disjE)

```

```

  apply clarsimp

```

```

  apply force

```

```

    apply force
  apply clarsimp
  apply (rename-tac k)
  apply (drule-tac x=k in spec)
  apply (clarsimp simp: not-less)
  apply (subgoal-tac k = n)
  apply clarsimp
  apply clarsimp
done
done

```

end

```

theory HeapLift
  imports
    In-Out-Parameters
    Split-Heap
    AbstractArrays
begin

```

21.2 Refinement Lemmas

lemma *ucast-ucast-id*:

```

  LENGTH('a) ≤ LENGTH('b) ⇒ ucast (ucast (x::'a::len word)::'b::len word) =
  x
  by (auto intro: ucast-up-ucast-id simp: is-up-def source-size-def target-size-def
  word-size)

```

lemma *lense-ucast-signed*:

```

  lense (unsigned :: 'a::len word ⇒ 'a signed word) (λv x. unsigned (v (unsigned
  x)))
  by (rule lenseI-equiv) (simp-all add: ucast-ucast-id)

```

lemma *pointer-lense-ucast-signed*:

```

  fixes r :: 'h ⇒ 'a::len8 word ptr ⇒ 'a word
  assumes pointer-lense r w
  shows pointer-lense
    (λh p. UCAST('a → 'a signed) (r h (PTR-COERCE('a signed word → 'a word)
  p)))
    (λp m. w (PTR-COERCE('a signed word → 'a word) p)
    (λw. UCAST('a signed → 'a) (m (UCAST('a → 'a signed) w))))

```

proof –

```

  interpret pointer-lense r w by fact
  note scast-ucast-norm[simp del]
  note ucast-ucast-id[simp]
  show ?thesis
  apply unfold-locales

```

```

apply (simp add: read-write-same)
apply (simp add: write-same)
apply (simp add: comp-def)
apply (simp add: write-other-commute typ-uinfo-t-signed-word-word-conv
        flip: size-of-tag typ-uinfo-size)
done
qed

lemma (in xmem-type) length-to-bytes:
  length (to-bytes (v:'a) bs) = size-of TYPE('a)
by (simp add: to-bytes-def lense.access-result-size)

lemma (in xmem-type) heap-update-padding-eq:
  length bs = size-of TYPE('a)  $\implies$ 
  heap-update-padding p v bs h = heap-update p v (heap-update-list (ptr-val p) bs
  h)
using u.max-size
by (simp add: heap-update-padding-def heap-update-def size-of-def
      heap-list-heap-update-list-id heap-update-list-overwrite)

lemma (in xmem-type) heap-update-padding-eq':
  length bs = size-of TYPE('a)  $\implies$ 
  heap-update-padding p v bs = heap-update p v  $\circ$  heap-update-list (ptr-val p) bs
by (simp add: fun-eq-iff heap-update-padding-eq)

lemma split-disj-asm:  $P (x \vee y) = (\neg (x \wedge \neg P x \vee \neg x \wedge \neg P y))$ 
by (smt (verit, best))

lemma comp-commute-of-fold:
assumes x:  $x = \text{fold } f \text{ } xs$ 
assumes xs: list-all  $(\lambda x. f x \circ a = a \circ f x) \text{ } xs$ 
shows  $x \circ a = a \circ x$ 
unfolding x using xs by (induction xs) (simp-all add: fun-eq-iff)

definition padding-closed-under-all-fields where
  padding-closed-under-all-fields t  $\longleftrightarrow$ 
   $(\forall s f n bs bs'. \text{field-lookup } t f 0 = \text{Some } (s, n) \longrightarrow$ 
   $\text{eq-upto-padding } t \text{ } bs \text{ } bs' \longrightarrow \text{eq-upto-padding } s \text{ } (\text{take } (\text{size-td } s) \text{ } (\text{drop } n \text{ } bs))$ 
   $(\text{take } (\text{size-td } s) \text{ } (\text{drop } n \text{ } bs')))$ 

lemma padding-closed-under-all-fields-typ-uinfo-t:
  padding-closed-under-all-fields (typ-uinfo-t TYPE('a::xmem-type))
unfolding padding-closed-under-all-fields-def
proof safe
fix s f n bs bs' assume s-n:  $\text{field-lookup } (\text{typ-uinfo-t } \text{TYPE('a)}) f 0 = \text{Some } (s,$ 
  n)
and bs-bs':  $\text{eq-upto-padding } (\text{typ-uinfo-t } \text{TYPE('a)}) \text{ } bs \text{ } bs'$ 
then have len:  $\text{length } bs = \text{size-of } \text{TYPE('a)} \text{ } \text{length } bs' = \text{size-of } \text{TYPE('a)}$ 
by (auto simp: eq-upto-padding-def size-of-def)

```

```

from  $s$ - $n$ [THEN field-lookup-uinfo-Some-rev] obtain  $k$  where
   $k$ : field-lookup (typ-info-t  $TYPE('a)$ )  $f$   $0 = Some$  ( $k$ ,  $n$ ) and  $k$ - $s$ : export-uinfo
 $k = s$ 
  by auto
  have [simp]: size-td  $k = size-td$   $s$  by (simp flip: k-s)
  from xmem-type-field-lookup-eq-upto-padding-focus[OF k len bs-bs']
  show eq-upto-padding  $s$  (take (size-td  $s$ ) (drop n bs)) (take (size-td  $s$ ) (drop n
 $bs$ ^))
  unfolding  $k$ - $s$  by simp
qed

```

```

lemma (in open-types) plift-heap-update-list-eq-upto-padding:
  assumes  $t$ : mem-type-u  $t$  and  $t'$ : padding-closed-under-all-fields  $t$ 
  assumes  $a$ : ptr-valid-u  $t$  (hrs-htd  $h$ )  $a$ 
  assumes  $bs$ - $bs'$ : eq-upto-padding  $t$   $bs$   $bs'$ 
  shows plift (hrs-mem-update (heap-update-list  $a$   $bs$ )  $h$ ) =
    (plift (hrs-mem-update (heap-update-list  $a$   $bs'$ )  $h$ ))::'a::xmem-type ptr  $\Rightarrow$  'a option)
  apply (rule plift-eq-plift, simp-all add: h-val-def hrs-mem-update)
proof -
  from  $bs$ - $bs'$  have [simp]: length  $bs = size-td$   $t$  length  $bs' = size-td$   $t$ 
  by (simp-all add: eq-upto-padding-def)
  have [arith]: size-td  $t < addr-card$ 
  using mem-type-u.max-size[OF t] by simp
  have  $a$ -bnd: size-of  $TYPE('a) \leq addr-card$ 
  using max-size[where 'a='a] by arith
  let  $?A = typ-uinfo-t$   $TYPE('a)$ 

  fix  $p$  :: 'a ptr assume  $p$ : ptr-valid (hrs-htd  $h$ )  $p$ 
  from ptr-valid-u-cases-weak[OF t a this[unfolded ptr-valid-def]]
  show from-bytes (heap-list (heap-update-list  $a$   $bs$  (hrs-mem  $h$ )) (size-of  $TYPE('a)$ )
(ptr-val  $p$ )) =
  (from-bytes (heap-list (heap-update-list  $a$   $bs'$  (hrs-mem  $h$ )) (size-of  $TYPE('a)$ )
(ptr-val  $p$ ))::'a)
  proof (elim disjE exE conjE)
  assume disjnt  $\{a..+size-td\ t\} \{ptr-val\ p..+size-td\ (typ-uinfo-t\ TYPE('a))\}$ 
  with  $bs$ - $bs'$  show ?thesis
  unfolding heap-upd-list-def
  by (subst ( $1\ 2$ ) heap-list-update-disjoint-same; simp add: size-of-def disjnt-def)
next
  fix  $path$  assume  $path$ : addressable-field  $t$   $path$   $?A$  and
   $p$ -eq: ptr-val  $p = a + word-of-nat$  (field-offset-untyped  $t$   $path$ )
  let  $?n = field-offset-untyped$   $t$   $path$ 
  have  $sz$ : size-of  $TYPE('a) + ?n \leq size-td$   $t$ 
  using field-lookup-offset-size'[OF addressable-fieldD-field-lookup'[OF path]]
  by (simp add: size-of-def)
  let  $?s = typ-uinfo-t$   $TYPE('a)$ 
  from addressable-fieldD-field-lookup'[OF path]  $t'$   $bs$ - $bs'$  have  $*$ :

```

```

    eq-upto-padding ?s (take (size-td ?s) (drop ?n bs)) (take (size-td ?s) (drop ?n
bs^))
  unfolding padding-closed-under-all-fields-def
  by (auto simp flip: typ-uinfo-size)

show ?thesis unfolding p-eq
  using eq-upto-padding-from-bytes-eq[OF *] sz
  apply (subst (1 2) heap-list-update-list)
  apply (simp-all add: size-of-def)
  done
next
fix path assume path: addressable-field ?A path t
  and p-eq: a = ptr-val p + word-of-nat (field-offset-untyped ?A path)
let ?n = field-offset-untyped ?A path
have sz: size-td t + ?n ≤ size-of TYPE('a)
  using field-lookup-offset-size'[OF addressable-fieldD-field-lookup'[OF path]]
  by (simp add: size-of-def)
from field-lookup-uinfo-Some-rev[OF addressable-fieldD-field-lookup'[OF path]]
obtain k
  where k: field-lookup (typ-info-t TYPE('a)) path 0 = Some (k, ?n)
  and eq-t: export-uinfo k = t by blast
then have [simp]: size-td k = size-td t
  by (simp flip: eq-t)

have *: eq-upto-padding (typ-uinfo-t TYPE('a))
  (super-update-bs bs (heap-list (hrs-mem h) (size-of TYPE('a)) (ptr-val p)) ?n)
  (super-update-bs bs' (heap-list (hrs-mem h) (size-of TYPE('a)) (ptr-val p))
?n)
  by (subst xmem-type-field-lookup-eq-upto-padding-super-update-bs[OF k(1)])
  (simp-all add: eq-t bs-bs')

note 1 = c-guard-no-wrap'[OF ptr-valid-c-guard, OF p]
show ?thesis unfolding p-eq using sz
  apply (subst (1 2) heap-update-list-super-update-bs-heap-list[OF 1])
  apply (simp-all add: heap-list-heap-update-list-id[OF a-bnd])
  apply (intro eq-upto-padding-from-bytes-eq[OF *])
  done
qed
qed

lemma (in open-types) read-dedicated-heap-heap-update-list-eq-upto-padding[simp]:
  assumes t: mem-type-u t and t': padding-closed-under-all-fields t
  assumes a: ptr-valid-u t (hrs-htd h) a
  assumes bs-bs': eq-upto-padding t bs bs'
  shows read-dedicated-heap (hrs-mem-update (heap-update-list a bs) h) =
    (read-dedicated-heap (hrs-mem-update (heap-update-list a bs') h)::'a::xmem-type
ptr ⇒ 'a) ⟷ True
  by (simp add: plift-heap-update-list-eq-upto-padding[OF assms] read-dedicated-heap-def
fun-eq-iff)

```

definition $L2Tcorres\ st\ A\ C = corresXF\ st\ (\lambda r\ -. r)\ (\lambda r\ -. r)\ (\lambda -. True)\ A\ C$

lemma $L2Tcorres-id$:

$L2Tcorres\ id\ C\ C$

by ($metis\ L2Tcorres-def\ corresXF-id$)

lemma $L2Tcorres-fail$:

$L2Tcorres\ st\ L2-fail\ X$

apply ($clarsimp\ simp: L2Tcorres-def\ L2-defs$)

apply ($rule\ corresXF-fail$)

done

lemma $admissible-nondet-ord-L2Tcorres$ [$corres-admissible$]:

$ccpo.admissible\ Inf\ (\geq)\ (\lambda A. L2Tcorres\ st\ A\ C)$

unfolding $L2Tcorres-def$

apply ($rule\ admissible-nondet-ord-corresXF$)

done

lemma $L2Tcorres-top$ [$corres-top$]: $L2Tcorres\ st\ \top\ C$

by ($auto\ simp\ add: L2Tcorres-def\ corresXF-def$)

definition $abs-guard\ st\ A\ C \equiv \forall s. A\ (st\ s) \longrightarrow C\ s$

definition $abs-expr\ st\ P\ A\ C \equiv \forall s. P\ (st\ s) \longrightarrow C\ s = A\ (st\ s)$

definition $abs-modifies\ st\ P\ A\ C \equiv \forall s. P\ (st\ s) \longrightarrow st\ (C\ s) = A\ (st\ s)$

definition $struct-rewrite-guard\ A\ C \equiv \forall s. A\ s \longrightarrow C\ s$

definition $struct-rewrite-expr\ P\ A\ C \equiv \forall s. P\ s \longrightarrow C\ s = A\ s$

definition $struct-rewrite-modifies\ P\ A\ C \equiv \forall s. P\ s \longrightarrow C\ s = A\ s$

named-theorems $heap-abs$

named-theorems $heap-abs-fo$

named-theorems $derived-heap-defs$ **and**

$valid-array-defs$ **and**

$heap-upd-cong$ **and**

$valid-same-typ-descs$

lemma $deepen-heap-upd-cong$: $f = f' \implies upd\ f\ s = upd\ f'\ s$

by $simp$

lemma $deepen-heap-map-cong$: $f = f' \implies upd\ f\ p\ s = upd\ f'\ p\ s$

by $simp$

lemma *abs-expr-fun-app2* [*heap-abs-fo*]:

[[*abs-expr st P f f'*;
abs-expr st Q g g']] \implies
abs-expr st ($\lambda s. P s \wedge Q s$) ($\lambda s a. f s a (g s a)$) ($\lambda s a. f' s a \$ g' s a$)
by (*simp add: abs-expr-def*)

lemma *abs-expr-fun-app* [*heap-abs-fo*]:

[[*abs-expr st Y x x'*; *abs-expr st X f f'*]] \implies
abs-expr st ($\lambda s. X s \wedge Y s$) ($\lambda s. f s (x s)$) ($\lambda s. f' s \$ x' s$)
apply (*clarsimp simp: abs-expr-def*)
done

lemma *abs-expr-Pair* [*heap-abs*]:

abs-expr st X f1 f1' \implies *abs-expr st Y f2 f2'* \implies
abs-expr st ($\lambda s. X s \wedge Y s$) ($\lambda s. (f1 s, f2 s)$) ($\lambda s. (f1' s, f2' s)$)
apply (*clarsimp simp: abs-expr-def*)
done

lemma *abs-expr-constant* [*heap-abs*]:

abs-expr st ($\lambda-. True$) ($\lambda s. a$) ($\lambda s. a$)
apply (*clarsimp simp: abs-expr-def*)
done

lemma *abs-guard-expr* [*heap-abs*]:

abs-expr st P a' a \implies *abs-guard st* ($\lambda s. P s \wedge a' s$) *a*
by (*simp add: abs-expr-def abs-guard-def*)

lemma *abs-guard-constant* [*heap-abs*]:

abs-guard st ($\lambda-. P$) ($\lambda-. P$)
by (*clarsimp simp: abs-guard-def*)

lemma *abs-guard-conj* [*heap-abs*]:

[[*abs-guard st G G'*; *abs-guard st H H'*]] \implies
abs-guard st ($\lambda s. G s \wedge H s$) ($\lambda s. G' s \wedge H' s$)
by (*clarsimp simp: abs-guard-def*)

lemma *L2Tcorres-modify* [*heap-abs*]:

[[*struct-rewrite-modifies P b c*; *abs-guard st P' P*;
abs-modifies st Q a b]] \implies
L2Tcorres st (*L2-seq* (*L2-guard* ($\lambda s. P' s \wedge Q s$)) ($\lambda-. (L2-modify a)$))
(*L2-modify c*)
apply (*auto intro!: refines-bind-guard-right refines-modify*
simp: corresXF-refines-conv
L2Tcorres-def L2-defs abs-modifies-def abs-guard-def struct-rewrite-modifies-def
struct-rewrite-guard-def)

done

lemma *L2Tcorres-gets* [*heap-abs*]:

[[*struct-rewrite-expr* *P* *b* *c*; *abs-guard* *st* *P'* *P*;
abs-expr *st* *Q* *a* *b*]] \implies

L2Tcorres *st* (*L2-seq* (*L2-guard* ($\lambda s. P' s \wedge Q s$)) ($\lambda-. L2\text{-gets } a \ n$)) (*L2-gets* *c* *n*)

apply (*auto intro!*; *refines-bind-guard-right* *refines-gets*

simp: *corresXF-refines-conv* *L2Tcorres-def* *L2-defs* *abs-expr-def* *abs-guard-def* *struct-rewrite-expr-def* *struct-rewrite-guard-def*)

done

lemma *L2Tcorres-gets-const* [*heap-abs*]:

L2Tcorres *st* (*L2-gets* ($\lambda-. a$) *n*) (*L2-gets* ($\lambda-. a$) *n*)

apply (*simp add*: *corresXF-refines-conv* *refines-gets* *L2Tcorres-def* *L2-defs*)

done

lemma *L2Tcorres-guard* [*heap-abs*]:

[[*struct-rewrite-guard* *b* *c*; *abs-guard* *st* *a* *b*]] \implies

L2Tcorres *st* (*L2-guard* *a*) (*L2-guard* *c*)

apply (*simp add*: *corresXF-def* *L2Tcorres-def* *L2-defs* *abs-guard-def* *struct-rewrite-guard-def*)

done

lemma *L2Tcorres-while* [*heap-abs*]:

assumes *body-corres* [*simplified THIN-def,rule-format*]:

PROP THIN ($\bigwedge x. L2\text{Tcorres } st (B' x) (B x)$)

and *cond-rewrite* [*simplified THIN-def,rule-format*]:

PROP THIN ($\bigwedge r. \text{struct-rewrite-expr } (G r) (C' r) (C r)$)

and *guard-abs* [*simplified THIN-def,rule-format*]:

PROP THIN ($\bigwedge r. \text{abs-guard } st (G' r) (G r)$)

and *guard-impl-cond* [*simplified THIN-def,rule-format*]:

PROP THIN ($\bigwedge r. \text{abs-expr } st (H r) (C'' r) (C' r)$)

shows *L2Tcorres* *st* (*L2-guarded-while* ($\lambda i s. G' i s \wedge H i s$) *C''* *B' i n*) (*L2-while* *C* *B i n*)

proof –

have *cond-match*: $\bigwedge r s. G' r (st s) \wedge H r (st s) \implies C'' r (st s) = C r s$

using *cond-rewrite* *guard-abs* *guard-impl-cond*

by (*clarsimp simp*: *abs-expr-def* *abs-guard-def* *struct-rewrite-expr-def*)

have *corresXF* *st* ($\lambda r -. r$) ($\lambda r -. r$) ($\lambda-. \text{True}$)

(*do* { $- \leftarrow \text{guard } (\lambda s. G' i s \wedge H i s)$;

whileLoop *C''*

($\lambda i. \text{do } \{ r \leftarrow B' i$;

$- \leftarrow \text{guard } (\lambda s. G' r s \wedge H r s)$;

return *r*

$\}$) *i*

$\}$)

(*whileLoop* *C* *B i*)

```

apply (rule corresXF-guard-imp)
apply (rule corresXF-guarded-while [where  $P = \lambda - . \text{True}$  and  $P' = \lambda - . \text{True}$ ])
  apply (clarsimp cong: corresXF-cong)
  apply (rule corresXF-guard-imp)
  apply (rule body-corres [unfolded L2Tcorres-def])
  apply simp
  apply (clarsimp simp: cond-match)
apply clarsimp
apply (simp add: runs-to-partial-def-old split: xval-splits)
apply simp
apply simp
apply simp
done

```

thus *?thesis*

by (*clarsimp simp: L2Tcorres-def L2-defs gets-return top-fun-def*)

qed

named-theorems *abs-spec*

definition *abs-spec* $st P (A :: ('a \times 'a) \text{ set}) (C :: ('c \times 'c) \text{ set})$
 $\equiv (\forall s t. P (st s) \longrightarrow ((s, t) \in C \longrightarrow ((st s, st t) \in A)))$
 $\wedge (\forall s. P (st s) \longrightarrow (\exists x. (st s, x) \in A) \longrightarrow (\exists x. (s, x) \in C))$

lemma *L2Tcorres-spec* [*heap-abs*]:

\llbracket *abs-spec* $st P A C$ \rrbracket

$\implies L2Tcorres\ st\ (L2\text{-seq}\ (L2\text{-guard}\ P)\ (\lambda\cdot.\ (L2\text{-spec}\ A)))\ (L2\text{-spec}\ C)$

unfolding *corresXF-refines-conv L2Tcorres-def L2-defs*

apply (*clarsimp simp add: abs-spec-def*)

apply (*intro refines-bind-guard-right refines-bind-bind-exn-wp refines-state-select*)

apply (*force intro!: refines-select simp add: abs-spec-def split: xval-splits*)

apply *blast*

done

definition *abs-assume* $st P (A :: 'a \Rightarrow ('b \times 'a) \text{ set}) (C :: 'c \Rightarrow ('b \times 'c) \text{ set})$

$\equiv (\forall s t r. P (st s) \longrightarrow (((r, t) \in C s) \longrightarrow ((r, st t) \in A (st s))))$

lemma *refines-assume-result-and-state'*:

refines (*assume-result-and-state* P) (*assume-result-and-state* Q) $s\ t\ R$

if *sim-set* $(\lambda(v, s) (w, t). R (Result\ v, s) (Result\ w, t)) (P\ s) (Q\ t)$

using *that*

by (*force simp: refines-def-old sim-set-def rel-set-def case-prod-unfold*)

lemma *L2Tcorres-assume* [*heap-abs*]:

```

[[ abs-assume st P A C ]]
  => L2Tcorres st (L2-seq (L2-guard P) (λ-. (L2-assume A))) (L2-assume C)
unfolding corresXF-refines-conv L2Tcorres-def L2-defs
apply (clarsimp simp add: abs-assume-def) thm refines-mono [OF - refines-assume-result-and-state]
apply (intro refines-bind-guard-right refines-bind-bind-exn-wp refines-assume-result-and-state'
)
apply (auto simp add: sim-set-def)
done

```

```

lemma abs-spec-constant [heap-abs]:
  abs-spec st (λ-. True) {(a, b). C} {(a, b). C}
apply (clarsimp simp: abs-spec-def)
done

```

```

lemma L2Tcorres-condition [heap-abs]:
  [[PROP THIN (Trueprop (L2Tcorres st L L'))];
   PROP THIN (Trueprop (L2Tcorres st R R'))];
  PROP THIN (Trueprop (struct-rewrite-expr P C' C));
  PROP THIN (Trueprop (abs-guard st P' P));
  PROP THIN (Trueprop (abs-expr st Q C'' C'))]] =>
  L2Tcorres st (L2-seq (L2-guard (λs. P' s ∧ Q s)) (λ-. L2-condition C'' L R))
(L2-condition C L' R')
unfolding THIN-def L2-defs L2Tcorres-def corresXF-refines-conv
apply clarsimp
apply (intro refines-bind-guard-right refines-condition)
apply (auto simp add: abs-expr-def abs-guard-def struct-rewrite-expr-def struct-rewrite-guard-def)
done

```

```

lemma L2Tcorres-seq [heap-abs]:
  [[PROP THIN (Trueprop (L2Tcorres st L' L)); PROP THIN (λr. L2Tcorres st
(R' r) (R r)) ]]
  => L2Tcorres st (L2-seq L' R') (L2-seq L R)
unfolding THIN-def L2-defs L2Tcorres-def corresXF-refines-conv
apply clarsimp
apply (intro refines-bind-bind-exn-wp)
subgoal for t
  apply (erule-tac x=t in alle)
  apply (rule refines-weaken)
  apply assumption
  apply (auto split: xval-splits)
done
done

```

```

lemma L2Tcorres-guarded-simple [heap-abs]:
  assumes b-c: struct-rewrite-guard b c
  assumes a-b: abs-guard st a b
  assumes f-g: λs s'. c s => s' = st s => L2Tcorres st f g
  shows L2Tcorres st (L2-guarded a f) (L2-guarded c g)

```

unfolding *L2-guarded-def L2-defs L2Tcorres-def corresXF-refines-conv*
using *b-c a-b f-g*
by (*fastforce simp add: refines-def-old L2Tcorres-def corresXF-refines-conv reaches-bind succeeds-bind*
struct-rewrite-guard-def abs-guard-def abs-expr-def split: xval-splits)

lemma *L2Tcorres-catch [heap-abs]*:

$$\llbracket PROP\ THIN\ (Trueprop\ (L2Tcorres\ st\ L\ L'));$$

$$PROP\ THIN\ (\bigwedge r.\ L2Tcorres\ st\ (R\ r)\ (R'\ r))$$

$$\rrbracket \implies L2Tcorres\ st\ (L2-catch\ L\ R)\ (L2-catch\ L'\ R')$$
unfolding *THIN-def*
apply (*clarsimp simp: L2Tcorres-def L2-defs*)
apply (*rule corresXF-guard-imp*)
apply (*erule corresXF-except [where P'=λx y s. x = y and Q=λ-. True]*)
apply (*simp add: corresXF-refines-conv*)
apply (*simp add: runs-to-partial-def-old split: xval-splits*)
apply *simp*
apply *simp*
done

lemma *corresXF-return-same*:

$$corresXF\ st\ (\lambda r\ -. r)\ (\lambda r\ -. r)\ (\lambda -. True)\ (return\ e)\ (return\ e)$$
by (*clarsimp simp add: corresXF-def*)

lemma *corresXF-yield-same*:

$$corresXF\ st\ (\lambda r\ -. r)\ (\lambda r\ -. r)\ (\lambda -. True)\ (yield\ e)\ (yield\ e)$$
by (*auto simp add: corresXF-refines-conv intro!: refines-yield split: xval-splits*)

lemma *L2-try-catch: L2-try L = L2-catch L (λe. yield (to-xval e))*
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add: runs-to-iff*)
apply (*auto simp add: runs-to-def-old unnest-exn-def to-xval-def split: xval-splits sum.splits*)
done

lemma *L2Tcorres-try [heap-abs]*:

$$\llbracket L2Tcorres\ st\ L\ L' \rrbracket \implies L2Tcorres\ st\ (L2-try\ L)\ (L2-try\ L')$$
apply (*simp add: L2-try-catch*)
apply (*erule L2Tcorres-catch [simplified THIN-def]*)
apply (*unfold L2Tcorres-def top-fun-def top-bool-def*)
apply (*rule corresXF-yield-same*)
done

lemma *L2Tcorres-unknown [heap-abs]*:

$$L2Tcorres\ st\ (L2-unknown\ ns)\ (L2-unknown\ ns)$$
apply (*clarsimp simp: L2-unknown-def*)
apply (*clarsimp simp: L2Tcorres-def*)
apply (*auto intro!: corresXF-select-select*)

done

lemma *L2Tcorres-throw* [*heap-abs*]:
 L2Tcorres st (L2-throw x n) (L2-throw x n)
 apply (*clarsimp simp: L2Tcorres-def L2-defs*)
 apply (*rule corresXF-throw*)
 apply *simp*
done

lemma *L2Tcorres-split* [*heap-abs*]:
 $\llbracket \bigwedge x y. L2Tcorres\ st\ (P\ x\ y)\ (P'\ x\ y) \rrbracket \Longrightarrow$
 $L2Tcorres\ st\ (case\ a\ of\ (x,\ y)\ \Rightarrow\ P\ x\ y)\ (case\ a\ of\ (x,\ y)\ \Rightarrow\ P'\ x\ y)$
 apply (*clarsimp simp: split-def*)
done

lemma *L2Tcorres-seq-unused-result* [*heap-abs*]:
 $\llbracket PROP\ THIN\ (Trueprop\ (L2Tcorres\ st\ L\ L'));\ PROP\ THIN\ (Trueprop\ (L2Tcorres\ st\ R\ R')) \rrbracket$
 $\Longrightarrow\ L2Tcorres\ st\ (L2-seq\ L\ (\lambda-. R))\ (L2-seq\ L'\ (\lambda-. R'))$
 apply (*rule L2Tcorres-seq, auto*)
done

lemma *abs-expr-split* [*heap-abs*]:
 $\llbracket \bigwedge a\ b. abs-expr\ st\ (P\ a\ b)\ (A\ a\ b)\ (C\ a\ b) \rrbracket$
 $\Longrightarrow\ abs-expr\ st\ (case\ r\ of\ (a,\ b)\ \Rightarrow\ P\ a\ b)$
 $\quad (case\ r\ of\ (a,\ b)\ \Rightarrow\ A\ a\ b)\ (case\ r\ of\ (a,\ b)\ \Rightarrow\ C\ a\ b)$
 apply (*auto simp: split-def*)
done

lemma *abs-guard-split* [*heap-abs*]:
 $\llbracket \bigwedge a\ b. abs-guard\ st\ (A\ a\ b)\ (C\ a\ b) \rrbracket$
 $\Longrightarrow\ abs-guard\ st\ (case\ r\ of\ (a,\ b)\ \Rightarrow\ A\ a\ b)\ (case\ r\ of\ (a,\ b)\ \Rightarrow\ C\ a\ b)$
 apply (*auto simp: split-def*)
done

lemma *L2Tcorres-abstract-fail* [*heap-abs*]:
 L2Tcorres st L2-fail L2-fail
 apply (*clarsimp simp: L2Tcorres-def L2-defs*)
 apply (*rule corresXF-fail*)
done

lemma *abs-expr-id* [*heap-abs*]:
 abs-expr id (λ-. True) A A
 apply (*clarsimp simp: abs-expr-def*)
done

lemma *abs-expr-lambda-null* [*heap-abs*]:
 abs-expr st P A C \Longrightarrow abs-expr st P (λs r. A s) (λs r. C s)
 apply (*clarsimp simp: abs-expr-def*)

done

lemma *abs-modify-id* [*heap-abs*]:
 abs-modifies id ($\lambda\cdot$. *True*) *A A*
 apply (*clarsimp simp: abs-modifies-def*)
 done

lemma *corresXF-exec-concrete* [*intro?*]:
 corresXF id ret-xf ex-xf P A C \implies *corresXF st ret-xf ex-xf P (exec-concrete st A) C*
 by (*force simp add: corresXF-refines-conv refines-def-old reaches-exec-concrete succeeds-exec-concrete-iff split: xval-splits*)

lemma *L2Tcorres-exec-concrete* [*heap-abs*]:
 L2Tcorres id A C \implies *L2Tcorres st (exec-concrete st (L2-call A emb ns)) (L2-call C emb ns)*
 apply (*clarsimp simp: L2Tcorres-def L2-call-def map-exn-catch-conv*)
 apply (*rule corresXF-exec-concrete*)
 apply (*rule CorresXF.corresXF-except [where P' = $\lambda x y s. x = y$]*)
 apply *assumption*
 subgoal
 by (*auto simp add: corresXF-refines-conv*)
 subgoal
 by (*auto simp add: runs-to-partial-def-old split: xval-splits*)
 subgoal by simp
 done

lemma *L2Tcorres-exec-concrete-simpl* [*heap-abs*]:
 L2Tcorres id A C \implies *L2Tcorres st (exec-concrete st (L2-call-L1 arg-xf gs ret-xf A)) (L2-call-L1 arg-xf gs ret-xf C)*
 apply (*clarsimp simp: L2Tcorres-def L2-call-L1-def*)
 apply (*rule corresXF-exec-concrete*)
 apply (*clarsimp simp add: corresXF-refines-conv*)
 apply (*rule refines-bind-bind-exn-wp*)
 apply (*clarsimp split: xval-splits*)
 apply (*rule refines-get-state*)
 apply (*clarsimp split: xval-splits*)
 apply (*rule refines-bind-bind-exn-wp*)
 apply (*clarsimp split: xval-splits*)
 apply (*rule refines-select*)
 apply (*clarsimp split: xval-splits*)
 subgoal for x
 apply (*rule exI[where x=x]*)
 apply (*erule-tac x=x in allE*)
 apply (*clarsimp*)
 apply (*rule refines-run-bind*)
 apply (*clarsimp split: exception-or-result-splits*)
 apply (*erule refines-weaken*)
 apply (*clarsimp split: xval-splits*)

```

apply (rule refines-bind-bind-exn-wp)
apply (clarsimp split: xval-splits)
apply (rule refines-set-state)
apply (clarsimp split: xval-splits)
done
done

```

```

lemma corresXF-exec-abstract [intro?]:
  corresXF st ret-xf ex-xf P A C  $\implies$  corresXF id ret-xf ex-xf P (exec-abstract st A) C
  by (force simp: corresXF-refines-conv refines-def-old reaches-exec-abstract split: xval-splits)

```

```

lemma L2Tcorres-exec-abstract [heap-abs]:
  L2Tcorres st A C  $\implies$  L2Tcorres id (exec-abstract st (L2-call A emb ns)) (L2-call C emb ns)
  apply (clarsimp simp: L2-call-def map-exn-catch-conv L2Tcorres-def)
  apply (rule corresXF-exec-abstract)
  apply (rule CorresXF.corresXF-except [ where  $P' = \lambda x y s. x = y$  ])
  apply assumption
  subgoal by (auto simp add: corresXF-refines-conv)
  subgoal by (auto simp add: runs-to-partial-def-old split: xval-splits)
  subgoal by simp
  done

```

```

lemma L2Tcorres-call [heap-abs]:
  L2Tcorres st A C  $\implies$  L2Tcorres st (L2-call A emb ns) (L2-call C emb ns)
  unfolding L2Tcorres-def L2-call-def map-exn-catch-conv
  apply (rule CorresXF.corresXF-except [ where  $P' = \lambda x y s. x = y$  ])
  apply assumption
  subgoal by (auto simp add: corresXF-refines-conv)
  subgoal by (auto simp add: runs-to-partial-def-old split: xval-splits)
  subgoal by simp
  done

```

named-theorems

```

valid-implies-c-guard and
read-commutes and
write-commutes and
field-write-commutes and
write-valid-preservation and
lift-heap-update-padding-heap-update-conv

```

```

locale valid-implies-cguard =

```



```

fixes st::'s ⇒ 't
fixes v::'t ⇒ 'a::c-type ptr ⇒ bool
assumes valid-implies-c-guard[valid-implies-c-guard]: v (st s) p ⇒ c-guard p

locale read-simulation =
  fixes st::'s ⇒ 't
  fixes v::'t ⇒ 'a::c-type ptr ⇒ bool
  fixes r::'t ⇒ 'a ptr ⇒ 'a
  fixes t-hrs::'s ⇒ heap-raw-state
  assumes read-commutes[read-commutes]: v (st s) p ⇒
    r (st s) p = h-val (hrs-mem (t-hrs s)) p

locale write-simulation =
  heap-raw-state t-hrs t-hrs-upd
for
  t-hrs::('s ⇒ heap-raw-state) and
  t-hrs-upd::(heap-raw-state ⇒ heap-raw-state) ⇒ 's ⇒ 's +
fixes st::'s ⇒ 't
fixes v::'t ⇒ 'a::mem-type ptr ⇒ bool
fixes w::'a ptr ⇒ ('a ⇒ 'a) ⇒ 't ⇒ 't

  assumes write-padding-commutes[write-commutes]: v (st s) p ⇒ length bs =
size-of TYPE('a) ⇒
  st (t-hrs-upd (hrs-mem-update (heap-update-padding p x bs)) s) =
    w p ( $\lambda$ -. x) (st s)

begin
lemma write-commutes[write-commutes]:
  assumes valid: v (st s) p
  shows st (t-hrs-upd (hrs-mem-update (heap-update p x)) s) =
    w p ( $\lambda$ -. x) (st s)

proof –
  have eq: hrs-mem-update (heap-update p x) =
    hrs-mem-update ( $\lambda$ h. heap-update-padding p x (heap-list h (size-of TYPE('a))
(ptr-val p)) h)
    using heap-update-heap-update-padding-conv
    by metis

  show ?thesis
    apply (simp only: eq)
    apply (subst write-padding-commutes [symmetric, where bs=heap-list (hrs-mem
(t-hrs s)) (size-of TYPE('a)) (ptr-val p))])
    apply (rule valid)
    apply clarsimp
    by (metis (no-types, lifting) heap.upd-cong)
qed

lemma lift-heap-update-padding-heap-update-conv[lift-heap-update-padding-heap-update-conv]:
  v (st s) p ⇒ length bs = size-of TYPE('a) ⇒

```

```

      st (t-hrs-upd (hrs-mem-update (heap-update-padding p x bs)) s) =
      st (t-hrs-upd (hrs-mem-update (heap-update p x)) s)
using write-padding-commutes write-commutes by auto

lemma write-commutes-atomic:  $\forall s p x. v (st s) p \longrightarrow$ 
  st (t-hrs-upd (hrs-mem-update (heap-update p x)) s) =
      w p ( $\lambda \cdot x$ ) (st s)
using write-commutes by blast

end

locale write-preserves-valid =
  fixes v :: 't  $\Rightarrow$  'a::c-type ptr  $\Rightarrow$  bool
  fixes w :: 'a ptr  $\Rightarrow$  ('b  $\Rightarrow$  'b)  $\Rightarrow$  't  $\Rightarrow$  't
  assumes valid-preserved: v (w p' f s) p = v s p
begin
lemma valid-preserved-pointless[simp]: v (w p' f s) = v s
  by (rule ext) (rule valid-preserved)
end

locale valid-only-typ-desc-dependent =
  fixes t-hrs :: ('s  $\Rightarrow$  heap-raw-state)
  fixes st :: 's  $\Rightarrow$  't
  fixes v :: 't  $\Rightarrow$  'a::c-type ptr  $\Rightarrow$  bool
  assumes valid-same-typ-desc [valid-same-typ-descs]: hrs-htd (t-hrs s) = hrs-htd
(t-hrs t)  $\Longrightarrow$  v (st s) p = v (st t) p

locale heap-typing-simulation =
  open-types  $\mathcal{T}$  + t-hrs: heap-raw-state t-hrs t-hrs-upd + heap-typing-state heap-typing
heap-typing-upd
for
   $\mathcal{T}$  and
  t-hrs :: ('s  $\Rightarrow$  heap-raw-state) and
  t-hrs-upd :: (heap-raw-state  $\Rightarrow$  heap-raw-state)  $\Rightarrow$  ('s  $\Rightarrow$  's) and
  heap-typing :: 't  $\Rightarrow$  heap-typ-desc and
  heap-typing-upd :: (heap-typ-desc  $\Rightarrow$  heap-typ-desc)  $\Rightarrow$  't  $\Rightarrow$  't +
fixes st :: 's  $\Rightarrow$  't
assumes heap-typing-commutes[simp]: heap-typing (st s) = hrs-htd (t-hrs s)
assumes lift-heap-update-list-stack-byte-independent:
  ( $\bigwedge i. i < \text{length } bs \Longrightarrow \text{root-ptr-valid (hrs-htd (t-hrs s)) ((p::\text{stack-byte ptr}) +_p$ 
int i))  $\Longrightarrow$ 
  st (t-hrs-upd (hrs-mem-update (heap-update-list (ptr-val p) bs)) s) = st s
assumes st-eq-upto-padding:
  mem-type-u t  $\Longrightarrow$  padding-closed-under-all-fields t  $\Longrightarrow$ 
  ptr-valid-u t (hrs-htd (t-hrs s)) a  $\Longrightarrow$  eq-upto-padding t bs bs'  $\Longrightarrow$ 
  st (t-hrs-upd (hrs-mem-update (heap-update-list a bs)) s) =
  st (t-hrs-upd (hrs-mem-update (heap-update-list a bs')) s)

```

begin

lemma *heap-typing-upd-commutes*: $\text{heap-typing } (\text{heap-typing-upd } f \text{ } (st \ s)) = \text{hrs-htd } (t\text{-hrs } (t\text{-hrs-upd } (\text{hrs-htd-update } f) \ s))$
apply (*simp add: hrs-htd-update*)
done

lemma *write-simulation-alt*:

assumes $v: \bigwedge s \ p. \ v \ (st \ s) \ p \implies \text{ptr-valid } (\text{hrs-htd } (t\text{-hrs } s)) \ p$
assumes $*$: $\bigwedge s \ (p::'a::\text{xmem-type } \text{ptr}) \ x. \ v \ (st \ s) \ p \implies$
 $st \ (t\text{-hrs-upd } (\text{hrs-mem-update } (\text{heap-update } p \ x)) \ s) = w \ p \ (\lambda-. \ x) \ (st \ s)$
shows *write-simulation t-hrs t-hrs-upd st v w*

proof

fix $s \ p \ x$ **and** $bs :: \text{byte list}$ **assume** $p: v \ (st \ s) \ p$ **and** $bs: \text{length } bs = \text{size-of } \text{TYPE}('a)$

have [*simp*]: $t\text{-hrs-upd } (\text{hrs-mem-update } (\text{heap-update } p \ x)) \ s =$
 $t\text{-hrs-upd } (\text{hrs-mem-update } (\text{heap-update-list } (\text{ptr-val } p)$
 $(\text{to-bytes } x \ (\text{heap-list } (\text{hrs-mem } (t\text{-hrs } s)) \ (\text{size-of } \text{TYPE}('a)) \ (\text{ptr-val } p)))) \ s$
by (*rule t-hrs.heap.upd-cong*) (*simp add: heap-update-def*)

show $st \ (t\text{-hrs-upd } (\text{hrs-mem-update } (\text{heap-update-padding } p \ x \ bs)) \ s) = w \ p \ (\lambda-. \ x) \ (st \ s)$

apply (*subst * [OF p, symmetric]*)
apply (*simp add: heap-update-padding-def [abs-def]*)
apply (*rule st-eq-upto-padding [where t=typ-uinfo-t TYPE('a)]*)
apply (*rule typ-uinfo-t-mem-type*)
apply (*rule padding-closed-under-all-fields-typ-uinfo-t*)
apply (*subst ptr-valid-def [symmetric]*)
apply (*simp add: v p*)
unfolding *to-bytes-def typ-uinfo-t-def*
apply (*rule field-lookup-access-ti-eq-upto-padding [where f=[] and 'b='a]*)
apply (*simp-all add: bs size-of-def*)
done

qed

end

locale *typ-heap-simulation* =
heap-raw-state t-hrs t-hrs-update +
read-simulation st v r t-hrs +
write-simulation t-hrs t-hrs-update st v w +
write-preserves-valid v w +
valid-implies-cguard st v +
valid-only-typ-desc-dependent t-hrs st v +
pointer-lense r w
for
 $st:: 's \Rightarrow 't$ **and**
 $r:: 't \Rightarrow ('a::\text{xmem-type } \text{ptr}) \Rightarrow 'a$ **and**

$w :: 'a \text{ ptr} \Rightarrow ('a \Rightarrow 'a) \Rightarrow 't \Rightarrow 't$ **and**
 $v :: 't \Rightarrow ('a :: \text{xmem-type}) \text{ ptr} \Rightarrow \text{bool}$ **and**
 $t\text{-hrs} :: 's \Rightarrow \text{heap-raw-state}$ **and**
 $t\text{-hrs-update} :: (\text{heap-raw-state} \Rightarrow \text{heap-raw-state}) \Rightarrow 's \Rightarrow 's$

begin

lemma *write-valid-preservation* [*write-valid-preservation*]:
shows $v (st (t\text{-hrs-update} (hrs\text{-mem-update} (heap\text{-update } q \ x)) \ s)) \ p = v (st \ s) \ p$
by (*metis hrs-htd-mem-update get-upd valid-same-typ-desc*)

lemma *write-padding-valid-preservation* [*write-valid-preservation*]:
shows $v (st (t\text{-hrs-update} (hrs\text{-mem-update} (heap\text{-update-padding } q \ x \ bs)) \ s)) \ p$
 $= v (st \ s) \ p$
by (*metis hrs-htd-mem-update get-upd valid-same-typ-desc*)

end

locale *stack-simulation* =
heap-typing-simulation \mathcal{T} *t-hrs* *t-hrs-update* *heap-typing* *heap-typing-upd* *st* +
typ-heap-typing *r* *w* *heap-typing* *heap-typing-upd* \mathcal{S}
for
 \mathcal{T} **and**
 $st :: 's \Rightarrow 't$ **and**
 $r :: 't \Rightarrow ('a :: \text{xmem-type}) \text{ ptr} \Rightarrow 'a$ **and**
 $w :: 'a \text{ ptr} \Rightarrow ('a \Rightarrow 'a) \Rightarrow 't \Rightarrow 't$ **and**
 $t\text{-hrs} :: 's \Rightarrow \text{heap-raw-state}$ **and**
 $t\text{-hrs-update} :: (\text{heap-raw-state} \Rightarrow \text{heap-raw-state}) \Rightarrow 's \Rightarrow 's$ **and**
 $heap\text{-typing} :: 't \Rightarrow \text{heap-typ-desc}$ **and**
 $heap\text{-typing-upd} :: (\text{heap-typ-desc} \Rightarrow \text{heap-typ-desc}) \Rightarrow 't \Rightarrow 't$ **and**
 $\mathcal{S} :: \text{addr set} +$

assumes *sim-stack-alloc*:
 $\bigwedge p \ d \ vs \ bs \ s \ n.$
 $(p, d) \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a) (hrs\text{-htd } (t\text{-hrs } s)) \Longrightarrow \text{length } vs = n \Longrightarrow$
 $\text{length } bs = n * \text{size-of } \text{TYPE}('a) \Longrightarrow$
 $st (t\text{-hrs-update} (hrs\text{-mem-update} (\text{fold } (\lambda i. \text{heap-update-padding } (p +_p \ \text{int } i) \ (vs!i) \ (\text{take } (\text{size-of } \text{TYPE}('a)) \ (\text{drop } (i * \text{size-of } \text{TYPE}('a)) \ bs)))) [0..<n]) \circ$
 $hrs\text{-htd-update } (\lambda-. \ d)) \ s) =$
 $(\text{fold } (\lambda i. \ w (p +_p \ \text{int } i) \ (\lambda-. \ (vs \ ! \ i))) [0..<n]) (heap\text{-typing-upd } (\lambda-. \ d) \ (st$
 $s))$

assumes *sim-stack-release*: $\bigwedge p \ s \ n. (\bigwedge i. i < n \Longrightarrow \text{root-ptr-valid } (hrs\text{-htd } (t\text{-hrs}$
 $s)) (p +_p \ \text{int } i)) \Longrightarrow$
 $\text{length } bs = n * \text{size-of } \text{TYPE}('a) \Longrightarrow$
 $st (t\text{-hrs-update} (hrs\text{-mem-update} (\text{heap-update-list } (ptr\text{-val } p) \ bs) \circ hrs\text{-htd-update}$
 $(\text{stack-releases } n \ p)) \ s) =$
 $((heap\text{-typing-upd } (\text{stack-releases } n \ p) \ (\text{fold } (\lambda i. \ w (p +_p \ \text{int } i) \ (\lambda-$
 $c\text{-type-class.zero})) [0..<n] \ (st \ s))))$

assumes *stack-byte-zero*: $\bigwedge p d i. (p, d) \in \text{stack-allocs } n \mathcal{S} \text{ TYPE}('a) \text{ (hrs-htd } (t\text{-hrs } s)) \implies i < n \implies r (st\ s) (p +_p \text{ int } i) = \text{ZERO}('a)$

lemma (in *typ-heap-simulation*) *L2Tcorres-IO-modify-paddingE* [*heap-abs*]:
assumes *abs-expr st P a c*
shows *L2Tcorres st (L2-seq (L2-guard ($\lambda t. v\ t\ p \wedge P\ t$)) ($\lambda-. (L2\text{-modify } (\lambda s. w\ p\ (\lambda-. a\ s)\ s))))$)*
(IO-modify-heap-paddingE (p::'a ptr) c)
using *assms*
using *length-to-bytes write-padding-commutes unfolding liftE-IO-modify-heap-padding*
by (*auto simp add: abs-expr-def L2Tcorres-def corresXF-refines-conv L2-defs*
IO-modify-heap-padding-def refines-def-old reaches-bind succeeds-bind split:
xval-splits)

locale *typ-heap-typing-stack-simulation* =
typ-heap-simulation st r w v t-hrs t-hrs-update +
stack-simulation \mathcal{T} st r w t-hrs t-hrs-update heap-typing heap-typing-upd \mathcal{S}
for
 \mathcal{T} **and**
st:: 's \Rightarrow 't and
r:: 't \Rightarrow ('a::xmem-type) ptr \Rightarrow 'a and
w:: 'a ptr \Rightarrow ('a \Rightarrow 'a) \Rightarrow 't \Rightarrow 't and
v:: 't \Rightarrow ('a::xmem-type) ptr \Rightarrow bool and
t-hrs :: 's \Rightarrow heap-raw-state and
t-hrs-update:: (heap-raw-state \Rightarrow heap-raw-state) \Rightarrow 's \Rightarrow 's and
heap-typing :: 't \Rightarrow heap-typ-desc and
heap-typing-upd :: (heap-typ-desc \Rightarrow heap-typ-desc) \Rightarrow 't \Rightarrow 't and
 \mathcal{S} :: addr set
begin

sublocale *monolithic: stack-heap-raw-state t-hrs t-hrs-update \mathcal{S}*
by (*unfold-locales*)

definition *rel-split-heap* $\equiv \lambda s_c s_a. s_a = st\ s_c$

lemma *rel-split-heap-stack-free-eq*:
rel-split-heap $s_c s_a \implies \text{stack-free (hrs-htd (t-hrs } s_c)) = \text{stack-free (heap-typing } s_a)$
by (*simp only: rel-split-heap-def heap-typing-commutes*)

definition *rel-stack-free-eq* **where**
rel-stack-free-eq $s_c s_a \equiv \text{stack-free (hrs-htd (t-hrs } s_c)) = \text{stack-free (heap-typing } s_a)$

lemma *rel-prod-rel-split-heap-conv*:

$rel\text{-}prod (=) rel\text{-}split\text{-}heap = (\lambda(v, t) (r, s).$
 $s = st\ t \wedge (case\ v\ of\ Exn\ x \Rightarrow r = Exn\ x \mid Result\ x \Rightarrow r = Result\ x))$
by (*auto simp add: rel-split-heap-def rel-prod-conv fun-eq-iff split: prod.splits xval-splits*)

lemma *L2Tcorres-refines*:

$L2Tcorres\ st\ f_a\ f_c \Longrightarrow refines\ f_c\ f_a\ s\ (st\ s)\ (rel\text{-}prod\ (=)\ rel\text{-}split\text{-}heap)$
by (*simp add: L2Tcorres-def corresXF-refines-conv rel-prod-rel-split-heap-conv*)

lemma *refines-L2Tcorres*:

assumes $f: \bigwedge s. refines\ f_c\ f_a\ s\ (st\ s)\ (rel\text{-}prod\ (=)\ rel\text{-}split\text{-}heap)$
shows $L2Tcorres\ st\ f_a\ f_c$
using f
by (*simp add: L2Tcorres-def corresXF-refines-conv rel-prod-rel-split-heap-conv*)

lemma *L2Tcorres-refines-conv*:

$L2Tcorres\ st\ f_a\ f_c \longleftrightarrow (\forall s. refines\ f_c\ f_a\ s\ (st\ s)\ (rel\text{-}prod\ (=)\ rel\text{-}split\text{-}heap))$
by (*auto simp add: L2Tcorres-refines refines-L2Tcorres*)

lemma *sim-guard-with-fresh-stack-ptr*:

fixes $f_c:: 'a\ ptr \Rightarrow ('b, 'c, 's)\ exn\text{-}monad$
assumes $init: init_a\ (st\ s) = init_c\ s$
assumes $f: \bigwedge s\ p::'a\ ptr. refines\ (f_c\ p)\ (f_a\ p)\ s\ (st\ s)\ (rel\text{-}prod\ (=)\ rel\text{-}split\text{-}heap)$
shows $refines$
 $(monolithic.with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_c\ f_c)$
 $(guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_a\ f_a)\ s\ (st\ s)$
 $(rel\text{-}prod\ (=)\ rel\text{-}split\text{-}heap)$
unfolding $monolithic.with\text{-}fresh\text{-}stack\text{-}ptr\text{-}def\ guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\text{-}def$
 $stack\text{-}ptr\text{-}acquire\text{-}def\ stack\text{-}ptr\text{-}release\text{-}def\ assume\text{-}stack\text{-}alloc\text{-}def$
apply (*rule refines-bind-bind-exn [where Q= (rel-prod (=) rel-split-heap)]*)
subgoal
apply (*rule refines-assume-result-and-state'*)
using $sim\text{-}stack\text{-}alloc\ stack\text{-}byte\text{-}zero$
by (*fastforce simp add: sim-set-def rel-split-heap-def init split: xval-splits*)
apply $simp$
apply $simp$
apply $simp$
apply (*rule refines-rel-prod-guard-on-exit [where S'=rel-split-heap]*)
apply (*subst (asm) rel-split-heap-def*)
apply $simp$
apply (*rule f*)
subgoal by (*auto simp add: rel-split-heap-def sim-stack-release*)
subgoal by (*simp add: Ex-list-of-length*)
done

lemma *sim-with-fresh-stack-ptr*:

fixes $f_c:: 'a\ ptr \Rightarrow ('b, 'c, 's)\ exn\text{-}monad$
assumes $init: init_a\ (st\ s) = init_c\ s$
assumes $f: \bigwedge s\ p::'a\ ptr. refines\ (f_c\ p)\ (f_a\ p)\ s\ (st\ s)\ (rel\text{-}prod\ (=)\ rel\text{-}split\text{-}heap)$

assumes *typing-unchanged*: $\bigwedge s p::'a \text{ ptr. } (f_c p) \cdot s \text{ ?}\{\lambda r t. \text{typing.unchanged-typing-on } \mathcal{S} s t\}$
shows *refines*
 (*monolithic.with-fresh-stack-ptr* *n* *init_c* *f_c*)
 (*with-fresh-stack-ptr* *n* *init_a* *f_a*) *s* (*st s*)
 (*rel-prod* (=) *rel-split-heap*)
apply (*simp* *add*: *monolithic.with-fresh-stack-ptr-def with-fresh-stack-ptr-def*
 stack-ptr-acquire-def stack-ptr-release-def assume-stack-alloc-def)
apply (*rule* *refines-bind-bind-exn* [**where** $Q = \lambda(r,t) (r',t')$.
 (*rel-prod* (=) *rel-split-heap*) (*r,t*) (*r',t'*) \wedge
 ($\exists p. r = \text{Result } p \wedge (\forall i < n. \text{ptr-span } (p +_p \text{int } i) \subseteq \mathcal{S} \wedge \text{root-ptr-valid}$
 (*heap-typing* *t'*) ($p +_p \text{int } i$)))]], *simp-all*)

subgoal
 apply (*rule* *refines-assume-result-and-state'*)
 using *sim-stack-alloc stack-byte-zero stack-allocs-S*
 apply (*clarsimp* *simp* *add*: *sim-set-def init rel-split-heap-def, safe*)
 apply *blast+*
 by (*smt* (*verit*) *hrs-htd-update stack-allocs-cases*)

subgoal for *p t t'*
 apply (*rule* *refines-runs-to-partial-rel-prod-on-exit* [**where** $S' = \text{rel-split-heap}$
and $P = \text{typing.unchanged-typing-on } \mathcal{S} t$])
 apply (*subst* (*asm*) *rel-split-heap-def*)
 apply *simp*
 apply (*rule* *f*)
 apply (*rule* *typing-unchanged*)
 subgoal for *s_c s_a t_c*
 apply *clarsimp*
 apply (*clarsimp* *simp* *add*: *rel-split-heap-def*)
 apply (*subst* *sim-stack-release*)
 subgoal for *bs i*
 using *typing.unchanged-typing-on-root-ptr-valid-preservation* [**where** $S = \mathcal{S}$
and $s = t$ **and** $t = s_c$ **and** $p = (p +_p \text{int } i)$]
 by *auto*
 subgoal by *auto*
 subgoal by *auto*
 done
subgoal by (*simp* *add*: *Ex-list-of-length*)
done
done

lemma *sim-assume-with-fresh-stack-ptr*:
 fixes $f_c::'a \text{ ptr} \Rightarrow ('b, 'c, 's) \text{ exn-monad}$
 assumes *init*: $\text{init}_a (st s) = \text{init}_c s$
 assumes *f*: $\bigwedge s p::'a \text{ ptr. } \text{refines } (f_c p) (f_a p) s (st s) (\text{rel-prod } (=) \text{rel-split-heap})$
 assumes *typing-unchanged*: $\bigwedge s p::'a \text{ ptr. } (f_c p) \cdot s \text{ ?}\{\lambda r t. \text{typing.unchanged-typing-on } \mathcal{S} s t\}$
shows *refines*

```

      (monolithic.with-fresh-stack-ptr n initc fc)
      (assume-with-fresh-stack-ptr n inita fa) s (st s)
      (rel-prod (=) rel-split-heap)
unfolding monolithic.with-fresh-stack-ptr-def assume-with-fresh-stack-ptr-def
      stack-ptr-acquire-def stack-ptr-release-def assume-stack-alloc-def
apply (rule refines-bind-bind-exn [where Q= λ(r,t) (r',t').
      (rel-prod (=) rel-split-heap) (r,t) (r',t') ∧
      (∃ p. r = Result p ∧ (∀ i < n. ptr-span (p +p int i) ⊆ S ∧ root-ptr-valid
      (heap-typing t') (p +p int i)))]])

subgoal
  apply (rule refines-assume-result-and-state')
  using sim-stack-alloc stack-byte-zero stack-allocs-S
  apply (clarsimp simp add: sim-set-def init rel-split-heap-def hrs-htd-update
  stack-allocs-root-ptr-valid-same)
  apply blast
  done
apply simp
apply simp
apply simp
subgoal for p q t t'
  apply (rule refines-runs-to-partial-rel-prod-assume-on-exit [where S'=rel-split-heap
and P=typing.unchanged-typing-on S t])
  apply (subst (asm) rel-split-heap-def )
  apply simp
  apply (rule f)
  apply (rule typing-unchanged)
subgoal for sc sa tc
  apply clarsimp
  apply (clarsimp simp add: rel-split-heap-def)
  apply (subst sim-stack-release)
  subgoal for bs i
    using typing.unchanged-typing-on-root-ptr-valid-preservation [where S=S
and s=t and t=sc and p= (p +p int i)]
    by auto
    subgoal by auto
    subgoal by auto
    done
  subgoal by (simp add: Ex-list-of-length)
  subgoal for sc sa
    apply clarsimp
    apply (clarsimp simp add: rel-split-heap-def)
    subgoal for i
      using typing.unchanged-typing-on-root-ptr-valid-preservation [where S=S
and s=t and t=sc and p= (p +p int i)]
      by auto
    done
  done
done

```


lemma *L2Tcorres-guard-with-fresh-stack-ptr* [*heap-abs*]:
assumes *rew*: *struct-rewrite-expr* *P* *init_c'* *init_c*
assumes *grd*: *abs-guard* *st* *P'* *P*
assumes *expr*: *abs-expr* *st* *Q* *init_a* *init_c'*
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* ($\bigwedge p::'a$ *ptr*. *L2Tcorres* *st* (*f_a* *p*) (*f_c* *p*))
shows *L2Tcorres* *st* (*L2-seq* (*L2-guard* (λs . *P' s* \wedge *Q s*))
(λ -. (*guard-with-fresh-stack-ptr* *n* *init_a* (*L2-VARS* *f_a* *nm*))))
(*monolithic.with-fresh-stack-ptr* *n* *init_c* (*L2-VARS* *f_c* *nm*)))
apply (*rule refines-L2Tcorres*)
apply (*simp add*: *L2-seq-def* *L2-guard-def* *L2-VARS-def*)
apply (*rule refines-bind-guard-right*)
apply *clarsimp*
apply (*rule sim-guard-with-fresh-stack-ptr*)
subgoal for *s*
using *rew* *grd* *expr*
by (*auto simp add*: *struct-rewrite-expr-def* *abs-guard-def* *abs-expr-def*)
subgoal for *s* *s'* *p*
apply (*rule L2Tcorres-refines*)
apply (*rule f*)
done
done

lemma *L2Tcorres-with-fresh-stack-ptr*:
assumes *typing-unchanged*: $\bigwedge s$ *p*::'a *ptr*. (*f_c* *p*) \cdot *s* $\{ \lambda r$ *t*. *typing.unchanged-typing-on* *S* *s* *t* $\}$
assumes *rew*: *struct-rewrite-expr* *P* *init_c'* *init_c*
assumes *grd*: *abs-guard* *st* *P'* *P*
assumes *expr*: *abs-expr* *st* *Q* *init_a* *init_c'*
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* ($\bigwedge p::'a$ *ptr*. *L2Tcorres* *st* (*f_a* *p*) (*f_c* *p*))
shows *L2Tcorres* *st* (*L2-seq* (*L2-guard* (λs . *P' s* \wedge *Q s*))
(λ -. (*with-fresh-stack-ptr* *n* *init_a* (*L2-VARS* *f_a* *nm*))))
(*monolithic.with-fresh-stack-ptr* *n* *init_c* (*L2-VARS* *f_c* *nm*)))
apply (*rule refines-L2Tcorres*)
apply (*simp add*: *L2-seq-def* *L2-guard-def* *L2-VARS-def*)
apply (*rule refines-bind-guard-right*)
apply *clarsimp*
apply (*rule sim-with-fresh-stack-ptr*)
subgoal for *s*
using *rew* *grd* *expr*
by (*auto simp add*: *struct-rewrite-expr-def* *abs-guard-def* *abs-expr-def*)
subgoal for *s* *s'* *p*
apply (*rule L2Tcorres-refines*)
apply (*rule f*)
done

using *typing-unchanged* by *blast*

lemma *L2Tcorres-assume-with-fresh-stack-ptr*[*heap-abs*]:
assumes *typing-unchanged*: $\bigwedge s p :: 'a \text{ ptr}. (f_c p) \cdot s \text{ ?}\{\lambda r t. \text{typing.unchanged-typing-on } S \ s \ t\}$
assumes *rew*: *struct-rewrite-expr* *P* *init_c'* *init_c*
assumes *grd*: *abs-guard* *st* *P'* *P*
assumes *expr*: *abs-expr* *st* *Q* *init_a* *init_c'*
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* ($\bigwedge p :: 'a \text{ ptr}. L2Tcorres \ st \ (f_a \ p) \ (f_c \ p)$)
shows *L2Tcorres* *st* (*L2-seq* (*L2-guard* ($\lambda s. P' \ s \wedge Q \ s$))
($\lambda -. (assume-with-fresh-stack-ptr \ n \ init_a \ (L2-VARS \ f_a \ nm))$))
(*monolithic.with-fresh-stack-ptr* *n* *init_c* (*L2-VARS* *f_c* *nm*)))
apply (*rule refines-L2Tcorres*)
apply (*simp add: L2-seq-def L2-guard-def L2-VARS-def*)
apply (*rule refines-bind-guard-right*)
apply *clarsimp*
apply (*rule sim-assume-with-fresh-stack-ptr*)
subgoal for *s*
 using *rew* *grd* *expr*
 by (*auto simp add: struct-rewrite-expr-def abs-guard-def abs-expr-def*)
subgoal for *s* *s'* *p*
 apply (*rule L2Tcorres-refines*)
 apply (*rule f*)
 done
using *typing-unchanged* by *blast*

lemma *unchanged-typing-commutes*: *typing.unchanged-typing-on* *S* *s* *t* \implies *unchanged-typing-on* *S* (*st* *s*) (*st* *t*)
 using *heap-typing-commutes* [of *s*] *heap-typing-commutes* [of *t*]
 by (*auto simp add: unchanged-typing-on-def typing.unchanged-typing-on-def*)
end

named-theorems *read-stack-byte-ZERO-base*
 and *read-stack-byte-ZERO-step*
 and *read-stack-byte-ZERO-step-subst*

lemma (in *open-types*) *ptr-span-with-stack-byte-type-implies-ptr-invalid*:
fixes *p* :: (*'a* :: {*mem-type*, *stack-type*}) *ptr*
assumes *: $\forall a \in \text{ptr-span } p. \text{root-ptr-valid } d \ (PTR \ (\text{stack-byte}) \ a)$
shows $\neg \text{ptr-valid-u} \ (\text{typ-uinfo-t } TYPE('a)) \ d \ (\text{ptr-val } p)$
by (*metis* *assms disjoint-iff fold-ptr-valid' in-ptr-span-itself ptr-exhaust-eq ptr-valid-stack-byte-disjoint*)

lemma (in *open-types*)
ptr-span-with-stack-byte-type-implies-ZERO[*read-stack-byte-ZERO-base*]:

fixes $p :: ('a :: \{\text{mem-type}, \text{stack-type}\}) \text{ ptr}$
assumes $\forall a \in \text{ptr-span } p. \text{root-ptr-valid } (\text{hrs-htd } d) (\text{PTR } (\text{stack-byte}) a)$
shows $\text{the-default } (\text{ZERO}('a)) (\text{plift } d p) = \text{ZERO}('a)$
using $\text{ptr-span-with-stack-byte-type-implies-ptr-invalid}[\text{OF } \text{assms}]$
by $(\text{simp add: fold-ptr-valid' plift-None})$

lemma $\text{ptr-span-array-ptr-index-subset-ptr-span}$:
fixes $p :: (('a :: \{\text{array-outer-max-size}\})['b :: \text{array-max-count}]) \text{ ptr}$
assumes $i < \text{CARD}('b)$
shows $\text{ptr-span } (\text{array-ptr-index } p \ c \ i) \subseteq \text{ptr-span } p$
using assms
apply $(\text{simp add: array-ptr-index-def ptr-add-def})$
apply $(\text{rule intvl-sub-offset})$
apply $(\text{rule order-trans}[\text{of } - \ i \ * \ \text{size-of } \text{TYPE}('a) \ + \ \text{size-of } \text{TYPE}('a)])$
apply $(\text{simp add: unat-le-helper})$
apply $(\text{simp add: add.commute less-le-mult-nat})$
done

lemma $\text{read-stack-byte-ZERO-array-intro}[\text{read-stack-byte-ZERO-step}]$:
fixes $q :: ('a :: \{\text{array-outer-max-size}\})['b :: \text{array-max-count}] \text{ ptr}$
assumes $\text{ptr-span-has-stack-byte-type}$:
 $\forall a \in \text{ptr-span } q. \text{root-ptr-valid } d (\text{PTR}(\text{stack-byte}) a)$
assumes $\text{subtype-reads-ZERO}$:
 $\bigwedge p :: 'a \text{ ptr}. \forall a \in \text{ptr-span } p. \text{root-ptr-valid } d (\text{PTR}(\text{stack-byte}) a) \implies r \ s \ p = \text{ZERO}('a)$
shows $(\text{ARRAY } i. \ r \ s \ (\text{array-ptr-index } q \ c \ i)) = \text{ZERO}('a['b])$
apply (rule array-ext)
apply $(\text{simp add: array-index-zero})$
apply $(\text{rule subtype-reads-ZERO})$
using $\text{ptr-span-has-stack-byte-type ptr-span-array-ptr-index-subset-ptr-span}$ **by**
 blast

lemma $\text{read-stack-byte-ZERO-array-2dim-intro}[\text{read-stack-byte-ZERO-step}]$:
fixes $q :: ('a :: \{\text{array-inner-max-size}\})['b :: \text{array-max-count}]['c :: \text{array-max-count}] \text{ ptr}$
assumes $\text{ptr-span-has-stack-byte-type}$:
 $\forall a \in \text{ptr-span } q. \text{root-ptr-valid } d (\text{PTR}(\text{stack-byte}) a)$
assumes $\text{subtype-reads-ZERO}$:
 $\bigwedge p :: 'a \text{ ptr}. \forall a \in \text{ptr-span } p. \text{root-ptr-valid } d (\text{PTR}(\text{stack-byte}) a) \implies r \ s \ p = \text{ZERO}('a)$
shows $(\text{ARRAY } i \ j. \ r \ s \ (\text{array-ptr-index } (\text{array-ptr-index } q \ c \ i) \ c \ j)) = \text{ZERO}('a['b]['c])$
apply (rule array-ext)
apply $(\text{simp add: array-index-zero})$
apply (rule array-ext)
apply $(\text{simp add: array-index-zero})$
apply $(\text{rule subtype-reads-ZERO})$
by $(\text{metis } (\text{no-types}, \text{opaque-lifting}) \text{subset-iff ptr-span-has-stack-byte-type ptr-span-array-ptr-index-subset-ptr-span})$

lemma *read-stack-byte-ZERO-field-intro*[*read-stack-byte-ZERO-step*]:
fixes $q :: 'a :: \text{mem-type ptr}$
assumes *ptr-span-has-stack-byte-type*:
 $\forall a \in \text{ptr-span } q. \text{root-ptr-valid } d \text{ (PTR(stack-byte) } a)$
assumes *subtype-reads-ZERO*:
 $\bigwedge p :: 'b :: \text{mem-type ptr}. \forall a \in \text{ptr-span } p. \text{root-ptr-valid } d \text{ (PTR(stack-byte) } a)$
 $\implies r \text{ s } p = \text{ZERO}('b)$
assumes *subtype-lookup*:
 $\text{field-lookup (typ-uinfo-t TYPE('a)) } f \ 0 = \text{Some (typ-uinfo-t TYPE('b), } n)$
shows $r \text{ s (PTR('b) (\&(q \rightarrow f)))} = \text{ZERO}('b)$
proof –
have $\text{ptr-span (PTR('b) (\&(q \rightarrow f)))} \subseteq \text{ptr-span } q$
using *TypHeapSimple.field-tag-sub'*[*OF subtype-lookup*]
by (*simp, metis size-of-fold*)
thus *?thesis*
using *ptr-span-has-stack-byte-type subtype-lookup subtype-reads-ZERO* **by** *blast*
qed

context *open-types*
begin

lemma *ptr-span-with-stack-byte-type-implies-read-dedicated-heap-ZERO*[*simp*]:
 $\forall a \in \text{ptr-span } p. \text{root-ptr-valid (hrs-htd } s) \text{ (PTR(stack-byte) } a) \implies$
 $\text{read-dedicated-heap } s \ p = \text{ZERO}('a::\{\text{stack-type, xmem-type}\})$
unfolding *read-dedicated-heap-def ptr-span-with-stack-byte-type-implies-ZERO*[*of*
p] *merge-addressable-fields.idem ..*

lemma *write-simulationI*:
fixes $R :: 's \Rightarrow 'a::\text{xmem-type ptr} \Rightarrow 'a$
assumes *fs*: $\text{map-of } \mathcal{T} \text{ (typ-uinfo-t TYPE('a))} = \text{Some } fs$
assumes *heap-typing-simulation* \mathcal{T} *t-hrs t-hrs-update heap-typing heap-typing-update*
 l
and $l\text{-w}$: $\text{list-all2 } (\lambda f \ w. \forall t \ u \ n \ h \ (p::'a \ \text{ptr}) \ x.$
 $\text{field-ti TYPE('a) } f = \text{Some } t \longrightarrow$
 $\text{field-lookup (typ-uinfo-t TYPE('a)) } f \ 0 = \text{Some } (u, n) \longrightarrow$
 $\text{ptr-valid-u } u \text{ (hrs-htd (t-hrs } h)) \ \&(p \rightarrow f) \longrightarrow$
 $l \text{ (t-hrs-update (hrs-mem-update (heap-upd-list (size-td } u) \ \&(p \rightarrow f) \text{ (access-ti$
 $t \ x))) } h)$
 $= w \ p \ x \ (l \ h) \ fs \ ws$
and $l\text{-u}$: $\bigwedge (p::'a \ \text{ptr}) \ (x::'a) \ (s::'b).$
 $\text{ptr-valid (hrs-htd (t-hrs } s)) \ p \implies$
 $l \text{ (t-hrs-update (write-dedicated-heap } p \ x) \ s) = u \text{ (upd-fun } p \ (\lambda \text{old. merge-addressable-fields}$
 $\text{old } x)) \ (l \ s)$
assumes V :
 $\bigwedge h \ p. V \ (l \ h) \ p \longleftrightarrow \text{ptr-valid (hrs-htd (t-hrs } h)) \ p$
assumes W :
 $\bigwedge p \ f \ h. W \ p \ f \ h =$
 $\text{fold } (\lambda w. w \ p \ (f \ (R \ h \ p))) \ ws \ (u \text{ (upd-fun } p \ (\lambda \text{old. merge-addressable-fields}$

```

old (f (R h p))) h)
  shows write-simulation t-hrs t-hrs-update l V W
proof –
interpret hrs: heap-typing-simulation  $\mathcal{T}$  t-hrs t-hrs-update heap-typing heap-typing-update
l
  by fact

have valid:
  list-all ( $\lambda f. \forall u n. \text{field-lookup} (\text{typ-uinfo-t TYPE}('a)) f 0 = \text{Some} (u, n) \longrightarrow$ 
    ptr-valid-u u h  $\&(p \rightarrow f)$ ) fs
  if *: ptr-valid-u (typ-uinfo-t TYPE('a)) h (ptr-val p) for h and p :: 'a ptr
  using ptr-valid-u-step[OF fs - - *]
  by (auto simp: list-all-iff field-lvalue-def field-offset-def)

have fold': l (fold
  ( $\lambda xa. \text{t-hrs-update}$ 
    (hrs-mem-update
      (heap-upd-list (size-td (the (field-ti TYPE('a) xa)))  $\&(p \rightarrow xa)$ 
        (access-ti (the (field-ti TYPE('a) xa)) x))))
    fs s) =
  fold ( $\lambda w. w p x$ ) ws (l s)
  if p: ptr-valid-u (typ-uinfo-t TYPE('a)) (hrs-htd (t-hrs s)) (ptr-val p)
  for p x s
  using l-w wf- $\mathcal{T}$ [OF fs] p[THEN valid]
proof (induction arbitrary: s)
  case (Cons f fs w ws)
  from Cons.prem1 obtain u n where f-u : field-lookup (typ-uinfo-t TYPE('a))
f 0 = Some (u, n)
  and [simp]: list-all ( $\lambda f. \exists a b. \text{field-lookup} (\text{typ-uinfo-t TYPE}('a)) f 0 = \text{Some}$ 
(a, b)) fs
  by auto
  from f-u[THEN field-lookup-uinfo-Some-rev] obtain k where
field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (k, n) and u-eq: u = export-uinfo
k
  by auto
  then have [simp]: field-ti TYPE('a) f = Some k by (simp add: field-ti-def)
  have [simp]: size-td k = size-td u
  by (simp add: u-eq)
  have [simp]: ptr-valid-u u (hrs-htd (t-hrs s))  $\&(p \rightarrow f)$ 
  using Cons.prem2 f-u by auto
  show ?case
  using Cons.prem1 Cons.hyps by (simp add: Cons.IH f-u)
qed simp

have fold:
  l ((fold (t-hrs-update  $\circ$ 
    ( $\lambda(f, u). \text{hrs-mem-update} (\text{heap-upd-list} (\text{size-td } u) \&(p \rightarrow f) (\text{access-ti } u$ 
x))))
    (addressable-fields TYPE('a))  $\circ$ 

```

```

    t-hrs-update (write-dedicated-heap p x) s) =
  fold (λw. w p x) ws (u (upd-fun p (λold. merge-addressable-fields old x)) (l s))
  if p: ptr-valid-u (typ-uinfo-t TYPE('a)) (hrs-htd (t-hrs s)) (ptr-val p)
  for p x s
  by (subst addressable-fields-def)
    (simp add: fs fold-map fold' p ptr-valid-def l-u cong: fold-cong)

```

```

show ?thesis
  apply (rule hrs.write-simulation-alt)
  apply (simp add: V)
  apply (subst hrs.mem-update-heap-update')
  apply (subst W)
  apply (subst (asm) V)
  apply (subst (asm) ptr-valid-def)
  apply (subst hrs.t-hrs.upd-comp[symmetric])
  apply (subst hrs.t-hrs.upd-comp-fold)
  apply (subst fold)
  apply simp-all
done

```

qed

end

```

locale stack-simulation-heap-typing =
  typ-heap-simulation st r w λt p. open-types.ptr-valid  $\mathcal{T}$  (heap-typing t) p t-hrs
  t-hrs-update +
  heap-typing-simulation  $\mathcal{T}$  t-hrs t-hrs-update heap-typing heap-typing-upd st +
  typ-heap-typing r w heap-typing heap-typing-upd  $\mathcal{S}$ 
for
  st:: 's ⇒ 't and
  r:: 't ⇒ ('a::{xmem-type, stack-type}) ptr ⇒ 'a and
  w:: 'a ptr ⇒ ('a ⇒ 'a) ⇒ 't ⇒ 't and
  t-hrs :: 's ⇒ heap-raw-state and
  t-hrs-update:: (heap-raw-state ⇒ heap-raw-state) ⇒ 's ⇒ 's and
  heap-typing :: 't ⇒ heap-typ-desc and
  heap-typing-upd :: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 't ⇒ 't and
   $\mathcal{S}$ :: addr set and
   $\mathcal{T}$ :: (typ-uinfo * qualified-field-name list) list +

```

assumes sim-stack-alloc-heap-typing:

```

  ∧p d s n.
  (p, d) ∈ stack-allocs n  $\mathcal{S}$  TYPE('a) (hrs-htd (t-hrs s)) ⇒
  st (t-hrs-update (hrs-mem-update (fold (λi. heap-update (p +p int i) c-type-class.zero)
  [0..<n]) ○ hrs-htd-update (λ-. d)) s) =
  (heap-typing-upd (λ-. d) (st s))

```

assumes sim-stack-release-heap-typing:

```

  ∧(p::'a ptr) s n. (∧i. i < n ⇒ root-ptr-valid (hrs-htd (t-hrs s)) (p +p int i)) ⇒
  st (t-hrs-update (hrs-htd-update (stack-releases n p)) s) =

```

$\text{heap-typing-upd } (\text{stack-releases } n \ p)$
 $(\text{st } (t\text{-hrs-update } (\text{hrs-mem-update } (\text{fold } (\lambda i. \text{heap-update } (p +_p \text{int } i) \text{c-type-class.zero})$
 $[0..<n])) \ s))$

assumes $\text{sim-stack-stack-byte-zero}[\text{read-stack-byte-ZERO-step}]$:

$\bigwedge p \ s. \forall a \in \text{ptr-span } p. \text{root-ptr-valid } (\text{hrs-htd } (t\text{-hrs } s)) \ (\text{PTR}(\text{stack-byte}) \ a) \implies$
 $r \ (\text{st } s) \ p = \text{ZERO}('a)$

begin

lemma $\text{fold-heap-update-simulation}$:

assumes $\text{valid}: \bigwedge i. i < n \implies \text{ptr-valid } (\text{heap-typing } (\text{st } s)) \ (p +_p \text{int } i)$
shows $\text{st } (t\text{-hrs-update } (\text{hrs-mem-update } (\text{fold } (\lambda i. \text{heap-update } (p +_p \text{int } i) \ (\text{vs}$
 $i)) \ [0..<n])) \ s) =$

$\text{fold } (\lambda i. w \ (p +_p \text{int } i) \ (\lambda-. \text{vs } i)) \ [0..<n] \ (\text{st } s)$

using valid

proof ($\text{induct } n \ \text{arbitrary}: \text{vs } s$)

case 0

then show $?case$

by ($\text{simp add: case-prod-unfold hrs-mem-update-def}$)

next

case ($\text{Suc } n$)

from Suc.premis **obtain**

$\text{valid}: \bigwedge i. i < \text{Suc } n \implies \text{ptr-valid } (\text{heap-typing } (\text{st } s)) \ (p +_p \text{int } i)$ **by** blast

from valid **have** $\text{valid}': \bigwedge i. i < n \implies \text{ptr-valid } (\text{heap-typing } (\text{st } s)) \ (p +_p \text{int } i)$

by auto

note $\text{hyp} = \text{Suc.hyps } [\text{OF } \text{valid}']$

show $?case$

apply ($\text{simp add: hyp } [\text{symmetric}]$)

apply ($\text{subst write-commutes } [\text{symmetric}]$)

using valid

apply simp

using $\text{TypHeapSimple.hrs-mem-update-comp hrs-mem-update-def}$

apply simp

done

qed

lemma $\text{fold-heap-update-padding-simulation}$:

assumes $\text{valid}: \bigwedge i. i < n \implies \text{ptr-valid } (\text{heap-typing } (\text{st } s)) \ (p +_p \text{int } i)$

assumes $\text{lbs: length } \text{bs} = n * \text{size-of } \text{TYPE}('a)$

shows $\text{st } (t\text{-hrs-update } (\text{hrs-mem-update } (\text{fold } (\lambda i. \text{heap-update-padding } (p +_p \text{int } i)$
 $i) \ (\text{vs } i) \ (\text{take } (\text{size-of } \text{TYPE}('a)) \ (\text{drop } (i * \text{size-of } \text{TYPE}('a)) \ \text{bs}))) \ [0..<n])) \ s)$
 $=$

$\text{fold } (\lambda i. w \ (p +_p \text{int } i) \ (\lambda-. \text{vs } i)) \ [0..<n] \ (\text{st } s)$

using valid lbs

proof ($\text{induct } n \ \text{arbitrary}: \text{bs } \text{vs } s$)

case 0

then show $?case$

```

    by (simp add: case-prod-unfold hrs-mem-update-def)
next
case (Suc n)
from Suc.prem1 obtain
  valid:  $\bigwedge i. i < Suc\ n \implies ptr\ valid\ (heap\ typing\ (st\ s))\ (p +_p\ int\ i)$  and
  lbs':  $length\ (take\ (n * (size\ of\ TYPE('a)))\ bs) = n * size\ of\ TYPE('a)$ 
by auto

from valid have valid':  $\bigwedge i. i < n \implies ptr\ valid\ (heap\ typing\ (st\ s))\ (p +_p\ int\ i)$ 
by auto
note hyp = Suc.hyps [OF valid' lbs']
have take-eq:  $\bigwedge i. i < n \implies take\ (size\ of\ TYPE('a))\ (drop\ (i * size\ of\ TYPE('a))\ (take\ (n * size\ of\ TYPE('a))\ bs)) =$ 
   $take\ (size\ of\ TYPE('a))\ (drop\ (i * size\ of\ TYPE('a))\ bs)$ 
by (metis Groups.mult-ac(2) mult-less-cancel1 not-less not-less-eq
  order-less-imp-le take-all take-drop-take times-nat.simps(2))

have fold-eq:  $\bigwedge h. fold$ 
   $(\lambda i. heap\ update\ padding\ (p +_p\ int\ i)\ (vs\ i)$ 
   $(take\ (size\ of\ TYPE('a))\ (drop\ (i * size\ of\ TYPE('a))\ (take\ (n$ 
   $*\ size\ of\ TYPE('a))\ bs))))$ 
   $[0..<n]\ h =$ 
  fold
   $(\lambda i. heap\ update\ padding\ (p +_p\ int\ i)\ (vs\ i)$ 
   $(take\ (size\ of\ TYPE('a))\ (drop\ (i * size\ of\ TYPE('a))\ bs)))$ 
   $[0..<n]\ h$ 

apply (rule fold-cong)
  apply (rule refl)
  apply (rule refl)
using take-eq
apply simp
done

show ?case
  apply (simp add: hyp [symmetric])
  apply (subst write-padding-commutes [symmetric, where bs = take (size-of
  TYPE('a)) (drop (n * size-of TYPE('a)) bs)])
  subgoal using valid
    by simp
  subgoal using Suc.prem1 by simp
  subgoal
    using TypHeapSimple.hrs-mem-update-comp hrs-mem-update-def
    by (simp add: fold-eq)
  done
qed

lemma sim-stack-alloc':

```



```

assumes alloc: (p, d) ∈ stack-allocs n S TYPE('a) (hrs-htd (t-hrs s))
assumes len: length vs = n
assumes lbs: length bs = n * size-of TYPE('a)
shows st (t-hrs-update (hrs-mem-update (fold (λi. heap-update-padding (p +p
int i) (vs!i) (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs))) [0..p int i) (λ-. (vs ! i))) [0..proof -
  {
    fix i
    assume i-bound: i < n
    have ptr-valid (heap-typing (st (t-hrs-update
      (hrs-mem-update (fold (λi. heap-update (p +p int i) c-type-class.zero)
[0..p int i)
    proof -
      from stack-allocs-cases [OF alloc] i-bound
      have ptr-valid: ptr-valid d (p +p int i)
      using root-ptr-valid-ptr-valid by auto
      thus ?thesis
      using heap-typing-upd-commutes by (simp, metis)
    qed
  } note valids = this

from stack-allocs-cases [OF alloc] obtain
  bound: unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card and
  not-null: ptr-val p ≠ 0
by (metis Ptr-ptr-val c-guard-NULL-simp)

show ?thesis
apply (simp add: sim-stack-alloc-heap-typing [OF alloc, symmetric])
apply (subst fold-heap-update-padding-simulation [OF valids lbs, symmetric])
  apply (simp)
  apply (simp add: len)
  apply (simp add: comp-def hrs-mem-update-comp')
apply (subst fold-heap-update-padding-heap-update-collapse [OF bound not-null])
using lbs
  apply (auto simp add: less-le-mult-nat nat-move-sub-le)
done
qed

lemma sim-stack-release':
fixes p :: 'a ptr
assumes roots: ∧i. i < n ⇒ root-ptr-valid (hrs-htd (t-hrs s)) (p +p int i)
shows st (t-hrs-update (hrs-htd-update (stack-releases n p)) s) =
      (heap-typing-upd (stack-releases n p) ((fold (λi. w (p +p int i) (λ-.

```

```

c-type-class.zero)) [0.. $n$ ]) (st s)))
proof -
  from roots root-ptr-valid-ptr-valid heap-typing-commutes
  have valids:  $\bigwedge i . i < n \implies$  ptr-valid (heap-typing (st s)) (p +p int i)
  by metis
  note commutes = fold-heap-update-simulation [OF valids, symmetric, of n, sim-
  plified]
  show ?thesis
  apply (simp add: commutes )
  apply (simp add: sim-stack-release-heap-typing [OF roots])
  done
qed

```

```

lemma sim-stack-release'':
  fixes p :: 'a ptr
  assumes roots:  $\bigwedge i . i < n \implies$  root-ptr-valid (hrs-htd (t-hrs s)) (p +p int i)
  assumes lbs: length bs = n * size-of TYPE('a)
  shows st (t-hrs-update (hrs-mem-update (heap-update-list (ptr-val p) bs) o hrs-htd-update
  (stack-releases n p)) s) =
    ((heap-typing-upd (stack-releases n p) ((fold ( $\lambda i . w$  (p +p int i) ( $\lambda .$ 
  c-type-class.zero)) [0.. $n$ ]) (st s))))
proof -
  {
    fix i
    assume bound-i: i < length bs
    have root-ptr-valid (hrs-htd (t-hrs (t-hrs-update (hrs-htd-update (stack-releases
  n p)) s)))
      ((PTR-COERCE('a  $\rightarrow$  stack-byte) p) +p int i)
    proof -
      have span: ptr-val (((PTR-COERCE('a  $\rightarrow$  stack-byte) p) +p int i))  $\in$  {ptr-val
  p.. $+n$  * size-of TYPE('a)}
      using lbs bound-i intvlI by (auto simp add: ptr-add-def)
      from roots have  $\forall i < n .$  c-guard (p +p int i)
      using root-ptr-valid-c-guard by blast
      from stack-releases-root-ptr-valid-footprint [OF span this]
      show ?thesis
      using typing.get-upd by force
    qed
  } note sb = this

  show ?thesis
  apply (simp add: lift-heap-update-list-stack-byte-independent [OF sb, simpli-
  fied])
  apply (simp add: sim-stack-release' [OF roots])
  done
qed

```

lemma *stack-byte-zero'*:
assumes $(p, d) \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a) \ (\text{hrs-htd } (t\text{-hrs } s))$
assumes $i < n$
shows $r \ (st \ s) \ (p +_p \ \text{int } i) = \text{ZERO}('a)$
by $(\text{rule } \text{sim-stack-stack-byte-zero})$
 $(\text{meson } \text{assms } \text{stack-allocs-cases } \text{stack-allocs-contained } \text{subsetD})$

sublocale *stack-simulation*
using *sim-stack-alloc' sim-stack-release'' stack-byte-zero'*
by $(\text{unfold-locales}) \ \text{auto}$

sublocale *typ-heap-typing-stack-simulation* $\mathcal{T} \ st \ r \ w \ \lambda t \ p. \ \text{open-types.ptr-valid } \mathcal{T}$
 $(\text{heap-typing } t) \ p \ t\text{-hrs } t\text{-hrs-update } \text{heap-typing } \text{heap-typing-upd } \mathcal{S}$
by (unfold-locales)

end

definition

valid-struct-field
 $:: \text{string list}$
 $\Rightarrow ((p::xmem\text{-type}) \Rightarrow (f::xmem\text{-type}))$
 $\Rightarrow ((f \Rightarrow 'f) \Rightarrow (p \Rightarrow 'p))$
 $\Rightarrow ('s \Rightarrow \text{heap-raw-state})$
 $\Rightarrow ((\text{heap-raw-state} \Rightarrow \text{heap-raw-state}) \Rightarrow 's \Rightarrow 's)$
 $\Rightarrow \text{bool}$

where

valid-struct-field field-name field-getter field-setter t-hrs t-hrs-update \equiv
 $(\text{lense } \text{field-getter } \text{field-setter})$
 $\wedge \text{field-ti } \text{TYPE}('p) \ \text{field-name} =$
 $\text{Some } (\text{adjust-ti } (\text{typ-info-t } \text{TYPE}('f)) \ \text{field-getter } (\text{field-setter} \circ (\lambda x \ . \ x)))$
 $\wedge (\forall p :: 'p \ \text{ptr}. \ \text{c-guard } p \longrightarrow \text{c-guard } (\text{Ptr } \&(p \rightarrow \text{field-name}) :: 'f \ \text{ptr}))$
 $\wedge \text{lense } t\text{-hrs } t\text{-hrs-update}$

lemma *typ-heap-simulation-get-hvalD*:

$\llbracket \text{typ-heap-simulation } st \ r \ w \ v$
 $t\text{-hrs } t\text{-hrs-update}; \ v \ (st \ s) \ p \rrbracket \Longrightarrow$
 $h\text{-val } (\text{hrs-mem } (t\text{-hrs } s)) \ p = r \ (st \ s) \ p$
by $(\text{clarsimp } \text{simp: } \text{read-simulation.read-commutes } [\text{OF } \text{typ-heap-simulation.axioms}(2)])$

lemma *valid-struct-fieldI* [*intro*]:

fixes *field-getter* $:: ('a::xmem\text{-type}) \Rightarrow ('f::xmem\text{-type})$

shows \llbracket

$$\begin{aligned}
& \bigwedge s f. f \text{ (field-getter } s) = \text{(field-getter } s) \implies \text{field-setter } f s = s; \\
& \bigwedge s f f'. f \text{ (field-getter } s) = f' \text{ (field-getter } s) \implies \text{field-setter } f s = \text{field-setter} \\
& f' s; \\
& \bigwedge s f. \text{field-getter (field-setter } f s) = f \text{ (field-getter } s); \\
& \bigwedge s f g. \text{field-setter } f \text{ (field-setter } g s) = \text{field-setter } (f \circ g) s; \\
& \text{field-ti TYPE('a) field-name =} \\
& \quad \text{Some (adjust-ti (typ-info-t TYPE('f)) field-getter (field-setter } \circ (\lambda x \cdot x)); \\
& \bigwedge (p::'a \text{ ptr}). c\text{-guard } p \implies c\text{-guard (Ptr } \&(p \rightarrow \text{field-name}) :: 'f \text{ ptr}); \\
& \bigwedge s x. t\text{-hrs (t-hrs-update } x s) = x \text{ (t-hrs } s); \\
& \bigwedge s f. f \text{ (t-hrs } s) = t\text{-hrs } s \implies t\text{-hrs-update } f s = s; \\
& \bigwedge s f f'. f \text{ (t-hrs } s) = f' \text{ (t-hrs } s) \implies t\text{-hrs-update } f s = t\text{-hrs-update } f' s; \\
& \bigwedge s f g. t\text{-hrs-update } f \text{ (t-hrs-update } g s) = t\text{-hrs-update } (\lambda x. f (g x)) s \\
& \rrbracket \implies \\
& \text{valid-struct-field field-name field-getter field-setter t-hrs t-hrs-update} \\
apply (unfold valid-struct-field-def lense-def o-def) \\
apply (safe | assumption | rule ext)+ \\
done$$

lemma *typ-heap-simulation-t-hrs-updateD*:

$$\llbracket \text{typ-heap-simulation } st \ r \ w \ v \\ t\text{-hrs } t\text{-hrs-update}; v \text{ (st } s) \ p \rrbracket \implies \\ st \text{ (t-hrs-update (hrs-mem-update (heap-update } p \ v')) \ s) =} \\ w \ p \ (\lambda x. \ v') \text{ (st } s)$$

by (clarsimp simp: write-simulation.write-commutes [OF typ-heap-simulation.axioms(3)])

lemma *heap-abs-expr-guard [heap-abs]*:

$$\llbracket \text{typ-heap-simulation } st \ \text{getter } \text{setter } v\text{getter } t\text{-hrs } t\text{-hrs-update}; \\ \text{abs-expr } st \ P \ x' \ x \rrbracket \implies \\ \text{abs-guard } st \ (\lambda s. \ P \ s \wedge v\text{getter } s \ (x' \ s)) \ (\lambda s. \ (c\text{-guard } (x \ s :: ('a::xmem-type) \\ ptr)))$$

apply (clarsimp simp: abs-expr-def abs-guard-def
simple-lift-def root-ptr-valid-def
valid-implies-cguard.valid-implies-c-guard [OF typ-heap-simulation.axioms(5)])

done

lemma *heap-abs-expr-h-val [heap-abs]*:

$$\llbracket \text{typ-heap-simulation } st \ r \ w \ v \ t\text{-hrs } t\text{-hrs-update}; \\ \text{abs-expr } st \ P \ x' \ x \rrbracket \implies \\ \text{abs-expr } st \\ (\lambda s. \ P \ s \wedge v \ s \ (x' \ s)) \\ (\lambda s. \ (r \ s \ (x' \ s))) \\ (\lambda s. \ (h\text{-val (hrs-mem (t-hrs } s))) \ (x \ s))$$

apply (clarsimp simp: abs-expr-def simple-lift-def)

apply (metis typ-heap-simulation-get-hvalD)

done

lemma *heap-abs-modifies-heap-update--unused*:

$$\llbracket \text{typ-heap-simulation } st \ r \ w \ v \ t\text{-hrs } t\text{-hrs-update};$$

```

abs-expr st Pb b' b;
abs-expr st Pc c' c ] ==>
abs-modifies st (λs. Pb s ∧ Pc s ∧ v s (b' s))
  (λs. w (b' s) (λx. (c' s)) s)
  (λs. t-hrs-update (hrs-mem-update (heap-update (b s :: ('a::xmem-type) ptr)
(c s))) s)
apply (clarsimp simp: typ-simple-heap-simps abs-expr-def abs-modifies-def)
apply (metis typ-heap-simulation-t-hrs-updateD)
done

```

definition *heap-lift--h-val* ≡ *h-val*

```

lemma heap-abs-modifies-heap-update [heap-abs]:
  [ typ-heap-simulation st r w v t-hrs t-hrs-update;
  abs-expr st Pb b' b;
  ∧v. abs-expr st Pc (c' v) (c v) ] ==>
  abs-modifies st (λs. Pb s ∧ Pc s ∧ v s (b' s))
    (λs. w (b' s) (λ-. (c' (r s (b' s)) s)) s)
    (λs. t-hrs-update (hrs-mem-update
      (heap-update (b s :: ('a::xmem-type) ptr)
        (c (heap-lift--h-val (hrs-mem (t-hrs s)) (b s)) s))) s)
apply (clarsimp simp: typ-simple-heap-simps abs-expr-def abs-modifies-def heap-lift--h-val-def)
subgoal for s
  apply (rule subst[where t = h-val (hrs-mem (t-hrs s)) (b' (st s))
    and s = r (st s) (b' (st s))])
  apply (clarsimp simp: read-simulation.read-commutes [OF typ-heap-simulation.axioms(2)])
  apply (simp add: write-simulation.write-commutes [OF typ-heap-simulation.axioms(3)])
  done
done

```

```

lemma struct-rewrite-guard-expr [heap-abs]:
  struct-rewrite-expr P a' a ==> struct-rewrite-guard (λs. P s ∧ a' s) a
by (simp add: struct-rewrite-expr-def struct-rewrite-guard-def)

```

```

lemma struct-rewrite-guard-constant [heap-abs]:
  struct-rewrite-guard (λ-. P) (λ-. P)
by (simp add: struct-rewrite-guard-def)

```

```

lemma struct-rewrite-guard-conj [heap-abs]:
  [ struct-rewrite-guard b' b; struct-rewrite-guard a' a ] ==>
  struct-rewrite-guard (λs. a' s ∧ b' s) (λs. a s ∧ b s)

```

by (clarsimp simp: struct-rewrite-guard-def)

lemma *struct-rewrite-guard-split* [heap-abs]:

[[$\wedge a b$. *struct-rewrite-guard* ($A a b$) ($C a b$)]]
⇒ *struct-rewrite-guard* (*case r of* (a, b) ⇒ $A a b$) (*case r of* (a, b) ⇒ $C a$
 b)

apply (auto simp: split-def)

done

lemma *struct-rewrite-guard-c-guard-field* [heap-abs]:

[[*valid-struct-field* *field-name* (*field-getter* :: ('a :: *xmem-type*) ⇒ ('f :: *xmem-type*))
field-setter *t-hrs* *t-hrs-update*;

struct-rewrite-expr $P p' p$;

struct-rewrite-guard $Q (\lambda s. c\text{-guard } (p' s))$] ⇒

struct-rewrite-guard ($\lambda s. P s \wedge Q s$)

($\lambda s. c\text{-guard } (Ptr (field-lvalue (p s :: 'a ptr) field-name) :: 'f ptr)$)

by (simp add: *valid-struct-field-def* *struct-rewrite-expr-def* *struct-rewrite-guard-def*)

lemma *align-of-array*: *align-of* $TYPE (('a :: array\text{-outer-max-size}) ['b' :: array\text{-max-count}])$
= *align-of* $TYPE ('a)$

by (simp add: *align-of-def* *align-td-array*)

lemma *c-guard-array*:

[[$0 \leq k$; $\text{nat } k < \text{CARD}('b)$; *c-guard* ($p :: (('a :: array\text{-outer-max-size}) ['b' :: array\text{-max-count}])$
ptr)]]

⇒ *c-guard* (*ptr-coerce* $p +_p k :: 'a ptr$)

apply (clarsimp simp: *CTypesDefs.ptr-add-def* *c-guard-def* *c-null-guard-def*)

apply (rule *conjI*[rotated])

apply (erule *contrapos-nn*)

apply (clarsimp simp: *intvl-def*)

subgoal for i

apply (rule *exI*[**where** $x = \text{nat } k * \text{size-of } TYPE ('a) + i$])

apply clarsimp

apply (rule *conjI*)

apply (simp add: *field-simps*)

apply (rule *less-le-trans*[**where** $y = \text{Suc } (\text{nat } k) * \text{size-of } TYPE ('a)$])

apply simp

apply (*metis* *less-eq-Suc-le* *mult-le-mono2* *mult.commute*)

done

apply (subgoal-tac *ptr-aligned* (*ptr-coerce* $p :: 'a ptr$))

apply (*frule-tac* $p = \text{ptr-coerce } p$ **and** $i = k$ **in** *ptr-aligned-plus*)

apply (clarsimp simp: *CTypesDefs.ptr-add-def*)

apply (clarsimp simp: *ptr-aligned-def* *align-of-array*)

done

lemma *struct-rewrite-guard-c-guard-Array-field* [heap-abs]:

[[*valid-struct-field* *field-name* (*field-getter* :: ('a :: *xmem-type*) ⇒ ('f :: *array-outer-max-size*
[$n :: array\text{-max-count}$])) *field-setter* *t-hrs* *t-hrs-update*;

$struct\text{-}rewrite\text{-}expr\ P\ p'\ p;$
 $struct\text{-}rewrite\text{-}guard\ Q\ (\lambda s. c\text{-}guard\ (p'\ s))\]\ \Longrightarrow$
 $struct\text{-}rewrite\text{-}guard\ (\lambda s. P\ s \wedge Q\ s \wedge 0 \leq k \wedge nat\ k < CARD('n))$
 $(\lambda s. c\text{-}guard\ (ptr\text{-}coerce\ (Ptr\ (field\text{-}lvalue\ (p\ s :: 'a\ ptr)\ field\text{-}name) :: ((f['n])$
 $ptr))\ +_p\ k :: 'f\ ptr))$
by (*simp del: ptr-coerce.simps add: valid-struct-field-def struct-rewrite-expr-def*
struct-rewrite-guard-def c-guard-array)

lemma *struct-rewrite-expr-id:*
 $struct\text{-}rewrite\text{-}expr\ (\lambda -. True)\ A\ A$
by (*simp add: struct-rewrite-expr-def*)

lemma *struct-rewrite-expr-fun-app2 [heap-abs-fo]:*
 $[[struct\text{-}rewrite\text{-}expr\ P\ f\ f';$
 $struct\text{-}rewrite\text{-}expr\ Q\ g\ g']]\ \Longrightarrow$
 $struct\text{-}rewrite\text{-}expr\ (\lambda s. P\ s \wedge Q\ s)\ (\lambda s\ a. f\ s\ a\ (g\ s\ a))\ (\lambda s\ a. f'\ s\ a\ \$\ g'\ s\ a)$
by (*simp add: struct-rewrite-expr-def*)

lemma *struct-rewrite-expr-fun-app [heap-abs-fo]:*
 $[[struct\text{-}rewrite\text{-}expr\ Y\ x\ x'; struct\text{-}rewrite\text{-}expr\ X\ f\ f']]\ \Longrightarrow$
 $struct\text{-}rewrite\text{-}expr\ (\lambda s. X\ s \wedge Y\ s)\ (\lambda s. f\ s\ (x\ s))\ (\lambda s. f'\ s\ \$\ x'\ s)$
by (*clarsimp simp: struct-rewrite-expr-def*)

lemma *struct-rewrite-expr-constant [heap-abs]:*
 $struct\text{-}rewrite\text{-}expr\ (\lambda -. True)\ (\lambda -. a)\ (\lambda -. a)$
by (*clarsimp simp: struct-rewrite-expr-def*)

lemma *struct-rewrite-expr-lambda-null [heap-abs]:*
 $struct\text{-}rewrite\text{-}expr\ P\ A\ C\ \Longrightarrow\ struct\text{-}rewrite\text{-}expr\ P\ (\lambda s -. A\ s)\ (\lambda s -. C\ s)$
by (*clarsimp simp: struct-rewrite-expr-def*)

lemma *struct-rewrite-expr-split [heap-abs]:*
 $[[\bigwedge a\ b. struct\text{-}rewrite\text{-}expr\ (P\ a\ b)\ (A\ a\ b)\ (C\ a\ b)]]$
 $\Longrightarrow\ struct\text{-}rewrite\text{-}expr\ (case\ r\ of\ (a, b) \Rightarrow P\ a\ b)$
 $(case\ r\ of\ (a, b) \Rightarrow A\ a\ b)\ (case\ r\ of\ (a, b) \Rightarrow C\ a\ b)$
apply (*auto simp: split-def*)
done

lemma *struct-rewrite-expr-basecase-h-val [heap-abs]:*
 $struct\text{-}rewrite\text{-}expr\ (\lambda -. True)\ (\lambda s. h\text{-}val\ (h\ s)\ (p\ s))\ (\lambda s. h\text{-}val\ (h\ s)\ (p\ s))$
by (*simp add: struct-rewrite-expr-def*)

lemma *struct-rewrite-expr-indirect-h-val [heap-abs]:*
 $struct\text{-}rewrite\text{-}expr\ P\ a\ c\ \Longrightarrow$

struct-rewrite-expr P ($\lambda s. h\text{-val } (h\ s) (a\ s)$) ($\lambda s. h\text{-val } (h\ s) (c\ s)$)
by (*simp add: struct-rewrite-expr-def*)

lemma *struct-rewrite-expr-field* [*heap-abs*]:

$\llbracket \text{valid-struct-field } \text{field-name } (\text{field-getter} :: ('a :: \text{xmem-type}) \Rightarrow ('f :: \text{xmem-type}))$
field-setter $t\text{-hrs } t\text{-hrs-update};$
struct-rewrite-expr $P\ p'\ p;$
struct-rewrite-expr $Q\ a\ (\lambda s. h\text{-val } (\text{hrs-mem } (t\text{-hrs } s)) (p'\ s)) \rrbracket$
 $\implies \text{struct-rewrite-expr } (\lambda s. P\ s \wedge Q\ s) (\lambda s. \text{field-getter } (a\ s))$
 $(\lambda s. h\text{-val } (\text{hrs-mem } (t\text{-hrs } s)) (\text{Ptr } (\text{field-lvalue } (p\ s) \text{field-name})))$
apply (*clarsimp simp: valid-struct-field-def struct-rewrite-expr-def*)
apply (*subst h-val-field-from-bytes'*)
apply *assumption*
apply (*rule export-tag-adjust-ti(1)[rule-format]*)
apply (*simp add: lense.fg-cons*)
apply *simp*
apply *simp*
done

lemma *abs-expr-field* [*heap-abs*]:

$\llbracket \text{valid-struct-field } \text{field-name } (\text{field-getter} :: ('a :: \text{xmem-type}) \Rightarrow ('f :: \text{xmem-type}))$
field-setter $t\text{-hrs } t\text{-hrs-update};$
abs-expr $st\ P\ a\ c \rrbracket$
 $\implies \text{abs-expr } st\ P\ (\lambda s. \text{field-getter } (a\ s)) (\lambda s. \text{field-getter } (c\ s))$
by (*clarsimp simp add: valid-struct-field-def abs-expr-def*)

lemma *struct-rewrite-expr-Array-field* [*heap-abs*]:

$\llbracket \text{valid-struct-field } \text{field-name}$
 $(\text{field-getter} :: ('a :: \text{xmem-type}) \Rightarrow 'f::\text{array-outer-max-size}$
 $['n::\text{array-max-count}])$
field-setter $t\text{-hrs } t\text{-hrs-update};$
struct-rewrite-expr $P\ p'\ p;$
struct-rewrite-expr $Q\ a\ (\lambda s. h\text{-val } (\text{hrs-mem } (t\text{-hrs } s)) (p'\ s)) \rrbracket$
 $\implies \text{struct-rewrite-expr } (\lambda s. P\ s \wedge Q\ s \wedge k \geq 0 \wedge \text{nat } k < \text{CARD}('n))$
 $(\lambda s. \text{index } (\text{field-getter } (a\ s)) (\text{nat } k))$
 $(\lambda s. h\text{-val } (\text{hrs-mem } (t\text{-hrs } s))$
 $(\text{ptr-coerce } (\text{Ptr } (\text{field-lvalue } (p\ s) \text{field-name}) :: ('f['n]) \text{ptr}) +_p k))$
apply (*cases k*)
apply (*clarsimp simp: struct-rewrite-expr-def simp del: ptr-coerce.simps*)
subgoal for $n\ s$
apply (*subst struct-rewrite-expr-field*
 $[\text{unfolded } \text{struct-rewrite-expr-def}, \text{simplified}, \text{rule-format}, \text{symmetric},$
 $\text{where } \text{field-getter} = \text{field-getter} \text{ and } P = P \text{ and } Q = Q \text{ and } p = p \text{ and}$
 $p' = p']$)
apply *assumption*
apply *simp*
apply *simp*
apply *simp*


```

apply (rule subst[where  $s = p\ s$  and  $t = p'\ s$ ])
apply simp
apply (rule heap-access-Array-element[symmetric])
apply simp
done
apply (simp add: struct-rewrite-expr-def)
done
declare struct-rewrite-expr-Array-field [unfolded ptr-coerce.simps, heap-abs]

```

```

lemma struct-rewrite-modifies-id [heap-abs]:
  struct-rewrite-modifies ( $\lambda$ -. True) A A
by (simp add: struct-rewrite-modifies-def)

```

```

lemma struct-rewrite-modifies-basecase [heap-abs]:
   $\llbracket$  typ-heap-simulation st (getter :: 's  $\Rightarrow$  'a ptr  $\Rightarrow$  ('a::xmem-type)) setter vgetter
  t-hrs t-hrs-update;
  struct-rewrite-expr P p' p;
  struct-rewrite-expr Q v' v  $\rrbracket \Longrightarrow$ 
  struct-rewrite-modifies ( $\lambda$ s. P s  $\wedge$  Q s)
  ( $\lambda$ s. t-hrs-update (hrs-mem-update (heap-update (p' s) (v' s :: 'a))) s)
  ( $\lambda$ s. t-hrs-update (hrs-mem-update (heap-update (p s) (v s :: 'a))) s)
by (simp add: struct-rewrite-expr-def struct-rewrite-modifies-def)

```

```

lemma heap-update-field-unpacked:
   $\llbracket$  field-ti TYPE('a::mem-type) f = Some (t :: 'a xtyp-info);
  c-guard (p :: 'a::mem-type ptr);
  export-uinfo t = export-uinfo (typ-info-t TYPE('b::mem-type))  $\rrbracket \Longrightarrow$ 
  heap-update (Ptr  $\&$ (p $\rightarrow$ f) :: 'b ptr) v hp =
  heap-update p (update-ti t (to-bytes-p v) (h-val hp p)) hp
oops

```

```

lemma heap-update-Array-element-unpacked:
   $n < \text{CARD}('b::\text{array-max-count}) \Longrightarrow$ 
  heap-update (ptr-coerce p' +p int n) w hp =
  heap-update (p'::('a::array-outer-max-size['b::array-max-count]) ptr)
  (Arrays.update (h-val hp p') n w) hp
oops

```

```

lemma read-write-valid-hrs-mem:
  lense hrs-mem hrs-mem-update
by (clarsimp simp: hrs-mem-def hrs-mem-update-def lense-def)

```

definition *heap-lift--wrap-h-val* $\equiv (=)$

lemma *heap-lift-wrap-h-val* [*heap-abs*]:

heap-lift--wrap-h-val (*heap-lift--h-val* *s p*) (*h-val* *s p*)

by (*simp add: heap-lift--h-val-def heap-lift--wrap-h-val-def*)

lemma *heap-lift-wrap-h-val-skip* [*heap-abs*]:

heap-lift--wrap-h-val (*h-val* *s* (*Ptr* (*field-lvalue* *p f*))) (*h-val* *s* (*Ptr* (*field-lvalue* *p f*)))

by (*simp add: heap-lift--wrap-h-val-def*)

lemma *heap-lift-wrap-h-val-skip-array* [*heap-abs*]:

heap-lift--wrap-h-val (*h-val* *s* (*ptr-coerce* *p +_p k*))

(*h-val* *s* (*ptr-coerce* *p +_p k*))

by (*simp add: heap-lift--wrap-h-val-def*)

lemma *struct-rewrite-modifies-field--unused*:

\llbracket *valid-struct-field* *field-name* (*field-getter* $:: ('a::xmem-type) \Rightarrow ('f::xmem-type)$)

field-setter *t-hrs* *t-hrs-update*;

struct-rewrite-expr *P p' p*;

struct-rewrite-expr *Q f' f*;

struct-rewrite-modifies *R*

($\lambda s.$ *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p'' s*))

(*u s* (*field-setter* ($\lambda-. f' s$)))))) *s*

($\lambda s.$ *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p' s*))

(*field-setter* ($\lambda-. f' s$) (*h-val* (*hrs-mem* (*t-hrs s*)) (*p' s*)))))) *s*;

struct-rewrite-guard *S* ($\lambda s.$ *c-guard* (*p' s*)) $\llbracket \implies$

struct-rewrite-modifies ($\lambda s.$ *P s* \wedge *Q s* \wedge *R s* \wedge *S s*)

($\lambda s.$ *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p'' s*))

(*u s* (*field-setter* ($\lambda-. f' s$)))))) *s*

($\lambda s.$ *t-hrs-update* (*hrs-mem-update* (*heap-update* (*Ptr* (*field-lvalue* (*p s*) *field-name*))

(*f s*))) *s*

apply (*clarsimp simp: struct-rewrite-expr-def struct-rewrite-guard-def struct-rewrite-modifies-def valid-struct-field-def*)

apply (*erule-tac* *x = s in alle*)**+**

apply (*erule impE, assumption*)**+**

apply (*erule-tac* *t = t-hrs-update* (*hrs-mem-update* (*heap-update* (*p'' s*))

(*u s* (*field-setter* ($\lambda-. f' s$)))))) *s*

and *s = t-hrs-update* (*hrs-mem-update* (*heap-update* (*p' s*))

(*field-setter* ($\lambda-. f' s$) (*h-val* (*hrs-mem* (*t-hrs s*)) (*p' s*)))))) *s*

in subst)

apply (*rule lense.upd-cong*[**where** *get = t-hrs* **and** *upd = t-hrs-update*])

apply *assumption*

apply (*rule lense.upd-cong*[*OF read-write-valid-hrs-mem*])

apply (*subst heap-update-field-root-conv''')*)

```

apply assumption+
apply (simp add: typ-uinfo-t-def lense.fg-cons)
apply (subst update-ti-update-ti-t)
apply (simp add: size-of-def)
apply (subst update-ti-s-adjust-ti-to-bytes-p)
apply (erule lense.fg-cons)
apply simp
done

```

lemma *struct-rewrite-modifies-Array-field--unused:*

```

[[ valid-struct-field field-name (field-getter :: ('a::xmem-type) => (('f::array-outer-max-size)['n::array-max-count
field-setter t-hrs t-hrs-update;

```

```

struct-rewrite-expr P p' p;
struct-rewrite-expr Q f' f;
struct-rewrite-modifies R
  ( $\lambda s. t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (p'' s)
    (u s (field\text{-setter } (\lambda a. Arrays.update a (nat k) (f' s)))))) s$ )
  ( $\lambda s. t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (p' s)
    (field\text{-setter } (\lambda a. Arrays.update a (nat k) (f' s))
    (h\text{-val } (hrs\text{-mem } (t\text{-hrs } s)) (p' s)))))) s$ );

```

```

struct-rewrite-guard S ( $\lambda s. c\text{-guard } (p' s)$ ) ] ==>

```

```

struct-rewrite-modifies ( $\lambda s. P s \wedge Q s \wedge R s \wedge S s \wedge 0 \leq k \wedge nat k < CARD('n)$ )

```

```

  ( $\lambda s. t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (p'' s)
    (u s (field\text{-setter } (\lambda a. Arrays.update a (nat k) (f' s)))))) s$ )

```

```

  ( $\lambda s. t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update }
    (ptr\text{-coerce } (Ptr (field\text{-lvalue } (p s) field\text{-name}) :: ('f['n] ptr) +_p k) (f s))$ )

```

s)

```

using ptr-coerce.simps [simp del]

```

```

apply (clarsimp simp: struct-rewrite-expr-def struct-rewrite-guard-def struct-rewrite-modifies-def
valid-struct-field-def)

```

subgoal for *s*

```

apply (erule-tac x = s in allE)+

```

```

apply (erule impE, assumption)+

```

```

apply (erule-tac t = t-hrs-update ( $\lambda s. hrs\text{-mem-update } (heap\text{-update } (p'' s)
  (u s (field\text{-setter } (\lambda a. Arrays.update a (nat k) (f' s)))))) s$ )

```

```

  and s = t-hrs-update ( $\lambda s. hrs\text{-mem-update } (heap\text{-update } (p' s)
    (field\text{-setter } (\lambda a. Arrays.update a (nat k) (f' s))
    (h\text{-val } (hrs\text{-mem } (t\text{-hrs } s)) (p' s)))))) s$ )

```

```

  in subst

```

```

apply (rule lense.upd-cong[where get = t-hrs and upd = t-hrs-update])

```

```

apply assumption

```

```

apply (rule lense.upd-cong[OF read-write-valid-hrs-mem])

```

```

apply (cases k, clarsimp)

```

```

apply (subst heap-update-array-element[symmetric])

```

```

  apply assumption

```

```

apply simp

```

```

apply (subst heap-update-field-root-conv''')

```

```

  apply assumption+

```

```

apply (simp add: typ-uinfo-t-def lense.fg-cons)

```

```

apply (subst h-val-field-from-bytes^)
  apply assumption+
  apply (simp add: lense.fg-cons)
apply clarsimp
apply (subst update-ti-update-ti-t)
  apply (simp add: size-of-def)
apply (subst update-ti-s-adjust-ti-to-bytes-p)
  apply (erule lense.fg-cons)
apply clarsimp
apply (subst lense.upd-cong[of field-getter field-setter])
  apply auto
done
done

```

lemma *struct-rewrite-modifies-field* [heap-abs]:

```

 $\llbracket$  valid-struct-field field-name (field-getter :: ('a::xmem-type)  $\Rightarrow$  ('f::xmem-type))
field-setter t-hrs t-hrs-update;
struct-rewrite-expr P p' p;
struct-rewrite-expr Q f' f;
 $\bigwedge$ s. heap-lift--wrap-h-val (h-val-p' s) (h-val (hrs-mem (t-hrs s)) (p' s));
struct-rewrite-modifies R
  ( $\lambda$ s. t-hrs-update (hrs-mem-update (heap-update (p'' s)
    (u s (field-setter (f' s)))))) s)
  ( $\lambda$ s. t-hrs-update (hrs-mem-update (heap-update (p' s)
    (field-setter (f' s) (h-val-p' s)))) s);
struct-rewrite-guard S ( $\lambda$ s. c-guard (p' s))  $\llbracket \implies$ 
struct-rewrite-modifies ( $\lambda$ s. P s  $\wedge$  Q s  $\wedge$  R s  $\wedge$  S s)
  ( $\lambda$ s. t-hrs-update (hrs-mem-update (heap-update (p'' s)
    (u s (field-setter (f' s)))))) s)
  ( $\lambda$ s. t-hrs-update (hrs-mem-update (heap-update (Ptr (field-lvalue (p s) field-name))
    (f s (h-val (hrs-mem (t-hrs s)) (Ptr (field-lvalue (p s) field-name))))))
s)
apply (clarsimp simp: struct-rewrite-expr-def struct-rewrite-guard-def struct-rewrite-modifies-def
valid-struct-field-def heap-lift--wrap-h-val-def)
apply (erule-tac x = s in alle)+
apply (erule impE, assumption)+
apply (erule-tac t = t-hrs-update (hrs-mem-update (heap-update (p'' s)
  (u s (field-setter (f' s)))))) s
  and s = t-hrs-update (hrs-mem-update (heap-update (p' s)
    (field-setter (f' s) (h-val (hrs-mem (t-hrs s)) (p' s)))))) s
  in subst)
apply (rule lense.upd-cong[where get = t-hrs and upd = t-hrs-update])
apply assumption
apply (rule lense.upd-cong[OF read-write-valid-hrs-mem])
apply (subst heap-update-field-root-conv'')
  apply assumption+
apply (simp add: typ-uinfo-t-def lense.fg-cons)

```

```

apply (subst h-val-field-from-bytes')
  apply assumption+
  apply (simp add: lense.fg-cons)
apply (subst update-ti-update-ti-t)
  apply (simp add: size-of-def)
apply (subst update-ti-s-adjust-ti-to-bytes-p)
  apply (erule lense.fg-cons)
apply (subst lense.upd-cong[where get = field-getter and upd = field-setter])
  apply auto
done

```

lemma *struct-rewrite-modifies-Array-field* [heap-abs]:

```

[[ valid-struct-field field-name (field-getter :: ('a::xmem-type) => (('f::array-outer-max-size)['n::array-max-count]
field-setter t-hrs t-hrs-update;
  struct-rewrite-expr P p' p;
  struct-rewrite-expr Q f' f;
   $\bigwedge s$ . heap-lift--wrap-h-val (h-val-p' s) (h-val (hrs-mem (t-hrs s)) (p' s));
  struct-rewrite-modifies R
    ( $\lambda s$ . t-hrs-update (hrs-mem-update (heap-update (p'' s)
      (u s (field-setter ( $\lambda a$ . Arrays.update a (nat k) (f' s (index a (nat
k)))))))) s)
    ( $\lambda s$ . t-hrs-update (hrs-mem-update (heap-update (p' s)
      (field-setter ( $\lambda a$ . Arrays.update a (nat k) (f' s (index a (nat k))))
      (h-val-p' s)))) s);
  struct-rewrite-guard S ( $\lambda s$ . c-guard (p' s)) ] ==>
struct-rewrite-modifies ( $\lambda s$ . P s  $\wedge$  Q s  $\wedge$  R s  $\wedge$  S s  $\wedge$  0  $\leq$  k  $\wedge$  nat k < CARD('n))
  ( $\lambda s$ . t-hrs-update (hrs-mem-update (heap-update (p'' s)
    (u s (field-setter ( $\lambda a$ . Arrays.update a (nat k) (f' s (index a (nat k)))))))) s)
  ( $\lambda s$ . t-hrs-update (hrs-mem-update (heap-update
    (ptr-coerce (Ptr (field-lvalue (p s) field-name) :: ('f['n] ptr) +p k)
    (f s (h-val (hrs-mem (t-hrs s)) (ptr-coerce (Ptr (field-lvalue (p s)
field-name) :: ('f['n] ptr) +p k :: 'f ptr)))))) s)
  using ptr-coerce.simps[simp del]
apply (clarsimp simp: struct-rewrite-expr-def struct-rewrite-guard-def struct-rewrite-modifies-def
valid-struct-field-def heap-lift--wrap-h-val-def)
  subgoal for s
    apply (erule-tac x = s in allE)+
    apply (erule impE, assumption)+
    apply (erule-tac t = t-hrs-update (hrs-mem-update (heap-update (p'' s)
      (u s (field-setter ( $\lambda a$ . Arrays.update a (nat k) (f' s (index a
(nat k)))))))) s
      and s = t-hrs-update (hrs-mem-update (heap-update (p' s)
        (field-setter ( $\lambda a$ . Arrays.update a (nat k) (f' s (index a (nat
k))))
          (h-val (hrs-mem (t-hrs s)) (p' s)))))) s
    in subst)
  apply (rule lense.upd-cong[where get = t-hrs and upd = t-hrs-update])

```

```

apply assumption
apply (rule lense.upd-cong[OF read-write-valid-hrs-mem])
apply (cases k, clarsimp)
apply (subst heap-update-array-element[symmetric])
  apply assumption
  apply simp
apply (subst heap-update-field-root-conv''')
  apply assumption+
  apply (simp add: typ-uinfo-t-def lense.fg-cons)
apply (subst h-val-field-from-bytes^')
  apply assumption+
  apply (simp add: lense.fg-cons)
apply (subst heap-access-Array-element[symmetric])
  apply simp
apply (subst h-val-field-from-bytes^')
  apply assumption+
  apply (simp add: lense.fg-cons)
apply clarsimp
apply (subst update-ti-update-ti-t)
  apply (simp add: size-of-def)
apply (subst update-ti-s-adjust-ti-to-bytes-p)
  apply (erule lense.fg-cons)
apply clarsimp
apply (subst lense.upd-cong[of field-getter field-setter])
  apply auto
done
done

```

definition

```

valid-globals-field ::
  ('s ⇒ 't)
  ⇒ ('s ⇒ 'a)
  ⇒ (('a ⇒ 'a) ⇒ 's ⇒ 's)
  ⇒ ('t ⇒ 'a)
  ⇒ (('a ⇒ 'a) ⇒ 't ⇒ 't)
  ⇒ bool

```

where

```

valid-globals-field st old-getter old-setter new-getter new-setter ≡
  (∀ s. new-getter (st s) = old-getter s)
  ∧ (∀ s v. new-setter v (st s) = st (old-setter v s))

```

lemma *abs-expr-globals-getter* [*heap-abs*]:

```

[[ valid-globals-field st old-getter old-setter new-getter new-setter ]]
  ⇒ abs-expr st (λ-. True) new-getter old-getter
apply (clarsimp simp: valid-globals-field-def abs-expr-def)
done

```

lemma *abs-expr-globals-setter* [*heap-abs*]:
 [*valid-globals-field* *st* *old-getter* *old-setter* *new-getter* *new-setter*;
 $\bigwedge \text{old. abs-expr } st (P \text{ old}) (v \text{ old}) (v' \text{ old})$]
 $\implies \text{abs-modifies } st (\lambda s. \forall \text{old. } P \text{ old } s) (\lambda s. \text{new-setter } (\lambda \text{old. } v \text{ old } s) s) (\lambda s. \text{old-setter } (\lambda \text{old. } v' \text{ old } s) s)$
apply (*clarsimp simp: valid-globals-field-def abs-expr-def abs-modifies-def*)
done

lemma *struct-rewrite-expr-globals-getter* [*heap-abs*]:
 [*valid-globals-field* *st* *old-getter* *old-setter* *new-getter* *new-setter*]
 $\implies \text{struct-rewrite-expr } (\lambda-. \text{True}) \text{old-getter old-getter}$
apply (*clarsimp simp: struct-rewrite-expr-def*)
done

lemma *struct-rewrite-modifies-globals-setter* [*heap-abs*]:
 [*valid-globals-field* *st* *old-getter* *old-setter* *new-getter* *new-setter*;
 $\bigwedge \text{old. struct-rewrite-expr } (P \text{ old}) (v' \text{ old}) (v \text{ old})$]
 $\implies \text{struct-rewrite-modifies } (\lambda s. \forall \text{old. } P \text{ old } s) (\lambda s. \text{old-setter } (\lambda \text{old. } v' \text{ old } s) s) (\lambda s. \text{old-setter } (\lambda \text{old. } v \text{ old } s) s)$
apply (*clarsimp simp: valid-globals-field-def struct-rewrite-expr-def struct-rewrite-modifies-def*)
done

lemma *abs-spec-may-not-modify-globals*[*heap-abs*]:
 $\text{abs-spec } st (\lambda-. \text{True}) \{(a, b). \text{meq } b \ a\} \{(a, b). \text{meq } b \ a\}$
apply (*clarsimp simp: abs-spec-def meq-def*)
done

lemma *abs-spec-modify-global*[*heap-abs*]:
 $\text{valid-globals-field } st \text{old-getter old-setter new-getter new-setter} \implies$
 $\text{abs-spec } st (\lambda-. \text{True}) \{(a, b). C \ a \ b\} \{(a, b). C' \ a \ b\} \implies$
 $\text{abs-spec } st (\lambda-. \text{True}) \{(a, b). \text{mex } (\lambda x. C \ (\text{new-setter } (\lambda-. x) \ a) \ b)\} \{(a, b). \text{mex } (\lambda x. C' \ (\text{old-setter } (\lambda-. x) \ a) \ b)\}$
apply (*fastforce simp: abs-spec-def mex-def valid-globals-field-def*)
done

lemma *uint-scst*:
 $\text{uint } (\text{scast } x :: 'a \ \text{word}) = \text{uint } (x :: 'a::\text{len signed word})$
apply (*subst down-cast-same [symmetric]*)
apply (*clarsimp simp: cast-simps*)
apply (*subst uint-up-ucast*)
apply (*clarsimp simp: cast-simps*)
apply *simp*
done

lemma *to-bytes-signed-word*:

to-bytes ($x :: 'a::len8$ signed word) $p = to-bytes$ (*scast* $x :: 'a$ word) p
by (*clarsimp simp: uint-scast to-bytes-def typ-info-word word-rsplit-def*)

lemma *from-bytes-signed-word*:

length $p = len-of$ *TYPE*('a) div 8 \implies
(*from-bytes* $p :: 'a::len8$ signed word) = *ucast* (*from-bytes* $p :: 'a$ word)
by (*clarsimp simp: from-bytes-def word-rcat-def scast-def cast-simps typ-info-word*)

lemma *hrs-mem-update-signed-word*:

hrs-mem-update (*heap-update* (*ptr-coerce* $p :: 'a::len8$ word ptr) (*scast* $val :: 'a::len8$ word))
= *hrs-mem-update* (*heap-update* p ($val :: 'a::len8$ signed word))
apply (*rule ext*)
apply (*clarsimp simp: hrs-mem-update-def split-def*)
apply (*clarsimp simp: heap-update-def to-bytes-signed-word size-of-def typ-info-word*)
done

lemma *h-val-signed-word*:

(*h-val* a $p :: 'a::len8$ signed word) = *ucast* (*h-val* a (*ptr-coerce* $p :: 'a$ word ptr))
apply (*clarsimp simp: h-val-def*)
apply (*subst from-bytes-signed-word*)
apply (*clarsimp simp: size-of-def typ-info-word*)
apply (*clarsimp simp: size-of-def typ-info-word*)
done

lemma *align-of-signed-word*:

align-of *TYPE*('a::len8 signed word) = *align-of* *TYPE*('a word)
by (*clarsimp simp: align-of-def typ-info-word*)

lemma *size-of-signed-word*:

size-of *TYPE*('a::len8 signed word) = *size-of* *TYPE*('a word)
by (*clarsimp simp: size-of-def typ-info-word*)

lemma *c-guard-ptr-coerce*:

[*align-of* *TYPE*('a) = *align-of* *TYPE*('b);
size-of *TYPE*('a) = *size-of* *TYPE*('b)] \implies
c-guard (*ptr-coerce* $p :: ('b::c-type)$ ptr) = *c-guard* ($p :: ('a::c-type)$ ptr)
apply (*clarsimp simp: c-guard-def ptr-aligned-def c-null-guard-def*)
done

lemma *word-rsplit-signed*:

(*word-rsplit* (*ucast* $v' :: ('a::len)$ signed word) $:: 8$ word list) = *word-rsplit* ($v' :: 'a$ word)
apply (*clarsimp simp: word-rsplit-def*)
apply (*clarsimp simp: cast-simps*)

done

lemma *heap-update-signed-word*:

$heap_update\ (ptr_coerce\ p :: 'a\ word\ ptr)\ (scast\ v) = heap_update\ (p :: ('a::len8)\ signed\ word\ ptr)\ v$

$heap_update\ (ptr_coerce\ p' :: 'a\ signed\ word\ ptr)\ (ucast\ v') = heap_update\ (p' :: ('a::len8)\ word\ ptr)\ v'$

apply (*auto simp: heap-update-def to-bytes-def typ-info-word word-rsplit-def cast-simps uint-scast*)

done

lemma *heap-update-padding-signed-word*:

$heap_update_padding\ (ptr_coerce\ p :: 'a\ word\ ptr)\ (scast\ v)\ bs = heap_update_padding\ (p :: ('a::len8)\ signed\ word\ ptr)\ v\ bs$

$heap_update_padding\ (ptr_coerce\ p' :: 'a\ signed\ word\ ptr)\ (ucast\ v')\ bs = heap_update_padding\ (p' :: ('a::len8)\ word\ ptr)\ v'\ bs$

by (*auto simp: heap-update-padding-def to-bytes-def typ-info-word word-rsplit-def cast-simps uint-scast*)

lemma *typ-heap-simulation-c-guard*:

$\llbracket\ typ_heap_simulation\ st\ r\ w\ v\ t_hrs\ t_hrs_update;\ v\ (st\ s)\ p\ \rrbracket \implies c_guard\ p$

by (*clarsimp simp: valid-implies-cguard.valid-implies-c-guard [OF typ-heap-simulation.axioms(5)]*)

abbreviation (*input*)

$scast_f :: (('a::len)\ signed\ word\ ptr \Rightarrow 'a\ signed\ word) \Rightarrow ('a\ word\ ptr \Rightarrow 'a\ word)$

where

$scast_f\ f \equiv (\lambda p. scast\ (f\ (ptr_coerce\ p)))$

abbreviation (*input*)

$ucast_f :: (('a::len)\ word\ ptr \Rightarrow 'a\ word) \Rightarrow ('a\ signed\ word\ ptr \Rightarrow 'a\ signed\ word)$

where

$ucast_f\ f \equiv (\lambda p. ucast\ (f\ (ptr_coerce\ p)))$

abbreviation (*input*)

$cast_f' :: ('a\ ptr \Rightarrow 'x) \Rightarrow ('b\ ptr \Rightarrow 'x)$

where

$cast_f'\ f \equiv (\lambda p. f\ (ptr_coerce\ p))$

lemma *read-write-validE-weak*:

$\llbracket\ lense\ r\ w;\ \llbracket\ \bigwedge f\ s. r\ (w\ f\ s) = f\ (r\ s);\ \bigwedge f\ s. f\ (r\ s) = (r\ s) \implies w\ f\ s = s\ \rrbracket \implies R\ \rrbracket \implies R$

apply *atomize-elim*

apply (*unfold lense-def*)

apply *metis*

done

lemma *lense-transcode*:

$\llbracket \text{lense } r \ w; \bigwedge v. f' (f \ v) = v; \bigwedge v. f (f' \ v) = v \rrbracket \implies$
 $\text{lense } (\lambda s. f' (r \ s)) (\lambda g \ s. w (\lambda \text{old}. f (g (f' \ \text{old}))) \ s)$
apply (*unfold lense-def*)
apply (*simp add:comp-def*)
done

lemma *typ-heap-simulation-signed-word*:

notes *align-of-signed-word* [*simp*]
size-of-signed-word [*simp*]
heap-update-signed-word [*simp*]

shows

$\llbracket \text{typ-heap-simulation } st$
 $(r :: 's \Rightarrow ('a::\text{len}8) \ \text{word } ptr \Rightarrow 'a \ \text{word}) \ w$
 $v \ t\text{-hrs } t\text{-hrs-update} \rrbracket$
 $\implies \text{typ-heap-simulation } st$
 $(\lambda s \ p. \ \text{ucast } (r \ s \ (ptr\text{-coerce } p)) :: 'a \ \text{signed word})$
 $(\lambda p \ f. \ (w \ (ptr\text{-coerce } p) \ ((\lambda x. \ \text{scast } (f \ (\text{ucast } x))))))$
 $(\lambda s \ p. \ v \ s \ (ptr\text{-coerce } p))$
 $t\text{-hrs } t\text{-hrs-update}$

apply (*clarsimp simp: typ-heap-simulation-def*
map.compositionality o-def c-guard-ptr-coerce)

apply (*intro conjI impI*)

subgoal

apply (*clarsimp simp add: read-simulation-def*)
apply (*drule-tac x=s in spec*)+
apply (*drule-tac x=ptr-coerce p in spec*)+
by (*simp add: h-val-signed-word*)

subgoal

apply (*clarsimp simp add: write-simulation-def write-simulation-axioms-def*)

subgoal for $s \ p \ bs \ x$

apply (*erule-tac x=s in alle*)
apply (*erule-tac x=(PTR-COERCE('a signed word \rightarrow 'a word) p) in alle*)
apply *clarsimp*
apply (*erule-tac x=bs in alle*)
apply *clarsimp*
apply (*erule-tac x= SCAST('a signed \rightarrow 'a) x in alle*)
using *heap-update-padding-signed-word*
by (*metis (mono-tags, lifting) hrs-mem-update-cong*)

done

subgoal

by (*clarsimp simp add: write-preserved-valid-def*)

subgoal

apply (*clarsimp simp add: valid-implies-cguard-def*)

```

apply (drule spec, drule spec, erule (1) impE)+
apply (subst (asm) c-guard-ptr-coerce, simp, simp)
by blast
subgoal
apply (simp (no-asm-use) add: valid-only-typ-desc-dependent-def)
by blast
subgoal
apply (simp (no-asm-use) add: pointer-lense-def)
apply clarsimp
by (metis comp-apply ucast-scast-id)
done

```

```

lemma c-guard-ptr-ptr-coerce:
   $\llbracket c\text{-guard } (a :: ('a::c\text{-type}) \text{ ptr ptr}); \text{ptr-val } a = \text{ptr-val } b \rrbracket \implies$ 
   $c\text{-guard } (b :: ('b::c\text{-type}) \text{ ptr ptr})$ 
by (clarsimp simp: c-guard-def ptr-aligned-def c-null-guard-def)

```

```

abbreviation (input)
  ptr-coerce-f :: ('a ptr ptr  $\Rightarrow$  'a ptr)  $\Rightarrow$  ('b ptr ptr  $\Rightarrow$  'b ptr)
where
  ptr-coerce-f f  $\equiv$  ( $\lambda p. \text{ptr-coerce } (f (\text{ptr-coerce } p))$ )

```

```

abbreviation (input)
  ptr-coerce-range-f :: ('a ptr  $\Rightarrow$  bool)  $\Rightarrow$  ('b ptr  $\Rightarrow$  bool)
where
  ptr-coerce-range-f f  $\equiv$  ( $\lambda p. (f (\text{ptr-coerce } p))$ )

```

```

lemma typ-heap-simulation-ptr-coerce:
   $\llbracket \text{typ-heap-simulation } st$ 
    ( $r :: 's \Rightarrow ('a::c\text{-type}) \text{ ptr ptr} \Rightarrow 'a \text{ ptr}$ )  $w$ 
     $v \text{ t-hrs t-hrs-update} \rrbracket$ 
   $\implies \text{typ-heap-simulation } st$ 
    ( $\lambda s p. \text{ptr-coerce } (r s (\text{ptr-coerce } p)) :: ('b::c\text{-type}) \text{ ptr}$ )
    ( $\lambda p f. (w (\text{ptr-coerce } p) ((\lambda x. \text{ptr-coerce } (f (\text{ptr-coerce } x))))))$ )
    ( $\lambda s p. v s (\text{ptr-coerce } p)$ )
     $t\text{-hrs } t\text{-hrs-update}$ 

```

```

apply (clarsimp simp: typ-heap-simulation-def fun-upd-def)
apply (intro conjI impI)
subgoal
by (clarsimp simp: read-simulation-def h-val-def typ-info-ptr from-bytes-def)
subgoal
apply (clarsimp simp add: write-simulation-def write-simulation-axioms-def)
apply (erule allE, erule allE, erule (1) impE)+
apply (erule-tac x=bs in allE)
apply clarsimp
apply (erule-tac x=ptr-coerce x in allE)
apply (clarsimp simp: heap-update-padding-def [abs-def] to-bytes-def typ-info-ptr)
done

```

```

subgoal
  apply (clarsimp simp add: write-preserves-valid-def)
  done
subgoal
  apply (clarsimp simp add: valid-implies-cguard-def)
  apply (drule spec, drule spec, erule (1) impE)+
  apply (subst (asm) c-guard-ptr-coerce, simp, simp)
  by blast
subgoal
  apply (simp (no-asm-use) add: valid-only-typ-desc-dependent-def)
  by blast
subgoal
  apply (simp (no-asm-use) add: pointer-lense-def)
  apply clarsimp
  by (metis comp-apply ptr-coerce-id ptr-coerce-idem)
done

```

```

lemmas signed-typ-heap-simulations =
  typ-heap-simulation-signed-word
  typ-heap-simulation-ptr-coerce [where 'a=( $'x::len8$ ) word and 'b=( $'x::len8$ )
signed word]
  typ-heap-simulation-ptr-coerce [where 'a=( $'x::len8$ ) word ptr and 'b=( $'x::len8$ )
signed word ptr]
  typ-heap-simulation-ptr-coerce [where 'a=( $'x::len8$ ) word ptr ptr and 'b=( $'x::len8$ )
signed word ptr ptr]
  typ-heap-simulation-ptr-coerce [where 'a=( $'x::len8$ ) word ptr ptr ptr and 'b=( $'x::len8$ )
signed word ptr ptr ptr]

```

```

lemma ptr-coerce-eq:
  (ptr-coerce x = ptr-coerce y) = (x = y)
  by (cases x, cases y, auto)

```

```

lemma signed-word-heap-opt [L2opt]:
  (scast ((( $\lambda x. ucast (a (ptr-coerce x))$ ) (p := v :: 'a::len signed word)) (b :: 'a signed
word ptr))))
  = ((a(ptr-coerce p := (scast v :: 'a word))) ((ptr-coerce b) :: 'a word ptr))
  by (auto simp: fun-upd-def ptr-coerce-eq)

```

```

lemma signed-word-heap-ptr-coerce-opt [L2opt]:
  (ptr-coerce ((( $\lambda x. ptr-coerce (a (ptr-coerce x))$ ) (p := v :: 'a ptr)) (b :: 'a ptr ptr))))
  = ((a(ptr-coerce p := (ptr-coerce v :: 'b ptr))) ((ptr-coerce b) :: 'b ptr ptr))
  by (auto simp: fun-upd-def ptr-coerce-eq)

```

```

declare ptr-coerce-idem [L2opt]

```

declare *scast-ucast-id* [*L2opt*]
declare *ucast-scast-id* [*L2opt*]

lemma *heap-abs-expr-c-guard-array* [*heap-abs*]:
 $\llbracket \text{typ-heap-simulation } st \ r \ w \ v \ t\text{-hrs } t\text{-hrs-update};$
 $\text{abs-expr } st \ P \ x' \ (\lambda s. \text{ptr-coerce } (x \ s) \ :: \ 'a \ \text{ptr}) \rrbracket \implies$
 $\text{abs-guard } st$
 $(\lambda s. P \ s \wedge (\forall a \in \text{set } (\text{array-addr} (x' \ s) \ \text{CARD}('b)). v \ s \ a))$
 $(\lambda s. \text{c-guard } (x \ s \ :: \ ('a::\text{array-outer-max-size}, 'b::\text{array-max-count}) \ \text{array} \ \text{ptr}))$
apply (*clarsimp simp: abs-expr-def abs-guard-def simple-lift-def root-ptr-valid-def*)
apply (*subgoal-tac* $\forall a \in \text{set } (\text{array-addr} (x' \ (st \ s)) \ \text{CARD}('b)). \text{c-guard } a$)
apply (*erule allE, erule (1) impE*)
apply (*rule c-guard-array-c-guard*)
apply (*subst (asm) (2) set-array-addr*)
apply *force*
apply *clarsimp*
apply (*erule (1) my-BallE*)
apply (*drule (1) typ-heap-simulation-c-guard*)
apply *simp*
done

lemma *fold-over-st*:
 $\llbracket xs = ys; P \ s;$
 $\bigwedge x. x \in \text{set } xs \wedge P \ s \implies P \ (g \ x \ s) \wedge f \ x \ (st \ s) = st \ (g \ x \ s)$
 $\rrbracket \implies \text{fold } f \ xs \ (st \ s) = st \ (\text{fold } g \ ys \ s)$
apply (*erule subst*)
apply (*induct xs arbitrary: s*)
apply *simp*
apply *simp*
done

lemma *fold-lift-write*:
 $\llbracket xs = ys; \text{lense } r \ w$
 $\rrbracket \implies \text{fold } (\lambda i. w \ (f \ i)) \ xs \ s = w \ (\text{fold } f \ ys) \ s$
apply (*erule subst*)
apply (*induct xs arbitrary: s*)
apply (*simp add: lense.upd-same*)
apply (*simp add: lense.upd-compose*)
done

lemma *fold-heap-update-list-nmem-same*:
 $\llbracket n * \text{size-of } \text{TYPE}('a \ :: \ \text{mem-type}) < \text{addr-card};$
 $n * \text{size-of } \text{TYPE}('a) \leq k; k < \text{addr-card};$
 $\bigwedge i \ h. \text{length } (\text{pad } i \ h) = \text{size-of } \text{TYPE}('a) \rrbracket \implies$
 $h \ (\text{ptr-val } (p \ :: \ 'a \ \text{ptr}) + \text{of-nat } k) =$
 $(\text{fold } (\lambda i \ h. \text{heap-update-list } (\text{ptr-val } (p \ +_p \ \text{int } i)))$

```

      (to-bytes (val i h :: 'a) (pad i h)) h) [0..<n] h) (ptr-val p + of-nat k)
apply (induct n arbitrary: k)
apply simp
apply (clarsimp simp: CTypesDefs.ptr-add-def simp del: mult-Suc)
apply (subst heap-update-nmem-same)
apply (subst len)
apply simp
apply (simp add: intvl-def)
apply (intro allI impI)
apply (subst (asm) of-nat-mult[symmetric] of-nat-add[symmetric])+
apply (rename-tac j)
apply (erule-tac Q = of-nat k = of-nat (n * size-of TYPE('a) + j) in contra-
pos-pn)
apply (subst of-nat-inj)
apply (subst len-of-addr-card)
apply simp
apply (subst len-of-addr-card)
apply simp
apply simp
apply simp
done

```

lemma *heap-list-of-disjoint-fold-heap-update-list*:

```

[[ n * size-of TYPE('a :: mem-type) < addr-card;
  n * size-of TYPE('a) + k < addr-card;
   $\bigwedge i h. \text{length (pad i h) = size-of TYPE('a)}$  ]]  $\implies$ 
heap-list (fold ( $\lambda i h. \text{heap-update-list (ptr-val ((p :: 'a ptr) +_p \text{int } i))$ 
      (to-bytes (val i h :: 'a) (pad i h)) h) [0..<n] h)
      k (ptr-val (p +_p int n))
= heap-list h k (ptr-val (p +_p int n))
apply (rule nth-equalityI, force)
subgoal for i
apply (clarsimp simp: heap-list-nth)
apply (rule subst[where t = ptr-val (p +_p int n) + of-nat i
  and s = ptr-val p + of-nat (n * size-of TYPE('a) + i)])
apply (clarsimp simp: CTypesDefs.ptr-add-def)
apply (rule fold-heap-update-list-nmem-same[symmetric])
apply simp-all
done
done

```

lemma *fold-heap-update-list*:

```

n * size-of TYPE('a :: mem-type) < 2^addr-bitsize  $\implies$ 
fold ( $\lambda i h. \text{heap-update-list (ptr-val ((p :: 'a ptr) +_p \text{int } i))$ 
      (to-bytes (val i :: 'a)
      (heap-list h (size-of TYPE('a)) (ptr-val (p +_p int i)))) h)
  [0..<n] h =
fold ( $\lambda i. \text{heap-update-list (ptr-val (p +_p int } i))$ )

```

```

      (to-bytes (val i)
        (heap-list h (size-of TYPE('a)) (ptr-val (p +p int i))))
    [0..<n] h
  apply (induct n)
  apply simp
  apply clarsimp
  apply (subst heap-list-of-disjoint-fold-heap-update-list)
  apply (simp add: len-of-addr-card[symmetric])+
  done

```

lemma *access-ti-list-array-unpacked:*

```

[[  $\forall n. \text{size-td-tuple } (f\ n) = v3; \text{length } xs = v3 * n;$ 
   $\forall m\ xs. \text{length } xs = v3 \wedge m < n \longrightarrow$ 
     $\text{access-ti-tuple } (f\ m) (FCP\ g)\ xs = h\ m\ xs$ 
]]  $\implies$ 
  access-ti-list (map f [0 ..<n]) (FCP g) xs
  = foldl (@) [] (map ( $\lambda m. h\ m\ (\text{take } v3\ (\text{drop } (v3 * m)\ xs))$ ) [0 ..<n])
  apply (subgoal-tac  $\forall ys. \text{size-td-list } (\text{map } f\ ys) = v3 * \text{length } ys$ )
  prefer 2
  subgoal
    apply (rule allI)
    subgoal for ys by (induct ys) auto
    done
  apply (induct n arbitrary: xs)
  apply simp
  apply (simp add: access-ti-append)
  apply (rule foldl-cong)
  apply simp
  apply (rule map-cong[OF refl])
  apply (subst take-drop)
  apply (subst take-take)
  apply (subst min-absorb1)
  apply clarsimp
  apply (metis Suc-leI mult-Suc-right nat-mult-le-cancel-disj)
  apply (subst take-drop[symmetric])
  apply (rule refl)
  apply simp
  done

```

lemma *concat-nth-chunk:*

```

[[  $\forall x \in \text{set } xs. \text{length } (f\ x) = \text{chunk}; n < \text{length } xs$  ]
 $\implies \text{take } \text{chunk}\ (\text{drop } (n * \text{chunk})\ (\text{concat } (\text{map } f\ xs))) = f\ (xs\ !\ n)$ 
  apply (induct xs arbitrary: n, force)
  subgoal for x xs n
    apply (cases n)
    apply clarsimp
    apply clarsimp
  done

```

done

lemma *array-update-split*:

\llbracket *typ-heap-simulation* st ($r :: 's \Rightarrow ('a::array-outer-max-size) ptr \Rightarrow 'a$) w
 v *t-hrs t-hrs-update*;
 $\forall x \in set$ (*array-addr*s (*ptr-coerce* p) $CARD('b::array-max-count)$).
 v (st s) x
 $\rrbracket \Longrightarrow st$ (*t-hrs-update* (*hrs-mem-update* (*heap-update* p ($arr :: 'a['b]$))) s) =
 fold ($\lambda i. w$ (*ptr-coerce* $p +_p$ *int* i) ($\lambda x. index$ arr i))
 [$0 ..< CARD('b)$] (st s)

apply (*clarsimp simp: typ-heap-simulation-def heap-raw-state-def valid-implies-cguard-def*
read-simulation-def write-preserved-valid-def)

apply (*drule write-simulation.write-commutes-atomic*)

apply (*subst fold-over-st[OF refl,*
 where $P = \lambda s. \forall x \in set$ (*array-addr*s (*ptr-coerce* p) $CARD('b)$). v (st s) x
 and $g = \lambda i. t-hrs-update$ (*hrs-mem-update* (*heap-update*
 (*ptr-coerce* $p +_p$ *int* i) (*index* arr i))))])

apply *simp*

subgoal for sa x

apply (*subgoal-tac* v (st sa) (*ptr-coerce* $p +_p$ *int* x))

apply *clarsimp*

apply (*clarsimp simp: set-array-addr*s)

apply *metis*

done

apply (*rule arg-cong[where f = st]*)

apply (*subst hrs-mem-update-def*) $+$

apply (*subst fold-lift-write[OF refl, where w = t-hrs-update]*)

apply *assumption*

apply (*rule arg-cong[where f = $\lambda f. t-hrs-update$ f s]*)

apply (*rule ext*)

apply (*subst fold-lift-write[OF refl,*

where $r = fst$ **and** $w = \lambda f s. case$ s *of* (h, z) \Rightarrow (f h, z))])

apply (*simp (no-asm) add: lense-def*)

apply *clarsimp*

apply (*clarsimp simp: heap-update-def[abs-def]*)

apply (*subst coerce-heap-update-to-heap-updates[unfolded foldl-conv-fold,*

where $chunk = size-of$ $TYPE('a)$ **and** $m = CARD('b)$])

apply (*rule size-of-array[simplified mult commute]*)

apply *simp*


```

apply (subst fold-heap-update-list[OF array-count-size])
apply (rule fold-cong[OF refl refl])
subgoal for a x

  apply (clarsimp simp: CTypesDefs.ptr-add-def)
  apply (rule arg-cong[where f = heap-update-list (ptr-val p + of-nat x * of-nat
(size-of TYPE('a)))]])

  apply (subst fcp-eta[where g = arr, symmetric])
  apply (clarsimp simp: to-bytes-def typ-info-array array-tag-def array-tag-n-eq
simp del: fcp-eta)
  apply (subst access-ti-list-array-unpacked)
    apply clarsimp
    apply (rule refl)
    apply (simp add: size-of-def)
    apply clarsimp
    apply (rule refl)
  apply (clarsimp simp: foldl-conv-concat)

  apply (subgoal-tac
     $\bigwedge x. x < \text{CARD}('b) \implies$ 
    size-td (typ-info-t TYPE('a))
     $\leq \text{CARD}('b) * \text{size-td} (\text{typ-info-t TYPE}('a)) - \text{size-td} (\text{typ-info-t}$ 
TYPE('a)) * x)
    prefer 2
    apply (subst le-diff-conv2)
    apply simp
    apply (subst mult.commute, subst mult-Suc[symmetric])
    apply (rule mult-le-mono1)
    apply simp

  apply (subst concat-nth-chunk)
    apply clarsimp
    apply (subst fd-cons-length)
    apply simp
    apply (simp add: size-of-def)
    apply (simp add: size-of-def)
    apply simp
  apply (subst drop-heap-list-le)
  apply (simp add: size-of-def)
  apply (subst take-heap-list-le)
  apply (simp add: size-of-def)
  apply (clarsimp simp: size-of-def)
  apply (subst mult.commute, rule refl)
  done
done

```

lemma fold-update-id:

```

[[ lense getter setter;
    $\forall i \in \text{set } xs. \forall j \in \text{set } xs. (i = j) = (\text{ind } i = \text{ind } j);$ 
    $\forall i \in \text{set } xs. \text{val } i = \text{getter } s (\text{ind } i)$ 
]]  $\implies \text{fold } (\lambda i. \text{setter } (\lambda x. x(\text{ind } i := \text{val } i))) xs s = s$ 
apply (induct xs)
apply simp
apply (rename-tac a xs)
apply clarsimp
apply (subgoal-tac setter  $(\lambda x. x(\text{ind } a := \text{getter } s (\text{ind } a))) s = s$ )
apply simp
apply (subst (asm) lense-def)
apply simp
done

```

lemma fold-update-id':

```

[[ pointer-lense r w;
    $\forall i \in \text{set } xs. \forall j \in \text{set } xs. (i = j) = (\text{ind } i = \text{ind } j);$ 
    $\forall i \in \text{set } xs. \text{val } i = r s (\text{ind } i)$ 
]]  $\implies \text{fold } (\lambda i. w (\text{ind } i) (\lambda-. \text{val } i)) xs s = s$ 
apply (induct xs)
apply simp
apply (rename-tac a xs)
apply clarsimp
apply (subgoal-tac w (ind a)  $(\lambda x. r s (\text{ind } a)) s = s$ )
apply simp
apply (subst (asm) pointer-lense-def)
apply simp
done

```

lemma array-count-index:

```

[[  $i < \text{CARD}('b::\text{array-max-count}); j < \text{CARD}('b)$  ]]
 $\implies (i = j) =$ 
   $((\text{of-nat } (i * \text{size-of TYPE}('a::\text{array-outer-max-size})) :: \text{addr})$ 
   $= \text{of-nat } (j * \text{size-of TYPE}('a)))$ 
apply (rule subst[where  $t = i = j$  and  $s = i * \text{size-of TYPE}('a) = j * \text{size-of TYPE}('a)$ ])
apply clarsimp
apply (subgoal-tac  $\bigwedge i. i < \text{CARD}('b) \implies i * \text{size-of TYPE}('a) < 2 \wedge \text{LENGTH}(\text{addr-bitsize})$ )
apply (rule of-nat-inj[symmetric]; force)
apply (rule subst[where  $t = \text{len-of TYPE}(\text{addr-bitsize})$  and  $s = \text{addr-bitsize}$ ],
force)
apply (rule less-trans)
apply (erule-tac  $b = \text{CARD}('b)$  in mult-strict-right-mono)
apply (rule sz-nzero)
apply (rule array-count-size)
done

```

lemma *le-outside-intvl*: $p < a \implies 0 \notin \{a ..+b\} \implies p \notin \{a ..+b\}$
apply (*clarsimp simp: intvl-def not-le not-less*)
by (*metis Word-Lemmas-Internal.word-wrap-of-natD add-increasing2 le0 le-iff-add less-le not-less*)

lemma *intvl-mult-split*:

$\{p ..+ a * b\} = (\bigcup_{i < b}. \{p + \text{of-nat } (i * a) ..+ a\})$

proof *cases*

assume $a: 0 < a$

note *of-nat-add*[*simp del, symmetric, simp*] *of-nat-mult*[*simp del, symmetric, simp*]

show *?thesis using a*

apply (*clarsimp simp: intvl-def, intro set-eqI iffI; clarsimp*)

subgoal for i

by (*intro bexI[of - i div a] exI[of - i mod a]*)

(*simp-all add: dme pos-mod-bound less-mult-imp-div-less ac-simps*)

subgoal for $j\ k$

by (*intro exI[of - j * a + k] (simp add: nat-index-bound)*)

done

qed *simp*

lemma *intvl-mul-disjnt*:

fixes $n\ i :: 'a::\text{len word}$

assumes $n: \text{unat } n < b$ **and** $i: \text{unat } i < b$ **and** $b: \text{sz} * b \leq 2^{\text{LENGTH}('a)}$

assumes $ni: n \neq i$

shows $\{n * \text{word-of-nat } \text{sz} ..+ \text{sz}\} \cap \{i * \text{word-of-nat } \text{sz} ..+ \text{sz}\} = \{\}$

proof $-$

{ fix $j\ l$ **assume** $j: j < \text{sz}$ **and** $l: l < \text{sz}$

assume $n * \text{word-of-nat } \text{sz} + \text{word-of-nat } j = i * \text{word-of-nat } \text{sz} + \text{word-of-nat } l$

then have $(\text{word-of-nat } (\text{unat } n * \text{sz} + j) :: 'a\ \text{word}) = \text{word-of-nat } (\text{unat } i * \text{sz} + l)$ **by** *simp*

moreover have $\text{unat } n * \text{sz} + j < 2^{\text{LENGTH}('a)}$

by (*intro less-le-trans[OF nat-index-bound b] n j*)

moreover have $\text{unat } i * \text{sz} + l < 2^{\text{LENGTH}('a)}$

by (*intro less-le-trans[OF nat-index-bound b] i l*)

ultimately have $(\text{unat } n * \text{sz} + j) \text{ div } \text{sz} = (\text{unat } i * \text{sz} + l) \text{ div } \text{sz}$

by (*subst (asm) of-nat-inj*) *simp-all*

then have $\text{unat } n = \text{unat } i$

using $j\ l$ **by** *simp*

with ni **have** *False* **by** *simp* }

then show *?thesis*

by (*auto simp: intvl-def*)

qed

lemma *array-disj-helper*:

fixes $p :: ('a::\text{mem-type}['b])\ \text{ptr}$

assumes $ni: n < \text{CARD}('b)$ $i < \text{CARD}('b)$ $n \neq i$

assumes *valid*: $\forall x \in \text{set} (\text{array-addr} (\text{PTR-COERCE}('a['b] \rightarrow 'a) p) \text{CARD}('b)).$
c-guard *x*
shows $\{\text{ptr-val } p + \text{word-of-nat } n * \text{word-of-nat} (\text{size-of TYPE}('a))..+\text{size-of TYPE}('a)\} \cap$
 $\{\text{ptr-val } p + \text{word-of-nat } i * \text{word-of-nat} (\text{size-of TYPE}('a))..+\text{size-of TYPE}('a)\}$
 $= \{\}$
proof –
have [*arith*]: $\text{CARD}('b) \leq \text{size-of TYPE}('a) * \text{CARD}('b)$
using *sz-nzero*[**where** $'a='a$, *arith*] **by** *simp*

have $0 \notin (\bigcup b < \text{CARD}('b). \{\text{ptr-val } p + \text{of-nat} (b * \text{size-of TYPE}('a)) ..+ \text{size-of TYPE}('a)\})$
using *valid*
apply (*clarsimp simp: set-array-addr c-guard-def c-null-guard-def*)
subgoal premises *prems* **for** *b*
using *prems*(2–) *prems*(1)[*rule-format*, *OF exI*, *of*
 $\text{Ptr} (\text{ptr-val } p + \text{word-of-nat } b * \text{word-of-nat} (\text{size-of TYPE}('a))) b]$
by (*simp add: ptr-add-def*)
done
then have $0 \notin \{\text{ptr-val } p ..+ \text{size-of TYPE}('a) * \text{CARD}('b)\}$
unfolding *intvl-mult-split* .
from *zero-not-in-intvl-no-overflow*[*OF this*]
have $\text{size-of TYPE}('a) * \text{CARD}('b) \leq \text{addr-card}$
by (*simp add: addr-card*)
moreover note *ni*
ultimately show *?thesis*
by (*smt* (*verit*, *ccfv-SIG*) $\langle \text{CARD}('b) \leq \text{size-of TYPE}('a) * \text{CARD}('b) \rangle$ *addr-card-len-of-conv*
 $\text{intvl-disj-offset intvl-mul-disjnt order-less-le-trans unat-of-nat-len}$)
qed

theorem *heap-abs-array-update* [*heap-abs*]:
 $\llbracket \text{typ-heap-simulation } st (r :: 's \Rightarrow 'a \text{ ptr} \Rightarrow 'a) w$
 $v \text{ t-hrs } t\text{-hrs-update};$
 $\text{abs-expr } st \text{ Pb } b' b;$
 $\text{abs-expr } st \text{ Pn } n' n;$
 $\text{abs-expr } st \text{ Pv } y' y$
 $\rrbracket \Longrightarrow$
 $\text{abs-modifies } st (\lambda s. \text{Pb } s \wedge \text{Pn } s \wedge \text{Pv } s \wedge n' s < \text{CARD}('b) \wedge$
 $(\forall \text{ptr} \in \text{set} (\text{array-addr} (\text{ptr-coerce} (b' s)) \text{CARD}('b)). (v s$
 $\text{ptr}))$
 $(\lambda s. w (\text{ptr-coerce} (b' s) +_p \text{int} (n' s)) (\lambda v. y' s) s)$
 $(\lambda s. t\text{-hrs-update} (\text{hrs-mem-update} ($
 $\text{heap-update } (b s) (\text{Arrays.update} ((h\text{-val} (\text{hrs-mem} (t\text{-hrs } s)) (b s))$
 $:: ('a::\text{array-outer-max-size})['b::\text{array-max-count}]) (n s)$
 $(y s)))) s)$
apply (*clarsimp simp: abs-modifies-def abs-expr-def*)
subgoal for *s*

```

apply (subst array-update-split
  [where st = st and t-hrs = t-hrs and t-hrs-update = t-hrs-update])
apply assumption
apply assumption
apply (clarsimp simp: typ-heap-simulation-def valid-implies-cguard-def read-simulation-def
write-preserved-valid-def)
apply (drule write-simulation.write-commutes-atomic)
apply (subst fold-cong[OF refl refl,
  where  $g = \lambda i. w \text{ (ptr-coerce (b' (st s)) } +_p \text{ int } i) (\lambda x.$ 
    if  $i = n' (st s)$  then  $y' (st s)$  else  $r (st s) \text{ (ptr-coerce (b' (st s))$ 
    +p int i))])
subgoal for x
apply (cases x = n' (st s))
apply simp
apply (subst index-update2)
apply simp
apply simp
apply (rule arg-cong[where x = index (h-val (hrs-mem (t-hrs s)) (b' (st s)))
x])
apply (subst heap-access-Array-element)
apply simp
apply (clarsimp simp: set-array-addr)
apply metis
done

```

```

apply (subst split-upt-on-n[where n = n s])
apply simp
apply clarsimp
thm fold-update-id

```

```

apply (subst fold-update-id'[where s = st s])
apply assumption
apply (clarsimp simp: CTypesDefs.ptr-add-def)
apply (subst of-nat-mult[symmetric])+
apply (rule array-count-index)
apply (erule less-trans, assumption)+
apply clarsimp

```

```

apply (subst fold-update-id')
apply assumption
apply (clarsimp simp: CTypesDefs.ptr-add-def)
apply (subst of-nat-mult[symmetric])+
apply (erule array-count-index)
apply assumption
apply clarsimp

```

```

apply (subst pointer-lense.read-write-other[where  $r = r$  and  $w = w$ ])
  apply assumption
apply (clarsimp simp: CTypesDefs.ptr-add-def disjnt-def)
apply (rule array-disj-helper)
  apply simp
  apply assumption
apply simp
apply blast
apply (rule refl)
apply (rule refl)
done
done

```

```

lemma heap-abs-array-access[heap-abs]:
  [| typ-heap-simulation st ( $r :: 's \Rightarrow ('a::xmem-type) ptr \Rightarrow 'a$ )  $w$ 
     $v$  t-hrs t-hrs-update;
    abs-expr st  $Pb$   $b' b$ ;
    abs-expr st  $Pn$   $n' n$ 
  |]  $\Rightarrow$ 
  abs-expr st ( $\lambda s. Pb\ s \wedge Pn\ s \wedge n'\ s < CARD('b::finite) \wedge v\ s$  (ptr-coerce ( $b'\ s$ )
+p int ( $n'\ s$ )))
  ( $\lambda s. r\ s$  (ptr-coerce ( $b'\ s$ ) +p int ( $n'\ s$ )))
  ( $\lambda s. index$  (h-val (hrs-mem (t-hrs s)) ( $b\ s :: ('a['b]) ptr$ )) ( $n\ s$ ))
apply (clarsimp simp: typ-heap-simulation-def abs-expr-def valid-implies-cguard-def
read-simulation-def write-simulation-def write-preserves-valid-def)
apply (subst heap-access-Array-element)
apply simp
apply (simp add: set-array-addr)
done

```

```

lemma the-fun-upd-lemma1:
  ( $\lambda x. the$  ( $f\ x$ ))( $p := v$ ) = ( $\lambda x. the$  (( $f$  ( $p := Some\ v$ ))  $x$ ))
by auto

```

```

lemma the-fun-upd-lemma2:
   $\exists z. f\ p = Some\ z \Rightarrow$ 
  ( $\lambda x. \exists z. (f$  ( $p := Some\ v$ ))  $x = Some\ z$ ) = ( $\lambda x. \exists z. f\ x = Some\ z$ )
by auto

```

```

lemma the-fun-upd-lemma3:
  (( $\lambda x. the$  ( $f\ x$ ))( $p := v$ ))  $x = the$  (( $f$  ( $p := Some\ v$ ))  $x$ )
by simp

```

```

lemma the-fun-upd-lemma4:

```

$\exists z. f p = \text{Some } z \implies$
 $(\exists z. (f (p := \text{Some } v)) x = \text{Some } z) = (\exists z. f x = \text{Some } z)$
by *simp*

lemmas *the-fun-upd-lemmas* =
the-fun-upd-lemma1
the-fun-upd-lemma2
the-fun-upd-lemma3
the-fun-upd-lemma4

lemma *word-update*:
 $\bigwedge ptr :: ('a :: len) \text{ signed word ptr.}$
 $(\lambda(x :: 'a \text{ word ptr} \Rightarrow 'a \text{ word}) p :: 'a \text{ word ptr.}$
 $\text{if } ptr\text{-coerce } p = ptr \text{ then } scast (n :: 'a \text{ signed word}) \text{ else } x (ptr\text{-coerce } p))$
 $=$
 $(\lambda(old :: 'a \text{ word ptr} \Rightarrow 'a \text{ word}) x :: 'a \text{ word ptr.}$
 $\text{if } x = ptr\text{-coerce } ptr \text{ then } scast n \text{ else } old x)$
by *force*

Proof taken from $\llbracket ?n < CARD(?'b); CARD(?'b) * \text{size-of } TYPE(?'a) < 2^{addr\text{-bitsize}} \rrbracket \implies \text{heap-update } ?p (\text{Arrays.update } ?arr ?n ?v) ?hp = \text{heap-update } (\text{array-ptr-index } ?p \text{ False } ?n) ?v (\text{heap-update } ?p ?arr ?hp)$

lemma (in *array-outer-max-size*) *array-index-unat-conv*:
assumes *x-bound*: $x < CARD('b::array\text{-max-count})$
assumes *k-bound*: $k < \text{size-of } TYPE('a)$
shows $unat (of\text{-nat } x * of\text{-nat } (\text{size-of } TYPE('a)) + (of\text{-nat } k :: \text{addr}))$
 $= x * \text{size-of } TYPE('a) + k$

proof –
have *size*: $CARD('b) * \text{size-of } TYPE('a) < 2^{\wedge} \text{addr-bitsize}$
using *array-count-size* **by** *blast*
show *?thesis*
using *size x-bound k-bound*
apply (*cases size-of TYPE('a), simp-all*)
apply (*cases CARD('b), simp-all*)
apply (*subst unat-add-lem[THEN iffD1]*)
apply (*simp add: unat-word-ariths unat-of-nat less-Suc-eq-le*)
apply (*subgoal-tac Suc x * size-of TYPE('a) < 2^{\wedge} addr-bitsize, simp-all*)
apply (*erule order-le-less-trans[rotated], simp add: add-mono*)
apply (*subst unat-mult-lem[THEN iffD1]*)
apply (*simp add: unat-of-nat unat-add-lem[THEN iffD1]*)
apply (*rule order-less-le-trans, erule order-le-less-trans[rotated], rule add-mono, simp+*)
apply (*simp add: less-Suc-eq-le trans-le-add2*)
apply *simp*
apply (*simp add: unat-of-nat unat-add-lem[THEN iffD1]*)
done
qed

lemma *ptr-span-array-index-disj*:
fixes $p::('a::array-outer-max-size['b::array-max-count])$ ptr
assumes $n-bound: n < CARD ('b)$
assumes $i-bound: i < n$
shows $disjnt (ptr-span (array-ptr-index p False n)) (ptr-span (array-ptr-index p False i))$
using $n-bound i-bound$
apply (*clarsimp simp add: array-ptr-index-def ptr-add-def intvl-def disjnt-def*)
apply (*intro set-eqI*)
apply *clarsimp*
apply (*drule word-unat.Rep-inject[THEN iffD2]*)
apply (*clarsimp simp: array-index-unat-conv [where 'b='b] nat-eq-add-iff1*)
by (*metis mult.commute nat-index-bound not-add-less1*)

lemma *ptr-span-array-ptr-index-disj*:
fixes $p::('a::array-outer-max-size['b::array-max-count])$ ptr
fixes $q::('a['b::array-max-count])$ ptr
assumes $n-bound: n < CARD ('b)$
assumes $m-bound: m < CARD ('b)$
assumes $disj:disjnt (ptr-span p) (ptr-span q)$
shows $disjnt (ptr-span (array-ptr-index p False n)) (ptr-span (array-ptr-index q False m))$
proof (*rule ccontr*)
assume $\neg disjnt (ptr-span (array-ptr-index p False n)) (ptr-span (array-ptr-index q False m))$

then obtain $k k'$ **where**
 $k-bound: k < size-of TYPE('a)$ **and**
 $k'-bound: k' < size-of TYPE('a)$ **and**
 $eq: ptr-val p + word-of-nat n * word-of-nat (size-of TYPE('a)) + word-of-nat k =$
 $ptr-val q + word-of-nat m * word-of-nat (size-of TYPE('a)) + word-of-nat k'$
by (*auto simp add: intvl-def array-ptr-index-def ptr-add-def disjnt-def*)

define i **where** $i = unat ((word-of-nat n::addr) * word-of-nat (size-of TYPE('a)) + word-of-nat k)$
have $i-bound: i < CARD('b) * size-of TYPE('a)$
using $n-bound k-bound$
by(*simp add: i-def array-index-unat-conv [OF n-bound k-bound] mult.commute nat-index-bound*)

define j **where** $j = unat ((word-of-nat m::addr) * word-of-nat (size-of TYPE('a)) + word-of-nat k')$
have $j-bound: j < CARD('b) * size-of TYPE('a)$
using $m-bound k'-bound$
by(*simp add: j-def array-index-unat-conv [OF m-bound k'-bound] mult.commute nat-index-bound*)


```

from i-bound j-bound disj have ptr-val p + word-of-nat i ≠ ptr-val q + word-of-nat
j
  by (auto simp add: intvl-def disjnt-def)
  with eq show False
  by (simp add: i-def j-def add.commute add.left-commute)
qed

```

```

named-theorems select-conv and select-conv-independent and valid-conv and
valid-array-conv and
gen-update-commute and gen-outer-update-commute and update-commute

```

```

locale pointer-array-lense = pointer-lense r w
  for r:: 's ⇒ 'a:: array-outer-max-size ptr ⇒ 'a
  and w:: 'a ptr ⇒ ('a ⇒ 'a) ⇒ 's ⇒ 's
begin

```

```

definition heap-array :: 's ⇒ ('a['b::array-max-count]) ptr ⇒ 'a['b] where
heap-array s p = FCP (λi. r s (array-ptr-index p False i))

```

```

lemmas [read-stack-byte-ZERO-step-subst] = heap-array-def

```

```

definition heap-array-map :: ('a['b]) ptr ⇒ ('a['b::array-max-count] ⇒ 'a['b]) ⇒
's ⇒ 's where
heap-array-map p f s ≡
  fold (λi. w (array-ptr-index p False i) (λ-. (f (heap-array s p)).[i])) [0..<CARD('b)]
s

```

```

lemma element-heap-eq-heap-array-eq [select-conv]: r s = r s' ⇒ heap-array s =
heap-array s'
  apply (rule ext)
  apply (simp add: heap-array-def)
  done

```

```

lemma fold-write-independent-field:
  fixes p::('a['b::array-max-count]) ptr
  assumes field-independent: (∧p x s. f (w p (λ-. x) s) = f s)
  assumes n-bound: n ≤ CARD('b)
  shows f (fold (λi. w (array-ptr-index p False i) (λ-. g (heap-array s p).[i]))
  [0..<n] t) = f t
  using n-bound
proof (induct n arbitrary: t)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then show ?case using field-independent
  by simp
qed

```

```

lemma heap-array-map-independent-field [select-conv-independent]:
  assumes field-independent:  $(\bigwedge p x s. f (w p (\lambda-. x) s) = f s)$ 
  shows  $f (heap\text{-}array\text{-}map p g s) = f s$ 
  apply (simp add: heap-array-map-def)
  apply (rule fold-write-independent-field)
  apply (rule field-independent)
  apply simp
  done

lemma fold-write-independent-field-upd-commute:
  fixes  $p::('a['b::array\text{-}max\text{-}count]) ptr$ 
  assumes write-commute:  $\bigwedge p x s. (f\text{-}upd (w p (\lambda-. x) s)) = w p (\lambda-. x) (f\text{-}upd s)$ 
  assumes n-bound:  $n \leq CARD('b)$ 
  shows  $f\text{-}upd (fold (\lambda i. w (array\text{-}ptr\text{-}index p False i) (\lambda-. g (heap\text{-}array s p).[i])) [0..<n] t) =$ 
 $fold (\lambda i. w (array\text{-}ptr\text{-}index p False i) (\lambda-. g (heap\text{-}array s p).[i])) [0..<n]$ 
 $(f\text{-}upd t)$ 
  using n-bound
  proof (induct n arbitrary: t)
  case 0
  then show ?case by simp
  next
  case (Suc n)
  then show ?case using write-commute
  by simp
  qed

lemma heap-array-map-independent-field-commute [gen-update-commute]:
  assumes read-independent:  $\bigwedge s. r (f\text{-}upd s) = r s$ 
  assumes write-independent:  $\bigwedge p x s. (f\text{-}upd (w p (\lambda-. x) s)) = w p (\lambda-. x) (f\text{-}upd s)$ 
  shows  $f\text{-}upd (heap\text{-}array\text{-}map p g s) = (heap\text{-}array\text{-}map p g (f\text{-}upd s))$ 
  apply (simp add: heap-array-map-def element-heap-eq-heap-array-eq [OF read-independent])
  apply (rule fold-write-independent-field-upd-commute)
  apply (rule write-independent)
  apply simp
  done

lemma heap-array-index[simp]:  $i < CARD('b) \implies$ 
 $heap\text{-}array s (p::('a['b::array\text{-}max\text{-}count]) ptr).[i] = r s (array\text{-}ptr\text{-}index p False$ 
 $i)$ 
  by (simp add: heap-array-def)

lemma read-write-same-fold-aux:
  fixes  $p::('a['b::array\text{-}max\text{-}count]) ptr$ 
  assumes n-bound:  $n \leq CARD('b)$ 
  assumes i-bound:  $i < n$ 

```

```

  shows r (fold (λi. w (array-ptr-index p False i) (λ-. f (heap-array s p).[i]))
    [0.. $n$ ] s) (array-ptr-index p False i) = f (heap-array s p).[i]
  using n-bound i-bound
  proof (induct n arbitrary: i s)
    case 0
    then show ?case by simp
  next
    case (Suc n)
    from Suc.premis obtain i-bound: i < Suc n and Suc-n-bound: Suc n ≤ CARD
      ('b) and
      n-bound: n ≤ CARD('b) and n-bound': n < CARD('b) by auto
    have size: CARD('b) * size-of TYPE('a) < 2 ^ addr-bitsize
      using Padding-Equivalence.array-count-size by blast
    show ?case
    proof (cases i=n)
      case True
      show ?thesis
      apply (simp add: True)
      apply (simp add: read-write-same)
      done
    next
      case False
      from False i-bound have i-bound': i < n by simp
      have disj: disjnt (ptr-span (array-ptr-index p False n)) (ptr-span (array-ptr-index
        p False i))
        by (rule ptr-span-array-index-disj [OF n-bound' i-bound'])
      show ?thesis
      apply (simp add: False)
      apply (simp add: read-write-other disj)
      apply (rule Suc.hyps [OF n-bound i-bound'])
      done
    qed
  qed

```

```

lemma array-read-write-same: heap-array (heap-array-map p f s) p = f (heap-array
  s p)
  apply (subst cart-eq)
  apply clarsimp
  apply (simp add: heap-array-map-def)
  apply (rule read-write-same-fold-aux)
  by simp-all

```

```

lemma read-write-other-fold-aux:
  fixes p::('a['b::array-max-count]) ptr
  fixes p'::('a['b::array-max-count]) ptr
  assumes n-bound: n ≤ CARD ('b)
  assumes i-bound: i < CARD ('b)
  assumes disj:disjnt (ptr-span p) (ptr-span p')
  shows r (fold (λi. w (array-ptr-index p False i) (λ-. f (heap-array s p).[i]))

```

```

[0.. $n$ ] s) (array-ptr-index p' False i) =
  r s (array-ptr-index p' False i)
  using n-bound i-bound
proof (induct n arbitrary: i s)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.premis obtain Suc-n-bound: Suc n  $\leq$  CARD ('b) and
    n-bound: n  $\leq$  CARD ('b) and n-bound': n < CARD ('b) and
    i-bound: i < CARD ('b) by auto
  have size: CARD ('b) * size-of TYPE ('a) < 2 ^ addr-bitsize
  using Padding-Equivalence.array-count-size by blast

  from ptr-span-array-ptr-index-disj [OF n-bound' i-bound disj]
  show ?case
  apply (simp add: read-write-other)
  apply (rule Suc.hyps [OF n-bound i-bound])
  done
qed

```

```

lemma array-read-write-other:
  fixes p::('a['b::array-max-count]) ptr
  fixes p'::('a['b::array-max-count]) ptr
  assumes disj:disjnt (ptr-span p) (ptr-span p')
  shows heap-array (heap-array-map p f s) p' = heap-array s p'
  apply (subst cart-eq)
  apply clarsimp
  apply (simp add: heap-array-map-def)
  apply (rule read-write-other-fold-aux [OF - - disj])
  by simp-all

```

```

lemma write-cong-fold-aux:
  fixes p::('a['b::array-max-count]) ptr
  assumes f-f': f (heap-array s p) = f' (heap-array s p)
  assumes n-bound: n  $\leq$  CARD ('b)
  shows fold ( $\lambda$ i. w (array-ptr-index p False i) ( $\lambda$ -. f (heap-array s p).[i])) [0.. $n$ ]
s =
  fold ( $\lambda$ i. w (array-ptr-index p False i) ( $\lambda$ -. f' (heap-array s p).[i])) [0.. $n$ ] s
  using n-bound
proof (induct n arbitrary:)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have suc-n-bound: Suc n  $\leq$  CARD ('b) using Suc.premis .

```

```

define s' where s' = (fold ( $\lambda$ i. w (array-ptr-index p False i) ( $\lambda$ -. f' (heap-array
s p).[i])) [0.. $n$ ] s)

```

```

from f-f' suc-n-bound
have  $(\lambda-. f \text{ (heap-array } s \text{ } p).\text{[}n\text{]}) = (\lambda-. f' \text{ (heap-array } s \text{ } p).\text{[}n\text{]})$ 
  by auto
hence eq:  $(\lambda-. f \text{ (heap-array } s \text{ } p).\text{[}n\text{]}) (r \text{ } s' \text{ (array-ptr-index } p \text{ } \text{False } n)) =$ 
   $(\lambda-. f' \text{ (heap-array } s \text{ } p).\text{[}n\text{]}) (r \text{ } s' \text{ (array-ptr-index } p \text{ } \text{False } n))$  by metis
from Suc show ?case by (simp add: write-cong [OF eq])
qed

```

```

lemma array-write-cong:
  fixes p::('a['b::array-max-count]) ptr
  assumes eq:  $f \text{ (heap-array } s \text{ } p) = f' \text{ (heap-array } s \text{ } p)$ 
  shows  $\text{heap-array-map } p \text{ } f \text{ } s = \text{heap-array-map } p \text{ } f' \text{ } s$ 
  apply (simp add: heap-array-map-def)
  apply (rule write-cong-fold-aux)
  apply (rule eq)
  apply simp
  done

```

```

lemma array-write-same-triv[simp]:
  fixes p::('a['b::array-max-count]) ptr
  shows  $\text{heap-array-map } p \text{ } (\lambda-. f \text{ (heap-array } s \text{ } p)) \text{ } s = \text{heap-array-map } p \text{ } f \text{ } s$ 
  using array-write-cong by fastforce

```

```

lemma array-write-fold-same-aux:
  fixes p::('a['b::array-max-count]) ptr
  assumes f-id:  $f \text{ (heap-array } s \text{ } p) = \text{heap-array } s \text{ } p$ 
  assumes n-bound:  $n \leq \text{CARD('b)}$ 
  shows  $\text{fold } (\lambda i. w \text{ (array-ptr-index } p \text{ } \text{False } i) (\lambda-. f \text{ (heap-array } s \text{ } p).\text{[}i\text{]})) [0..<n]$ 
   $s = s$ 
  using n-bound
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prems f-id
  have f-n-id:  $f \text{ (heap-array } s \text{ } p).\text{[}n\text{]} = r \text{ } s \text{ (array-ptr-index } p \text{ } \text{False } n)$ 
  by simp
  from Suc show ?case
  by (simp add: write-same f-n-id)
qed

```

```

lemma array-write-same:
  fixes p::('a['b::array-max-count]) ptr
  assumes f-id:  $f \text{ (heap-array } s \text{ } p) = \text{heap-array } s \text{ } p$ 
  shows  $\text{heap-array-map } p \text{ } f \text{ } s = s$ 
  apply (simp add: heap-array-map-def)

```

apply (rule array-write-fold-same-aux)
apply (rule f-id)
by simp

lemma array-write-triv [simp]:
fixes p::('a['b::array-max-count]) ptr
shows heap-array-map p (λ-. heap-array s p) s = s **and** heap-array-map p (λx.
x) s = s
by (simp-all add: array-write-same)

lemma write-fold-other-commute:
fixes p::nat ⇒ 'a ptr
and q:: 'a ptr
assumes disj: $\bigwedge i. i < n \implies \text{disjnt } (\text{ptr-span } q) (\text{ptr-span } (p \ i))$
shows
w q f (fold (λi. w (p i) (g i)) [0.. n] s) =
fold (λi. w (p i) (g i)) [0.. n] (w q f s)
using disj
proof (induct n arbitrary: s)
case 0
then show ?case **by** simp
next
case (Suc n)
from Suc.premis **obtain**
q-n-disj: disjnt (ptr-span q) (ptr-span (p n)) **and**
disj': $\bigwedge i. i < n \implies \text{disjnt } (\text{ptr-span } q) (\text{ptr-span } (p \ i))$ **by** auto

note n-commute = write-other-commute[OF q-n-disj, symmetric]
show ?case
apply simp
apply (subst n-commute)
apply (subst Suc.hyps [OF disj'])
apply (simp-all)
done
qed

lemma write-fold-arr-commute:
fixes p::('a['b::array-max-count]) ptr
and arr:: 'a['b::array-max-count]
assumes n-bound: $n < \text{CARD}('b)$
shows
w (array-ptr-index p False n) (λ-. arr.[n])
(fold (λi. w (array-ptr-index p False i) (λ-. arr.[i]))
[0.. n] s) =
fold
(λi. w (array-ptr-index p False i) (λ-. arr.[i]))
[0.. n] (w (array-ptr-index p False n) (λ-. arr.[n]) s)

apply (*rule write-fold-other-commute*)
using *n-bound*
by (*simp add: ptr-span-array-index-disj*)

lemma *array-fold-fuse-aux*:
fixes $p::('a['b::array-max-count])\ ptr$
fixes $f::'a['b] \Rightarrow 'a['b]$
fixes $g::'a['b] \Rightarrow 'a['b]$
assumes *n-bound*: $n \leq CARD('b)$
shows *fold*
 $(\lambda i. w (array-ptr-index\ p\ False\ i) (\lambda-. f (g (heap-array\ s\ p)).[i]))$
 $[0..<n]$
fold
 $(\lambda i. w (array-ptr-index\ p\ False\ i) (\lambda-. g (heap-array\ s\ p).[i]))$
 $[0..<n]\ t) =$
fold
 $(\lambda i. w (array-ptr-index\ p\ False\ i) (\lambda-. f (g (heap-array\ s\ p)).[i]))$
 $[0..<n]\ t$
using *n-bound*
proof (*induct n arbitrary: t*)
case 0
then show ?*case* **by** *simp*
next
case (*Suc n*)
from *Suc.prem*s **have** *n-bound*: $n \leq CARD('b)$ **and** *n-bound'*: $n < CARD('b)$
by *auto*
show ?*case*
apply (*simp add:*)

apply (*subst (2) write-fold-arr-commute [OF n-bound']*)
apply (*subst Suc.hyps [OF n-bound]*)
apply (*subst write-fold-arr-commute [OF n-bound']*)
apply (*subst write-comp*)
apply (*subst (1) write-fold-arr-commute [OF n-bound']*)
by (*meson comp-apply*)
qed

lemma *array-write-comp*:
fixes $p::('a['b::array-max-count])\ ptr$
shows $heap-array-map\ p\ f (heap-array-map\ p\ g\ s) = heap-array-map\ p (f\ o\ g)\ s$
proof –
from *array-read-write-same [of p g s]*
have *g-eq*: $heap-array (heap-array-map\ p\ g\ s)\ p = g (heap-array\ s\ p)$.
show ?*thesis*

```

apply (subst (1 3) heap-array-map-def)
apply (simp add: g-eq)
apply (subst (1) heap-array-map-def)
apply (rule array-fold-fuse-aux)
by simp
qed

```

lemma *fold-fold-other-commute*:

```

fixes p::nat  $\Rightarrow$  'a ptr
and q:: nat  $\Rightarrow$  'a ptr
assumes disj:  $\bigwedge i j. i < n \implies j < m \implies \text{disjnt } (\text{ptr-span } (p \ i)) (\text{ptr-span } (q \ j))$ 
shows
  fold ( $\lambda i. w (p \ i) (f \ i)$ ) [0.. $n$ ] (fold ( $\lambda i. w (q \ i) (g \ i)$ ) [0.. $m$ ] s) =
  fold ( $\lambda i. w (q \ i) (g \ i)$ ) [0.. $m$ ] (fold ( $\lambda i. w (p \ i) (f \ i)$ ) [0.. $n$ ] s)
using disj
proof (induct n arbitrary: m s)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prem1 obtain
    disj':  $\bigwedge i j. i < n \implies j < m \implies \text{disjnt } (\text{ptr-span } (p \ i)) (\text{ptr-span } (q \ j))$  and
    m-disj:  $\bigwedge j. j < m \implies \text{disjnt } (\text{ptr-span } (p \ n)) (\text{ptr-span } (q \ j))$  by auto
  show ?case
    apply simp
    apply (subst write-fold-other-commute [OF m-disj, symmetric])
    apply assumption
    apply (subst Suc.hyps [OF disj'])
    apply assumption
    apply assumption
    apply (rule refl)
  done
qed

```

lemma *array-write-other-commute*:

```

fixes p::('a['b::array-max-count]) ptr
fixes q::('a['b::array-max-count]) ptr
assumes disj: disjnt (ptr-span p) (ptr-span q)
shows heap-array-map q g (heap-array-map p f s) = heap-array-map p f (heap-array-map q g s)
proof –
  from array-read-write-other [OF disj]
  have g-eq: g (heap-array (heap-array-map p f s) q) = g (heap-array s q) by simp

  from array-read-write-other disj
  have f-eq: f (heap-array (heap-array-map q g s) p) = f (heap-array s p)
    by (metis Int-commute disjnt-def)

```



```

show ?thesis
  apply (subst (1 3) heap-array-map-def)
  apply (simp add: g-eq f-eq)
  apply (simp add: heap-array-map-def)
  apply (rule fold-fold-other-commute)
  using disj
  by (metis Int-commute ptr-span-array-ptr-index-disj disjnt-def)
qed

sublocale array: pointer-lense heap-array heap-array-map
  apply (unfold-locales)
    apply (rule array-read-write-same)
    apply (rule array-write-same, assumption)
    apply (rule array-write-comp)
  apply (rule array-write-other-commute, assumption)
  done

end

locale pointer-two-dimensional-array-lense = pointer-array-lense r w
  for r:: 's ⇒ 'a:: array-inner-max-size ptr ⇒ 'a
  and w:: 'a ptr ⇒ ('a ⇒ 'a) ⇒ 's ⇒ 's
  begin
    sublocale outer: pointer-array-lense heap-array heap-array-map
      by (intro-locales)

  lemmas outer.heap-array-map-independent-field-commute [gen-outer-update-commute]

  end

locale valid-array-base =
  fixes vgetter:: ('t ⇒ 'a::array-outer-max-size ptr ⇒ bool)
  begin

  definition valid-array :: 't ⇒ ('a['b::array-max-count]) ptr ⇒ bool where
    [valid-array-defs]: valid-array s p = (∀ i < CARD('b). vgetter s (array-ptr-index p
    False i))

  lemma element-vgetter-eq-valid-array-eq [valid-array-conv]: vgetter s = vgetter s'
  ⇒ valid-array s = valid-array s'
    apply (rule ext)
    apply (simp add: valid-array-def)
    done

  end

locale valid-pointer-array-lense = pointer-array-lense r w +
  valid-array-base vgetter +
  write-preserves-valid vgetter w

```

```

for r:: 's ⇒ 'a:: array-outer-max-size ptr ⇒ 'a
and w:: 'a ptr ⇒ ('a ⇒ 'a) ⇒ 's ⇒ 's
and vgetter:: ('s ⇒ 'a ptr ⇒ bool)
begin

lemma setter-fold-keeps-vgetter:
  fixes p':: ('a['b::array-max-count]) ptr
  fixes p:: 'a ptr
  assumes n-bound: n ≤ CARD('b)
  shows vgetter (fold (λi. w (array-ptr-index p' False i) (λ-. f (heap-array s
p').[i])) [0..<n] s) p = vgetter s p
  using n-bound
proof (induct n arbitrary: s)
  case 0
  then show ?case by (simp )
next
  case (Suc n)
  from Suc.prem1 obtain
    suc-n-bound: Suc n ≤ CARD('b) and
    n-bound: n ≤ CARD('b) by auto
  show ?case
    apply simp
    apply (rule Suc.hyps [OF n-bound])
  done
qed

lemma heap-array-map-keeps-vgetter:
  fixes p:: ('a['b::array-max-count]) ptr
  fixes p':: ('a['b]) ptr
  shows valid-array (heap-array-map p' f s) p = valid-array s p
  by (clarsimp simp add: valid-array-def heap-array-map-def setter-fold-keeps-vgetter)

sublocale array: write-preserves-valid valid-array heap-array-map
  apply (unfold-locales)
  apply (rule heap-array-map-keeps-vgetter)
  done

lemma heap-array-map-keeps-vgetter-element[simp]:
  fixes p':: ('a['b::array-max-count]) ptr
  shows vgetter (heap-array-map p' f s) = vgetter s
  apply (rule ext)
  by (simp add: heap-array-map-def setter-fold-keeps-vgetter)

lemma getter-keeps-valid-array[simp]:
  fixes p':: 'a ptr
  shows (valid-array::'s ⇒ ('a['b::array-max-count]) ptr ⇒ bool) (w p' f s) =
valid-array s
  apply (rule ext)
  by (simp add: valid-array-def)

```

```

lemma getter-keeps-valid-array-pointwise[:
  fixes  $p':: 'a \text{ ptr}$ 
  fixes  $p :: ('a['b::\text{array-max-count}]) \text{ ptr}$ 
  shows ( $\text{valid-array}::'s \Rightarrow ('a['b::\text{array-max-count}]) \text{ ptr} \Rightarrow \text{bool}$ ) ( $w \text{ p' f s}$ )  $p =$ 
 $\text{valid-array s p}$ 
  by (simp add: valid-array-def)
end

locale valid-pointer-two-dimensional-array-lense = pointer-two-dimensional-array-lense
 $r \ w \ +$ 
  valid-array-base vgetter +
  write-preserves-valid vgetter w
  for  $r:: 's \Rightarrow 'a:: \text{array-inner-max-size ptr} \Rightarrow 'a$ 
  and  $w:: 'a \text{ ptr} \Rightarrow ('a \Rightarrow 'a) \Rightarrow 's \Rightarrow 's$ 
  and  $vgetter:: ('s \Rightarrow 'a \text{ ptr} \Rightarrow \text{bool})$ 
begin
sublocale inner: valid-pointer-array-lense  $r \ w \ vgetter$ 
  by (intro-locals)

sublocale outer: valid-pointer-array-lense heap-array heap-array-map valid-array
  by (intro-locals)

end

locale array-typ-heap-simulation =
  lense t-hrs t-hrs-update +
  read-simulation st v r t-hrs +
  write-simulation t-hrs t-hrs-update st v w +
  write-preserves-valid v w +
  valid-implies-cguard st v +
  valid-only-typ-desc-dependent t-hrs st v +
  pointer-array-lense r w +
  valid-array-base v
  for
     $st:: 's \Rightarrow 't$  and
     $r:: 't \Rightarrow 'a:: \text{array-outer-max-size ptr} \Rightarrow 'a$  and
     $w:: 'a \text{ ptr} \Rightarrow ('a \Rightarrow 'a) \Rightarrow 't \Rightarrow 't$  and
     $v:: 't \Rightarrow 'a \text{ ptr} \Rightarrow \text{bool}$  and
     $t\text{-hrs} :: 's \Rightarrow \text{heap-raw-state}$  and
     $t\text{-hrs-update}:: (\text{heap-raw-state} \Rightarrow \text{heap-raw-state}) \Rightarrow 's \Rightarrow 's$ 
begin

sublocale typ-heap-simulation
  by (intro-locals)

sublocale valid-pointer-array-lense  $r \ w \ v$ 
  by (intro-locals)

```

lemmas *typ-heap-simulation* = *typ-heap-simulation-axioms*

lemma *valid-array-implies-safe*: *valid-array* (st s) p \implies *c-guard* p
 using *valid-implies-c-guard*
 apply (*simp add: valid-array-def*)
 by (*metis TypHeapSimple.c-guard-array-c-guard array-ptr-index-simps(1)*)

lemma *heap-array-data-correct*:
 assumes *valid-arr*: *valid-array* (st s) p
 shows *heap-array* (st s) p = *h-val* (*hrs-mem* (t-*hrs* s)) p
 apply (*subst cart-eq*)
 apply *clarsimp*
 apply (*subst heap-access-Array-element'*)
 apply *simp*
 using *read-commutes valid-arr*
 by (*auto simp add: valid-array-def*)

lemma *write-fold-commutes*:
 fixes *p*: ('a['b::array-max-count]) ptr
 assumes *n-bound*: $n \leq \text{CARD}('b)$
 assumes *valid*: *valid-array* (st s) p
 shows st (t-*hrs-update* (*hrs-mem-update* (fold (λi . *heap-update* (*array-ptr-index* p False i) (x.[i]) [0.. n])) s) =
 fold (λi . *w* (*array-ptr-index* p False i) (λ -. x.[i]) [0.. n] (st s))
 using *n-bound valid*
proof (*induct n arbitrary: s*)
 case 0
 then show ?*case*
 by *simp*
next
 case (*Suc n*)
 from *Suc.prem*s **obtain** *suc-n-bound*: $\text{Suc } n \leq \text{CARD}('b)$ **and** *n-bound*: $n \leq \text{CARD}('b)$ **and** *n-bound'*: $n < \text{CARD}('b)$
 and *valid*: *valid-array* (st s) p **by** *auto*

have *vgetter*: *v* (st (t-*hrs-update* (*hrs-mem-update* (fold (λi . *heap-update* (*array-ptr-index* p False i) (x.[i]) [0.. n])) s))
 (*array-ptr-index* p False n)
 apply (*subst Suc.hyps [OF n-bound valid]*)
 apply (*subst setter-fold-keeps-vgetter [OF n-bound]*)
 using *valid n-bound'* **apply** (*simp add: valid-array-def*)
 done
show ?*case*
 apply *simp*
 apply (*subst Suc.hyps [OF n-bound valid, symmetric]*)
 apply (*subst write-commutes [symmetric]*)
 apply (*rule vgetter*)

by (*simp add: hrs-mem-update-def split-def comp-def*)
qed

lemma *heap-array-map-commutes:*

fixes *p*: ('a['b::array-max-count]) *ptr*
assumes *valid*: *valid-array* (*st s*) *p*
shows *st* (*t-hrs-update* (*hrs-mem-update* (*heap-update p x*) *s*) = *heap-array-map*
p (λ -. *x*) (*st s*)

proof –

from *valid-array-implies-safe* [*OF valid*] **have** *cgrd*: *c-guard p* .

show *?thesis*

apply (*simp add: heap-array-map-def heap-update-array-pointless* [*OF cgrd*])

apply (*rule write-fold-commutes*)

apply *simp*

apply (*rule valid*)

done

qed

lemma *write-padding-fold-commutes:*

fixes *p*: ('a['b::array-max-count]) *ptr*
assumes *n-bound*: $n \leq \text{CARD}('b)$
assumes *valid*: *valid-array* (*st s*) *p*
assumes *lbs*: $\text{length } bs = \text{CARD}('b) * \text{size-of } \text{TYPE}('a)$
shows *st* (*t-hrs-update*

(*hrs-mem-update*
(*fold*
(λ *i*. *heap-update-padding* (*array-ptr-index p False i*) (*x*.[*i*])
(*take* ($\text{size-of } \text{TYPE}('a)$) (*drop* ($i * \text{size-of } \text{TYPE}('a)$) *bs*)))
[$0..<n$]))
s) =

fold (λ *i*. *w* (*array-ptr-index p False i*) (λ -. *x*.[*i*])) [$0..<n$] (*st s*)

using *n-bound valid*

proof (*induct n arbitrary: s*)

case *0*

then show *?case*

by *simp*

next

case (*Suc n*)

from *Suc.prem*s **obtain** *suc-n-bound*: $\text{Suc } n \leq \text{CARD}('b)$ **and** *n-bound*: $n \leq$
 $\text{CARD}('b)$ **and** *n-bound'*: $n < \text{CARD}('b)$

and *valid*: *valid-array* (*st s*) *p* **by** *auto*

from *lbs suc-n-bound*

have *lbs'*: $\text{length} (\text{take} (\text{size-of } \text{TYPE}('a)) (\text{drop} (n * \text{size-of } \text{TYPE}('a)) bs)) =$
 $\text{size-of } \text{TYPE}('a)$

by (*auto simp add: less-le-mult-nat nat-move-sub-le*)

```

have vgetter: v (st (t-hrs-update
  (hrs-mem-update
    (λh. fold (λi. heap-update-padding (array-ptr-index p False i) (x.[i]
      (take (size-of TYPE('a)) (drop (i * size-of TYPE('a))
        bs))))
      [0..<n] h))
    s))
  (array-ptr-index p False n)
apply (subst Suc.hyps [OF n-bound valid] )

apply (subst setter-fold-keeps-vgetter [OF n-bound])
using valid n-bound' apply (simp add: valid-array-def)
done

show ?case apply simp
apply (subst Suc.hyps[OF n-bound valid, symmetric])

  apply (subst write-padding-commutes[where bs = ⟨take (size-of TYPE('a))
    (drop (n * size-of TYPE('a)) bs)⟩, OF vgetter lbs', symmetric])
  apply simp
  apply (simp only: hrs-mem-update-comp)
  apply (simp only: comp-def)
  done
qed

```

```

lemma heap-array-padding-map-commutes:
  fixes p:: ('a['b::array-max-count]) ptr
  assumes valid: valid-array (st s) p
  assumes bound: length bs = size-of TYPE('a['b])
  shows st (t-hrs-update (hrs-mem-update (heap-update-padding p x bs)) s) =
  heap-array-map p (λ-. x) (st s)
proof –
  from valid-array-implies-safe [OF valid] have cgrd: c-guard p .
  from bound have bound': length bs = CARD('b) * size-of TYPE('a)
  by auto
  show ?thesis
  apply (simp add: heap-array-map-def heap-update-padding-array-pointless [OF
    cgrd bound'])
  apply (simp add: write-padding-fold-commutes [OF - valid bound'])
  done
qed

```

```

lemma array-valid-same-typ-desc:
  fixes p:: ('a['b::array-max-count]) ptr
  assumes typ-eq: hrs-htd (t-hrs s) = hrs-htd (t-hrs t)
  shows valid-array (st s) p = valid-array (st t) p
  apply (simp add: valid-array-def)

```

```

using typ-eq valid-same-typ-desc by blast

sublocale array: typ-heap-simulation st heap-array heap-array-map valid-array t-hrs
t-hrs-update
  apply (unfold-locales)
    apply (rule heap-array-data-correct, assumption)
    apply (rule heap-array-padding-map-commutes, assumption, assumption)
    apply (rule valid-array-implies-safe, assumption)
    apply (rule array-valid-same-typ-desc, assumption)
  done

end

locale two-dimensional-array-typ-heap-simulation =
  lense t-hrs t-hrs-update +
  read-simulation st v r t-hrs +
  write-simulation t-hrs t-hrs-update st v w +
  write-preserves-valid v w +
  valid-implies-cguard st v +
  valid-only-typ-desc-dependent t-hrs st v +
  pointer-two-dimensional-array-lense r w
  for
    st:: 's ⇒ 't and
    r:: 't ⇒ 'a::array-inner-max-size ptr ⇒ 'a and
    w:: 'a ptr ⇒ ('a ⇒ 'a) ⇒ 't ⇒ 't and
    v:: 't ⇒ 'a ptr ⇒ bool and
    t-hrs :: 's ⇒ heap-raw-state and
    t-hrs-update:: (heap-raw-state ⇒ heap-raw-state) ⇒ 's ⇒ 's
  begin

sublocale inner: array-typ-heap-simulation st r w v t-hrs t-hrs-update
  by (intro-locales)

lemmas inner-typ-heap-simulation = inner.typ-heap-simulation

sublocale outer: array-typ-heap-simulation st heap-array heap-array-map inner.valid-array
t-hrs t-hrs-update
  by (intro-locales)

lemmas outer-typ-heap-simulation = outer.typ-heap-simulation

end

lemma root-ptr-valid-field-lvalue:
  fixes p::'a::mem-type ptr
  fixes q:: 'b::mem-type ptr
  assumes root-p: root-ptr-valid d p
  assumes root-q: root-ptr-valid d q

```

assumes *fl*: *field-lookup* (*typ-info-t* *TYPE*('b)) *f* 0 = *Some* (*t*, *k*)
assumes *other*: *typ-uinfo-t* *TYPE*('a) ≠ *typ-uinfo-t* *TYPE*('b)
shows *ptr-val* *p* = $\&(q \rightarrow f) \longleftrightarrow$ *False*
proof
assume *p*: *ptr-val* *p* = $\&(q \rightarrow f)$
from *field-tag-sub* [*OF fl*] **have** $\{\&(q \rightarrow f)..+size-td\ t\} \subseteq$ *ptr-span* *q*.
moreover
from *root-ptr-valid-cases* [*OF root-p root-q*] *other* **have** *ptr-span* *p* ∩ *ptr-span* *q*
= {} **by** *blast*
ultimately
show *False*
using *p*
by (*metis* (*mono-tags*, *lifting*) *disjoint-iff* *field-lookup-wf-size-desc-gt fl* *intvl-inter*
intvl-start-inter *mem-type-self* *subset-iff* *wf-size-desc*)
qed *simp*

lemma *root-ptr-valid-field-lvalue'*:
fixes *q*:: 'b::*mem-type* *ptr*
assumes *root-p*: *root-ptr-valid* *d* (*PTR* ('a::*mem-type*)($\&(q \rightarrow f)$))
assumes *root-q*: *root-ptr-valid* *d* *q*
assumes *fl*: *field-lookup* (*typ-info-t* *TYPE*('b)) *f* 0 = *Some* (*t*, *k*)
assumes *other*: *typ-uinfo-t* *TYPE*('a) ≠ *typ-uinfo-t* *TYPE*('b)
shows *False*
using *root-ptr-valid-field-lvalue* [*OF root-p root-q fl other*]
by *simp*

lemma *root-ptr-valid-array-ptr-index-dim1*:
fixes *q*:: ('a::*array-outer-max-size*['c::*array-max-count*]) *ptr*
assumes *root-p*: *root-ptr-valid* *d* (*array-ptr-index* *q* *False* *i*)
assumes *root-q*: *root-ptr-valid* *d* *q*
assumes *other*: *typ-uinfo-t* *TYPE*('a) ≠ *typ-uinfo-t* *TYPE*('a['c])
assumes *i-bound*: *i* < *CARD*('c)
shows *False*
proof –
from *field-lookup-array* [*OF i-bound*]
obtain *t k* **where**
fl: *field-lookup* (*typ-info-t* *TYPE*('a['c])) [*replicate* *i* (*CHR* "1")] 0 = *Some* (*t*,
k)
by *blast*
from *root-ptr-valid-field-lvalue* [*OF root-p root-q fl other*] *i-bound*
show *?thesis*
by (*simp* *add*: *array-ptr-index-field-lvalue-conv*)
qed

lemma *root-ptr-valid-field-lvalue-array-ptr-index-dim1*:
fixes *q*:: 'b::*mem-type* *ptr*
assumes *root-p*: *root-ptr-valid* *d* (*array-ptr-index* (*PTR*('a['c]) $\&(q \rightarrow f)$) *False* *i*)

assumes *root-q*: *root-ptr-valid* *d q*
assumes *other*: *typ-uinfo-t* *TYPE('a)* \neq *typ-uinfo-t* *TYPE('b)*
assumes *i-bound*: $i < \text{CARD}('c)$
assumes *fl*: *field-lookup* (*typ-info-t* *TYPE('b)*) *f* $0 = \text{Some}$ (*t*, *k*)
assumes *t*: *export-uinfo* *t* = *typ-uinfo-t* (*TYPE('a::array-outer-max-size*[*'c::array-max-count*]))
shows *False*
proof –
from *field-lookup-array* [*OF i-bound*]
obtain *s n* **where**
fl-i: *field-lookup* (*typ-info-t* *TYPE('a['c])*) [*replicate i* (*CHR "1"*)] *k* = *Some*
(*s*, *n*)
by *blast*

from *fl-i t* **obtain** *s' n'* **where**
fl-i': *field-lookup* *t* [*replicate i* (*CHR "1"*)] *k* = *Some* (*s'*, *n'*)
by (*metis* (*no-types*, *lifting*) *Padding-Equivalence.field-lookup-export-uinfo-Some-rev*
field-lookup-export-uinfo-Some fold-typ-uinfo-t)

from *field-lookup-prefix-Some''(1)*[*rule-format*, **where** *f=f* **and** *g=[replicate i*
CHR "1"]
and *m=0*, *OF fl*]
have *fl-app*: *field-lookup* (*typ-info-t* *TYPE('b)*) (*f* @ [*replicate i* *CHR "1"*]) $0 =$
Some (*s'*, *n'*)
by (*simp add: fl-update fl-i'*)
have *conv*: $\&(PTR('a['c]) \&(q \rightarrow f) \rightarrow [replicate i CHR "1"])$ = $\&(q \rightarrow f @ [replicate$
*i CHR "1"])
apply (*subst field-lvalue-append*)
apply *simp-all*
apply (*rule field-lookup-field-ti'*)
apply (*rule fl*)
apply (*simp add: typ-uinfo-t-def i-bound t*)
apply (*rule field-lookup-field-ti'*)
apply (*rule field-lookup-array* [*OF i-bound*])
done

from *root-ptr-valid-field-lvalue* [*OF root-p root-q fl-app other*] **show** *False*
by (*simp add: array-ptr-index-field-lvalue-conv i-bound conv*)
qed*

lemma *root-ptr-valid-field-lvalue-array-ptr-index-dim1'*:
fixes *q*:: *'b::mem-type ptr*
assumes *root-p*: *root-ptr-valid* *d* (*array-ptr-index* (*PTR(('a::array-outer-max-size*[*'c::array-max-count*]))
 $\&(q \rightarrow f)$) *False i*)
assumes *root-q*: *root-ptr-valid* *d q*
assumes *other*: *typ-uinfo-t* *TYPE('a)* \neq *typ-uinfo-t* *TYPE('b)*
assumes *i-bound*: $i < \text{CARD}('c)$
assumes *fl*: *field-lookup* (*typ-info-t* *TYPE('b)*) *f* $0 = \text{Some}$ (*adjust-ti* (*typ-info-t*
(*TYPE('a['c])*)) *acc upd*, *k*)
assumes *fg*: *fg-cons* *acc upd*

shows *False*
using *root-ptr-valid-field-lvalue-array-ptr-index-dim1* [*OF* *root-p* *root-q* *other* *i-bound* *fl*]
by (*simp add: fg*)

lemma *root-ptr-valid-array-ptr-index-dim2*:

fixes *q*:: ((*'a*::array-inner-max-size[*'c*::array-max-count])[*'d*::array-max-count]) *ptr*
assumes *root-p*: *root-ptr-valid* *d* (*array-ptr-index* (*array-ptr-index* *q* *False* *i*) *False* *j*)

assumes *root-q*: *root-ptr-valid* *d* *q*
assumes *other*: *typ-ufinfo-t* *TYPE*('a') \neq *typ-ufinfo-t* *TYPE*('a'[*'c*][*'d*])
assumes *i-bound*: *i* < *CARD*('d')
assumes *j-bound*: *j* < *CARD*('c')
shows *False*

proof –

from *field-lookup-array* [*OF* *i-bound*, of 0, **where** *'a*=*'a*[*'c*]]
have *fl-i*: *field-lookup* (*typ-ufinfo-t* *TYPE*((*'a*[*'c*])[*'d*])) [*replicate* *i* *CHR* "*'1*'"] 0 =
Some (*adjust-ti* (*typ-ufinfo-t* *TYPE*('a'[*'c*])) ($\lambda x. x.[i]$) ($\lambda x f. \text{Arrays.update}$ *f* *i* *x*),

i * *size-of* *TYPE*('a'[*'c*])) **by** *simp*

from *field-lookup-array* [*OF* *j-bound*, of *i* * *size-of* *TYPE*('a'[*'c*]), **where** *'a*=*'a*]
have *fl-j*: *field-lookup* (*typ-ufinfo-t* *TYPE*('a'[*'c*])) [*replicate* *j* *CHR* "*'1*'"] (*i* * *size-of* *TYPE*('a'[*'c*])) =

Some (*adjust-ti* (*typ-ufinfo-t* *TYPE*('a')) ($\lambda x. x.[j]$) ($\lambda x f. \text{Arrays.update}$ *f* *j* *x*),

i * *size-of* *TYPE*('a'[*'c*]) + *j* * *size-of* *TYPE*('a')) **by** *simp*

have *append*: $\&(PTR('a'[*'c*]) \&(q \rightarrow [replicate\ i\ CHR\ "'1'"] \rightarrow [replicate\ j\ CHR\ "'1'"])$ =

$\&(q \rightarrow [replicate\ i\ CHR\ "'1'"] , replicate\ j\ CHR\ "'1'"])$

apply (*subst field-lvalue-append*)

apply *simp-all*

apply (*rule field-lookup-field-ti'*)

apply (*rule fl-i*)

apply (*simp add: typ-ufinfo-t-def i-bound*)

apply (*rule field-lookup-field-ti'*)

apply (*rule field-lookup-array* [*OF* *j-bound*])

done

have *sz-eq*: *size-of* *TYPE*('a'[*'c*]) = (*CARD*('c') * *size-of* *TYPE*('a'))

using *size-of-array* **by** *blast*

from *field-lookup-prefix-Some''(1)*[*rule-format*, **where** *f*=[*replicate* *i* *CHR* "*'1*'"]
and *g*=[*replicate* *j* *CHR* "*'1*'"]

and *t* = (*typ-ufinfo-t* *TYPE*((*'a*[*'c*])[*'d*])) **and** *m*=0, *OF* *fl-i*

obtain *t* *k*

where *fl*: *field-lookup* (*typ-ufinfo-t* *TYPE*('a'[*'c*][*'d*])) [*replicate* *i* *CHR* "*'1*'"] , *replicate* *j* *CHR* "*'1*'"] 0 = *Some* (*t*, *k*)

apply (*simp add: i-bound*)

apply (*simp add: fl-update sz-eq*)

```

    using fl-j [simplified sz-eq]
    apply simp
    done
  show ?thesis
    using root-ptr-valid-field-lvalue [OF root-p root-q fl other] i-bound j-bound
    apply (simp add: array-ptr-index-field-lvalue-conv append )
    done
qed

lemma root-ptr-valid-field-lvalue-array-ptr-index-dim2:
  fixes q: 'b::mem-type ptr
  assumes root-p: root-ptr-valid d (array-ptr-index (array-ptr-index (PTR('a['c]['d])
&(q→f)) False i) False j)
  assumes root-q: root-ptr-valid d q
  assumes other: typ-ufinfo-t TYPE('a) ≠ typ-ufinfo-t TYPE('b)
  assumes i-bound: i < CARD('d)
  assumes j-bound: j < CARD('c)
  assumes fl: field-lookup (typ-ufinfo-t TYPE('b)) f 0 = Some (t, k)
  assumes t: export-ufinfo t = typ-ufinfo-t (TYPE(('a::array-inner-max-size['c::array-max-count]))['d::array-max-count])
  shows False
proof -
  from field-lookup-array [OF i-bound]
  obtain s n where
    fl-i: field-lookup (typ-ufinfo-t TYPE('a['c]['d])) [replicate i (CHR "1")] k = Some
(s, n) and
    s: s = adjust-ti (typ-ufinfo-t TYPE('a['c])) (λx. x.[i]) (λx f. Arrays.update f i x)
  by blast

  from fl-i t s obtain s' n' where
    fl-i': field-lookup t [replicate i (CHR "1")] k = Some (s', n') and
    s': export-ufinfo s' = typ-ufinfo-t TYPE('a['c])
  by (smt (verit, best) Padding-Equivalence.field-lookup-export-ufinfo-Some-rev
    export-tag-adjust-ti(1) fg-cons-array field-lookup-array fold-typ-ufinfo-t i-bound
wf-fd field-lookup-typ-ufinfo-t-Some)

  from field-lookup-prefix-Some''(1)[rule-format, where f=f and g=[replicate i
CHR "1']]
  and m=0, OF fl]
  have fl-app1: field-lookup (typ-ufinfo-t TYPE('b)) (f @ [replicate i CHR "1"]) 0
= Some (s', n')
  by (simp add: fl-update fl-i')

  from field-lookup-array [OF j-bound]
  obtain s'' n'' where
    fl-j: field-lookup (typ-ufinfo-t TYPE('a['c])) [replicate j (CHR "1")] n' = Some
(s'', n'')
  by blast

  from fl-j s' obtain s''' n''' where

```

fl-j: *field-lookup* *s'* [*replicate j (CHR "1'")*] *n' = Some (s''', n''')*
by (*metis (no-types, lifting) CTypes.field-lookup-export-uinfo-Some-rev field-lookup-export-uinfo-Some fold-ty-p-uinfo-t*)

from *field-lookup-prefix-Some''(1)*[*rule-format*, **where** *f=f @ [replicate i CHR "1'"]* **and** *g=[replicate j CHR "1'"]*
and *m=0, OF fl-app1*
have *fl-app2: field-lookup (typ-uinfo-t TYPE('b)) (f @ [replicate i CHR "1'", replicate j CHR "1'"]) 0 = Some (s''', n''')*
by (*simp add: fl-update fl-j'*)

have *conv: &(PTR('a['c]) &(PTR('a['c]['d]) &(q→f)→[replicate i CHR "1'"]→[replicate j CHR "1'"])* =

&(q→f @ [replicate i CHR "1'", replicate j CHR "1'"])
apply (*subst field-lvalue-append*)
apply (*rule field-lookup-field-ti'*)
apply (*rule fl*)
apply (*simp add: typ-uinfo-t-def i-bound t*)
apply (*rule field-lookup-field-ti'*)
apply (*rule field-lookup-offset-shift'*)
apply (*rule fl-i*)

apply (*subst field-lvalue-append*)
apply (*rule field-lookup-field-ti'*)
apply (*rule field-lookup-offset-shift'*)
apply (*rule fl-app1*)
apply (*simp add: s'*)
apply (*rule field-lookup-field-ti'*)
apply (*rule field-lookup-offset-shift'*)
apply (*rule fl-j*)
apply *simp*
done

from *root-ptr-valid-field-lvalue [OF root-p root-q fl-app2 other] show False*
by (*simp add: array-ptr-index-field-lvalue-conv i-bound j-bound conv*)

qed

context *open-types*
begin

definition

typ-heap-simulation-of-field (st::'s ⇒ 't) t-hrs t-hrs-update heap-typing-upd f' r' w' ←→

(∀ d p f. heap-typing-upd d o w' p f = w' p f o heap-typing-upd d) ∧
(∀ t u n.

field-ti TYPE('a::mem-type) f' = Some t →

field-lookup (typ-uinfo-t TYPE('a)) f' 0 = Some (u, n) →

partial-pointer-lense (merge-ti t) r' w' ∧

(∀ (p::'a ptr) s. (∀ a∈ptr-span p. root-ptr-valid (hrs-htd (t-hrs s)) (PTR(stack-byte) a)) →

```

    r' (st s) p ZERO('a) = ZERO('a) ∧
    (∀ (p::'a ptr) x h. ptr-valid-u u (hrs-htd (t-hrs h)) &(p→f') →
    st (t-hrs-update (hrs-mem-update (heap-upd-list (size-td u) &(p→f') (access-ti
    t x))) h) =
    w' p x (st h)))

```

end

definition

```

pointer-writer-disjnt ::
('a::c-type ptr ⇒ 't upd) ⇒ ('b::c-type ptr ⇒ 't upd) ⇒ bool

```

where

```

pointer-writer-disjnt f g ←→
(∀ p q. disjnt (ptr-span p) (ptr-span q) → f p ∘ g q = g q ∘ f p)

```

definition

```

pointer-writer-disjnt-eq ::
('a::c-type ptr ⇒ 'x ⇒ 't upd) ⇒ ('b::c-type ptr ⇒ 'y ⇒ 't upd) ⇒ bool

```

where

```

pointer-writer-disjnt-eq f g ←→ (∀ p q x y.
disjnt (ptr-span p) (ptr-span q) ∨ ptr-val p = ptr-val q → f p x ∘ g q y = g q
y ∘ f p x)

```

named-theorems pointer-writer-disjnt-intros

lemma pointer-writer-disjnt-eq:

```

assumes nm: ∃ n1 n2.
  field-lookup (typ-uinfo-t TYPE('a::mem-type)) f1 0 =
  Some (typ-uinfo-t TYPE('b::mem-type), n1) ∧
  field-lookup (typ-uinfo-t TYPE('a)) f2 0 = Some (typ-uinfo-t TYPE('c::mem-type),
n2)

```

```

assumes w1-w2: ∧ x y. pointer-writer-disjnt (λp. w1 p x) (λq. w2 q y)

```

```

shows disj-fn f1 f2 → pointer-writer-disjnt-eq

```

```

(λ(p::'a ptr). w1 (PTR('b) &(p→f1)))

```

```

(λ(p::'a ptr). w2 (PTR('c) &(p→f2))) (is ?disj → ?goal)

```

proof

```

assume ?disj

```

```

from nm obtain n1 n2 where fl:

```

```

  field-lookup (typ-uinfo-t TYPE('a::mem-type)) f1 0 =

```

```

  Some (typ-uinfo-t TYPE('b::mem-type), n1)

```

```

  field-lookup (typ-uinfo-t TYPE('a)) f2 0 = Some (typ-uinfo-t TYPE('c::mem-type),
n2)

```

```

by auto

```

```

from fl[THEN field-tag-sub']

```

```

have ptr-span (PTR('b) &(p→f1)) ⊆ ptr-span p ∧ ptr-span (PTR('c) &(q→f2))

```

```

⊆ ptr-span q

```

```

for p q :: 'a ptr

```

```

by (auto intro: field-tag-sub' simp: size-of-def del: subsetI)

```

```

then have ne[simp]: disjnt (ptr-span p) (ptr-span q) ⇒

```

```

    disjnt (ptr-span (PTR('b) &(p→f1))) (ptr-span (PTR('c) &(q→f2))) for p q ::
'a ptr
  by (blast intro: disjnt-subset1 disjnt-subset2)
moreover
from fl obtain u1 u2 where fl-t:
  field-lookup (typ-info-t TYPE('a)) f1 0 = Some (u1, n1)
  field-lookup (typ-info-t TYPE('a)) f2 0 = Some (u2, n2)
and export: export-uinfo u1 = typ-uinfo-t TYPE('b) export-uinfo u2 = typ-uinfo-t
TYPE('c)
  by (auto dest!: field-lookup-export-uinfo-Some-rev simp: typ-uinfo-t-def)
from fa-fu-lookup-disj-interD[OF fl-t ‹disj-fn f1 f2›]
have disjnt (ptr-span (PTR('b) &(p→f1))) (ptr-span (PTR('c) &(p→f2))) for
p :: 'a ptr
  apply (simp add: size-of-fold disjnt-def)
  apply (simp add: size-of-def field-lvalue-def field-offset-def field-offset-untyped-def
fl
      intvl-disj-offset
      flip: typ-uinfo-size export)
done
ultimately show ?goal using w1-w2
  by (auto simp: pointer-writer-disjnt-def pointer-writer-disjnt-eq-def)
qed

```

```

lemma pointer-writer-disjnt-sym:
  pointer-writer-disjnt f g  $\implies$  pointer-writer-disjnt g f
  by (auto simp add: pointer-writer-disjnt-def disjnt-sym)

```

```

lemma pointer-writer-disjnt-id-left:
  pointer-writer-disjnt ( $\lambda p h. h$ ) w
  by (simp add: pointer-writer-disjnt-def fun-eq-iff)

```

```

lemma pointer-writer-disjnt-id-left':
  pointer-writer-disjnt ( $\lambda p. id$ ) w
  by (simp add: pointer-writer-disjnt-def fun-eq-iff)

```

```

lemma pointer-writer-disjnt-comp-left:
  pointer-writer-disjnt w1 w  $\implies$  pointer-writer-disjnt w2 w  $\implies$ 
  pointer-writer-disjnt ( $\lambda p h. w1 p (w2 p h)$ ) w
  by (simp add: pointer-writer-disjnt-def fun-eq-iff)

```

```

lemma pointer-writer-disjnt-comp-left':
  pointer-writer-disjnt w1 w  $\implies$  pointer-writer-disjnt w2 w  $\implies$ 
  pointer-writer-disjnt ( $\lambda p. w1 p \circ w2 p$ ) w
  by (simp add: pointer-writer-disjnt-def fun-eq-iff)

```

```

lemma pointer-writer-disjnt-fold-left:
  list-all ( $\lambda w'. pointer-writer-disjnt (f w') w$ ) ws  $\implies$ 
  pointer-writer-disjnt ( $\lambda p. fold (\lambda w. f w p) ws$ ) w
  by (induction ws) (auto intro: pointer-writer-disjnt-comp-left' pointer-writer-disjnt-id-left')

```

lemma *pointer-writer-disjntI*:

$(\bigwedge p q h. w1 p (w2 q h) = w2 q (w1 p h)) \implies \text{pointer-writer-disjnt } w1 w2$
by (*auto simp: pointer-writer-disjnt-def*)

lemma *pointer-writer-disjntD*:

$\text{pointer-writer-disjnt } w1 w2 \implies \text{disjnt } (\text{ptr-span } p) (\text{ptr-span } q) \implies$
 $w1 p (w2 q h) = w2 q (w1 p h)$
by (*auto simp: pointer-writer-disjnt-def fun-eq-iff*)

lemma *pointer-writer-disjnt-ptr-map-left*:

fixes $w1 :: 'a::\text{mem-type ptr} \Rightarrow 's \text{ upd}$ **and** $w2 :: 'b::\text{c-type ptr} \Rightarrow 's \text{ upd}$
assumes $w1\text{-}w2$: *pointer-writer-disjnt* $w1 w2$
assumes $\bigwedge (p::'c::\text{c-type ptr}). \text{ptr-span } (F p) \subseteq \text{ptr-span } p$
shows *pointer-writer-disjnt* $(\lambda(p::'c \text{ ptr}). w1 (F p)) w2$
using *assms* **by** (*auto simp: pointer-writer-disjnt-def disjnt-subset1*)

lemma *pointer-writer-disjnt-ptr-left*:

fixes $w1 :: 'a::\text{mem-type ptr} \Rightarrow 's \text{ upd}$ **and** $w2 :: 'b::\text{mem-type ptr} \Rightarrow 's \text{ upd}$
assumes $w1\text{-}w2$: *pointer-writer-disjnt* $w1 w2$
assumes n :
 $\exists n. \text{field-lookup } (\text{typ-uinfo-t } \text{TYPE}('c::\text{mem-type})) f 0 = \text{Some } (\text{typ-uinfo-t } \text{TYPE}('a), n)$
shows *pointer-writer-disjnt* $(\lambda(p::'c \text{ ptr}). w1 (\text{PTR}('a) \ \&(p \rightarrow f))) w2$
proof (*rule pointer-writer-disjnt-ptr-map-left[OF w1-w2]*)
show $\text{ptr-span } (\text{PTR}('a) \ \&(p \rightarrow f)) \subseteq \text{ptr-span } p$ **for** $p :: 'c \text{ ptr}$
using *field-tag-sub'* [**where** $'a = 'c$, *of f typ-uinfo-t TYPE('a) - p*] n
by (*auto simp: size-of-def*)
qed

lemma *pointer-writer-disjnt-ptr-corce*:

fixes $w1 :: 'a::\text{mem-type ptr} \Rightarrow 's \text{ upd}$ **and** $w2 :: 'b::\text{mem-type ptr} \Rightarrow 's \text{ upd}$
assumes $w1\text{-}w2$: *pointer-writer-disjnt* $w1 w2$
and *size*: $\text{size-of } \text{TYPE}('a) \leq \text{size-of } \text{TYPE}('c)$
shows *pointer-writer-disjnt* $(\lambda(p::'c::\text{c-type ptr}). w1 (\text{PTR-COERCE}('c \rightarrow 'a) p)) w2$
by (*rule pointer-writer-disjnt-ptr-map-left[OF w1-w2]*)
(simp add: intvl-start-le size)

lemma *pointer-writer-disjnt-ptr-corce-signed*:

fixes $w1 :: 'a::\text{len8 word ptr} \Rightarrow 's \text{ upd}$ **and** $w2 :: 'b::\text{mem-type ptr} \Rightarrow 's \text{ upd}$
assumes $w1\text{-}w2$: *pointer-writer-disjnt* $w1 w2$
shows *pointer-writer-disjnt* $(\lambda(p::'a \text{ signed word ptr}). w1 (\text{PTR-COERCE}('a \text{ signed word} \rightarrow 'a \text{ word}) p)) w2$
using $w1\text{-}w2$ **by** (*rule pointer-writer-disjnt-ptr-corce*) (*simp add: size-of-signed-word*)

lemma *pointer-writer-disjnt-ptr-corce-signed'*:

fixes $w1 :: 'a::\text{len8 word ptr} \Rightarrow 's \text{ upd}$ **and** $w2 :: 'b::\text{mem-type ptr} \Rightarrow 's \text{ upd}$
shows *pointer-writer-disjnt* $w2 w1 \implies$

pointer-writer-disjnt $w2$ ($\lambda(p::'a \text{ signed word ptr}).$
 $w1$ (*PTR-COERCE*('a signed word \rightarrow 'a word) p))
using *pointer-writer-disjnt-ptr-corce-signed*[of $w1$ $w2$] **by** (*simp add: pointer-writer-disjnt-sym*)

lemma (**in** *pointer-lense*) *pointer-writer-disjnt-replace-by-const*:
 $(\bigwedge x. \text{pointer-writer-disjnt } (\lambda p. w p (\lambda-. x)) w') \implies \text{pointer-writer-disjnt } (\lambda p. w$
 $p f) w'$
by (*auto simp add: pointer-writer-disjnt-def fun-eq-iff*)
(metis (no-types, lifting) read-write-same write-cong write-read)

lemma (**in** *pointer-lense*) *read-commute-of-pointer-writer-disjnt*:
assumes $w': \bigwedge f. \text{pointer-writer-disjnt } (\lambda p. w p f) w'$
assumes $p\text{-}q: \text{disjnt } (\text{ptr-span } p) (\text{ptr-span } q)$
shows $r (w' q h) p = r h p$
apply (*subst write-same*[of $\lambda-. r h p h p$, *OF refl, symmetric*])
apply (*subst pointer-writer-disjntD*[*OF w' p-q, symmetric*])
apply (*subst read-write-same*)
apply (*rule refl*)
done

lemma (**in** *pointer-lense*) *read-commute-of-pointer-writer-disjnt'*:
assumes $w': \bigwedge f. \text{pointer-writer-disjnt } w' (\lambda p. w p f)$
assumes $p\text{-}q: \text{disjnt } (\text{ptr-span } p) (\text{ptr-span } q)$
shows $r (w' q h) p = r h p$
using *read-commute-of-pointer-writer-disjnt*[*OF w' [THEN pointer-writer-disjnt-sym]*
 $p\text{-}q$].

lemma (**in** *pointer-lense*) *read-commute-of-commute*:
assumes $w': \bigwedge p f. w p f o w' = w' o w p f$
shows $r (w' h) p = r h p$
using *read-write-same*[of $p \lambda-. r h p w' h$] w'
by (*subst write-same*[of $\lambda-. r h p h p$, *OF refl, symmetric*]) (*simp add: fun-eq-iff*)

lemma (**in** *pointer-lense*) *read-commute-of-commute'*:
assumes $w': \bigwedge p f. w' o w p f = w p f o w'$
shows $r (w' h) p = r h p$
using *read-commute-of-commute*[*OF w' [symmetric]*].

lemma (**in** *lense*) *get-commute-of-commute*:
assumes $w': \bigwedge f. w' o \text{upd } f = \text{upd } f o w'$
shows $\text{get } (w' h) = \text{get } h$
using *get-upd*[of $\lambda-. \text{get } h w' h$] w' [*symmetric*]
by (*subst upd-same*[of $\lambda-. \text{get } h h$, *OF refl, symmetric*]) (*simp add: fun-eq-iff del: get-upd*)

lemma (**in** *pointer-lense*) *pointer-writer-disjnt-replace-dep-by-const*:
assumes $*$: $\bigwedge x. \text{pointer-writer-disjnt } (\lambda p. w p x) w'$
shows *pointer-writer-disjnt* ($\lambda p s. w p (f (r s p)) s$) w'
by (*auto simp add: pointer-writer-disjnt-def fun-eq-iff*)

read-commute-of-pointer-writer-disjnt[*OF* *]
pointer-writer-disjntD[*OF* *])

lemma (in *pointer-lense*) *pointer-writer-disjnt-upd*[*pointer-writer-disjnt-intros*]:
pointer-writer-disjnt ($\lambda p. w p x$) ($\lambda p. w p y$)
by (*simp add: pointer-writer-disjnt-def write-other-commute disjnt-def fun-eq-iff*)

lemma (in *partial-pointer-lense*) *read-commute-of-pointer-writer-disjnt*:
assumes w' : $\bigwedge f. \text{pointer-writer-disjnt } (\lambda p. w p f) w'$
assumes p - q : *disjnt* (*ptr-span* p) (*ptr-span* q)
shows $r (w' q h) p y = r h p y$
by (*metis m-r p-q pointer-writer-disjntD r-w w' w-r*)

lemma *pointer-lense-upd-fun-of-lense*:
fixes *get upd* **assumes** *lense get upd*
shows *pointer-lense get* ($\lambda p f. \text{upd } (\text{upd-fun } p f)$)
proof –
interpret *lense get upd* **by fact**
show *?thesis*
proof qed (*simp-all add: upd-same upd-fun.upd-same disj-ptr-span-ptr-neq
upd-fun-commute Int-commute disjnt-def*)
qed

locale *open-types-heap-typing-state* = *open-types* \mathcal{T} +
heap-typing-state *heap-typing* *heap-typing-upd*
for
 \mathcal{T} **and**
heap-typing :: $'t \Rightarrow \text{heap-typ-desc}$ **and**
heap-typing-upd :: ($\text{heap-typ-desc} \Rightarrow \text{heap-typ-desc}$) $\Rightarrow 't \Rightarrow 't$

locale *typ-heap-simulation-open-types* =
typ-heap-simulation *st r w v t-hrs t-hrs-update* +
open-types-heap-typing-state \mathcal{T} *heap-typing* *heap-typing-upd*
for
 \mathcal{T} **and**
st:: $'s \Rightarrow 't$ **and**
r:: $'t \Rightarrow ('a::\{\text{xmem-type, stack-type}\}) \text{ptr} \Rightarrow 'a$ **and**
w:: $'a \text{ptr} \Rightarrow ('a \Rightarrow 'a) \Rightarrow 't \Rightarrow 't$ **and**
v:: $'t \Rightarrow 'a \text{ptr} \Rightarrow \text{bool}$ **and**
t-hrs :: $'s \Rightarrow \text{heap-raw-state}$ **and**
t-hrs-update:: ($\text{heap-raw-state} \Rightarrow \text{heap-raw-state}$) $\Rightarrow 's \Rightarrow 's$ **and**
heap-typing :: $'t \Rightarrow \text{heap-typ-desc}$ **and**
heap-typing-upd :: ($\text{heap-typ-desc} \Rightarrow \text{heap-typ-desc}$) $\Rightarrow 't \Rightarrow 't$ +
assumes *heap-typing-commutes*[*simp*]: *heap-typing* (*st* s) = *hrs-htd* (*t-hrs* s)
assumes *heap-typing-upd-write-commute*:
 $\bigwedge p f h d. \text{heap-typing-upd } d (w p f h) = w p f (\text{heap-typing-upd } d h)$
assumes *ptr-valid-imp-v*: $\bigwedge p h. \text{ptr-valid } (\text{heap-typing } h) p \Longrightarrow v h p$
assumes *sim-stack-stack-byte-zero*:

$\bigwedge p s. \forall a \in \text{ptr-span } p. \text{root-ptr-valid } (\text{hrs-htd } (t\text{-hrs } s)) (\text{PTR}(\text{stack-byte}) a) \implies$
 $r (st\ s) p = \text{ZERO}(a)$

begin

lemma *heap-typing-write[simp]*: *heap-typing* (*w p f h*) = *heap-typing* *h*
apply (*subst upd-same[symmetric, of λ-. heap-typing h, OF refl]*)
apply (*subst heap-typing-upd-write-commute[symmetric]*)
apply (*rule get-upd*)
done

lemma *heap-typing-upd[simp]*: *r* (*heap-typing-upd d h*) = *r h*
apply (*rule ext*)
subgoal for *p*
apply (*subst write-same[symmetric, of λ-. r h p, OF refl]*)
apply (*subst heap-typing-upd-write-commute*)
apply (*rule read-write-same*)
done
done

lemma *unchanged-typing-root-ptr-valid-preserved*:
unchanged-typing-on A t1 t2 \implies *ptr-span p* \subseteq *A*
 \implies *root-ptr-valid* (*heap-typing t1*) *p* = *root-ptr-valid* (*heap-typing t2*) *p*
by (*simp add: unchanged-typing-on-def equal-on-def intvlI*
root-ptr-valid-def size-of-tag subsetD valid-root-footprint-def)

lemma *unchanged-typing-ptr-valid-preserved*:
assumes *unch*: *unchanged-typing-on UNIV t1 t2*
shows *ptr-valid* (*heap-typing t1*) *p* = *ptr-valid* (*heap-typing t2*) *p*
proof –
from *unch* **have** *heap-typing t1* = *heap-typing t2*
by (*simp add: fun-eq-iff unchanged-typing-on-def equal-on-def*)
thus *?thesis*
by *simp*
qed

lemma *unchanged-typing-on-write [unchanged-typing-on-simps]*:
unchanged-typing-on A t (*w p f t*)
using *heap-typing-upd-write-commute*
by (*simp add: unchanged-typing-on-def*)

lemma *heap-typing-fold-upd-write*: *heap-typing* (*fold* ($\lambda i. w (x\ i) (g\ i)$) [*0..<n*] *t*)
= *heap-typing t*
proof (*induct n*)
case *0*
then show *?case* **by** (*simp add: heap-typing-upd-write-commute*)
next
case (*Suc n*)
then show *?case* **by** (*simp add: heap-typing-upd-write-commute*)
qed

end

lemma *distinct-prop-conj*:

$distinct-prop (\lambda a b. R a b \wedge P a b) xs \longleftrightarrow distinct-prop R xs \wedge distinct-prop P xs$

by (*auto simp: distinct-prop-iff-nth*)

lemma *distinct-prop-zip-cons*:

$list-all (\lambda(c, d). P a b c d) (zip as bs) \implies$
 $distinct-prop (\lambda(a, b) (c, d). P a b c d) (zip as bs) \implies$
 $distinct-prop (\lambda(a, b) (c, d). P a b c d) (zip (a\#as) (b\#bs))$
by (*simp add: list-all-iff*)

lemma *distinct-prop-zip-nil*:

$distinct-prop (\lambda(a, b) (c, d). P a b c d) (zip [] [])$
by (*simp add: list-all-iff*)

lemma *list-all-zip-cons*:

$P a b \implies list-all (\lambda(a, b). P a b) (zip as bs) \implies$
 $list-all (\lambda(a, b). P a b) (zip (a\#as) (b\#bs))$
by *simp*

lemma *list-all-zip-nil*:

$list-all (\lambda(a, b). P a b) (zip [] [])$
by *simp*

named-theorems *disjnt-heap-fields-comp*

context *open-types*

begin

lemma *typ-heap-simulationI-part-addressable*:

fixes $R :: 't \Rightarrow 'a::\{xmem-type, stack-type\} ptr \Rightarrow 'a$

assumes $hrs: heap-typing-simulation \mathcal{T} hrs hrs-upd heap-typing heap-typing-upd$
 l

assumes $fs: map-of \mathcal{T} (typ-uinfo-t TYPE('a)) = Some fs$

assumes $lense\ g\ u$

assumes $len[simp]: length\ rs = length\ fs\ length\ ws = length\ fs$

assumes $*$:

$list-all (\lambda(f, r, w). typ-heap-simulation-of-field\ l\ hrs\ hrs-upd\ heap-typing-upd$
 $f\ r\ w) (zip\ fs\ (zip\ rs\ ws))$

and $**:$ $distinct-prop (\lambda(f1, w1) (f2, w2).$

$disj-fn\ f1\ f2 \longrightarrow pointer-writer-disjnt-eq\ w1\ w2)$

$(zip\ fs\ ws)$

and $ws-u:$ $list-all (\lambda w. \forall p a f. w\ p\ a \circ u\ f = u\ f \circ w\ p\ a)\ ws$

and $l-u:$ $\bigwedge(p::'a\ ptr) (x::'a) (s::'b).$

$PTR-VALID('a) (hrs-htd (hrs\ s))\ p \implies$

l ($hrs\text{-upd}$ ($write\text{-dedicated-heap}$ p x) s) = u ($upd\text{-fun}$ p ($\lambda old.$ $merge\text{-addressable-fields}$ old x)) (l s)

and $r\text{-stack}$:

$\bigwedge p$ $s.$ $\forall a \in ptr\text{-span } p.$ $root\text{-ptr-valid}$ ($hrs\text{-htd}$ (hrs s)) ($PTR(stack\text{-byte})$ a) \implies
 g (l s) p = $ZERO(-)$

and $heap\text{-typing-u}$: $\bigwedge x$ d $h.$ $heap\text{-typing-upd}$ d (u x h) = u x ($heap\text{-typing-upd}$ d h)

assumes V :
 $\bigwedge h$ $p.$ V h p \longleftrightarrow $PTR\text{-VALID}'(a)$ ($heap\text{-typing}$ h) p

assumes R :
 $\bigwedge h$ $p.$ R h p = $fold$ ($\lambda r.$ r h p) rs (g h p)

assumes W :
 $\bigwedge p$ f $h.$ W p f h =
 $fold$ ($\lambda w.$ w p (f (R h p))) ws (u ($upd\text{-fun}$ p ($\lambda old.$ $merge\text{-addressable-fields}$ old (f (R h p)))) h)

shows $typ\text{-heap-simulation-open-types}$ \mathcal{T} l R W V hrs $hrs\text{-upd}$ $heap\text{-typing}$ $heap\text{-typing-upd}$

\wedge
 $(pointer\text{-lense } g$ (λp $f.$ u ($upd\text{-fun}$ p f))) \wedge
 $(\forall w.$ $(\forall x.$ $list\text{-all}$ ($\lambda w'.$ $pointer\text{-writer-disjnt}$ ($\lambda p.$ w' p x) w) ws) \longrightarrow
 $(\forall x.$ $pointer\text{-writer-disjnt}$ ($\lambda p.$ u ($upd\text{-fun}$ p ($\lambda -.$ x))) w) \longrightarrow
 $(\forall f.$ $pointer\text{-writer-disjnt}$ ($\lambda p.$ W p f) w)) \wedge
 $(\forall w$ $p.$ $(\forall x.$ $list\text{-all}$ ($\lambda w'.$ w' p x o w = w o w' p x) ws) \longrightarrow
 $(\forall x.$ u ($upd\text{-fun}$ p ($\lambda -.$ x)) o w = w o u ($upd\text{-fun}$ p ($\lambda -.$ x))) \longrightarrow
 $(\forall f.$ W p f o w = w o W p f))
is $?t1$ \wedge $?t3$ \wedge
 $(\forall w.$ $?ws$ w \longrightarrow $?u$ w \longrightarrow $(\forall f.$ $?t2$ f w)) \wedge
 $(\forall w$ $p.$ $?ws2$ w p \longrightarrow $?u2$ w p \longrightarrow $(\forall f.$ $?t4$ w p f))

proof ($intro$ $conjI$ $allI$ $impI$)

interpret hrs : $heap\text{-typing-simulation}$ \mathcal{T} hrs $hrs\text{-upd}$ $heap\text{-typing}$ $heap\text{-typing-upd}$

l **by fact**

interpret $lense$ g u **by fact**

have [$simp$]: $length$ ($addressable\text{-fields}$ $TYPE'(a)$) = $length$ fs
by ($simp$ add : $addressable\text{-fields-def}$ fs)

have $list\text{-all}$ ($\lambda f.$ $\exists u$ $n.$ $field\text{-lookup}$ ($typ\text{-uinfo-t}$ $TYPE'(a)$) f 0 = $Some$ (u , n))
 $(map$ fst (zip fs (zip rs ws)))
using $wf\text{-T}[OF$ $fs]$ **by auto**

then have $fs\text{-exists}$:
 $list\text{-all}$ ($\lambda x.$ $\exists u$ $n.$ $field\text{-lookup}$ ($typ\text{-uinfo-t}$ $TYPE'(a)$) (fst x) 0 = $Some$ (u , n))
 $(zip$ fs (zip rs ws))
by ($simp$ add : $list.pred\text{-map comp-def del: len}$)

have $rs\text{-ws}$:
 $list\text{-all}$ ($\lambda(m, r, w).$ $partial\text{-pointer-lense}$ m r w) (zip (map $merge\text{-ti}$ (map snd ($addressable\text{-fields}$ $TYPE'(a)$)))) (zip rs ws))
apply ($simp$ add : $zip\text{-map1 list.pred\text{-map addressable-fields-def fs split-beta' comp-def}$)
using $list\text{-all-conj}[OF$ $* fs\text{-exists}]$

```

apply (rule list.pred-mono-strong)
apply (auto simp: field-ti-def field-lookup-typ-uinfo-t-Some typ-heap-simulation-of-field-def
          dest!: field-lookup-uinfo-Some-rev)
done

have dist: distinct-prop disj-fn fs
using  $\mathcal{T}$ -distinct[OF fs] .

interpret pointer-lense R W
apply (rule pointer-lense-of-partials-cover[
  of g u map merge-ti (map snd (addressable-fields TYPE('a))) rs ws, OF  $\lambda$ lense
  g u) - - - - - R])
subgoal by simp
subgoal by simp
subgoal by fact
subgoal for a b c
apply (simp add: distinct-prop-map addressable-fields-def fs)
apply (rule distinct-prop-mono[OF - dist])
using wf- $\mathcal{T}$ [OF fs]
apply (intro disjnt-scene-merge-ti[THEN disjnt-sceneD,
  OF option.collapse[symmetric] option.collapse[symmetric]])
apply (auto simp: list.pred-set field-ti-def disj-fn-def
          dest!: field-lookup-uinfo-Some-rev)
done
subgoal
using ws-u ** dist
apply (clarsimp simp add: list-all-iff pointer-writer-disjnt-eq-def[abs-def])
apply (clarsimp simp add: distinct-prop-iff-nth fun-eq-iff distinct-conv-nth
          disj-fn-def)
apply metis
done
subgoal by (simp add: merge-ti-list-def fold-map comp-def)
subgoal by (rule W)
done

have list-all ( $\lambda$ (f, w).  $\forall t u n h p x.$ 
  field-ti TYPE('a) f = Some t  $\longrightarrow$ 
  field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (u, n)  $\longrightarrow$ 
  ptr-valid-u u (hrs-htd (hrs h))  $\&$ (p $\rightarrow$ f)  $\longrightarrow$ 
  l (hrs-upd (hrs-mem-update (heap-upd-list (size-td u)  $\&$ (p $\rightarrow$ f)) (access-ti t
x))) h) =
  w p x (l h)
  (map ( $\lambda$ frw. (fst frw, snd (snd frw))) (zip fs (zip rs ws)))
unfolding list.pred-map
using * by (rule list.pred-mono-strong) (auto simp: typ-heap-simulation-of-field-def)
also have map ( $\lambda$ frw. (fst frw, snd (snd frw))) (zip fs (zip rs ws)) = zip fs ws
by (simp add: list-eq-iff-nth-eq)
finally have fs-ws: list-all2 ( $\lambda$ f w.  $\forall t u n h p x.$ 
  field-ti TYPE('a) f = Some t  $\longrightarrow$ 

```

$field_lookup (typ_uinfo_t \text{TYPE}(a)) f 0 = \text{Some } (u, n) \longrightarrow$
 $ptr_valid_u u (hrs_htd (hrs h)) \ \&(p \rightarrow f) \longrightarrow$
 $l (hrs_upd (hrs_mem_update (heap_upd_list (size_td u) \ \&(p \rightarrow f) (access_ti t$
 $x))) h) =$
 $w \ p \ x \ (l \ h)$
 $fs \ ws$
by (*subst (asm) list-all-zip-iff-list-all2; simp*)

have $V-l: \bigwedge h \ p. V (l \ h) \ p \longleftrightarrow PTR_VALID(a) (hrs_htd (hrs h)) \ p$
by (*simp add: V*)

interpret *write-simulation hrs hrs-upd l V W*
apply (*rule write-simulationI[OF fs hrs, of ws u V - R]*)
apply (*rule fs-ws*)
apply (*rule l-u*)
apply *assumption*
apply (*rule V-l*)
apply (*rule W*)
done

show *?t3*
by (*rule pointer-lense-upd-fun-of-lense*) *fact*
then interpret *u: pointer-lense g λp f. u (upd-fun p f)* .

show *?t2 f w if *: ?ws w ?u w for w f*
by (*rule pointer-writer-disjnt-replace-by-const*)
*(use * in <auto simp add: W[abs-def]*
intro!: pointer-writer-disjnt-comp-left[OF pointer-writer-disjnt-fold-left
u.pointer-writer-disjnt-replace-by-const]>)

show *disj: ?t4 w p f if *: ?u2 w p ?ws2 w p for p f w*
proof (*standard, unfold comp-def*)

fix *h*
have $w\text{-ws}: w (fold (\lambda w. w \ p \ c) \ ws \ h) = fold (\lambda w. w \ p \ c) \ ws (w \ h)$ **for** $c \ h$
using **[rule-format, of c]*
by (*induction ws arbitrary: h (auto simp: fun-eq-iff)*)

have $w\text{-u}[simp]: w (u (upd-fun p (\lambda-. c)) h) = u (upd-fun p (\lambda-. c)) (w h)$ **for**
 $c \ h$
using **(1) by (auto simp: fun-eq-iff)*

have $g\text{-w}[simp]: g (w \ h) \ p = g \ h \ p$ **for** h
by (*subst u.write-same[of λ-. g h p h p, OF refl, symmetric]*)
(simp add: u.read-write-same)

have $w\text{-u}'[simp]: w (u (upd-fun p f) h) = u (upd-fun p f) (w h)$ **for** $f \ h$
by (*subst (1 2) u.write-cong[where f'=λ-. f (g h p)] simp-all*)

have $w\text{-W-const}: w (W \ p (\lambda x. c) h) = W \ p (\lambda x. c) (w \ h)$ **for** $c \ h$

```

    by (simp add: W w-ws)

  have R-eq: R (w h) p = R h p
    apply (subst write-same[of λ-. R h p h p, OF refl, symmetric])
    apply (subst w-W-const)
    apply (subst read-write-same)
    apply (rule refl)
  done

  show W p f (w h) = w (W p f h)
    by (simp add: R-eq write-cong[of f - - λ-. f (R h p)] w-W-const)
qed

show ?t1
proof
  fix h p assume V-p: V (l h) p
  then show c-guard p by (simp add: ptr-valid.ptr-valid-c-guard V)

  have l-h: l h = W p (λx. h-val (hrs-mem (hrs h)) p) (l h)
    apply (subst write-commutes[OF V-p, symmetric])
    apply (subst hrs.t-hrs.heap.upd-same)
    apply (cases hrs h)
  apply (simp-all add: hrs-mem-update-def hrs-mem-def xmem-type-class.heap-update-id)
  done

  show R (l h) p = h-val (hrs-mem (hrs h)) p
    apply (subst l-h)
    apply (subst read-write-same)
    apply simp
  done

next
  fix p :: 'a ptr and s
  assume p: ∀ a ∈ ptr-span p. root-ptr-valid (hrs-htd (hrs s)) (PTR(stack-byte) a)
  moreover
  { fix f w and r :: 't ⇒ 'a ptr ⇒ 'a ⇒ 'a and u n
    assume typ-heap-simulation-of-field l hrs hrs-upd heap-typing-upd f r w
      field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (u, n)
    with p have r (l s) p ZERO('a) = ZERO('a)
      unfolding typ-heap-simulation-of-field-def
    by (auto simp: field-ti-def field-lookup-typ-uinfo-t-Some typ-heap-simulation-of-field-def
      dest!: field-lookup-uinfo-Some-rev) }
  note this[simp]
  from * fs-exists len have fold (λr. r (l s) p) rs ZERO('a) = ZERO('a)
    by (induction fs arbitrary: rs ws)
      (auto simp: length-Suc-conv simp del: len)
  ultimately show R (l s) p = ZERO('a)
    unfolding R by (simp add: r-stack)
next
  { fix d p f

```

```

have  $W p f o \text{ heap-typing-upd } d = \text{heap-typing-upd } d o W p f$ 
  apply (rule disj)
  subgoal using heap-typing-u by (simp add: fun-eq-iff)
  using * len
  apply (induction fs arbitrary: ws rs)
  apply (auto simp add: length-Suc-conv typ-heap-simulation-of-field-def simp
del: len)
  done
  then show  $\text{heap-typing-upd } d (W p f h) = W p f (\text{heap-typing-upd } d h)$  for  $h$ 
    by (simp add: fun-eq-iff) }
  note  $\text{heap-typing-}W = \text{this}$ 

show  $V (W p' f h) p = V h p$  for  $p' f h p$ 
  unfolding  $V$ 
  apply (subst hrs.upd-same[symmetric, of  $\lambda-. \text{heap-typing } h, OF refl$ ])
  apply (subst heap-typing-}W[symmetric])
  apply (subst hrs.get-upd)
  ..
qed (simp-all add: V)
qed

lemma typ-heap-simulationI-no-addressable:
  fixes  $R :: 't \Rightarrow 'a::\{\text{xmem-type, stack-type}\} \text{ ptr} \Rightarrow 'a$ 
  assumes heap-typing-simulation  $\mathcal{T}$  hrs hrs-upd heap-typing heap-typing-upd l
  assumes  $R\text{-u: lense } R u$ 
  assumes fs: map-of  $\mathcal{T}$  (typ-winfo-t  $\text{TYPE}('a)$ ) = None
  and  $l\text{-u: } \bigwedge(p::'a \text{ ptr}) (x::'a) (s::'b).$ 
     $\text{PTR-VALID}('a) (\text{hrs-htd } (\text{hrs } s)) p \implies$ 
     $l (\text{hrs-upd } (\text{write-dedicated-heap } p x) s) = u (\text{upd-fun } p (\lambda \text{old. merge-addressable-fields}$ 
old } x)) (l s)
  and  $\text{heap-typing-u: } \bigwedge x d h. \text{heap-typing-upd } d (u x h) = u x (\text{heap-typing-upd}$ 
d } h)
  and r-stack:
     $\bigwedge p s. \forall a \in \text{ptr-span } p. \text{root-ptr-valid } (\text{hrs-htd } (\text{hrs } s)) (\text{PTR}(\text{stack-byte}) a) \implies$ 
     $R (l s) p = \text{ZERO}(-)$ 
  assumes  $V$ :
     $\bigwedge h p. V h p \iff \text{PTR-VALID}('a) (\text{heap-typing } h) p$ 
  assumes  $W$ :
     $\bigwedge p f h. W p f h = u (\lambda h'. h'(p := f (h' p))) h$ 
shows typ-heap-simulation-open-types  $\mathcal{T} l R W V \text{hrs hrs-upd heap-typing heap-typing-upd}$ 
proof –
  interpret hrs: heap-typing-simulation  $\mathcal{T}$  hrs hrs-upd heap-typing heap-typing-upd
l by fact
  interpret lense  $R u$  by fact

  have [simp]: merge-addressable-fields =  $(\lambda a b::'a. b)$ 
    by (simp add: fun-eq-iff addressable-fields-def fs)

  interpret pointer-lense  $R W$ 

```



```

using pointer-lense-of-lense-flt[where 'd='a, OF R-u, of []]
by (simp add: W[abs-def] upd-fun-def[abs-def])

interpret write-simulation hrs hrs-upd l V W
apply (rule hrs.write-simulation-alt)
subgoal by (simp add: V)
subgoal for h p x
  unfolding W V
  using l-u[of h p x]
  by (simp add: write-dedicated-heap-def heap-upd-const upd-fun-def[abs-def])
done

show ?thesis
proof
  fix h p assume V-p: V (l h) p
  then show c-guard p by (simp add: ptr-valid.ptr-valid-c-guard V)

  have l-h: l h = W p (λx. h-val (hrs-mem (hrs h)) p) (l h)
    apply (subst write-commutes[OF V-p, symmetric])
    apply (subst hrs.t-hrs.heap.upd-same)
    apply (cases hrs h)
  apply (simp-all add: hrs-mem-update-def hrs-mem-def xmem-type-class.heap-update-id)
  done

  show R (l h) p = h-val (hrs-mem (hrs h)) p
    apply (subst l-h)
    apply (subst read-write-same)
    apply simp
  done

next
  show V (W p' f h) p = V h p for p' f h p
    unfolding W V
    apply (subst hrs.upd-same[symmetric, of λ-. heap-typing h, OF refl])
    apply (subst heap-typing-u[symmetric])
    apply (subst hrs.get-upd)
  ..
  qed (simp-all add: V W heap-typing-u r-stack)
qed

lemma typ-heap-simulationI-all-addressable:
  fixes R :: 't ⇒ 'a::{xmem-type, stack-type} ptr ⇒ 'a
  assumes hrs: heap-typing-simulation T hrs hrs-upd heap-typing heap-typing-upd
  l
  assumes fs: map-of T (typ-uinfo-t TYPE('a)) = Some fs
  assumes len[simp]: length rs = length fs length ws = length fs
  assumes *:
    list-all (λ(f, r, w). typ-heap-simulation-of-field l hrs hrs-upd heap-typing-upd f
  r w) (zip fs (zip rs ws))
    and **: distinct-prop (λ(f1, w1) (f2, w2)).

```

$disj\text{-}fn\ f1\ f2 \longrightarrow pointer\text{-}writer\text{-}disjnt\text{-}eq\ w1\ w2$
 $(zip\ fs\ ws)$
assumes $all: \bigwedge a\ b. fold\ (\lambda x. merge\text{-}ti\ (the\ (field\text{-}ti\ TYPE('a)\ x))\ a)\ fs\ b = a$
assumes $V:$
 $\bigwedge h\ p. V\ h\ p \longleftrightarrow PTR\text{-}VALID('a)\ (heap\text{-}typing\ h)\ p$
assumes $R:$
 $\bigwedge h\ p\ x. R\ h\ p = fold\ (\lambda r. r\ h\ p)\ rs\ x$
assumes $W:$
 $\bigwedge p\ f\ h. W\ p\ f\ h = fold\ (\lambda w. w\ p\ (f\ (R\ h\ p)))\ ws\ h$
shows $typ\text{-}heap\text{-}simulation\text{-}open\text{-}types\ \mathcal{T}\ l\ R\ W\ V\ hrs\ hrs\text{-}upd\ heap\text{-}typing\ heap\text{-}typing\text{-}upd$
 \wedge
 $(\forall w. (\forall x. list\text{-}all\ (\lambda w'. pointer\text{-}writer\text{-}disjnt\ (\lambda p. w'\ p\ x)\ w)\ ws) \longrightarrow$
 $(\forall f. pointer\text{-}writer\text{-}disjnt\ (\lambda p. W\ p\ f)\ w)) \wedge$
 $(\forall (w::'t \Rightarrow 't)\ p.$
 $(\forall x. list\text{-}all\ (\lambda w'. w'\ p\ x \circ w = w \circ w'\ p\ x)\ ws) \longrightarrow$
 $(\forall f. W\ p\ f \circ w = w \circ W\ p\ f))$
 $(is\ ?t1 \wedge (\forall w. ?ws\ w \longrightarrow (\forall f. ?t2\ f\ w)) \wedge (\forall w\ p. ?ws2\ p\ w \longrightarrow (\forall f. ?t3\ p\ f$
 $w)))$
proof $(intro\ conjI\ allI\ impI)$
interpret $hrs: heap\text{-}typing\text{-}simulation\ \mathcal{T}\ hrs\ hrs\text{-}upd\ heap\text{-}typing\ heap\text{-}typing\text{-}upd$
l by fact

have $[simp]: length\ (addressable\text{-}fields\ TYPE('a)) = length\ fs$
by $(simp\ add: addressable\text{-}fields\text{-}def\ fs)$

have $list\text{-}all\ (\lambda f. \exists u\ n. field\text{-}lookup\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ f\ 0 = Some\ (u, n))$
 $(map\ fst\ (zip\ fs\ (zip\ rs\ ws)))$
using $wf\text{-}\mathcal{T}[OF\ fs]$ **by auto**
then have $fs\text{-}exists:$
 $list\text{-}all\ (\lambda x. \exists u\ n. field\text{-}lookup\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ (fst\ x)\ 0 = Some\ (u, n))$
 $(zip\ fs\ (zip\ rs\ ws))$
by $(simp\ add: list.\text{pred}\text{-}map\ comp\text{-}def\ del: len)$

have $coer\text{-}eq[simp]: merge\text{-}addressable\text{-}fields = (\lambda a\ b::'a. a)$
by $(simp\ add: merge\text{-}ti\text{-}list\text{-}def\ addressable\text{-}fields\text{-}def\ fold\text{-}map\ fs\ fun\text{-}eq\text{-}iff\ comp\text{-}def$
 $all)$

have $dist: distinct\text{-}prop\ disj\text{-}fn\ fs$
using $\mathcal{T}\text{-}distinct[OF\ fs]$.

interpret $pointer\text{-}lense\ R\ W$
apply $(rule\ pointer\text{-}lense\text{-}of\text{-}partials[$
 $of\ map\ merge\text{-}ti\ (map\ snd\ (addressable\text{-}fields\ TYPE('a)))\ rs\ ws, OF\ \text{-}\text{-}\text{-}\text{-}\text{-}\text{-}\text{-}$
 $\text{-}\ R])$
subgoal by simp
subgoal by simp
subgoal
apply $(simp\ add: zip\text{-}map1\ list.\text{pred}\text{-}map\ addressable\text{-}fields\text{-}def\ fs\ split\text{-}beta'$
 $comp\text{-}def)$

```

    using list-all-conj[OF * fs-exists]
    apply (rule list.pred-mono-strong)
  apply (auto simp: field-ti-def field-lookup-typ-uinfo-t-Some typ-heap-simulation-of-field-def
    dest!: field-lookup-uinfo-Some-rev)
  done
subgoal for a b c
  apply (simp add: distinct-prop-map addressable-fields-def fs)
  apply (rule distinct-prop-mono[OF - dist])
  using wf- $\mathcal{T}$ [OF fs]
  apply (intro disjnt-scene-merge-ti[THEN disjnt-sceneD,
    OF option.collapse[symmetric] option.collapse[symmetric]])
  apply (auto simp: list.pred-set field-ti-def disj-fn-def
    dest!: field-lookup-uinfo-Some-rev)
  done
subgoal
  using ** dist
  apply (clarsimp simp add: distinct-prop-iff-nth fun-eq-iff pointer-writer-disjnt-eq-def[abs-def]
    distinct-conv-nth disj-fn-def)
  apply metis
  done
subgoal by (simp add: addressable-fields-def fs fold-map comp-def all)
subgoal by (rule W)
done

have list-all ( $\lambda(f, w). \forall t u n h p x.$ 
  field-ti TYPE('a) f = Some t  $\longrightarrow$ 
  field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (u, n)  $\longrightarrow$ 
  ptr-valid-u u (hrs-htd (hrs h)) &(p $\rightarrow$ f)  $\longrightarrow$ 
  l (hrs-upd (hrs-mem-update (heap-upd-list (size-td u) &(p $\rightarrow$ f)) (access-ti t
x))) h) =
  w p x (l h))
  (map ( $\lambda$ frw. (fst frw, snd (snd frw))) (zip fs (zip rs ws)))
  unfolding list.pred-map
  using * by (rule list.pred-mono-strong) (auto simp: typ-heap-simulation-of-field-def)
also have map ( $\lambda$ frw. (fst frw, snd (snd frw))) (zip fs (zip rs ws)) = zip fs ws
  by (simp add: list-eq-iff-nth-eq)
finally have fs-ws: list-all2 ( $\lambda$ f w.  $\forall t u n h p x.$ 
  field-ti TYPE('a) f = Some t  $\longrightarrow$ 
  field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (u, n)  $\longrightarrow$ 
  ptr-valid-u u (hrs-htd (hrs h)) &(p $\rightarrow$ f)  $\longrightarrow$ 
  l (hrs-upd (hrs-mem-update (heap-upd-list (size-td u) &(p $\rightarrow$ f)) (access-ti t
x))) h) =
  w p x (l h))
  fs ws
  by (subst (asm) list-all-zip-iff-list-all2; simp)

have V-l:  $\bigwedge h p. V (l h) p \longleftrightarrow PTR-VALID('a) (hrs-htd (hrs h)) p$ 
  by (simp add: V)

```

```

interpret write-simulation hrs hrs-upd l V W
  apply (rule write-simulationI[OF fs hrs, of ws  $\lambda x y. y V - R$ ])
  apply (rule fs-ws)
  apply (simp add: write-dedicated-heap-def heap-upd-id hrs-mem-update-id3 flip:
id-def)
  apply (simp add: id-def)
  apply (rule V-l)
  apply (rule W)
  done

show ?t2 f w if *: ?ws w for w f
  apply (rule pointer-writer-disjnt-replace-by-const)
  apply (simp add: W[abs-def])
  apply (intro pointer-writer-disjnt-fold-left *[rule-format])
  done

show disj: ?t3 p f w if *: ?ws2 p w for p f w
proof (standard, unfold comp-def)
  fix h
  have w-W-const: w (W p ( $\lambda x. c$ ) h) = W p ( $\lambda x. c$ ) (w h) for c h
    using *[rule-format, of c]
    apply (simp add: W)
    apply (induction ws arbitrary: h)
    apply (auto simp: fun-eq-iff)
    done
  have R-eq: R (w h) p = R h p
    apply (subst write-same[of  $\lambda-. R h p h p$ , OF refl, symmetric])
    apply (subst w-W-const)
    apply (subst read-write-same)
    apply (rule refl)
    done

  show W p f (w h) = w (W p f h)
    by (simp add: R-eq write-cong[of f - -  $\lambda-. f (R h p)$ ] w-W-const)
qed

show ?t1
proof
  fix h p assume V-p: V (l h) p
  then show c-guard p by (simp add: ptr-valid.ptr-valid-c-guard V)

  have l-h: l h = W p ( $\lambda x. h\text{-val} (hrs\text{-mem} (hrs h)) p$ ) (l h)
    apply (subst write-commutes[OF V-p, symmetric])
    apply (subst hrs.t-hrs.heap.upd-same)
    apply (cases hrs h)
  apply (simp-all add: hrs-mem-update-def hrs-mem-def xmem-type-class.heap-update-id)
  done

  show R (l h) p = h-val (hrs-mem (hrs h)) p

```

```

    apply (subst l-h)
    apply (subst read-write-same)
    apply simp
  done
next
fix p :: 'a ptr and s
assume p:  $\forall a \in \text{ptr-span } p. \text{root-ptr-valid (hrs-htd (hrs s)) (PTR(stack-byte) a)}$ 
moreover
{ fix f w and r :: 't  $\Rightarrow$  'a ptr  $\Rightarrow$  'a  $\Rightarrow$  'a and u n
  assume typ-heap-simulation-of-field l hrs hrs-upd heap-typing-upd f r w
    field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (u, n)
  with p have r (l s) p ZERO('a) = ZERO('a)
    unfolding typ-heap-simulation-of-field-def
  by (auto simp: field-ti-def field-lookup-typ-uinfo-t-Some typ-heap-simulation-of-field-def
    dest!: field-lookup-uinfo-Some-rev) }
note this[simp]
from * fs-exists len have fold ( $\lambda r. r (l s) p$ ) rs ZERO('a) = ZERO('a)
  by (induction fs arbitrary: rs ws)
    (auto simp: length-Suc-conv simp del: len)
ultimately show R (l s) p = ZERO('a)
  unfolding R[of - - ZERO('a)] by simp
next
{ fix p f d
  have W p f o heap-typing-upd d = heap-typing-upd d o W p f
    apply (rule disj)
    using * len
    apply (induction fs arbitrary: ws rs)
    apply (auto simp add: length-Suc-conv typ-heap-simulation-of-field-def simp
del: len)
  done
  then show heap-typing-upd d (W p f h) = W p f (heap-typing-upd d h) for h
    by (simp add: fun-eq-iff) }
note heap-typing-W = this

show V (W p' f h) p = V h p for p' f h p
  unfolding V
  apply (subst hrs.upd-same[symmetric, of  $\lambda-. \text{heap-typing } h, OF refl$ ])
  apply (subst heap-typing-W[symmetric])
  apply (subst hrs.get-upd)
  ..
qed (simp-all add: V)
qed

end

definition
  field-with-lense ::
    qualified-field-name  $\Rightarrow$  ('a::c-type  $\Rightarrow$  'b::c-type)  $\Rightarrow$  (('b  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'a)  $\Rightarrow$ 
  bool

```

where

field-with-lense $f g u \longleftrightarrow$
field-ti $TYPE('a) f = Some (adjust-ti (typ-info-t TYPE('b)) g (u o (\lambda x -. x)))$
 \wedge
lense $g u$

lemma *update-desc-id*: *update-desc* $(\lambda x. x) (\lambda x -. x) = id$
by (*simp add: fun-eq-iff update-desc-def*)

lemma *field-with-lense-Nil*: *field-with-lense* $\square (\lambda x. x) (\lambda f x. f x)$
by (*simp add: field-with-lense-def lense-def comp-def adjust-ti-def update-desc-id map-td-id(1)*)

lemma *field-with-lense-Cons*:

fixes $g :: 'a::mem-type \Rightarrow 'b::mem-type$ **and** $gs :: 'b \Rightarrow 'c::mem-type$
assumes fs : *field-with-lense* $fs gs us$
assumes f : *field-ti* $TYPE('a) [f] = Some (adjust-ti (typ-info-t TYPE('b)) g (u o (\lambda x -. x)))$
assumes $g-u$: *fg-cons* $g (u o (\lambda x -. x))$
assumes u : $\bigwedge f x. u f x = u (\lambda -. f (g x)) x$
shows *field-with-lense* $(f \# fs) (\lambda x. gs (g x)) (\lambda f. u (us f))$
unfolding *field-with-lense-def*

proof

from fs **obtain** n **where** fs :

field-lookup $(typ-info-t TYPE('b)) fs 0 =$
 $Some (adjust-ti (typ-info-t TYPE('c)) gs (us o (\lambda x -. x)), n)$
and $gs-us$: *lense* $gs us$
by (*auto simp add: field-with-lense-def field-ti-def split: option.splits*)

have $g-u$: *lense* $g u$

using $g-u$
apply (*rule lense-of-fg-cons'*)
apply (*rule u*)
done

then interpret *lense* $g u$.

show *lense* $(\lambda x. gs (g x)) (\lambda f. u (us f))$
using $g-u$ $gs-us$ **by** (*rule lense-compose*)

have $(\lambda x. gs (g x)) = gs \circ g$

$(\lambda v x. u (\lambda -. us (\lambda -. v) (g x)) x) = (\lambda f. u (us f)) \circ (\lambda x -. x)$

by (*simp-all add: fun-eq-iff cong: upd-cong*)

with *field-ti-append-field-lookup*[*OF* f , *of* $fs 0$]

show *field-ti* $TYPE('a) (f \# fs) =$

$Some (adjust-ti (typ-info-t TYPE('c)) (\lambda x. gs (g x)) ((\lambda f. u (us f)) \circ (\lambda x -. x)))$

unfolding *field-lookup-adjust-ti2*[*OF* fs]

by (*simp add: adjust-ti-adjust-ti*)

qed

lemma (in *typ-heap-simulation-open-types*) *typ-heap-simulation-of-field*:
assumes *f*: *field-with-lense* *f* *g* *u*
assumes *v*: $\bigwedge h p. \text{PTR-VALID}('a) (hrs\text{-}htd (t\text{-}hrs h)) p \implies v (st\ h) p$
assumes *g-zero*: $g \text{ ZERO}('x::\text{mem-type}) = \text{ZERO}('a)$
shows *typ-heap-simulation-of-field* *st* *t-hrs* *t-hrs-update* *heap-typing-upd* *f*
 $(\lambda h p. u (\lambda\cdot. r\ h (\text{PTR}('a) \ \&(p\rightarrow f))))$
 $(\lambda p x. w (\text{PTR}('a) \ \&(p\rightarrow f)) (\lambda\cdot. g\ x))$

proof –
have *g-u*: *lense* *g* *u* **using** *f* **by** (*simp* *add*: *field-with-lense-def*)
then interpret *lense* *g* *u* .

have [*simp*]: *fg-cons* *g* (*u* $\circ (\lambda x \cdot x)$)
by (*rule* *fg-cons*)
obtain *n* **where** [*simp*]:
field-lookup (*typ-uinfo-t* *TYPE*('x)) *f* 0 = *Some* (*typ-uinfo-t* *TYPE*('a), *n*)
using *f*
by (*auto* *simp*: *field-ti-def* *field-lookup-typ-uinfo-t-Some* *field-with-lense-def*
split: *option.splits*)
have *sz*: *size-td* (*typ-info-t* *TYPE*('a)) = *size-of* *TYPE*('a)
by (*simp* *add*: *size-of-def*)

have *p-f-eq*: $\&(p\rightarrow f) = \text{ptr-val} (\text{PTR}('c) \ \&(p\rightarrow f))$ **for** *p* :: 'x *ptr*
by *simp*

have *ptr-span-subset*: $\text{ptr-span} (\text{PTR}('a) \ \&(p\rightarrow f)) \subseteq \text{ptr-span } p$ **for** *p* :: 'x *ptr*
using *field-tag-sub'* [**where** 'a='x, *of* *f* - - *p*] **by** (*simp* *add*: *size-of-def*)

have *u-zero*: $u (\lambda\cdot. \text{ZERO}('a)) \text{ZERO}('x) = \text{ZERO}('x)$
by (*simp* *add*: *upd-same-g-zero*)

show *?thesis* **using** *f*
apply (*clarsimp* *simp*: *typ-heap-simulation-of-field-def* *field-with-lense-def*, *intro*
conjI *allI* *impI*)
subgoal **by** (*simp* *add*: *heap-typing-upd-write-commute* *fun-eq-iff*)
subgoal
apply (*rule* *partial-pointer-lenseI*[*OF* *g-u*])
apply (*rule* *pointer-lense-field-lvalue*)
apply *assumption*
apply (*simp* *add*: *size-of-def*)
done

subgoal **for** *p*
using *ptr-span-subset*[*of* *p*]
by (*subst* *sim-stack-stack-byte-zero'*) (*auto* *simp*: *u-zero*)

subgoal
apply (*subst* *sz*)
apply (*subst* *p-f-eq*)
apply (*subst* *heap-update-eq-heap-upd-list*[*symmetric*])
apply (*rule* *write-commutes*)
apply (*simp* *add*: *v* *ptr-valid-def*)

done
done
qed

context *pointer-array-lense*
begin

lemma *pointer-writer-disjnt-heap-array-map*[*pointer-writer-disjnt-intros*]:
 assumes $w': \bigwedge x. \text{pointer-writer-disjnt } (\lambda p. w \ p \ x) \ w'$
 shows $\text{pointer-writer-disjnt } (\lambda p. \text{heap-array-map } p \ x) \ w'$
 apply (rule *array.pointer-writer-disjnt-replace-by-const*)
 apply (simp only: *heap-array-map-def*[*abs-def*])
 apply (rule *pointer-writer-disjnt-fold-left*)
 apply (clarsimp simp add: *list-all-iff*)
 apply (rule *pointer-writer-disjnt-ptr-map-left*[*OF w' ptr-span-array-ptr-index-subset-ptr-span*])
 apply simp
 done

lemma *pointer-writer-disjnt-heap-array-map-right*[*pointer-writer-disjnt-intros*]:
 $(\bigwedge x. \text{pointer-writer-disjnt } w' (\lambda p. w \ p \ x)) \implies \text{pointer-writer-disjnt } w' (\lambda p. \text{heap-array-map } p \ x)$
 by (metis *pointer-writer-disjnt-heap-array-map pointer-writer-disjnt-sym*)

lemma *disjnt-comp-heap-array-map*[*disjnt-heap-fields-comp*]:
 assumes $w': \bigwedge q \ x. w \ q \ x \circ w' = w' \circ w \ q \ x$
 shows $\text{heap-array-map } p \ x \circ w' = w' \circ \text{heap-array-map } p \ x$
 proof –
 have $\text{heap-array-map } p \ (\lambda-. \ c) \circ w' = w' \circ \text{heap-array-map } p \ (\lambda-. \ c)$ for c
 apply (rule *comp-commute-of-fold*)
 apply (subst *heap-array-map-def*[*abs-def*], rule *refl*)
 apply (simp add: *w' list-all-iff*)
 done
 then have $*$: $w' (\text{heap-array-map } p \ (\lambda-. \ c) \ h) = \text{heap-array-map } p \ (\lambda-. \ c) (w' \ h)$
 for $c \ h$
 by (simp add: *fun-eq-iff*)

have [*simp*]: $\text{heap-array } (w' \ h) \ p = \text{heap-array } h \ p$ for h
 apply (subst *array.write-same*[*symmetric, of \lambda-. heap-array h p, OF refl*])
 apply (subst $*$)
 apply (subst *array.read-write-same*)
 apply (rule *refl*)
 done
 show ?thesis
 apply (rule *ext*)
 subgoal for h
 apply (clarsimp simp add: *fun-eq-iff*)
 apply (subst (1 2) *array.write-cong*[*where f'=\lambda-. x (heap-array h p)*])
 apply (simp-all flip: $*$)
 done

done
qed

end

lemma (in open-types) *valid-array-of-ptr-valid-arrayI*:
fixes $r :: 't \Rightarrow 'a::\{\text{xmem-type, array-outer-max-size}\}$ $\text{ptr} \Rightarrow 'a$
fixes $p :: ('a['n::\{\text{finite, array-max-count}\}]) \text{ptr}$
assumes $f: \text{map-of } \mathcal{T} (\text{typ-uinfo-t TYPE}('a['n])) = \text{Some} (\text{array-fields CARD}('n))$
assumes $p: \text{PTR-VALID}('a['n]) \text{h}' p$
assumes $v: \bigwedge p. \text{PTR-VALID}('a) \text{h}' p \implies v \text{h} p$
shows *valid-array-base.valid-array* $v \text{h} p$
unfolding *valid-array-base.valid-array-def*
proof (intro allI impI v)
fix n **assume** $n: n < \text{CARD}('n)$
note $\text{fl-array} = \text{field-lookup-array}[\text{OF } n, \text{THEN field-lookup-typ-uinfo-t-Some}]$

have $[\text{replicate } n \text{ CHR } "1"] \in \text{set} (\text{array-fields CARD}('n))$
using n **by** (auto simp: array-fields-def)

note $*$ = *ptr-valid-u-step* $[\text{OF } f \text{ this fl-array } p[\text{unfolded ptr-valid-def}]]$
have *field-offset-untyped* (typ-uinfo-t TYPE('a['n])) $[\text{replicate } n \text{ CHR } "1"] =$
 $n * \text{size-of TYPE}('a)$
by (simp add: field-offset-untyped-def fl-array)
with $*$ **show** $\text{PTR-VALID}('a) \text{h}' (\text{array-ptr-index } p \text{ False } n)$
by (simp add: n array-ptr-index-def ptr-add-def ptr-valid-def typ-uinfo-t-def)
qed

lemma (in open-types) *valid-array-of-ptr-valid-array1* [simp]:
fixes $r :: 't \Rightarrow 'a::\{\text{xmem-type, array-outer-max-size}\}$ $\text{ptr} \Rightarrow 'a$
fixes $p :: ('a['n::\{\text{finite, array-max-count}\}]) \text{ptr}$
assumes $f:$
 $\text{map-of } \mathcal{T} (\text{typ-uinfo-t TYPE}('a['n])) = \text{Some} (\text{array-fields CARD}('n))$
assumes $p: \text{PTR-VALID}('a['n]) (\text{heap-typing } h) p$
shows *valid-array-base.valid-array* $(\lambda h. \text{PTR-VALID}('a) (\text{heap-typing } h)) \text{h} p$
by (rule *valid-array-of-ptr-valid-arrayI* [of heap-typing h, OF f p])

lemma (in open-types) *valid-array-of-ptr-valid-array2* [simp]:
fixes $r :: 't \Rightarrow 'a::\{\text{xmem-type, array-inner-max-size}\}$ $\text{ptr} \Rightarrow 'a$
fixes $p :: ('a['n::\{\text{finite, array-max-count}\}][['m::\{\text{finite, array-max-count}\}]) \text{ptr}$
assumes $f2:$
 $\text{map-of } \mathcal{T} (\text{typ-uinfo-t TYPE}('a['n][['m])) = \text{Some} (\text{array-fields CARD}('m))$
assumes $f1:$
 $\text{map-of } \mathcal{T} (\text{typ-uinfo-t TYPE}('a['n])) = \text{Some} (\text{array-fields CARD}('n))$
assumes $p: \text{PTR-VALID}('a['n][['m]) (\text{heap-typing } h) p$
shows *valid-array-base.valid-array* (*valid-array-base.valid-array*
 $(\lambda h. \text{PTR-VALID}('a) (\text{heap-typing } h))) \text{h} p$
apply (rule *valid-array-of-ptr-valid-arrayI* [of heap-typing h, OF f2 p])
apply (rule *valid-array-of-ptr-valid-array1* [OF f1])

apply *assumption*
done

lemma (*in open-types*) *ptr-valid-unsigned*[*simp*]:
 $PTR-VALID('a::len8\ signed\ word)\ h\ p \longleftrightarrow$
 $PTR-VALID('a::len8\ word)\ h\ (PTR-COERCE('a\ signed\ word \rightarrow 'a\ word)\ p)$
by (*simp add: typ-uinfo-t-signed-word-word-conv ptr-valid-def*)

lemma *ucast-zero-word*:
 $UCAST('a::len8 \rightarrow 'a\ signed)\ ZERO('a\ word) = ZERO('a\ signed\ word)$
using *len8-bytes*[**where** $'a='a$]
apply (*simp add: zero-def*)
apply (*subst from-bytes-signed-word*)
apply (*simp-all add: size-of-def typ-info-word ucast-ucast-id*)
done

definition *signed-heap*:
 $('s \Rightarrow 'a::len\ word\ ptr \Rightarrow 'a\ word) \Rightarrow ('s \Rightarrow 'a\ signed\ word\ ptr \Rightarrow 'a\ signed\ word)$
where
 $signed-heap\ h\ s = UCAST('a::len \rightarrow 'a\ signed)\ o\ (h\ s)\ o\ PTR-COERCE('a\ signed\ word \rightarrow 'a\ word)$

lemma *signed-heap-apply*[*simp*]:
 $signed-heap\ h\ s\ p = UCAST('a::len \rightarrow 'a\ signed)\ (h\ s\ (PTR-COERCE('a\ signed\ word \rightarrow 'a\ word)\ p))$
by (*simp add: signed-heap-def*)

lemma *signed-typ-heap-simulation-of-typ-heap-simulation*:
fixes $r :: - \Rightarrow - \Rightarrow 'a::len8\ word$
assumes r :
 $typ-heap-simulation-open-types\ \mathcal{T}\ st\ r\ w\ v\ t-hrs\ t-hrs-update\ heap-typing\ heap-typing-upd$
shows $typ-heap-simulation-open-types\ \mathcal{T}\ st$
 $(signed-heap\ r)$
 $(\lambda p\ m.\ w\ (PTR-COERCE('a\ signed\ word \rightarrow 'a\ word)\ p)\ (\lambda w.\ UCAST('a\ signed\ word \rightarrow 'a)\ (m\ (UCAST('a \rightarrow 'a\ signed)\ w))))$
 $(\lambda h\ p.\ v\ h\ (PTR-COERCE('a\ signed\ word \rightarrow 'a\ word)\ p))$
 $t-hrs\ t-hrs-update\ heap-typing\ heap-typing-upd$

proof –
interpret $typ-heap-simulation-open-types\ \mathcal{T}\ st\ r\ w\ v\ t-hrs\ t-hrs-update\ heap-typing\ heap-typing-upd$
by fact
interpret *cast: pointer-lense*
 $signed-heap\ r$
 $\lambda p\ m.\ w\ (PTR-COERCE('a\ signed\ word \rightarrow 'a\ word)\ p)$
 $(\lambda w.\ UCAST('a\ signed \rightarrow 'a)\ (m\ (UCAST('a \rightarrow 'a\ signed)\ w)))$
unfolding *signed-heap-def comp-def*
by (*rule pointer-lense-ucast-signed*) *unfold-locales*
have [*simp*]: $c-guard\ p \longleftrightarrow c-guard\ (PTR-COERCE('a\ signed\ word \rightarrow 'a\ word)\ p)$
for p

```

apply (intro c-guard-ptr-coerce[symmetric])
apply (simp-all add: size-of-signed-word align-of-signed-word)
done
note scast-ucast-norm[simp del]
note ucast-ucast-id[simp]
show ?thesis
  apply unfold-locales
  apply (simp-all add: ptr-valid-unsigned ptr-valid-imp-v
    read-commutes h-val-signed-word write-padding-commutes heap-typing-upd-write-commute)
  apply (subst heap-update-padding-signed-word(1)[symmetric])
  apply (subst write-padding-commutes)
  apply (simp-all add: size-of-signed-word scast-ucast-down-same valid-implies-c-guard)
  apply (rule valid-same-typ-desc, simp)
  apply (simp add: sim-stack-stack-byte-zero')
  apply (simp add: ucast-zero-word)
done
qed

lemma array-typ-heap-simulation-of-typ-heap-simulation:
  fixes r :: -  $\Rightarrow$  -  $\Rightarrow$  'a::{stack-type, xmem-type, array-outer-max-size}
  assumes typ-heap-simulation-open-types  $\mathcal{T}$  st r w v t-hrs t-hrs-update heap-typing
  heap-typing-upd
  assumes f: map-of  $\mathcal{T}$  (typ-uinfo-t TYPE('a['n>::{finite, array-max-count}))) =
    Some (array-fields CARD('n))
  shows
    typ-heap-simulation-open-types  $\mathcal{T}$  st
    (pointer-array-lense.heap-array r :: -  $\Rightarrow$  ('a['n]) ptr  $\Rightarrow$  -)
    (pointer-array-lense.heap-array-map r w)
    (valid-array-base.valid-array v)
    t-hrs t-hrs-update heap-typing heap-typing-upd
proof -
  interpret sim: typ-heap-simulation-open-types  $\mathcal{T}$  st r w v t-hrs t-hrs-update by
  fact
  interpret array-typ-heap-simulation st r w v t-hrs t-hrs-update ..
  show ?thesis
  proof
    fix p :: ('a['n]) ptr and d f
    have heap-array-map p f o heap-typing-upd d = heap-typing-upd d o heap-array-map
  p f
    by (intro disjnt-comp-heap-array-map)
    (simp add: fun-eq-iff sim.heap-typing-upd-write-commute)
    then show heap-typing-upd d (heap-array-map p f h) = heap-array-map p f
  (heap-typing-upd d h) for h
    by (simp add: fun-eq-iff)
    show heap-array (st s) p = ZERO(-)
    if p:  $\forall a \in \text{ptr-span } p. \text{root-ptr-valid (hrs-htd (t-hrs s)) (PTR(stack-byte) a)}$  for
  s
    apply (intro array-ext)
    apply (simp add: array-index-zero)

```

```

apply (intro sim.sim-stack-stack-byte-zero')
subgoal for i
  using ptr-span-array-ptr-index-subset-ptr-span[of i p False] p
  apply auto
  done
done
show sim.ptr-valid (heap-typing h) p  $\implies$  valid-array h p for h
  by (rule sim.valid-array-of-ptr-valid-arrayI[OF f, of heap-typing h])
    (simp-all add: sim.ptr-valid-imp-v)
qed simp
qed

lemma (in typ-heap-simulation-open-types) stack-simulation-heap-typingI:
  assumes hrs: heap-typing-simulation T t-hrs t-hrs-update heap-typing heap-typing-upd
  st
  assumes sim-stack-alloc-heap-typing:
     $\bigwedge p d s n.$ 
     $(p, d) \in \text{stack-allocs } n \mathcal{S} \text{ TYPE}'a) (\text{hrs-htd } (t\text{-hrs } s)) \implies$ 
     $st (\text{t-hrs-update } (\text{hrs-mem-update } (\text{fold } (\lambda i. \text{heap-update } (p +_p \text{int } i)$ 
c-type-class.zero) [0..n])  $\circ$  hrs-htd-update ( $\lambda \cdot. d$ )) s) =
    (heap-typing-upd ( $\lambda \cdot. d$ ) (st s))
  assumes sim-stack-release-heap-typing:
     $\bigwedge (p::'a \text{ ptr}) s n. (\bigwedge i. i < n \implies \text{root-ptr-valid } (\text{hrs-htd } (t\text{-hrs } s)) (p +_p \text{int } i))$ 
 $\implies$ 
     $st (\text{t-hrs-update } (\text{hrs-htd-update } (\text{stack-releases } n p)) s) =$ 
    heap-typing-upd (stack-releases n p)
    ( $st (\text{t-hrs-update } (\text{hrs-mem-update } (\text{fold } (\lambda i. \text{heap-update } (p +_p \text{int } i)$ 
c-type-class.zero) [0..n])) s))
  shows stack-simulation-heap-typing st r w t-hrs t-hrs-update heap-typing heap-typing-upd
  S T
proof –
  interpret hrs: heap-typing-simulation T t-hrs t-hrs-update heap-typing heap-typing-upd
  st by fact
  interpret write-simulation t-hrs t-hrs-update st  $\lambda t p.$  open-types.ptr-valid T
  (heap-typing t) p w
  apply (rule hrs.write-simulation-alt)
  apply simp
  apply (simp-all add: ptr-valid.ptr-valid-c-guard ptr-valid-imp-v write-commutes
  read-commutes)
  done
  interpret typ-heap-simulation
  st r w  $\lambda t p.$  open-types.ptr-valid T (heap-typing t) p t-hrs t-hrs-update
  by unfold-locales
    (simp-all add: ptr-valid.ptr-valid-c-guard ptr-valid-imp-v write-commutes
  read-commutes)
  show ?thesis
  proof
  qed (simp-all add: sim-stack-stack-byte-zero' sim-stack-alloc-heap-typing sim-stack-release-heap-typing)
qed

```

end

theory *NatBitwise*

imports

More-Lib

begin

instantiation *nat* :: *lsb*

begin

definition

lsb *x* = *lsb* (*int* *x*)

instance

by *intro-classes* (*meson even-of-nat lsb-nat-def lsb-odd*)

end

instantiation *nat* :: *msb*

begin

definition

msb *x* = *msb* (*int* *x*)

instance ..

end

instantiation *nat* :: *set-bit*

begin

definition

set-bit *x* *y* *z* = *nat* (*set-bit* (*int* *x*) *y* *z*)

instance

by *intro-classes*

(*metis* (*mono-tags*) *set-bit-nat-def bin-nth-sc-gen bin-sc-pos*
bit-nat-iff exp-eq-0-imp-not-bit int-eq-iff)

end

lemma *nat-2p-eq-shiffl*:

$(2::nat)^x = 1 \ll x$

by *simp*

lemma *shiffl-nat-def*:

$(x::nat) \ll y = nat (int x \ll y)$

by (*simp add: nat-int-mul push-bit-eq-mult shiffl-def*)

lemma *nat-shiffl-less-cancel*:

$n \leq m \implies ((x :: nat) \ll n < y \ll m) = (x < y \ll (m - n))$

apply (*simp add: nat-int-comparison(2) shiffl-nat-def shiffl-def*)

by (*metis int-shiffl-less-cancel shiffl-def*)

lemma *nat-shiffl-lt-2p-bits*:

$(x::nat) < 1 \ll n \implies \forall i \geq n. \neg x !! i$

apply (*clarsimp simp: shiffl-nat-def zless-nat-eq-int-zless
dest!: le-Suc-ex*)

by (*metis bit-take-bit-iff not-add-less1 take-bit-nat-eq-self-iff*)

lemmas *nat-eq-test-bit = bit-eq-iff*

lemmas *nat-eq-test-bitI = bit-eq-iff[THEN iffD2, rule-format]*

end

Chapter 22

WA phase: Word Abstraction

```
theory WordAbstract
imports
  L2Peephole
  NatBitwise
begin
```

22.1 Basic Definitions

definition WORD-MAX $x \equiv ((2 \wedge (\text{len-of } x - 1) - 1) :: \text{int})$

definition WORD-MIN $x \equiv (- (2 \wedge (\text{len-of } x - 1)) :: \text{int})$

definition UWORD-MAX $x \equiv ((2 \wedge (\text{len-of } x)) - 1 :: \text{nat})$

lemma WORD-values [simplified]:

$$\text{WORD-MAX } (\text{TYPE}(8 \text{ signed})) = (2 \wedge 7 - 1)$$

$$\text{WORD-MAX } (\text{TYPE}(16 \text{ signed})) = (2 \wedge 15 - 1)$$

$$\text{WORD-MAX } (\text{TYPE}(32 \text{ signed})) = (2 \wedge 31 - 1)$$

$$\text{WORD-MAX } (\text{TYPE}(64 \text{ signed})) = (2 \wedge 63 - 1)$$

$$\text{WORD-MIN } (\text{TYPE}(8 \text{ signed})) = - (2 \wedge 7)$$

$$\text{WORD-MIN } (\text{TYPE}(16 \text{ signed})) = - (2 \wedge 15)$$

$$\text{WORD-MIN } (\text{TYPE}(32 \text{ signed})) = - (2 \wedge 31)$$

$$\text{WORD-MIN } (\text{TYPE}(64 \text{ signed})) = - (2 \wedge 63)$$

$$\text{UWORD-MAX } (\text{TYPE}(8)) = (2 \wedge 8 - 1)$$

$$\text{UWORD-MAX } (\text{TYPE}(16)) = (2 \wedge 16 - 1)$$

$$\text{UWORD-MAX } (\text{TYPE}(32)) = (2 \wedge 32 - 1)$$

$$\text{UWORD-MAX } (\text{TYPE}(64)) = (2 \wedge 64 - 1)$$

by (auto simp: WORD-MAX-def WORD-MIN-def UWORD-MAX-def)

lemmas WORD-values-add1 =

WORD-values [THEN arg-cong [where $f = \lambda x. x + 1$],
simplified semiring-norm, simplified numeral-One]

lemmas WORD-values-minus1 =

WORD-values [*THEN arg-cong* [**where** $f = \lambda x. x - 1$],
simplified semiring-norm, simplified numeral-One nat-numeral]

lemmas *WORD-values-fold* [*L1unfold*] =
WORD-values [*symmetric*]
WORD-values-add1 [*symmetric*]
WORD-values-minus1 [*symmetric*]

lemma *WORD-signed-to-unsigned* [*simp*]:
WORD-MAX TYPE('a signed) = WORD-MAX TYPE('a::len)
WORD-MIN TYPE('a signed) = WORD-MIN TYPE('a::len)
UWORD-MAX TYPE('a signed) = UWORD-MAX TYPE('a::len)
by (*auto simp: WORD-MAX-def WORD-MIN-def UWORD-MAX-def*)

lemma *INT-MIN-comparisons* [*simp*]:
 $\llbracket a \leq - (2 \wedge (\text{len-of } \text{TYPE}('a) - 1)) \rrbracket \implies a \leq \text{WORD-MIN } (\text{TYPE}('a::\text{len}))$
 $a < - (2 \wedge (\text{len-of } \text{TYPE}('a) - 1)) \implies a < \text{WORD-MIN } (\text{TYPE}('a::\text{len}))$
 $a \geq - (2 \wedge (\text{len-of } \text{TYPE}('a) - 1)) \implies a \geq \text{WORD-MIN } (\text{TYPE}('a::\text{len}))$
 $a > - (2 \wedge (\text{len-of } \text{TYPE}('a) - 1)) \implies a \geq \text{WORD-MIN } (\text{TYPE}('a::\text{len}))$
by (*auto simp: WORD-MIN-def*)

lemma *INT-MAX-comparisons* [*simp*]:
 $a \leq (2 \wedge (\text{len-of } \text{TYPE}('a) - 1)) - 1 \implies a \leq \text{WORD-MAX } (\text{TYPE}('a::\text{len}))$
 $a < (2 \wedge (\text{len-of } \text{TYPE}('a) - 1)) - 1 \implies a < \text{WORD-MAX } (\text{TYPE}('a::\text{len}))$
 $a \geq (2 \wedge (\text{len-of } \text{TYPE}('a) - 1)) - 1 \implies a \geq \text{WORD-MAX } (\text{TYPE}('a::\text{len}))$
 $a > (2 \wedge (\text{len-of } \text{TYPE}('a) - 1)) - 1 \implies a \geq \text{WORD-MAX } (\text{TYPE}('a::\text{len}))$
by (*auto simp: WORD-MAX-def*)

lemma *UINT-MAX-comparisons* [*simp*]:
 $x \leq (2 \wedge (\text{len-of } \text{TYPE}('a))) - 1 \implies x \leq \text{UWORD-MAX } (\text{TYPE}('a::\text{len}))$
 $x < (2 \wedge (\text{len-of } \text{TYPE}('a))) - 1 \implies x < \text{UWORD-MAX } (\text{TYPE}('a::\text{len}))$
 $x \geq (2 \wedge (\text{len-of } \text{TYPE}('a))) - 1 \implies x \geq \text{UWORD-MAX } (\text{TYPE}('a::\text{len}))$
 $x > (2 \wedge (\text{len-of } \text{TYPE}('a))) - 1 \implies x > \text{UWORD-MAX } (\text{TYPE}('a::\text{len}))$
by (*auto simp: UWORD-MAX-def*)

lemma *is-up-SCAST-same-signed* [*simp*]: *is-up* (*SCAST* (('a::len) \rightarrow 'a signed))
unfolding *is-up*
by *simp*

lemma *sint-ucast-signed* [*simp, L2opt*]: *sint* (*UCAST* ('a::len \rightarrow 'a signed) x) =
sint x
using *is-up-SCAST-same-signed scast-ucast-norm(2) sint-up-scast*
by (*metis scast-scast-id(1)*)

22.2 Abstracting values and expressions

definition *introduce-typ-abs-fn* $f \equiv \text{True}$

declare *introduce-typ-abs-fn-def* [*simp*]

lemma *introduce-typ-abs-fn*:
introduce-typ-abs-fn *f*
by *simp*

definition

abstract-bool-binop :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'c* ⇒ *'a*)
⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'c* ⇒ *'c* ⇒ *bool*) ⇒ *bool*

where

abstract-bool-binop *P f X X'* ≡ ∀ *a b*. *P* (*f a*) (*f b*) → (*X' a b* = *X* (*f a*) (*f b*))

definition

abstract-binop :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'c* ⇒ *'a*)
⇒ (*'a* ⇒ *'a* ⇒ *'a*) ⇒ (*'c* ⇒ *'c* ⇒ *'c*) ⇒ *bool*

where

abstract-binop *P f X X'* ≡ ∀ *a b*. *P* (*f a*) (*f b*) → (*f* (*X' a b*) = *X* (*f a*) (*f b*))

definition *abstract-val* *P a f b* ≡ *P* → (*a* = *f b*)

definition *abs-var a f b* ≡ *abstract-val True a f b*

lemmas *basic-abstract-defs* =

abstract-bool-binop-def
abstract-binop-def
abstract-val-def
abs-var-def

lemma *abstract-val-trivial*:

abstract-val True (*f b*) *f b*
by (*simp add: basic-abstract-defs*)

lemma *abstract-binop-is-abstract-val*:

abstract-binop P f X X' = (∀ *a b*. *abstract-val* (*P* (*f a*) (*f b*)) (*X* (*f a*) (*f b*)) *f* (*X' a b*))
by (*auto simp add: basic-abstract-defs*)

lemma *abstract-expr-bool-binop*:

[[*abstract-bool-binop E f X X'*;
introduce-typ-abs-fn f;
abstract-val P a f a';
abstract-val Q b f b']] ⇒
abstract-val (*P* ∧ *Q* ∧ *E a b*) (*X a b*) *id* (*X' a' b'*)

by (clarsimp simp add: basic-abstract-defs)

lemma *abstract-expr-binop*:

[[*abstract-binop* $E f X X'$;
 abstract-val $P a f a'$;
 abstract-val $Q b f b'$]] \implies
 abstract-val $(P \wedge Q \wedge E a b) (X a b) f (X' a' b')$

by (clarsimp simp add: basic-abstract-defs)

lemma *unat-abstract-bool-binops*:

abstract-bool-binop $(\lambda - -. True) (unat :: ('a::len) word \Rightarrow nat) (<) (<)$
abstract-bool-binop $(\lambda - -. True) (unat :: ('a::len) word \Rightarrow nat) (\leq) (\leq)$
abstract-bool-binop $(\lambda - -. True) (unat :: ('a::len) word \Rightarrow nat) (=) (=)$

by (auto simp: word-less-nat-alt word-le-nat-alt eq-iff basic-abstract-defs)

lemmas *unat-mult-simple = iffD1 [OF unat-mult-lem [unfolded word-bits-len-of]]*

lemma *le-to-less-plus-one*:

$((a::nat) \leq b) = (a < b + 1)$

by *arith*

lemma *unat-abstract-binops*:

abstract-binop $(\lambda a b. a + b \leq \text{WORD-MAX TYPE}('a::len)) (unat :: 'a word \Rightarrow nat) (+) (+)$

abstract-binop $(\lambda a b. a * b \leq \text{WORD-MAX TYPE}('a)) (unat :: 'a word \Rightarrow nat) (*) (*)$

abstract-binop $(\lambda a b. a \geq b) (unat :: 'a word \Rightarrow nat) (-) (-)$

abstract-binop $(\lambda a b. True) (unat :: 'a word \Rightarrow nat) (div) (div)$

abstract-binop $(\lambda a b. True) (unat :: 'a word \Rightarrow nat) (mod) (mod)$

by (auto simp: unat-plus-iff' unat-div unat-mod WORD-MAX-def le-to-less-plus-one
 WordAbstract.unat-mult-simple word-bits-def unat-sub word-le-nat-alt
 basic-abstract-defs)

lemma *unat-of-int*:

$[[i \geq 0; i < 2 \wedge \text{LENGTH}('a)]] \implies unat (of-int i :: 'a::len word) = nat i$

by (metis nat-less-numeral-power-cancel-iff of-nat-nat unat-of-nat-len)

lemma *unat-of-int-signed*:

$[[i \geq 0; i < 2 \wedge \text{LENGTH}('a)]] \implies unat (of-int i :: 'a::len signed word) = nat i$

by (simp add: unat-of-int)

lemma *nat-sint*:

$0 <= s (x :: 'a::len signed word) \implies nat (sint x) = unat x$

apply (subst unat-of-int-signed[where 'a='a, symmetric])

 apply (simp add: word-sle-def)

 apply (rule less-trans[OF sint-lt])

 apply simp

apply simp

done

lemma *int-unat-nonneg*:

$0 \leq s (x :: 'a::len \text{ signed word}) \implies \text{int} (\text{unat } x) = \text{sint } x$
by (*simp add: int-unat word-sle-msb-le sint-eq-uint*)

lemma *uint-sint-nonneg*:

$0 \leq s (x :: 'a::len \text{ signed word}) \implies \text{uint } x = \text{sint } x$
by (*simp add: int-unat word-sle-msb-le sint-eq-uint*)

lemma *unat-bitwise-abstract-binops*:

abstract-binop ($\lambda a b. \text{True}$) (*unat* :: *'a::len word* \Rightarrow *nat*) *Bit-Operations.and*
Bit-Operations.and
abstract-binop ($\lambda a b. \text{True}$) (*unat* :: *'a::len word* \Rightarrow *nat*) *Bit-Operations.or* *Bit-Operations.or*
abstract-binop ($\lambda a b. \text{True}$) (*unat* :: *'a::len word* \Rightarrow *nat*) *Bit-Operations.xor*
Bit-Operations.xor
apply (*simp add: unsigned-and-eq uint-nat unat-of-int basic-abstract-defs*)
apply (*simp add: unsigned-or-eq uint-nat unat-of-int basic-abstract-defs*)
apply (*simp add: unsigned-xor-eq uint-nat unat-of-int basic-abstract-defs*)
done

lemma *abstract-val-unsigned-bitNOT*:

abstract-val *P* *x* *unat* (*x'* :: *'a::len word*) \implies
abstract-val *P* (*UWORD-MAX TYPE*(*'a*) - *x*) *unat* (*NOT* *x'*)
apply (*clarsimp simp: UWORD-MAX-def NOT-eq basic-abstract-defs*)
by (*metis nat-le-Suc-less diff-Suc-eq-diff-pred mask-eq-sum-exp mask-eq-sum-exp-nat*

minus-diff-commute unat-lt2p unat-minus-one-word unat-sub-if-size unsigned-0)

lemma *sint-abstract-bool-binops*:

abstract-bool-binop ($\lambda - . \text{True}$) (*sint* :: *'a::len signed word* \Rightarrow *int*) (<) (*word-sless*)
abstract-bool-binop ($\lambda - . \text{True}$) (*sint* :: *'a signed word* \Rightarrow *int*) (\leq) (*word-sle*)
abstract-bool-binop ($\lambda - . \text{True}$) (*sint* :: *'a signed word* \Rightarrow *int*) (=) (=)
by (*auto simp: word-sless-def word-sle-def less-le basic-abstract-defs*)

lemma *sint-abstract-binops*:

abstract-binop ($\lambda a b. \text{WORD-MIN TYPE}$ (*'a::len*) $\leq a + b \wedge a + b \leq \text{WORD-MAX TYPE}$ (*'a*)) (*sint* :: *'a signed word* \Rightarrow *int*) (+) (+)
abstract-binop ($\lambda a b. \text{WORD-MIN TYPE}$ (*'a*) $\leq a * b \wedge a * b \leq \text{WORD-MAX TYPE}$ (*'a*)) (*sint* :: *'a signed word* \Rightarrow *int*) (*) (*)
abstract-binop ($\lambda a b. \text{WORD-MIN TYPE}$ (*'a*) $\leq a - b \wedge a - b \leq \text{WORD-MAX TYPE}$ (*'a*)) (*sint* :: *'a signed word* \Rightarrow *int*) (-) (-)
abstract-binop ($\lambda a b. \text{WORD-MIN TYPE}$ (*'a*) $\leq a \text{ sdiv } b \wedge a \text{ sdiv } b \leq \text{WORD-MAX TYPE}$ (*'a*)) (*sint* :: *'a signed word* \Rightarrow *int*) (sdiv) (sdiv)
abstract-binop ($\lambda a b. \text{WORD-MIN TYPE}$ (*'a*) $\leq a \text{ smod } b \wedge a \text{ smod } b \leq \text{WORD-MAX TYPE}$ (*'a*)) (*sint* :: *'a signed word* \Rightarrow *int*) (smod) (smod)
by (*auto simp: signed-arith-sint word-size WORD-MIN-def WORD-MAX-def basic-abstract-defs*)

lemma *sint-bitwise-abstract-binops*:
abstract-binop ($\lambda a b. \text{True}$) (*sint* :: 'a::len signed word \Rightarrow int) *Bit-Operations.and*
Bit-Operations.and
abstract-binop ($\lambda a b. \text{True}$) (*sint* :: 'a::len signed word \Rightarrow int) *Bit-Operations.or*
Bit-Operations.or
abstract-binop ($\lambda a b. \text{True}$) (*sint* :: 'a::len signed word \Rightarrow int) *Bit-Operations.xor*
Bit-Operations.xor
apply (*fastforce intro: int-eq-test-bitI*
simp: nth-sint bin-nth-ops basic-abstract-defs)
done

lemma *abstract-val-signed-bitNOT*:
abstract-val *P* *x* *sint* (*x'* :: 'a::len signed word) \Longrightarrow
abstract-val *P* (*NOT* *x*) *sint* (*NOT* *x'*)
by (*auto intro: int-eq-test-bitI*
simp add: nth-sint bin-nth-ops word-nth-neq basic-abstract-defs min-less-iff-disj)

lemma *abstract-val-signed-unary-minus*:
 $\llbracket \text{abstract-val } P \ r \ \text{sint } r' \rrbracket \Longrightarrow$
abstract-val (*P* \wedge ($- r$) \leq *WORD-MAX TYPE('a)*) ($- r$) *sint* ($- (r' :: ('a$
:: len) signed word))
apply (*clarsimp simp add: basic-abstract-defs*)
using *sint-range-size* [**where** *w=r'*]
apply $-$
apply (*subst signed-arith-sint*)
apply (*clarsimp simp: word-size WORD-MAX-def*)
apply *simp*
done

lemma *bang-big-nonneg*:
 $\llbracket 0 \leq s \ (x::'a::len \text{signed word}); n \geq \text{size } x - 1 \rrbracket \Longrightarrow (x !! n) = \text{False}$
apply (*cases n = size x - 1*)
apply (*simp add: word-size msb-nth*[**where** 'a='a signed, symmetric, simplified]
word-sle-msb-le)
apply (*simp add: test-bit-bl*)
apply *arith*
done

lemma *int-shiftr-nth*[*simp*]:
 $(i \gg n) !! m = i !! (n + m)$ **for** *i* :: int
by (*simp add: shiftr-def bin-nth-shiftr*)

lemma *int-shiffl-nth*[*simp*]:
 $(i \ll n) !! m = (n \leq m \wedge i !! (m - n))$ **for** *i* :: int
by (*simp add: shiffl-def bin-nth-shiffl*)

lemma *sint-shiftr-nonneg*:

$\llbracket 0 \leq_s (x :: 'a::len \text{ signed word}); 0 \leq n; n < LENGTH('a) \rrbracket \implies \text{sint } (x \gg n) = \text{sint } x \gg n$

apply (*rule int-eq-test-bitI*)

apply (*clarsimp simp: bang-big-nonneg[simplified word-size] nth-sint nth-shiftr field-simps*)

simp del: bit-signed-iff)

done

lemma *abstract-val-unsigned-unary-minus*:

$\llbracket \text{abstract-val } P \ r \ \text{unat } r' \rrbracket \implies$

$\text{abstract-val } P \ (\text{if } r = 0 \text{ then } 0 \text{ else } UWORD\text{-MAX } TYPE('a::len) + 1 - r)$
 $\text{unat } (- (r' :: 'a \text{ word}))$

by (*clarsimp simp: unat-minus' word-size unat-eq-zero UWORD-MAX-def basic-abstract-defs*)

lemma *abstract-val-signed-shiftr-signed*:

$\llbracket \text{abstract-val } Px \ x \ \text{sint } (x' :: ('a :: len) \text{ signed word});$

$\text{abstract-val } Pn \ n \ \text{sint } (n' :: ('b :: len) \text{ signed word}) \rrbracket \implies$

$\text{abstract-val } (Px \wedge Pn \wedge 0 \leq x \wedge 0 \leq n \wedge n < LENGTH('a))$

$(x \gg (nat \ n)) \ \text{sint } (x' \gg (\text{unat } n'))$

apply (*clarsimp simp only: abstract-val-def*)

apply (*subst nat-sint, simp add: word-sle-def*)

apply (*subst sint-shiftr-nonneg*)

apply (*simp add: word-sle-def*)

apply *simp*

apply (*subst SMT.nat-int-comparison(2)*)

apply (*subst int-unat-nonneg*)

apply (*simp add: word-sle-def*)

apply *assumption*

apply (*rule refl*)

done

lemma *abstract-val-signed-shiftr-unsigned*:

$\llbracket \text{abstract-val } Px \ x \ \text{sint } (x' :: ('a :: len) \text{ signed word});$

$\text{abstract-val } Pn \ n \ \text{unat } (n' :: ('b :: len) \text{ word}) \rrbracket \implies$

$\text{abstract-val } (Px \wedge Pn \wedge 0 \leq x \wedge n < LENGTH('a))$

$(x \gg n) \ \text{sint } (x' \gg \text{unat } n')$

apply (*clarsimp simp: shiftr-int-def basic-abstract-defs*)

apply (*subst sint-shiftr-nonneg*)

apply (*simp add: word-sle-def*)

apply *simp*

apply *assumption*

apply (*clarsimp simp: shiftr-int-def*)

done

```

lemma foo:  $\neg n - i < \text{LENGTH}('a::\text{len}) - \text{Suc } 0 \implies$ 
   $n < \text{LENGTH}('a) - \text{Suc } 0 = \text{False}$ 
apply simp
done

lemma sint-shiffl-nonneg:
   $\llbracket 0 \leq s(x :: 'a::\text{len signed word}); n < \text{LENGTH}('a); \text{sint } x \ll n < 2^{\text{LENGTH}('a)}$ 
 $- 1 \rrbracket \implies$ 
   $\text{sint } (x \ll n) = \text{sint } x \ll n$ 
apply (rule int-eq-test-bitI)
subgoal for na
apply (simp add: nth-sint nth-shiffl word-sle-def int-shiffl-less-cancel int-2p-eq-shiffl
  bang-big-nonneg[simplified word-size]
  del: bit-signed-iff shiffl-1)

apply (intro impI iffI conjI; (solves simp)?)
apply (drule(1) int-shiffl-lt-2p-bits[rotated])
apply (clarsimp simp: nth-sint)
apply (drule-tac x= $\text{LENGTH}('a) - 1 - n$  in spec)
apply (subgoal-tac  $\text{LENGTH}('a) - 1 - n < \text{LENGTH}('a) - 1$ )
apply simp
apply arith
apply (drule(1) int-shiffl-lt-2p-bits[rotated])
apply (clarsimp simp: nth-sint)
apply (drule-tac x=na - n in spec)
apply simp
apply (cases n = 0)
apply (simp add: word-sle-msb-le[where x=0, simplified word-sle-def, simpli-
fied] msb-nth)
apply (drule(1) int-shiffl-lt-2p-bits[rotated])
apply (clarsimp simp: nth-sint)
apply (drule-tac x= $\text{LENGTH}('a) - 1 - n$  in spec)
apply (subgoal-tac  $\text{LENGTH}('a) - 1 - n < \text{LENGTH}('a) - 1$ )
apply simp
apply simp
done
done

lemma abstract-val-signed-shiffl-signed:
   $\llbracket \text{abstract-val } Px \ x \ \text{sint } (x' :: ('a :: \text{len}) \ \text{signed word});$ 
   $\text{abstract-val } Pn \ n \ \text{sint } (n' :: ('b :: \text{len}) \ \text{signed word}) \rrbracket \implies$ 
   $\text{abstract-val } (Px \wedge Pn \wedge 0 \leq x \wedge 0 \leq n \wedge n < \text{int } \text{LENGTH}('a) \wedge x \ll \text{nat}$ 
 $n < 2^{\text{LENGTH}('a) - 1})$ 
   $(x \ll \text{nat } n) \ \text{sint } (x' \ll \text{unat } n')$ 
apply (clarsimp simp add: basic-abstract-defs)
by (metis One-nat-def len-gt-0 nat-int nat-sint signed-0 sint-shiffl-nonneg word-sle-eq
  zless-nat-conj)

```

lemma *abstract-val-signed-shiffl-unsigned*:
 $\llbracket \text{abstract-val } Px \ x \ \text{sint} \ (x' :: ('a :: \text{len}) \ \text{signed} \ \text{word});$
 $\text{abstract-val } Pn \ n \ \text{unat} \ (n' :: ('b :: \text{len}) \ \text{word}) \rrbracket \implies$
 $\text{abstract-val} \ (Px \wedge Pn \wedge 0 \leq x \wedge n < \text{LENGTH}('a) \wedge x \ll n < 2^{\wedge}(\text{LENGTH}('a)$
 $- 1))$
 $(x \ll n) \ \text{sint} \ (x' \ll \text{unat } n')$
by (*clarsimp simp: sint-shiffl-nonneg word-sle-def basic-abstract-defs*
nat-less-eq-zless[where z=int LENGTH('a), simplified])

lemma *abstract-val-unsigned-shiftr-unsigned*:
 $\llbracket \text{abstract-val } Px \ x \ \text{unat} \ (x' :: ('a :: \text{len}) \ \text{word});$
 $\text{abstract-val } Pn \ n \ \text{unat} \ (n' :: ('a :: \text{len}) \ \text{word}) \rrbracket \implies$
 $\text{abstract-val} \ (Px \wedge Pn) \ (x \gg n) \ \text{unat} \ (x' \gg \text{unat } n')$
apply (*clarsimp simp add: basic-abstract-defs*)
apply (*simp add: shiftr-div-2n' shiftr-int-def*)
using *shiftr-eq-div by blast*

lemma *abstract-val-unsigned-shiftr-signed*:
 $\llbracket \text{abstract-val } Px \ x \ \text{unat} \ (x' :: ('a :: \text{len}) \ \text{word});$
 $\text{abstract-val } Pn \ n \ \text{sint} \ (n' :: ('b :: \text{len}) \ \text{signed} \ \text{word}) \rrbracket \implies$
 $\text{abstract-val} \ (Px \wedge Pn \wedge 0 \leq n) \ (x \gg \text{nat } n) \ \text{unat} \ (x' \gg \text{unat } n')$
apply (*clarsimp simp: shiftr-div-2n' shiftr-int-def basic-abstract-defs*)
by (*simp add: nat-sint shiftr-nat-def word-sle-eq*)

lemma *abstract-val-unsigned-shiffl-unsigned*:
 $\llbracket \text{abstract-val } Px \ x \ \text{unat} \ (x' :: ('a :: \text{len}) \ \text{word});$
 $\text{abstract-val } Pn \ n \ \text{unat} \ (n' :: ('b :: \text{len}) \ \text{word}) \rrbracket \implies$
 $\text{abstract-val} \ (Px \wedge Pn \wedge n < \text{LENGTH}('a) \wedge x \ll n < 2^{\wedge}(\text{LENGTH}('a))$
 $(x \ll n) \ \text{unat} \ (x' \ll \text{unat } n')$
by (*clarsimp simp: shiffl-t2n Word-Lemmas-Internal.shiffl-nat-def unat-mult-simple*
field-simps basic-abstract-defs)

lemma *abstract-val-unsigned-shiffl-signed*:
 $\llbracket \text{abstract-val } Px \ x \ \text{unat} \ (x' :: ('a :: \text{len}) \ \text{word});$
 $\text{abstract-val } Pn \ n \ \text{sint} \ (n' :: ('b :: \text{len}) \ \text{signed} \ \text{word}) \rrbracket \implies$
 $\text{abstract-val} \ (Px \wedge Pn \wedge 0 \leq n \wedge n < \text{int} \ (\text{LENGTH}('a)) \wedge x \ll \text{nat } n <$
 $2^{\wedge}(\text{LENGTH}('a))$
 $(x \ll \text{nat } n) \ \text{unat} \ (x' \ll \text{unat } n')$
apply (*clarsimp simp: shiffl-t2n Word-Lemmas-Internal.shiffl-nat-def unat-mult-simple*
field-simps basic-abstract-defs)
apply (*simp add: sint-eq-uint word-msb-sint*)
by (*metis Word.of-nat-unat len-gt-0 nat-int unat-of-nat-len unat-power-lower*
word-arith-nat-mult
zless-nat-conj)

lemma *signed-shiffl-c-guard-simp* :
 $\llbracket \text{int } \text{bound} < 2^{\wedge}(\text{LENGTH}('a)); a * 2^{\wedge}b < \text{int } \text{bound}; 0 \leq a \rrbracket \implies$
 $\text{unat} \ (\text{of-int } a :: 'a::\text{len} \ \text{word}) * 2^{\wedge}b < \text{bound}$

```

apply (subst unat-of-int)
  apply assumption
  apply (drule(1) less-trans)
  apply (subgoal-tac a * 2b < 2LENGTH('a) * 2b)
  apply simp
  apply (erule less-le-trans)
  apply simp
  apply (subgoal-tac nat (a * 2b) < nat (int bound))
  apply (simp add: nat-power-eq nat-mult-distrib)
  apply (subst nat-mono-iff)
  apply (rule le-less-trans, assumption)
  apply (erule le-less-trans[rotated])
  apply (simp add: mult-left-mono[where a=1::int, simplified])
apply simp
done

```

```

lemmas abstract-val-signed-ops [simplified simp-thms] =
  abstract-expr-bool-binop [OF snat-abstract-bool-binops(1)]
  abstract-expr-bool-binop [OF snat-abstract-bool-binops(2)]
  abstract-expr-bool-binop [OF snat-abstract-bool-binops(3)]
  abstract-expr-binop [OF snat-abstract-binops(1)]
  abstract-expr-binop [OF snat-abstract-binops(2)]
  abstract-expr-binop [OF snat-abstract-binops(3)]
  abstract-expr-binop [OF snat-abstract-binops(4)]
  abstract-expr-binop [OF snat-abstract-binops(5)]
  abstract-expr-binop [OF sint-bitwise-abstract-binops(1)]
  abstract-expr-binop [OF sint-bitwise-abstract-binops(2)]
  abstract-expr-binop [OF sint-bitwise-abstract-binops(3)]
  abstract-val-signed-bitNOT
  abstract-val-signed-unary-minus
  abstract-val-signed-shiftr-signed
  abstract-val-signed-shiftr-unsigned
  abstract-val-signed-shiffl-signed
  abstract-val-signed-shiffl-unsigned

```

```

lemmas abstract-val-unsigned-ops [simplified simp-thms] =
  abstract-expr-bool-binop [OF unat-abstract-bool-binops(1)]
  abstract-expr-bool-binop [OF unat-abstract-bool-binops(2)]
  abstract-expr-bool-binop [OF unat-abstract-bool-binops(3)]
  abstract-expr-binop [OF unat-abstract-binops(1)]
  abstract-expr-binop [OF unat-abstract-binops(2)]
  abstract-expr-binop [OF unat-abstract-binops(3)]
  abstract-expr-binop [OF unat-abstract-binops(4)]
  abstract-expr-binop [OF unat-abstract-binops(5)]
  abstract-expr-binop [OF unat-bitwise-abstract-binops(1)]
  abstract-expr-binop [OF unat-bitwise-abstract-binops(2)]
  abstract-expr-binop [OF unat-bitwise-abstract-binops(3)]
  abstract-val-unsigned-bitNOT
  abstract-val-unsigned-unary-minus

```


abstract-val-unsigned-shiftr-signed
abstract-val-unsigned-shiftr-unsigned
abstract-val-unsigned-shiftl-signed
abstract-val-unsigned-shiftl-unsigned

lemma *mod-less*:

$(a :: \text{nat}) < c \implies a \text{ mod } b < c$
by (*metis less-trans mod-less-eq-dividend order-leE*)

lemma *abstract-val-ucast*:

$\llbracket \text{introduce-typ-abs-fn } (\text{unat} :: ('a::\text{len}) \text{ word} \Rightarrow \text{nat});$
 $\text{abstract-val } P \ v \ \text{unat } v' \rrbracket$
 $\implies \text{abstract-val } (P \wedge v \leq \text{nat } (\text{WORD-MAX TYPE}('a)))$
 $(\text{int } v) \ \text{sint } (\text{ucast } (v' :: 'a \ \text{word}) :: 'a \ \text{signed word})$
apply (*clarsimp simp: uint-nat [symmetric] basic-abstract-defs*)
apply (*subst sint-eq-uint*)
apply (*rule not-msb-from-less*)
apply (*clarsimp simp: word-less-nat-alt unat-ucast WORD-MAX-def le-to-less-plus-one*)
apply (*subst (asm) nat-diff-distrib*)
apply *simp*
apply *clarsimp*
apply *clarsimp*
apply (*metis of-nat-numeral nat-numeral nat-power-eq of-nat-0-le-iff*)
apply (*clarsimp simp: uint-up-ucast is-up*)
done

lemma *abstract-val-heap-sword-template*:

$\llbracket \text{introduce-typ-abs-fn } (\text{sint} :: ('a::\text{len}) \ \text{signed word} \Rightarrow \text{int});$
 $\text{abstract-val } P \ p' \ \text{id } p \rrbracket$
 $\implies \text{abstract-val } P \ (\text{sint } (\text{ucast } (\text{heap-get } s \ p' :: 'a \ \text{word}) :: 'a \ \text{signed word}))$
 $\text{sint } (\text{ucast } (\text{heap-get } s \ p) :: 'a \ \text{signed word})$
by (*simp add: basic-abstract-defs*)

lemma *abstract-val-scast*:

$\llbracket \text{introduce-typ-abs-fn } (\text{sint} :: ('a::\text{len}) \ \text{signed word} \Rightarrow \text{int});$
 $\text{abstract-val } P \ C' \ \text{sint } C \rrbracket$
 $\implies \text{abstract-val } (P \wedge 0 \leq C') \ (\text{nat } C') \ \text{unat } (\text{scast } (C :: ('a::\text{len}) \ \text{signed}$
 $\text{word}) :: ('a::\text{len}) \ \text{word})$
apply (*clarsimp simp: down-cast-same [symmetric] is-down unat-ucast basic-abstract-defs*)
apply (*subst sint-eq-uint*)
apply (*clarsimp simp: word-msb-sint*)
apply (*clarsimp simp: unat-def [symmetric]*)
apply (*subst word-unat.norm-Rep [symmetric]*)
apply *clarsimp*
done

lemma *abstract-val-scast-upcast*:

$\llbracket \text{len-of TYPE}('a::\text{len}) \leq \text{len-of TYPE}('b::\text{len});$

$$\text{abstract-val } P \ C' \ \text{sint } C \] \implies \text{abstract-val } P \ (C') \ \text{sint} \ (\text{scast } (C :: 'a \ \text{signed word}) :: 'b \ \text{signed word})$$
by (*clarsimp simp: down-cast-same [symmetric] sint-up-scast is-up basic-abstract-defs*)

lemma *abstract-val-scast-downcast:*

$$\llbracket \text{len-of } \text{TYPE}('b) < \text{len-of } \text{TYPE}('a::\text{len}); \text{abstract-val } P \ C' \ \text{sint } C \] \implies \text{abstract-val } P \ (\text{sbintrunc } ((\text{len-of } \text{TYPE}('b::\text{len}) - 1)) \ C') \ \text{sint} \ (\text{scast } (C :: 'a \ \text{signed word}) :: 'b \ \text{signed word})$$
by (*metis Word.of-int-sint abstract-val-def len-signed word-sbin.inverse-norm*)

lemma *abstract-val-ucast-upcast:*

$$\llbracket \text{len-of } \text{TYPE}('a::\text{len}) \leq \text{len-of } \text{TYPE}('b::\text{len}); \text{abstract-val } P \ C' \ \text{unat } C \] \implies \text{abstract-val } P \ (C') \ \text{unat} \ (\text{ucast } (C :: 'a \ \text{word}) :: 'b \ \text{word})$$
by (*clarsimp simp: is-up unat-ucast-upcast basic-abstract-defs*)

lemma *abstract-val-ucast-downcast:*

$$\llbracket \text{len-of } \text{TYPE}('b::\text{len}) < \text{len-of } \text{TYPE}('a::\text{len}); \text{abstract-val } P \ C' \ \text{unat } C \] \implies \text{abstract-val } P \ (C' \ \text{mod } (\text{UWORD-MAX } \text{TYPE}('b) + 1)) \ \text{unat} \ (\text{ucast } (C :: 'a \ \text{word}) :: 'b \ \text{word})$$
apply (*clarsimp simp: scast-def sint-uint UWORD-MAX-def basic-abstract-defs*)
unfolding *ucast-def unat-def*
apply (*subst int-word-uint*)
apply (*metis (mono-tags) uint-mod uint-power-lower unat-def unat-mod unat-power-lower*)
done

definition

$$\text{valid-typ-abs-fn } (P :: 'a \Rightarrow \text{bool}) \ (Q :: 'a \Rightarrow \text{bool}) \ (A :: 'c \Rightarrow 'a) \ (C :: 'a \Rightarrow 'c) \equiv (\forall v. P \ v \longrightarrow A \ (C \ v) = v) \wedge (\forall v. Q \ (A \ v) \longrightarrow C \ (A \ v) = v)$$

declare *valid-typ-abs-fn-def [simp]*

lemma *valid-typ-abs-fn-id:*

$$\text{valid-typ-abs-fn } (\lambda-. \ \text{True}) \ (\lambda-. \ \text{True}) \ \text{id} \ \text{id}$$
by *clarsimp*

lemma *valid-typ-abs-fn-unit:*

$$\text{valid-typ-abs-fn } (\lambda-. \ \text{True}) \ (\lambda-. \ \text{True}) \ \text{id} \ (\text{id} :: \text{unit} \Rightarrow \text{unit})$$
by *clarsimp*

lemma *valid-typ-abs-fn-unat:*

$$\text{valid-typ-abs-fn } (\lambda v. \ v \leq \text{UWORD-MAX } \text{TYPE}('a::\text{len})) \ (\lambda-. \ \text{True}) \ (\text{unat} :: 'a \ \text{word} \Rightarrow \text{nat}) \ (\text{of-nat} :: \text{nat} \Rightarrow 'a \ \text{word})$$
supply *unsigned-of-nat [simp del]*
by (*clarsimp simp: unat-of-nat-eq UWORD-MAX-def le-to-less-plus-one*)

lemma *valid-typ-abs-fn-sint*:

valid-typ-abs-fn ($\lambda v. \text{WORD-MIN TYPE}('a::\text{len}) \leq v \wedge v \leq \text{WORD-MAX TYPE}('a)$)
($\lambda-. \text{True}$) (*sint* :: 'a signed word \Rightarrow int) (*of-int* :: int \Rightarrow 'a signed word)
by (*clarsimp simp: sint-of-int-eq WORD-MIN-def WORD-MAX-def*)

lemma *valid-typ-abs-fn-tuple*:

$\llbracket \text{valid-typ-abs-fn } P\text{-}a \text{ } Q\text{-}a \text{ abs-}a \text{ conc-}a; \text{valid-typ-abs-fn } P\text{-}b \text{ } Q\text{-}b \text{ abs-}b \text{ conc-}b \rrbracket$
 \Longrightarrow
valid-typ-abs-fn ($\lambda(a, b). P\text{-}a \text{ } a \wedge P\text{-}b \text{ } b$) ($\lambda(a, b). Q\text{-}a \text{ } a \wedge Q\text{-}b \text{ } b$) (*map-prod*
abs-}a \text{ abs-}b) (*map-prod conc-}a \text{ conc-}b*)
by *clarsimp*

lemma *valid-typ-abs-fn-tuple-split*:

$\llbracket \text{valid-typ-abs-fn } P\text{-}a \text{ } Q\text{-}a \text{ abs-}a \text{ conc-}a; \text{valid-typ-abs-fn } P\text{-}b \text{ } Q\text{-}b \text{ abs-}b \text{ conc-}b \rrbracket$
 \Longrightarrow
valid-typ-abs-fn ($\lambda(a, b). P\text{-}a \text{ } a \wedge P\text{-}b \text{ } b$) ($\lambda(a, b). Q\text{-}a \text{ } a \wedge Q\text{-}b \text{ } b$) ($\lambda(a, b).$
(*abs-}a \text{ } a, \text{abs-}b \text{ } b*) (*map-prod conc-}a \text{ conc-}b*)
by *clarsimp*

lemma *introduce-typ-abs-fn-tuple*:

$\llbracket \text{introduce-typ-abs-fn } \text{abs-}a; \text{introduce-typ-abs-fn } \text{abs-}b \rrbracket \Longrightarrow$
introduce-typ-abs-fn (*map-prod abs-}a \text{ abs-}b*)
by *clarsimp*

lemma *valid-typ-abs-fn-sum*:

$\llbracket \text{valid-typ-abs-fn } P\text{-}a \text{ } Q\text{-}a \text{ abs-}a \text{ conc-}a; \text{valid-typ-abs-fn } P\text{-}b \text{ } Q\text{-}b \text{ abs-}b \text{ conc-}b \rrbracket$
 \Longrightarrow
valid-typ-abs-fn (*case-sum P-}a \text{ P-}b*) (*case-sum Q-}a \text{ Q-}b*) (*map-sum abs-}a*
abs-}b) (*map-sum conc-}a \text{ conc-}b*)
by (*auto simp add: map-sum-def split: sum.splits*)

lemma *introduce-typ-abs-fn-sum*:

$\llbracket \text{introduce-typ-abs-fn } \text{abs-}a; \text{introduce-typ-abs-fn } \text{abs-}b \rrbracket \Longrightarrow$
introduce-typ-abs-fn (*map-sum abs-}a \text{ abs-}b*)
by *clarsimp*

22.3 Refinement Lemmas

named-theorems *word-abs*

definition

corresTA $P \text{ } rx \text{ } ex \text{ } A \text{ } C \equiv \text{corresXF } (\lambda s. s) (\lambda r \text{ } s. \text{ } rx \text{ } r) (\lambda r \text{ } s. \text{ } ex \text{ } r) \text{ } P \text{ } A \text{ } C$

definition *rel-word-abs* $ex \text{ } rx \equiv \text{rel-xval } (\lambda c \text{ } a. \text{ } a = ex \text{ } c) (\lambda c \text{ } a. \text{ } a = rx \text{ } c)$

lemma *rel-word-abs-simps*[*simp*]:

rel-word-abs $ex \text{ } rx \text{ } (\text{Result } rc) \text{ } (\text{Exn } la) = \text{False}$
rel-word-abs $ex \text{ } rx \text{ } (\text{Exn } lc) \text{ } (\text{Result } ra) = \text{False}$

$rel\text{-}word\text{-}abs\ ex\ rx\ (Result\ rc)\ (Result\ ra) = (ra = rx\ rc)$
 $rel\text{-}word\text{-}abs\ ex\ rx\ (Exn\ lc)\ (Exn\ la) = (la = ex\ lc)$
by (*auto simp add: rel-word-abs-def*)

lemma *corresTA-refines*:

corresTA $P\ rx\ ex\ f_a\ f_c \implies P\ s \implies refines\ f_c\ f_a\ s\ s\ (rel\text{-}prod\ (rel\text{-}word\text{-}abs\ ex\ rx)\ (=))$

unfolding *corresTA-def*

apply (*clarsimp simp add: corresXF-refines-conv rel-word-abs-def rel-xval.simps*)

apply (*clarsimp simp add: refines-def-old rel-xval.simps reaches-succeeds*)

by (*smt (verit) le-boolE le-boolI' linorder-not-le xval-split*)

lemma *refines-corresTA*:

assumes *sim*: $\bigwedge s. P\ s \implies refines\ f_c\ f_a\ s\ s\ (rel\text{-}prod\ (rel\text{-}word\text{-}abs\ ex\ rx)\ (=))$

shows *corresTA* $P\ rx\ ex\ f_a\ f_c$

unfolding *corresTA-def*

apply (*clarsimp simp add: corresXF-refines-conv rel-word-abs-def rel-xval.simps*)

using *sim*

apply (*fastforce simp add: refines-def-old rel-xval.simps reaches-succeeds rel-word-abs-def split: xval-splits*)

done

lemma *corresTA-refines-conv*:

corresTA $P\ rx\ ex\ f_a\ f_c \longleftrightarrow (\forall s. P\ s \longrightarrow refines\ f_c\ f_a\ s\ s\ (rel\text{-}prod\ (rel\text{-}word\text{-}abs\ ex\ rx)\ (=)))$

using *corresTA-refines refines-corresTA by metis*

lemma *admissible-nondet-ord-corresTA* [*corres-admissible*]:

ccpo.admissible $Inf\ (\ge)\ (\lambda A. corresTA\ P\ rx\ ex\ A\ C)$

unfolding *corresTA-def*

apply (*rule admissible-nondet-ord-corresXF*)

done

lemma *corresTA-top* [*corres-top*]: *corresTA* $P\ rx\ st\ \top\ C$

by (*auto simp add: corresTA-def corresXF-def*)

lemma *corresTA-assume-and-weaken-pre*:

assumes $A\text{-}C$: $\bigwedge s. P\ s \implies corresTA\ Q\ rt\ ex\ A\ C$

assumes $P\text{-}Q$: $\bigwedge s. P\ s \implies Q\ s$

shows *corresTA* $P\ rt\ ex\ A\ C$

unfolding *corresTA-def*

apply (*rule corresXF-assume-pre*)

apply (*rule corresXF-guard-imp*)

apply (*rule A-C [unfolded corresTA-def]*)

apply *assumption*

apply (*rule P-Q*)

apply *assumption*

done

lemma *corresTA-L2-gets*:

```
[[  $\wedge s. \text{abstract-val } (Q\ s) (C\ s) \text{ rx } (C'\ s)$  ]]  $\implies$   
  corresTA Q rx ex (L2-gets ( $\lambda s. C\ s$ ) n) (L2-gets ( $\lambda s. C'\ s$ ) n)  
unfolding L2-defs  
apply (clarsimp simp add: corresTA-refines-conv)  
apply (rule refines-gets)  
apply (simp add: abstract-val-def)  
done
```

lemma *corresTA-L2-modify*:

```
[[  $\wedge s. \text{abstract-val } (P\ s) (m\ s) \text{ id } (m'\ s)$  ]]  $\implies$   
  corresTA P rx ex (L2-modify ( $\lambda s. m\ s$ ) (L2-modify ( $\lambda s. m'\ s$ )))  
unfolding L2-defs  
apply (clarsimp simp add: corresTA-refines-conv)  
apply (rule refines-modify)  
apply (simp add: abstract-val-def)  
done
```

lemma *refines-throw*: $R (Exn\ x, s) (Exn\ y, t) \implies \text{refines } (\text{throw } x) (\text{throw } y) s\ t$
R

by (*auto simp add: refines-def-old Exn-def*)

lemma *corresTA-L2-throw*:

```
[[ abstract-val Q C ex C' ]]  $\implies$   
  corresTA ( $\lambda-. Q$ ) rx ex (L2-throw C n) (L2-throw C' n)  
unfolding L2-defs  
apply (clarsimp simp add: corresTA-refines-conv)  
apply (simp add: abstract-val-def)  
done
```

lemma *corresTA-L2-skip*:

```
corresTA ( $\lambda-. \text{True}$ ) rx ex L2-skip L2-skip  
unfolding L2-defs  
by (auto simp add: corresTA-refines-conv refines-gets)
```

lemma *corresTA-L2-fail*:

```
corresTA ( $\lambda-. \text{True}$ ) rx ex L2-fail L2-fail  
unfolding L2-defs  
by (auto simp add: corresTA-refines-conv)
```

lemma *corresTA-L2-seq'*:

```
fixes L' :: ('e, 'c1, 's) exn-monad  
fixes R' :: 'c1  $\Rightarrow$  ('e, 'c2, 's) exn-monad  
fixes L :: ('ea, 'a1, 's) exn-monad  
fixes R :: 'a1  $\Rightarrow$  ('ea, 'a2, 's) exn-monad  
shows
```

[[*corresTA* *P* *rx1* *ex* *L* *L'*;
 $\bigwedge r. \text{corresTA } (Q \text{ (rx1 } r)) \text{ rx2 } \text{ex } (R \text{ (rx1 } r)) \text{ (R' } r) \text{]}] \implies$
corresTA *P* *rx2* *ex*
 $(L2\text{-seq } L \text{ (}\lambda r. L2\text{-seq } (L2\text{-guard } (\lambda s. Q \text{ } r \text{ } s)) \text{ (}\lambda-. R \text{ } r))$
 $(L2\text{-seq } L' \text{ (}\lambda r. R' \text{ } r))$
apply *atomize*
apply (*clarsimp simp: L2-seq-def L2-guard-def corresTA-def*)
apply (*erule corresXF-join [where P'=λx y s. rx1 y = x]*)
subgoal
by (*auto simp add: corresXF-def reaches-bind reaches-guard succeeds-bind split: xval-splits*)
subgoal
by (*auto simp add: runs-to-partial-def-old split: xval-splits*)
subgoal by *simp*
done

lemma *corresTA-L2-seq:*

[[*introduce-typ-abs-fn* *rx1*;
 $PROP \text{ THIN } (Trueprop \text{ (corresTA } P \text{ (rx1 } :: 'a \Rightarrow 'b) \text{ ex } L \text{ } L'))$;
 $PROP \text{ THIN } (\bigwedge r r'. \text{abs-var } r \text{ rx1 } r' \implies \text{corresTA } (Q \text{ } r) \text{ rx2 } \text{ex } (R \text{ } r) \text{ (R' } r')) \text{]}] \implies$
 $\text{corresTA } P \text{ rx2 } \text{ex } (L2\text{-seq } L \text{ (}\lambda r. L2\text{-seq } (L2\text{-guard } (Q \text{ } r)) \text{ (}\lambda-. R \text{ } r))$ $(L2\text{-seq } L' \text{ } R')$
unfolding *THIN-def*
by (*rule corresTA-L2-seq', (simp add: basic-abstract-defs)+*)

lemma *corresTA-L2-seq-unused-result:*

[[*introduce-typ-abs-fn* *rx1*;
 $PROP \text{ THIN } (Trueprop \text{ (corresTA } P \text{ (rx1 } :: 'a \Rightarrow 'b) \text{ ex } L \text{ } L'))$;
 $PROP \text{ THIN } (Trueprop \text{ (corresTA } Q \text{ rx2 } \text{ex } R \text{ } R')) \text{]}] \implies$
 $\text{corresTA } P \text{ rx2 } \text{ex } (L2\text{-seq } L \text{ (}\lambda r. L2\text{-seq } (L2\text{-guard } Q) \text{ (}\lambda-. R))$ $(L2\text{-seq } L' \text{ (}\lambda-. R'))$
unfolding *THIN-def*
by (*rule corresTA-L2-seq', simp+*)

lemma *corresTA-L2-seq-unit:*

fixes *L' :: ('e, unit, 's) exn-monad*
fixes *R' :: unit \Rightarrow ('e, 'r, 's) exn-monad*
fixes *L :: ('ea, unit, 's) exn-monad*
fixes *R :: ('ea, 'ra, 's) exn-monad*

shows

[[$PROP \text{ THIN } (Trueprop \text{ (corresTA } P \text{ id } \text{ex } L \text{ } L'))$;
 $PROP \text{ THIN } (Trueprop \text{ (corresTA } Q \text{ rx } \text{ex } R \text{ (R' ()))) \text{]}] \implies$
 $\text{corresTA } P \text{ rx } \text{ex}$
 $(L2\text{-seq } L \text{ (}\lambda r. L2\text{-seq } (L2\text{-guard } Q) \text{ (}\lambda-. R))$
 $(L2\text{-seq } L' \text{ } R')$

unfolding *THIN-def*

by (*rule corresTA-L2-seq', simp+*)

lemma *corresTA-L2-catch'*:
fixes $L' :: ('e1, 'c, 's) \text{exn-monad}$
fixes $R' :: 'e1 \Rightarrow ('e2, 'c, 's) \text{exn-monad}$
fixes $L :: ('e1a, 'ca, 's) \text{exn-monad}$
fixes $R :: 'e1a \Rightarrow ('e2a, 'ca, 's) \text{exn-monad}$
shows
 $\llbracket \text{corresTA } P \text{ rx } ex1 \text{ L } L';$
 $\bigwedge r. \text{corresTA } (Q (ex1 \ r)) \text{ rx } ex2 \ (R (ex1 \ r)) \ (R' \ r) \rrbracket \Longrightarrow$
 $\text{corresTA } P \text{ rx } ex2 \ (\text{L2-catch } L \ (\lambda r. \text{L2-seq } (\text{L2-guard } (\lambda s. Q \ r \ s)) \ (\lambda -. R \ r)))$
 $(\text{L2-catch } L' \ (\lambda r. R' \ r))$
apply *atomize*
apply (*clarsimp simp: L2-seq-def L2-catch-def L2-guard-def corresTA-def*)
apply (*erule corresXF-except [where P'= $\lambda x \ y \ s. ex1 \ y = x$]*)
subgoal
by (*auto simp add: corresXF-def reaches-bind reaches-guard succeeds-bind split:*
xval-splits)
subgoal
by (*auto simp add: runs-to-partial-def-old split: xval-splits*)
subgoal by *simp*
done

lemma *corresTA-L2-catch*:
 $\llbracket \text{introduce-typ-abs-fn } ex1;$
 $\text{PROP THIN } (\text{Trueprop } (\text{corresTA } P \text{ rx } ex1 \text{ L } L'));$
 $\text{PROP THIN } (\bigwedge r \ r'. \text{abs-var } r \ ex1 \ r' \Longrightarrow \text{corresTA } (Q \ r) \text{ rx } ex2 \ (R \ r) \ (R' \ r'))$
 $\rrbracket \Longrightarrow$
 $\text{corresTA } P \text{ rx } ex2 \ (\text{L2-catch } L \ (\lambda r. \text{L2-seq } (\text{L2-guard } (\lambda s. Q \ r \ s)) \ (\lambda -. R \ r)))$
 $(\text{L2-catch } L' \ (\lambda r. R' \ r))$
unfolding *THIN-def*
by (*rule corresTA-L2-catch', (simp add: basic-abstract-defs)+*)

term *corresTA P rx ex f g*

lemma *corresTA-yield*:
 $\text{abstract-val True } v' \ (\text{map-xval } ex \ rx) \ v \Longrightarrow \text{corresTA } P \text{ rx } ex \ (\text{yield } v') \ (\text{yield } v)$
apply (*auto simp add: corresTA-refines-conv rel-word-abs-def abstract-val-def*
rel-xval.simps map-xval-def
intro!: refines-yield split: xval-splits)
done

lemma *map-sum-apply*: $\text{map-sum } ex \ rx = (\lambda v. \text{case } v \text{ of } \text{Inl } l \Rightarrow \text{Inl } (ex \ l) \mid \text{Inr } r \Rightarrow \text{Inr } (rx \ r))$
by (*simp add: fun-eq-iff split-sum-all*)

```

lemma refines-try-rel-prod:
  assumes refines f g s t (rel-prod (rel-xval (rel-sum L R) R) S)
  shows refines (try f) (try g) s t (rel-prod (rel-xval L R) S)
  using assms
  apply (clarsimp simp add: refines-def-old reaches-try unnest-exn-def rel-xval.simps
split: xval-splits sum.splits)
  subgoal for r s' r'
    apply (erule-tac x=r' in allE)
    apply (erule-tac x=s' in allE)

  apply (cases r)
  subgoal for e
    apply (clarsimp simp add: default-option-def Exn-def, safe)
    apply (metis Exception-eq-Exception Exn-def Exn-neq-Result exception-or-result-cases
not-None-eq sum-all-ex(2))
    by (smt (verit, ccfv-threshold) Exn-eq-Exception(2) Exn-neq-Result
rel-sum.cases rel-sum-simps(2) theLeft.simps the-Exn-Exn(2))

  subgoal for v
    apply (clarsimp simp add: default-option-def Exn-def, safe)
    apply (metis Exception-eq-Exception Exn-def Exn-neq-Result exception-or-result-cases
not-None-eq sum-all-ex(2))
    apply (smt (verit, ccfv-threshold) Exn-def Result-neq-Exn rel-sum.cases
rel-sum-simps(3) unnest-exn-eq-simps(3))
    by (metis Exn-def Result-eq-Result Result-neq-Exn)
  done
done

lemma rel-map-xval-sum-rel-sum-conv:
  rel-xval (λc a. a = map-sum ex rx c) (λc a. a = rx c) =
  rel-xval (rel-sum (λc a. a = ex c) (λc a. a = rx c)) (λc a. a = rx c)
  apply (rule ext)+
  apply (auto simp add: rel-xval.simps rel-sum.simps)
  done

lemma corresTA-L2-try':
  assumes corres-L-L': corresTA P rx (map-sum ex rx) L L'
  shows corresTA P rx ex (L2-try L) (L2-try L')
  unfolding L2-defs
  apply (clarsimp simp add: corresTA-refines-conv rel-word-abs-def)
  apply (rule refines-try-rel-prod)
  using corres-L-L'
  apply (auto simp add: corresTA-refines-conv rel-word-abs-def rel-map-xval-sum-rel-sum-conv)
  done

lemma corresTA-L2-while:
  assumes init-corres: abstract-val Q i rx i'
  and cond-corres: PROP THIN (∧r r' s. abs-var r rx r'
   $\implies$  abstract-val (G r s) (C r s) id (C' r' s))

```


and *body-corres*: *PROP THIN* ($\bigwedge r r'. \text{abs-var } r \text{ rx } r'$
 $\implies \text{corresTA } (P \ r) \text{ rx } \text{ex } (B \ r) \ (B' \ r')$)

shows *corresTA* ($\lambda-. Q$) *rx ex*
 $(L2\text{-guarded-while } (\lambda r \ s. G \ r \ s) \ (\lambda r \ s. C \ r \ s) \ (\lambda r. L2\text{-seq } (L2\text{-guard } (\lambda s. P \ r \ s)) \ (\lambda-. B \ r))) \ i \ x)$
 $(L2\text{-while } (\lambda r \ s. C' \ r \ s) \ B' \ i' \ x)$

proof –

note *cond-corres* = *cond-corres* [*unfolded THIN-def*, *rule-format*]
note *body-corres* = *body-corres* [*unfolded THIN-def*, *rule-format*]
note *body-corres'* =
 $\text{corresXF-guarded-while-body } [OF \ \text{body-corres } [unfolded \ \text{corresTA-def}]]$

have *init-corres'*:
 $Q \implies i = \text{rx } i'$
using *init-corres*
by (*simp add: basic-abstract-defs*)

note *basic-abstract-defs* [*simp*]
show *?thesis*
thm *corresXF-assume-pre*
apply (*clarsimp simp: L2-defs corresTA-def gets-return*)
apply (*rule corresXF-assume-pre*)
thm *corresXF-guarded-while*
apply (*rule corresXF-guarded-while [where P= $\lambda r \ s. G \ (rx \ r) \ s]$*)
subgoal for $s \ s' \ x \ y$
apply (*cut-tac r'=x in body-corres, simp*)
apply (*fastforce simp add: corresTA-refines-conv corresXF-refines-conv re-fines-def-old reaches-bind succeeds-bind*
 $\text{rel-word-abs-def rel-xval.simps split: xval-splits}$)
done
subgoal
apply (*insert cond-corres*)[1]
apply (*clarsimp*)
done
subgoal
by (*auto simp add: runs-to-partial-def-old split: xval-splits*)
subgoal using *init-corres*
by (*clarsimp*)
subgoal using *init-corres'*
by *clarsimp*
done

qed

lemma *corresTA-L2-guard*:
 $\llbracket \bigwedge s. \text{abstract-val } (Q \ s) \ (G \ s) \ \text{id } (G' \ s) \rrbracket$
 $\implies \text{corresTA } (\lambda-. \text{True}) \ \text{rx } \text{ex } (L2\text{-guard } (\lambda s. G \ s \wedge Q \ s)) \ (L2\text{-guard } (\lambda s. G' \ s))$
unfolding *L2-defs*

apply (*auto simp add: corresTA-refines-conv abstract-val-def intro!: refines-guard*)
done

lemma *corresTA-L2-guard'*:

$\llbracket \bigwedge s. \text{abstract-val } (Q\ s) (G\ s) \text{ id } (G'\ s);$
 $\bigwedge s. R\ s \implies G\ s \wedge Q\ s \rrbracket$
 $\implies \text{corresTA } (\lambda-. \text{True}) \text{ rx ex } (\text{L2-guard } (\lambda s. R\ s)) (\text{L2-guard } (\lambda s. G'\ s))$

unfolding *L2-defs*

apply (*auto simp add: corresTA-refines-conv abstract-val-def intro!: refines-guard*)
done

lemma *corresTA-L2-guarded-simple*:

assumes $G\text{-}G'$: $\bigwedge s. \text{abstract-val } (Q\ s) (G\ s) \text{ id } (G'\ s)$
assumes $f\text{-}f'$: $\bigwedge s. G'\ s \implies Q\ s \implies G\ s \implies \text{corresTA } P \text{ rx ex } f\ f'$
shows $\text{corresTA } (\lambda-. \text{True}) \text{ rx ex } (\text{L2-guarded } (\lambda s. G\ s \wedge Q\ s \wedge P\ s) f) (\text{L2-guarded } G'\ f')$

unfolding *L2-defs L2-guarded-def*

apply (*clarsimp simp add: corresTA-refines-conv*)

apply (*rule refines-bind-guard-right*)

using $G\text{-}G'$ $f\text{-}f'$

by (*auto simp add: corresTA-refines-conv refines-def-old succeeds-bind reaches-bind abstract-val-def*)

lemma *corresTA-L2-spec*:

$(\bigwedge s\ t. \text{abstract-val } (Q\ s) (P\ s\ t) \text{ id } (P'\ s\ t)) \implies$
 $\text{corresTA } Q \text{ rx ex } (\text{L2-spec } \{(s, t). P\ s\ t\}) (\text{L2-spec } \{(s, t). P'\ s\ t\})$

unfolding *L2-defs*

by (*auto simp add: corresTA-refines-conv abstract-val-def refines-def-old reaches-bind succeeds-bind*)

lemma *corresTA-L2-assume*:

$(\bigwedge s\ r\ t. \text{abstract-val } (Q\ s) (P\ s) (\lambda X. (\lambda(x, y). (rx\ x, y)) ' X) (P'\ s)) \implies$
 $\text{corresTA } Q \text{ rx ex } (\text{L2-assume } P) (\text{L2-assume } P')$

unfolding *L2-defs*

apply (*auto simp add: corresTA-refines-conv abstract-val-def refines-def-old reaches-bind succeeds-bind*)

rel-word-abs-def rel-xval.simps)

done

lemma *corresTA-L2-condition*:

$\llbracket \text{PROP THIN } (\text{Trueprop } (\text{corresTA } P \text{ rx ex } L\ L'));$
 $\text{PROP THIN } (\text{Trueprop } (\text{corresTA } Q \text{ rx ex } R\ R'));$
 $\bigwedge s. \text{abstract-val } (T\ s) (C\ s) \text{ id } (C'\ s) \rrbracket$

$\implies \text{corresTA } T \text{ rx ex}$

$(\text{L2-condition } (\lambda s. C\ s))$

$(\text{L2-seq } (\text{L2-guard } P) (\lambda-. L))$

$(\text{L2-seq } (\text{L2-guard } Q) (\lambda-. R))$

$) (\text{L2-condition } (\lambda s. C'\ s) L'\ R')$

unfolding *THIN-def L2-defs*
by (*auto simp add: corresTA-refines-conv abstract-val-def intro!: refines-condition refines-bind-guard-right*)

lemma *L2-call-L2-defs: L2-call x emb ns = L2-catch x (λe. L2-throw (emb e) ns)*
unfolding *L2-defs L2-call-def*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add: runs-to-iff*)
apply (*auto simp add: runs-to-def-old map-exn-def split: xval-splits*)
done

lemma *corresTA-L2-call:*
 $\llbracket \text{corresTA } P \text{ } rx \text{ } ex' \text{ } A \text{ } B; \bigwedge r \text{ } r'. \text{ abs-var } r \text{ } ex' \text{ } r' \implies \text{ abstract-val } Q \text{ } (emb \text{ } r) \text{ } ex \text{ } (emb' \text{ } r') \rrbracket \implies$
 $\text{corresTA } (\lambda s. P \text{ } s \wedge Q) \text{ } rx \text{ } ex \text{ } (L2\text{-call } A \text{ } emb \text{ } ns) \text{ } (L2\text{-call } B \text{ } emb' \text{ } ns)$
unfolding *L2-call-def*
by (*force simp add: corresTA-refines-conv abstract-val-def abs-var-def refines-def-old reaches-map-value map-exn-def rel-word-abs-def rel-xval.simps*)

lemma *corresTA-L2-call':*
 $\llbracket \text{corresTA } P \text{ } f1 \text{ } ex' \text{ } A \text{ } B; \text{ valid-typ-abs-fn } Q1 \text{ } Q1' \text{ } f1 \text{ } f1'; \text{ valid-typ-abs-fn } Q2 \text{ } Q2' \text{ } f2 \text{ } f2'; \bigwedge r \text{ } r'. \text{ abs-var } r \text{ } ex' \text{ } r' \implies \text{ abstract-val } Q \text{ } (emb \text{ } r) \text{ } ex \text{ } (emb' \text{ } r') \rrbracket \implies$
 $\text{corresTA } (\lambda s. P \text{ } s \wedge Q) \text{ } f2 \text{ } ex$
 $(L2\text{-seq } (L2\text{-call } A \text{ } emb \text{ } ns) \text{ } (ETA\text{-TUPLED } (\lambda ret. (L2\text{-seq } (L2\text{-guard } (\lambda-. Q1' \text{ } ret)) (\lambda-. L2\text{-gets } (\lambda-. f2 \text{ } (f1' \text{ } ret)) \text{ } ns))))))$
 $(L2\text{-call } B \text{ } emb' \text{ } ns)$
unfolding *L2-call-def L2-defs*
apply (*clarsimp simp add: ETA-TUPLED-def corresTA-refines-conv abstract-val-def abs-var-def refines-def-old reaches-map-value map-exn-def reaches-bind succeeds-bind reaches-succeeds rel-word-abs-def rel-xval.simps*)
subgoal for *s s' r'*
apply (*erule-tac x=s in allE*)
apply *clarsimp*
apply (*cases r'*)
subgoal
apply (*simp add: default-option-def Exn-def [symmetric]*)
by (*smt (z3) Exn-def Exn-neq-Result case-exception-or-result-Exn case-xval-simps(1) the-Exn-Exn(1)*)
subgoal
apply *simp*
by (*metis (mono-tags, lifting) Exn-neq-Result case-exception-or-result-Result*)

case-xval-simps(2) the-Result-simp)

done
done

lemma *corresTA-L2-unknown:*

corresTA ($\lambda\cdot$. *True*) *rx ex (L2-unknown x) (L2-unknown x)*

unfolding *L2-defs*

by (*auto simp add: corresTA-refines-conv intro!: refines-select*)

lemma *corresTA-L2-call-exec-concrete:*

\llbracket *corresTA P rx ex' A B* ;

$\bigwedge r r'. \text{abs-var } r \text{ ex' } r' \implies \text{abstract-val } Q \text{ (emb } r) \text{ ex (emb' } r') \rrbracket \implies$

corresTA ($\lambda s. \forall s'. s = st \ s' \longrightarrow P \ s' \wedge Q$) *rx ex*

(*exec-concrete st (L2-call A emb ns)*)

(*exec-concrete st (L2-call B emb' ns)*)

unfolding *L2-defs L2-call-def*

apply (*clarsimp simp add: corresTA-refines-conv abstract-val-def abs-var-def*)

apply (*clarsimp simp add: refines-def-old succeeds-exec-concrete-iff reaches-exec-concrete*

reaches-map-value rel-word-abs-def rel-xval.simps map-exn-def split: xval-splits

)

subgoal for *r t t' r'*

apply (*erule-tac x=t in allE*)

apply *clarsimp*

apply (*cases r'*)

subgoal

apply (*clarsimp simp add: default-option-def Exn-def [symmetric]*)

by (*metis Exn-eq-Exn Exn-neq-Result*)

subgoal

apply *simp*

by (*metis Result-eq-Result Result-neq-Exn*)

done

done

lemma *corresTA-L2-call-exec-abstract:*

\llbracket *corresTA P rx ex' A B* ;

$\bigwedge r r'. \text{abs-var } r \text{ ex' } r' \implies \text{abstract-val } Q \text{ (emb } r) \text{ ex (emb' } r') \rrbracket \implies$

corresTA ($\lambda s. P \text{ (st } s) \wedge Q$) *rx ex*

(*exec-abstract st (L2-call A emb ns)*)

(*exec-abstract st (L2-call B emb' ns)*)

unfolding *L2-defs L2-call-def*

apply (*clarsimp simp add: corresTA-refines-conv abstract-val-def abs-var-def*)

apply (*clarsimp simp add: refines-def-old succeeds-exec-abstract-iff reaches-exec-abstract*

reaches-map-value rel-word-abs-def rel-xval.simps map-exn-def split: xval-splits

)

subgoal for *s r s' r'*

```

apply (erule-tac  $x=st$   $s$  in allE)
apply clarsimp
apply (cases  $r'$ )
subgoal
  apply (clarsimp simp add: default-option-def Exn-def [symmetric])
  by (metis Exn-eq-Exn Exn-neq-Result)
subgoal
  apply simp
  by (metis Result-eq-Result Result-neq-Exn)
done
done

```

```

lemma corresTA-L2-call-exec-concrete':
  [| corresTA  $P$   $f1$   $ex'$   $A$   $B$ ;
    valid-typ-abs-fn  $Q1$   $Q1'$   $f1$   $f1'$ ;
    valid-typ-abs-fn  $Q2$   $Q2'$   $f2$   $f2'$ ;
     $\bigwedge r r'. \text{abs-var } r \text{ } ex' \text{ } r' \implies \text{abstract-val } Q \text{ } (emb \text{ } r) \text{ } ex \text{ } (emb' \text{ } r')$ 
  |]  $\implies$ 
  corresTA ( $\lambda s. \forall s'. s = st \text{ } s' \longrightarrow P \text{ } s' \wedge Q$ )  $f2$   $ex$ 
    (L2-seq (exec-concrete st (L2-call A emb ns)) ( $\lambda ret. (L2-seq (L2-guard (\lambda-.$ 
Q1' ret)) ( $\lambda-. L2\text{-gets } (\lambda-. f2 \text{ } (f1' \text{ } ret)) \text{ } []))$ ))
    (exec-concrete st (L2-call B emb' ns))
  unfolding L2-defs L2-call-def
  apply (clarsimp simp add: corresTA-refines-conv abstract-val-def abs-var-def)
  apply (clarsimp simp add: refines-def-old succeeds-exec-concrete-iff reaches-exec-concrete
    reaches-bind succeeds-bind
    reaches-map-value rel-word-abs-def rel-xval.simps map-exn-def split: xval-splits
  )
  subgoal for  $r \text{ } t \text{ } t' \text{ } r'$ 
  apply (erule-tac  $x=t$  in allE)
  apply clarsimp
  apply (cases  $r'$ )
  subgoal
    apply (clarsimp simp add: default-option-def Exn-def [symmetric])
    by (smt (z3) Exn-def Exn-eq-Exn Exn-neq-Result case-exception-or-result-Exn)

  subgoal for  $v$ 
    apply clarsimp
    apply (erule-tac  $x=Result \text{ } v$  in allE)
    apply (erule-tac  $x=t'$  in allE)
    by (smt (z3) Exn-neq-Result Result-eq-Result case-exception-or-result-Result)
  done
done

```

```

lemma corresTA-L2-call-exec-abstract':

```

```

[[ corresTA P f1 ex' A B;
   valid-typ-abs-fn Q1 Q1' f1 f1';
   valid-typ-abs-fn Q2 Q2' f2 f2';
    $\bigwedge r r'. \text{abs-var } r \text{ ex' } r' \implies \text{abstract-val } Q \text{ (emb } r) \text{ ex (emb' } r')$ 
]]  $\implies$ 
corresTA ( $\lambda s. P \text{ (st } s) \wedge Q$ ) f2 ex
  (L2-seq (exec-abstract st (L2-call A emb ns)) ( $\lambda \text{ret. (L2-seq (L2-guard (\lambda -. Q1' \text{ret})) (\lambda -. L2-gets (\lambda -. f2 (f1' \text{ret})) []))$ 
    (exec-abstract st (L2-call B emb' ns))
unfolding L2-defs L2-call-def
apply (clarsimp simp add: corresTA-refines-conv abstract-val-def abs-var-def)
apply (clarsimp simp add: refines-def-old succeeds-exec-abstract-iff reaches-exec-abstract
  reaches-bind succeeds-bind
  reaches-map-value rel-word-abs-def rel-xval.simps map-exn-def split: xval-splits
)
subgoal for s r s' r'
apply (erule-tac x=st s in alle)
apply clarsimp
apply (cases r')
subgoal
apply (clarsimp simp add: default-option-def Exn-def [symmetric])
by (smt (z3) Exn-def Exn-eq-Exn Exn-neq-Result case-exception-or-result-Exn)
subgoal
apply clarsimp
by (smt (z3) Result-eq-Result Result-neq-Exn case-exception-or-result-Result)
done
done

```

Avoid higher-order unification issues by explicit application with (\$):

- in concrete program position enforces 'obvious' instantiation
- in abstract program position enforces introduction of two separate variables for a and b instead of a higher-order flex-flex pair.

lemma *abstract-val-fun-app:*

```

[[ abstract-val Q b id b'; abstract-val P a id a' ]]  $\implies$ 
  abstract-val (P  $\wedge$  Q) (f $ (a $ b)) f (a' $ b')
by (simp add: basic-abstract-defs)

```

lemma *corresTA-precond-to-guard:*

```

corresTA ( $\lambda s. P \text{ } s$ ) rx ex A A'  $\implies$  corresTA ( $\lambda -. \text{True}$ ) rx ex (L2-seq (L2-guard
( $\lambda s. P \text{ } s$ )) ( $\lambda -. A$ )) A'
unfolding L2-defs
by (auto simp add: corresTA-refines-conv intro: refines-bind-guard-right)

```

lemma *corresTA-precond-to-asm:*

$\llbracket \bigwedge s. P s \implies \text{corresTA } (\lambda\cdot. \text{True}) \text{ rx ex } A A' \rrbracket \implies \text{corresTA } P \text{ rx ex } A A'$
by (*clarsimp simp: corresXF-def corresTA-def*)

lemma *L2-guard-true: L2-seq (L2-guard ($\lambda\cdot. \text{True}$)) $A = A$ ()*
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*auto simp add: run-bind run-guard*)
done

lemma *corresTA-simp-trivial-guard:*
 $\text{corresTA } P \text{ rx ex } (L2\text{-seq } (L2\text{-guard } (\lambda\cdot. \text{True})) A) C \equiv \text{corresTA } P \text{ rx ex } (A ())$
 C
by (*simp add: L2-guard-true*)

lemma *corresTA-extract-preconds-of-call-init:*
 $\llbracket \text{corresTA } (\lambda s. P) \text{ rx ex } A A' \rrbracket \implies \text{corresTA } (\lambda s. P \wedge \text{True}) \text{ rx ex } A A'$
by *simp*

lemma *corresTA-extract-preconds-of-call-step:*
 $\llbracket \text{corresTA } (\lambda s. (\text{abs-var } a f a' \wedge R) \wedge C) \text{ rx ex } A A'; \text{abstract-val } Y a f a' \rrbracket$
 $\implies \text{corresTA } (\lambda s. R \wedge (Y \wedge C)) \text{ rx ex } A A'$
by (*clarsimp simp: corresXF-def corresTA-def basic-abstract-defs*)

lemma *corresTA-extract-preconds-of-call-final:*
 $\llbracket \text{corresTA } (\lambda s. (\text{abs-var } a f a') \wedge C) \text{ rx ex } A A'; \text{abstract-val } Y a f a' \rrbracket$
 $\implies \text{corresTA } (\lambda s. (Y \wedge C)) \text{ rx ex } A A'$
by (*clarsimp simp: corresXF-def corresTA-def basic-abstract-defs*)

lemma *corresTA-extract-preconds-of-call-final':*
 $\llbracket \text{corresTA } (\lambda s. \text{True} \wedge C) \text{ rx ex } A A' \rrbracket$
 $\implies \text{corresTA } (\lambda s. C) \text{ rx ex } A A'$
by (*clarsimp simp: corresXF-def corresTA-def*)

lemma *corresTA-extract-preconds-of-call-init-prems:*
 $\llbracket \text{corresTA } (\lambda s. P \wedge \text{True}) \text{ rx ex } A A' \rrbracket \implies \text{corresTA } (\lambda s. P) \text{ rx ex } A A'$
by *simp*

lemma *corresTA-extract-preconds-of-call-step-prems:*
 $\llbracket \bigwedge Y. \text{abstract-val } Y a f a' \implies \text{corresTA } (\lambda s. R \wedge (Y \wedge C)) \text{ rx ex } A A' \rrbracket$
 $\implies \text{corresTA } (\lambda s. (\text{abs-var } a f a' \wedge R) \wedge C) \text{ rx ex } A A'$
by (*clarsimp simp: corresXF-def corresTA-def basic-abstract-defs*)

lemma *corresTA-extract-preconds-of-call-final-prems:*
 $\llbracket \bigwedge Y. \text{abstract-val } Y a f a' \implies \text{corresTA } (\lambda s. (Y \wedge C)) \text{ rx ex } A A' \rrbracket$
 $\implies \text{corresTA } (\lambda s. (\text{abs-var } a f a') \wedge C) \text{ rx ex } A A'$
by (*auto simp: corresXF-def corresTA-def basic-abstract-defs split: prod.splits sum.splits*)

lemma *corresTA-extract-preconds-of-call-final'-prems:*

$\llbracket \text{corresTA } (\lambda s. C) \text{ rx ex } A \ A' \rrbracket$
 $\implies \text{corresTA } (\lambda s. \text{True} \wedge C) \text{ rx ex } A \ A'$
by (*clarsimp simp: corresXF-def corresTA-def*)

lemma *corresTA-case-prod:*

$\llbracket \text{introduce-typ-abs-fn rx1};$
 $\text{introduce-typ-abs-fn rx2};$
 $\text{abstract-val } (Q \ x) \ x \ (\text{map-prod rx1 rx2}) \ x';$
 $\bigwedge a \ b \ a' \ b'. \llbracket \text{abs-var } a \text{ rx1 } a'; \text{abs-var } b \text{ rx2 } b' \rrbracket$
 $\implies \text{corresTA } (P \ a \ b) \text{ rx ex } (M \ a \ b) \ (M' \ a' \ b') \rrbracket \implies$
 $\text{corresTA } (\lambda s. \text{case } x \text{ of } (a, b) \Rightarrow P \ a \ b \ s \wedge Q \ (a, b)) \text{ rx ex } (\text{case } x \text{ of } (a, b) \Rightarrow$
 $M \ a \ b) \ (\text{case } x' \text{ of } (a, b) \Rightarrow M' \ a \ b)$
apply (*clarsimp simp add: corresTA-def*)
apply (*rule corresXF-assume-pre*)
apply (*clarsimp simp: split-def map-prod-def basic-abstract-defs*)
done

lemma *abstract-val-case-prod:*

$\llbracket \text{abstract-val True } r \ (\text{map-prod } f \ g) \ r';$
 $\bigwedge a \ b \ a' \ b'. \llbracket \text{abs-var } a \ f \ a'; \text{abs-var } b \ g \ b' \rrbracket$
 $\implies \text{abstract-val } (P \ a \ b) \ (M \ a \ b) \ h \ (M' \ a' \ b') \rrbracket$
 $\implies \text{abstract-val } (P \ (\text{fst } r) \ (\text{snd } r))$
 $(\text{case } r \text{ of } (a, b) \Rightarrow M \ a \ b) \ h$
 $(\text{case } r' \text{ of } (a, b) \Rightarrow M' \ a \ b)$
apply (*cases r, cases r'*)
apply (*clarsimp simp: map-prod-def basic-abstract-defs*)
done

lemma *abstract-val-case-prod-fun-app:*

$\llbracket \text{abstract-val True } r \ (\text{map-prod } f \ g) \ r';$
 $\bigwedge a \ b \ a' \ b'. \llbracket \text{abs-var } a \ f \ a'; \text{abs-var } b \ g \ b' \rrbracket$
 $\implies \text{abstract-val } (P \ a \ b) \ (M \ a \ b \ s) \ h \ (M' \ a' \ b' \ s) \rrbracket$
 $\implies \text{abstract-val } (P \ (\text{fst } r) \ (\text{snd } r))$
 $((\text{case } r \text{ of } (a, b) \Rightarrow M \ a \ b) \ s) \ h$
 $((\text{case } r' \text{ of } (a, b) \Rightarrow M' \ a \ b) \ s)$
apply (*cases r, cases r'*)
apply (*clarsimp simp: map-prod-def basic-abstract-defs*)
done

lemma *abstract-val-of-nat:*

$\text{abstract-val } (r \leq \text{UWORD-MAX TYPE}('a::\text{len})) \ r \ \text{unat} \ (\text{of-nat } r \ :: \ 'a \ \text{word})$
by (*clarsimp simp: unat-of-nat-eq UWORD-MAX-def le-to-less-plus-one basic-abstract-defs*)

lemma *abstract-val-of-int:*

$\text{abstract-val } (\text{WORD-MIN TYPE}('a::\text{len}) \leq r \wedge r \leq \text{WORD-MAX TYPE}('a)) \ r$
 $\text{sint} \ (\text{of-int } r \ :: \ 'a \ \text{signed word})$
by (*clarsimp simp: sint-of-int-eq WORD-MIN-def WORD-MAX-def basic-abstract-defs*)

lemma *abstract-val-tuple*:
 $\llbracket \text{abstract-val } P \ a \ \text{absL } a'; \text{abstract-val } Q \ b \ \text{absR } b' \rrbracket \implies$
 $\text{abstract-val } (P \wedge Q) \ (a, b) \ (\text{map-prod } \text{absL } \text{absR}) \ (a', b')$
by (*clarsimp simp add: basic-abstract-defs*)

lemma *abstract-val-Inl*:
 $\llbracket \text{abstract-val } P \ a \ \text{absL } a' \rrbracket \implies$
 $\text{abstract-val } P \ (\text{Inl } a) \ (\text{map-sum } \text{absL } \text{absR}) \ (\text{Inl } a')$
by (*clarsimp simp add: basic-abstract-defs*)

lemma *abstract-val-Inr*:
 $\llbracket \text{abstract-val } P \ b \ \text{absR } b' \rrbracket \implies$
 $\text{abstract-val } P \ (\text{Inr } b) \ (\text{map-sum } \text{absL } \text{absR}) \ (\text{Inr } b')$
by (*clarsimp simp add: basic-abstract-defs*)

lemma *abstract-val-func*:
 $\llbracket \text{abstract-val } P \ a \ \text{id } a'; \text{abstract-val } Q \ b \ \text{id } b' \rrbracket$
 $\implies \text{abstract-val } (P \wedge Q) \ (f \ a \ b) \ \text{id } (f \ a' \ b')$
by (*clarsimp simp add: basic-abstract-defs*)

lemma *abstract-val-conj*:
 $\llbracket \text{abstract-val } P \ a \ \text{id } a'; \text{abstract-val } Q \ b \ \text{id } b' \rrbracket \implies$
 $\text{abstract-val } (P \wedge (a \longrightarrow Q)) \ (a \wedge b) \ \text{id } (a' \wedge b')$
apply (*clarsimp simp add: basic-abstract-defs*)
apply *blast*
done

lemma *abstract-val-disj*:
 $\llbracket \text{abstract-val } P \ a \ \text{id } a'; \text{abstract-val } Q \ b \ \text{id } b' \rrbracket \implies$
 $\text{abstract-val } (P \wedge (\neg a \longrightarrow Q)) \ (a \vee b) \ \text{id } (a' \vee b')$
apply (*clarsimp simp add: basic-abstract-defs*)
apply *blast*
done

lemma *abstract-val-unwrap*:
 $\llbracket \text{introduce-typ-abs-fn } f; \text{abstract-val } P \ a \ f \ b \rrbracket$
 $\implies \text{abstract-val } P \ a \ \text{id } (f \ b)$
by (*simp add: basic-abstract-defs*)

lemma *abstract-val-uint*:
 $\llbracket \text{introduce-typ-abs-fn } \text{unat}; \text{abstract-val } P \ x \ \text{unat } x' \rrbracket$
 $\implies \text{abstract-val } P \ (\text{int } x) \ \text{id } (\text{uint } x')$
by (*clarsimp simp add: basic-abstract-defs*)

lemma *abstract-val-lambda*:

$\llbracket \bigwedge v. \text{abstract-val } (P v) (a v) \text{ id } (a' v) \rrbracket \implies$
 $\text{abstract-val } (\forall v. P v) (\lambda v. a v) \text{ id } (\lambda v. a' v)$
by (*auto simp add: basic-abstract-defs*)

lemma *corresTA-call-L1*:
 $\text{abstract-val True arg-xf id arg-xf}' \implies$
 $\text{corresTA } (\lambda-. \text{True}) \text{ id id}$
 (*L2-call-L1 arg-xf gs ret-xf l1body*)
 (*L2-call-L1 arg-xf' gs ret-xf l1body*)
apply (*unfold corresTA-def abstract-val-def id-def*)
apply (*subst (asm) simp-thms*)
apply (*erule subst*)
apply (*rule corresXF-id[simplified id-def]*)
done

context *stack-heap-state*

begin

lemma *corresTA-with-fresh-stack-ptr[word-abs]*:
assumes $f[\text{simplified THIN-def, rule-format}]$: *PROP THIN* ($\bigwedge p. \text{corresTA } Q \text{ rx}$
 $ex (f_a p) (f_c p)$)
assumes *init*: $\bigwedge s. \text{abstract-val } (P s) (init_a s) \text{ id } (init_c s)$
shows $\text{corresTA } P \text{ rx } ex$
 (*with-fresh-stack-ptr n init_a (L2-VARS (λp. (L2-seq (L2-guard Q) (λ-. f_a*
 $p))) nm$)
 (*with-fresh-stack-ptr n init_c (L2-VARS f_c nm)*)
apply (*rule refines-corresTA*)
apply (*simp add: L2-seq-def L2-guard-def rel-word-abs-def*)
apply (*rule refines-rel-prod-with-fresh-stack-ptr*)
subgoal for s
using *init* [*of s*]
by (*simp add: abstract-val-def*)
subgoal for $s s' p$
apply (*rule refines-bind-guard-right*)
apply (*simp only: rel-word-abs-def [symmetric]*)
apply (*rule corresTA-refines*)
apply (*rule f*)
apply *simp*
done
done
end

context *typ-heap-typing*

begin

lemma *corresTA-guard-with-fresh-stack-ptr[word-abs]*:
assumes $f[\text{simplified THIN-def, rule-format}]$: *PROP THIN* ($\bigwedge p. \text{corresTA } Q \text{ rx}$
 $ex (f_a p) (f_c p)$)

assumes *init*: $\bigwedge s. \text{abstract-val } (P \ s) \ (init_a \ s) \ id \ (init_c \ s)$
shows *corresTA* $P \ rx \ ex$
 $(\lambda-. \ f_a \ p)) \ nm))$
 $(guard-with-fresh-stack-ptr \ n \ init_a \ (L2-VARS \ (\lambda p. \ (L2-seq \ (L2-guard \ Q)$
 $(\lambda-. \ f_a \ p))) \ nm))$
 $(guard-with-fresh-stack-ptr \ n \ init_c \ (L2-VARS \ f_c \ nm))$
apply (*rule refines-corresTA*)
apply (*simp add: L2-seq-def L2-guard-def rel-word-abs-def*)
apply (*rule refines-rel-xval-guard-with-fresh-stack-ptr*)
subgoal for s
using *init* [*of s*]
by (*simp add: abstract-val-def*)
subgoal for $s \ s' \ p$
apply (*rule refines-bind-guard-right*)
apply (*simp only: rel-word-abs-def [symmetric]*)
apply (*rule corresTA-refines*)
apply (*rule f*)
apply *simp*
done
done

lemma *corresTA-assume-with-fresh-stack-ptr[word-abs]*:
assumes $f[\text{simplified THIN-def, rule-format}]$: *PROP THIN* $(\bigwedge p. \text{corresTA } Q \ rx$
 $ex \ (f_a \ p) \ (f_c \ p))$
assumes *init*: $\bigwedge s. \text{abstract-val } (P \ s) \ (init_a \ s) \ id \ (init_c \ s)$
shows *corresTA* $P \ rx \ ex$
 $(assume-with-fresh-stack-ptr \ n \ init_a \ (L2-VARS \ (\lambda p. \ (L2-seq \ (L2-guard \ Q)$
 $(\lambda-. \ f_a \ p))) \ nm))$
 $(assume-with-fresh-stack-ptr \ n \ init_c \ (L2-VARS \ f_c \ nm))$
apply (*rule refines-corresTA*)
apply (*simp add: L2-seq-def L2-guard-def rel-word-abs-def*)
apply (*rule refines-rel-xval-assume-with-fresh-stack-ptr*)
subgoal for s
using *init* [*of s*]
by (*simp add: abstract-val-def*)
subgoal for $s \ s' \ p$
apply (*rule refines-bind-guard-right*)
apply (*simp only: rel-word-abs-def [symmetric]*)
apply (*rule corresTA-refines*)
apply (*rule f*)
apply *simp*
done
done

lemma *corresTA-with-fresh-stack-ptr[word-abs]*:
assumes $f[\text{simplified THIN-def, rule-format}]$: *PROP THIN* $(\bigwedge p. \text{corresTA } Q \ rx$
 $ex \ (f_a \ p) \ (f_c \ p))$
assumes *init*: $\bigwedge s. \text{abstract-val } (P \ s) \ (init_a \ s) \ id \ (init_c \ s)$
shows *corresTA* $P \ rx \ ex$
 $(with-fresh-stack-ptr \ n \ init_a \ (L2-VARS \ (\lambda p. \ (L2-seq \ (L2-guard \ Q) \ (\lambda-. \ f_a$

```

p))) nm))
  (with-fresh-stack-ptr n initc (L2-VARS fc nm))
apply (rule refines-corresTA)
apply (simp add: L2-seq-def L2-guard-def rel-word-abs-def)
apply (rule refines-rel-xval-with-fresh-stack-ptr)
subgoal for s
  using init [of s]
  by (simp add: abstract-val-def)
subgoal for s s' p
  apply (rule refines-bind-guard-right)
  apply (simp only: rel-word-abs-def [symmetric])
  apply (rule corresTA-refines)
  apply (rule f)
  apply simp
done
done
end

```

lemma *abstract-val-call-L1-args*:
 $abstract-val\ P\ x\ id\ x' \implies abstract-val\ P\ y\ id\ y' \implies$
 $abstract-val\ P\ (x\ and\ y)\ id\ (x'\ and\ y')$
by (simp add: basic-abstract-defs)

lemma *abstract-val-call-L1-arg*:
 $abs-var\ x\ id\ x' \implies abstract-val\ P\ (\lambda s. f\ s = x)\ id\ (\lambda s. f\ s = x')$
by (simp add: basic-abstract-defs)

lemma *abstract-val-abs-var [consumes 1]*:
 $\llbracket abs-var\ a\ f\ a' \rrbracket \implies abstract-val\ True\ a\ f\ a'$
by (clarsimp simp: fun-upd-def basic-abstract-defs split: if-splits)

lemma *abstract-val-abs-var-concretise [consumes 1]*:
 $\llbracket abs-var\ a\ A\ a';\ introduce-typ-abs-fn\ A;\ valid-typ-abs-fn\ PA\ PC\ A\ (C :: 'a \Rightarrow 'c) \rrbracket$
 $\implies abstract-val\ (PC\ a)\ (C\ a)\ id\ a'$
by (clarsimp simp: fun-upd-def basic-abstract-defs split: if-splits)

lemma *abstract-val-abs-var-give-up [consumes 1]*:
 $\llbracket abs-var\ a\ id\ a' \rrbracket \implies abstract-val\ True\ (A\ a)\ A\ a'$
by (clarsimp simp: fun-upd-def basic-abstract-defs split: if-splits)

lemma *abstract-val-abs-var-sint-unat [consumes 1]*:
 $\llbracket abs-var\ a\ sint\ a' \rrbracket \implies abstract-val\ (0 \leq a)\ (nat\ a)\ id\ (unat\ a')$
apply (simp add: basic-abstract-defs)
by (metis nat-uint-eq signed-0 sint-eq-uint word-msb-0 word-sle-eq word-sle-msb-le)

lemma *abstract-val-abs-var-uint-unat* [*consumes 1*]:
 $\llbracket \text{abs-var } a \text{ uint } a' \rrbracket \implies \text{abstract-val True (nat } a) \text{ id (unat } a')$
by (*simp add: basic-abstract-defs*)

lemma *abs-var-id*: $(\text{abs-var } a \text{ id } a') = (a' = a)$
by (*auto simp add: abs-var-def abstract-val-def*)

lemma *abstract-val-id*: *abstract-val P a id a*
by (*simp add: abstract-val-def*)

lemmas *abstract-val-id-unit-ptr = abstract-val-id* [**where** $a = a :: \text{unit ptr}$ **and** $P = \text{True}$] **for** a

lemma *abstract-val-id-True*: *abstract-val True a id a*
by (*rule abstract-val-id*)

lemmas *abs-var-id-unit-ptr = abs-var-id* [**where** $a = a :: \text{unit ptr}$] **for** a

lemma *len-of-word-comparisons* [*L2opt*]:

$\text{len-of TYPE}(64) \leq \text{len-of TYPE}(64)$
 $\text{len-of TYPE}(32) \leq \text{len-of TYPE}(64)$
 $\text{len-of TYPE}(16) \leq \text{len-of TYPE}(64)$
 $\text{len-of TYPE}(8) \leq \text{len-of TYPE}(64)$
 $\text{len-of TYPE}(32) \leq \text{len-of TYPE}(32)$
 $\text{len-of TYPE}(16) \leq \text{len-of TYPE}(32)$
 $\text{len-of TYPE}(8) \leq \text{len-of TYPE}(32)$
 $\text{len-of TYPE}(16) \leq \text{len-of TYPE}(16)$
 $\text{len-of TYPE}(8) \leq \text{len-of TYPE}(16)$
 $\text{len-of TYPE}(8) \leq \text{len-of TYPE}(8)$

$\text{len-of TYPE}(32) < \text{len-of TYPE}(64)$
 $\text{len-of TYPE}(16) < \text{len-of TYPE}(64)$
 $\text{len-of TYPE}(8) < \text{len-of TYPE}(64)$
 $\text{len-of TYPE}(16) < \text{len-of TYPE}(32)$
 $\text{len-of TYPE}(8) < \text{len-of TYPE}(32)$
 $\text{len-of TYPE}(8) < \text{len-of TYPE}(16)$

$\text{len-of TYPE}(a :: \text{len signed}) = \text{len-of TYPE}(a)$
 $(\text{len-of TYPE}(a) = \text{len-of TYPE}(a)) = \text{True}$
by *auto*

lemma *sbintrunc-eq*: $0 \leq i \implies i < 2^{\hat{n}} \implies \text{sbintrunc } n \ i = i$
apply (*induction n arbitrary: i*)
apply *auto*
done

lemma *uint-ucast*:

$uint (ucast\ x :: ('a :: len)\ word) = uint\ x\ mod\ 2^{\wedge}LENGTH('a)$
unfolding *uint-nat unat-ucast by (simp add: zmod-int)*

lemma *uint-scast'*:

$uint (SCAST('a::len \to 'b::len)\ c) = sint\ c\ mod\ 2^{\wedge}LENGTH('b)$
by (*metis Word.of-int-sint word-uint.eq-norm*)

lemma *uint-ucast'*:

$uint (UCAST('a::len \to 'b::len)\ c) = uint\ c\ mod\ 2^{\wedge}LENGTH('b)$
by (*metis Word.of-int-uint word-uint.eq-norm*)

lemma *sint-ucast'*:

$LENGTH('a) < LENGTH('b) \implies sint (UCAST('a::len \to 'b::len)\ c) = uint\ c$
by (*smt bintrunc-bintrunc-ge less-or-eq-imp-le signed-take-bit-eq uint-sint unsigned-ucast-eq unsigned-word-eqI*)

lemma *scast-ucast-eq-ucast [simp, L2opt]*:

$LENGTH('a::len) < LENGTH('b::len) \implies LENGTH('b) \leq LENGTH('c::len)$
 $\implies SCAST('b \to 'c) (UCAST('a \to 'b)\ x) = UCAST('a \to 'c)\ x$
unfolding *word-uint-eq-iff uint-scast' uint-ucast sint-ucast' ..*

lemma *scast-ucast-simps [simp, L2opt]*:

$\llbracket len-of\ TYPE('b) \leq len-of\ TYPE('a); len-of\ TYPE('c) \leq len-of\ TYPE('b) \rrbracket$
 $\implies (scast\ (ucast\ (a :: 'a::len\ word) :: 'b::len\ word) :: 'c::len\ word) = ucast\ a$
 $\llbracket len-of\ TYPE('c) \leq len-of\ TYPE('a); len-of\ TYPE('c) \leq len-of\ TYPE('b) \rrbracket$
 $\implies (scast\ (ucast\ (a :: 'a::len\ word) :: 'b::len\ word) :: 'c::len\ word) = ucast\ a$
 $\llbracket len-of\ TYPE('a) \leq len-of\ TYPE('b); len-of\ TYPE('c) \leq len-of\ TYPE('b) \rrbracket$
 $\implies (scast\ (ucast\ (a :: 'a::len\ word) :: 'b::len\ word) :: 'c::len\ word) = ucast\ a$
 $\llbracket len-of\ TYPE('a) \leq len-of\ TYPE('b) \rrbracket \implies (scast\ (scast\ (a :: 'a::len\ word) :: 'b::len\ word) :: 'c::len\ word) = scast\ a$
 $\llbracket len-of\ TYPE('b) \leq len-of\ TYPE('a); len-of\ TYPE('c) \leq len-of\ TYPE('b) \rrbracket$
 $\implies (ucast\ (scast\ (a :: 'a::len\ word) :: 'b::len\ word) :: 'c::len\ word) = scast\ a$
 $\llbracket len-of\ TYPE('c) \leq len-of\ TYPE('a); len-of\ TYPE('c) \leq len-of\ TYPE('b) \rrbracket$
 $\implies (ucast\ (scast\ (a :: 'a::len\ word) :: 'b::len\ word) :: 'c::len\ word) = scast\ a$
 $\llbracket len-of\ TYPE('a) \leq len-of\ TYPE('b); len-of\ TYPE('c) \leq len-of\ TYPE('b) \rrbracket$
 $\implies (ucast\ (scast\ (a :: 'a::len\ word) :: 'b::len\ word) :: 'c::len\ word) = scast\ a$
 $\llbracket len-of\ TYPE('c) \leq len-of\ TYPE('b) \rrbracket \implies (ucast\ (ucast\ (a :: 'a::len\ word) :: 'b::len\ word) :: 'c::len\ word) = ucast\ a$
 $\llbracket len-of\ TYPE('a) \leq len-of\ TYPE('b) \rrbracket \implies (ucast\ (ucast\ (a :: 'a::len\ word) :: 'b::len\ word) :: 'c::len\ word) = ucast\ a$

$\llbracket \text{len-of } TYPE('a) \leq \text{len-of } TYPE('b) \rrbracket \implies$
 $(\text{scast } (\text{scast } (a :: 'a::\text{len word}) :: 'b::\text{len word}) :: 'c::\text{len word}) = \text{scast } a$
by (*auto simp: is-up is-down*
scast-ucast-1 scast-ucast-3 scast-ucast-4
ucast-scast-1 ucast-scast-3 ucast-scast-4
scast-scast-a scast-scast-b
ucast-ucast-a ucast-ucast-b)

declare *len-signed* [L2opt]

lemmas [L2opt] = *zero-sle-ucast-up*

lemma *zero-sle-ucast-WORD-MAX* [L2opt]:
 $(0 \leq s ((\text{ucast } (b::('a::\text{len}) \text{ word})) :: ('a::\text{len}) \text{ signed word}))$
 $= (\text{uint } b \leq \text{WORD-MAX } (TYPE('a)))$
by (*clarsimp simp: WORD-MAX-def zero-sle-ucast*)

lemmas [L2opt] =
is-up is-down unat-ucast-upcast sint-ucast-eq-wint

lemmas [L2opt] =
ucast-down-add scast-down-add
ucast-down-minus scast-down-minus
ucast-down-mult scast-down-mult

lemma *eq-trivI*: $x = y \implies x = y$
by *simp*

lemmas [*word-abs*] =
corresTA-L2-gets
corresTA-L2-modify
corresTA-L2-throw
corresTA-L2-skip
corresTA-L2-fail
corresTA-L2-seq
corresTA-L2-seq-unit
corresTA-L2-catch
corresTA-L2-try'

corresTA-L2-while
corresTA-L2-guard
corresTA-L2-guarded-simple
corresTA-L2-spec
corresTA-L2-assume
corresTA-L2-condition

corresTA-L2-unknown
corresTA-case-prod
corresTA-L2-call-exec-concrete'
corresTA-L2-call-exec-concrete
corresTA-L2-call-exec-abstract'
corresTA-L2-call-exec-abstract
corresTA-L2-call'
corresTA-L2-call
corresTA-call-L1

lemmas *[word-abs]* =
abstract-val-tuple
abstract-val-Inl
abstract-val-Inr
abstract-val-conj
abstract-val-disj
abstract-val-case-prod
abstract-val-trivial
abstract-val-of-int
abstract-val-of-nat
abstract-val-call-L1-args

lemmas *abs-var-rules* =
abstract-val-call-L1-arg
abstract-val-abs-var-sint-unat
abstract-val-abs-var-uint-unat
abstract-val-abs-var-give-up
abstract-val-abs-var-concretise
abstract-val-abs-var

lemmas *word-abs-base [word-abs]* =
valid-typ-abs-fn-id [where 'a='a::c-type]
valid-typ-abs-fn-id [where 'a=bool]
valid-typ-abs-fn-id [where 'a='gx c-exntype]
valid-typ-abs-fn-tuple-split
valid-typ-abs-fn-tuple
valid-typ-abs-fn-sum
valid-typ-abs-fn-unit
valid-typ-abs-fn-sint
valid-typ-abs-fn-unat
len-of-word-comparisons

lemmas *word-abs-sword* =
abstract-val-signed-ops
abstract-val-scast
abstract-val-scast-upcast


```

abstract-val-scast-downcast
abstract-val-unwrap [where f=sint]
introduce-typ-abs-fn [where f=sint :: (sword64 ⇒ int)]
introduce-typ-abs-fn [where f=sint :: (sword32 ⇒ int)]
introduce-typ-abs-fn [where f=sint :: (sword16 ⇒ int)]
introduce-typ-abs-fn [where f=sint :: (sword8 ⇒ int)]

```

```

lemmas word-abs-word =
  abstract-val-unsigned-ops
  abstract-val-uint
  abstract-val-ucast
  abstract-val-ucast-upcast
  abstract-val-ucast-downcast
  abstract-val-unwrap [where f=unat]
  introduce-typ-abs-fn [where f=unat :: (word64 ⇒ nat)]
  introduce-typ-abs-fn [where f=unat :: (word32 ⇒ nat)]
  introduce-typ-abs-fn [where f=unat :: (word16 ⇒ nat)]
  introduce-typ-abs-fn [where f=unat :: (word8 ⇒ nat)]

```

```

lemmas word-abs-default =
  introduce-typ-abs-fn [where f=id :: ('a::c-type ⇒ 'a)]
  introduce-typ-abs-fn [where f=id :: (bool ⇒ bool)]
  introduce-typ-abs-fn [where f=id :: ('gx c-exntype ⇒ 'gx c-exntype)]
  introduce-typ-abs-fn [where f=id :: (unit ⇒ unit)]
  introduce-typ-abs-fn-tuple
  introduce-typ-abs-fn-sum

```

```

thm word-abs

```

```

lemma int-bounds-to-nat-boundsF: (n::int) < numeral B ⇒ 0 ≤ n ⇒ nat n <
numeral B

```

```

  by (simp add: nat-less-iff)

```

```

lemma int-bounds-one-to-nat: (n::int) < 1 ⇒ 0 ≤ n ⇒ nat n = 0

```

```

  by (simp add: nat-less-iff)

```

```

lemma id-map-prod-unfold: id = map-prod id id

```

```

  by (simp add: prod.map-id0)

```

```

lemma id-tuple-unfold: id = (λ(x, y). (id x, id y))

```

```

  by simp

```

```

end

```

```

theory WordPolish

```

imports *WordAbstract*
begin

definition [*simplified*]: *LONG-MAX* $\equiv (2 :: \text{int}) ^ 63 - 1$

definition [*simplified*]: *LONG-MIN* $\equiv - ((2 :: \text{int}) ^ 63)$

definition [*simplified*]: *ULONG-MAX* $\equiv (2 :: \text{nat}) ^ 64 - 1$

definition [*simplified*]: *INT-MAX* $\equiv (2 :: \text{int}) ^ 31 - 1$

definition [*simplified*]: *INT-MIN* $\equiv - ((2 :: \text{int}) ^ 31)$

definition [*simplified*]: *UINT-MAX* $\equiv (2 :: \text{nat}) ^ 32 - 1$

definition [*simplified*]: *SHORT-MAX* $\equiv (2 :: \text{int}) ^ 15 - 1$

definition [*simplified*]: *SHORT-MIN* $\equiv - ((2 :: \text{int}) ^ 15)$

definition [*simplified*]: *USHORT-MAX* $\equiv (2 :: \text{nat}) ^ 16 - 1$

definition [*simplified*]: *CHAR-MAX* $\equiv (2 :: \text{int}) ^ 7 - 1$

definition [*simplified*]: *CHAR-MIN* $\equiv - ((2 :: \text{int}) ^ 7)$

definition [*simplified*]: *UCHAR-MAX* $\equiv (2 :: \text{nat}) ^ 8 - 1$

lemma *WORD-MAX-simps*:

WORD-MAX TYPE(64) = *LONG-MAX*

WORD-MAX TYPE(32) = *INT-MAX*

WORD-MAX TYPE(16) = *SHORT-MAX*

WORD-MAX TYPE(8) = *CHAR-MAX*

by (*auto simp: LONG-MAX-def INT-MAX-def SHORT-MAX-def CHAR-MAX-def WORD-MAX-def*)

lemma *WORD-MIN-simps*:

WORD-MIN TYPE(64) = *LONG-MIN*

WORD-MIN TYPE(32) = *INT-MIN*

WORD-MIN TYPE(16) = *SHORT-MIN*

WORD-MIN TYPE(8) = *CHAR-MIN*

by (*auto simp: LONG-MIN-def INT-MIN-def SHORT-MIN-def CHAR-MIN-def WORD-MIN-def*)

lemma *UWORD-MAX-simps*:

UWORD-MAX TYPE(64) = *ULONG-MAX*

UWORD-MAX TYPE(32) = *UINT-MAX*

UWORD-MAX TYPE(16) = *USHORT-MAX*

UWORD-MAX TYPE(8) = *UCHAR-MAX*

by (*auto simp: ULONG-MAX-def UINT-MAX-def USHORT-MAX-def UCHAR-MAX-def UWORD-MAX-def*)

lemma *MIN-MAX-lemmas-schema*:

sint (*s::'a::len signed word*) \leq *WORD-MAX TYPE*('a)

WORD-MIN TYPE('a) \leq *sint* (*s::'a::len signed word*)

```

unat (u::'a::len word) ≤ UWORD-MAX TYPE('a)
¬ (sint (s::'a::len signed word) > WORD-MAX TYPE('a))
¬ (WORD-MIN TYPE('a) > sint (s::'a::len signed word))
¬ (unat (u::'a::len word) > UWORD-MAX TYPE('a))
WORD-MIN TYPE('a) ≤ WORD-MAX TYPE('a)
0 ≤ WORD-MAX TYPE('a)
WORD-MIN TYPE('a) ≤ 0
unfolding WORD-MAX-def WORD-MIN-def UWORD-MAX-def
using unat-lt2p[where x=u] less-eq-Suc-le
       sint-range-size [where w=s, simplified word-size, simplified]
by auto

```

```

lemmas MIN-MAX-lemmas-pre =
  MIN-MAX-lemmas-schema[where 'a=64]
  MIN-MAX-lemmas-schema[where 'a=32]
  MIN-MAX-lemmas-schema[where 'a=16]
  MIN-MAX-lemmas-schema[where 'a=8]

```

```

lemmas INT-MIN-MAX-lemmas [simp] =
  MIN-MAX-lemmas-pre[unfolded WORD-MAX-simps WORD-MIN-simps UWORD-MAX-simps]

```

end

Chapter 23

TS phase: Type Strengthening (find suitable target monad)

```
theory Refines-Spec  
  imports  
    Option-MonadND  
    L2ExceptionRewrite  
begin
```

```
lemma gets-the-ocondition:  
  Spec-Monad.gets-the (ocondition C T F) =  
    Spec-Monad.condition C (Spec-Monad.gets-the T) (Spec-Monad.gets-the F)  
by (simp add: spec-monad-ext-iff ocondition-def run-condition)
```

```
lemma gets-the-oreturn:  
  Spec-Monad.gets-the (oreturn x) = Spec-Monad.return x  
by (simp add: spec-monad-ext-iff)
```

```
lemma gets-the-obind:  
  Spec-Monad.gets-the (obind f g) =  
    Spec-Monad.bind (Spec-Monad.gets-the f) (λx. Spec-Monad.gets-the (g x))  
by (simp add: spec-monad-ext-iff run-bind obind-def split: option.splits)
```

```
setup <  
  let open Mutual-CCPO-Rec in  
    add-ccpo spec-monad-gfp (fn ctxt => fn (T as Type <spec-monad E A S>) =>  
      let  
        in  
        synth-gfp ctxt T
```

```

    end)
  |> Context.theory-map
end
>

```

lemma *rel-map-the-Result*[simp]: *rel-map the-Result (Result v) b* \longleftrightarrow *v = b*
by (*auto simp add: rel-map-def*)

lemma *holds-partial-post-state-Inf*:
assumes *X*: $\bigwedge x. x \in X \implies \text{holds-partial-post-state } P \ x$
shows *holds-partial-post-state* *P* $(\bigcap X)$
apply (*clarsimp simp add: Inf-post-state-def vimage-def*)
subgoal **premises** *prems* **for** *p x*
using *X*[*of Success x*] *prems* **by** *auto*
done

lemma *holds-post-state-Inf*:
assumes *X*: $\bigwedge x. x \in X \implies \text{holds-post-state } P \ x$
shows $X \neq \{\}$ $\implies \text{holds-post-state } P \ (\bigcap X)$
apply (*clarsimp simp add: Inf-post-state-def vimage-def, safe*)
subgoal **for** *x* **using** *X*[*of x*] **by** (*cases x*) *auto*
subgoal **using** *assms* **by** *force*
done

lemma *admissible-runs-to*[*corres-admissible*]:
ccpo.admissible *Inf* $(\lambda x y. y \leq x)$ $(\lambda x. x \cdot s \ \{\!\! \{ Q \!\!\})$
apply (*simp add: ccpo.admissible-def chain-def*)
apply *transfer*
apply (*auto intro: holds-post-state-Inf*)
done

lemma *spec-monad-Inf-run*: $(\text{run } (\bigcap A) \ s) = \bigcap ((\lambda f. (\text{run } f \ s)) \ ` A)$
by (*metis INF-apply Inf-spec-monad.rep-eq*)

lemma *outcomes-Inf-run-succeeds-conv*:
 $\text{outcomes } (\bigcap f \in A. \text{run } f \ t) = \text{outcomes } (\bigcap f \in \{f. f \in A \wedge \text{succeeds } f \ t\}. \text{run } f \ t)$
apply (*clarsimp simp add: Inf-post-state-def succeeds-def top-post-state-def*)
apply *fastforce*
done

lemma *succeeds-outcomes-Inf-Inter-conv*:
 $g \in A \implies \text{succeeds } g \ t \implies \text{outcomes } (\bigcap f \in \{f \in A. \text{succeeds } f \ t\}. \text{run } f \ t) =$
 $(\bigcap f \in \{f \in A. \text{succeeds } f \ t\}. \text{outcomes } (\text{run } f \ t))$
apply (*clarsimp simp add: Inf-post-state-def succeeds-def top-post-state-def, intro impI conjI set-eqI iffI*)
subgoal **by** (*auto simp add: vimage-def image-def split: post-state.splits*)
subgoal **by** (*smt (verit, ccfv-SIG) INF-top-conv(2) Inf-post-state-def mem-Collect-eq top-post-state-def*)

subgoal by (*smt (verit) Success-vimage-image-cancel image-cong mem-Collect-eq outcomes-succeeds-run-conv succeeds-def top-post-state-def*)
subgoal by (*smt (verit, best) Success-vimage-image-cancel image-cong mem-Collect-eq outcomes-succeeds-run-conv succeeds-def top-post-state-def*)
done

lemma *admissible-refines-spec-fun-of-rel:*

assumes *prj: fun-of-rel R prj*

shows *ccpo.admissible Inf (\geq)*

($\lambda(A::('e::default, 'a, 's) spec-monad) . refines C A s t (rel-prod R (=)))$)

proof (*rule ccpo.admissibleI*)

fix *A::('e, 'a, 's) spec-monad set*

assume *A-prop: $\forall g \in A. refines C g s t (rel-prod R (=))$*

assume *chain: Complete-Partial-Order.chain (\geq) A*

assume *non-empty: $A \neq \{\}$*

have *Inf-lower: $\bigwedge g. g \in A \implies \bigcap A \leq g$*

by (*simp add: Inf-lower*)

hence *run-Inf-lower: $\bigwedge g s. g \in A \implies run (\bigcap A) s \leq run g s$*

by (*simp add: le-funD less-eq-spec-monad.rep-eq*)

have (*$\bigwedge s. (\bigwedge g. g \in A \implies run g s = Failure) \implies run (\bigcap A) s = Failure$*)

apply (*simp add: spec-monad-Inf-run*)

by (*simp add: non-empty*)

hence *succeeds: $\bigwedge s. succeeds (\bigcap A) s \implies (\exists g \in A. succeeds g s)$*

by (*auto simp add: succeeds-def top-post-state-def*)

show *refines C ($\bigcap A$) s t (rel-prod R (=))*

apply (*clarsimp simp add: refines-def-old reaches-def*)

apply (*intro conjI allI impI*)

proof –

assume *succeeds ($\bigcap A$) t then*

show *succeeds C s*

using *succeeds A-prop non-empty*

by (*auto simp add: refines-def-old*)

next

fix *r s'*

assume *succeeds-Inf: succeeds ($\bigcap A$) t*

then obtain *g where $g \in A$ and succeeds-g: succeeds g t*

using *succeeds by blast*

assume *res: $(r, s') \in outcomes (run C s)$*

show *$\exists r'. (r', s') \in outcomes (run (\bigcap A) t) \wedge R r r'$*

proof –

from *A-prop res prj*

have *prj-prop: $\bigwedge g. g \in A \implies succeeds g t \implies (prj r, s') \in outcomes (run g$*

$t) \wedge R r (prj r)$
apply (*clarsimp simp add: refines-def-old*)
using *fun-of-rel-def*
by (*smt (verit) reaches-def*)
hence ($prj r, s') \in outcomes (run (\sqcap A) t)$
apply (*simp add: spec-monad-Inf-run*)
apply (*subst outcomes-Inf-run-succeeds-conv*)
apply (*subst succeeds-outcomes-Inf-Inter-conv [OF g succeeds-g]*)
apply *auto*
done
moreover from *prj-prop g succeeds-g* **have** $R r (prj r)$ **by** *blast*
ultimately show *?thesis* **by** *blast*
qed
qed
qed

lemma *admissible-refines-spec-funp*:
assumes *funp R*
shows *ccpo.admissible Inf (\geq)*
 $(\lambda(A::('e::default, 'a, 's) spec-monad) . refines C A s t (rel-prod R (=)))$
proof –
from *assms* **obtain** *prj* **where** *fun-of-rel R prj*
by (*auto simp add: funp-def*)
then show *?thesis*
by (*rule admissible-refines-spec-fun-of-rel*)
qed

lemma *admissible-refines-spec-res[corres-admissible]*:
shows *ccpo.admissible Inf (\geq)*
 $(\lambda(A::('a, 's) res-monad) . refines C A s t (rel-prod (rel-liftE) (=)))$
by (*rule admissible-refines-spec-funp*) (*intro funp-intros*)

lemma *admissible-refines-spec-exit-eq[corres-admissible]*:
shows *ccpo.admissible Inf (\geq)*
 $(\lambda(A::('e, 'a, 's) exn-monad) . refines C A s t (rel-prod (rel-xval (=) (=)) (=)))$
by (*rule admissible-refines-spec-funp*) (*intro funp-intros*)

lemma *admissible-refines-spec-exit-the-Nonlocal[corres-admissible]*:
shows *ccpo.admissible Inf (\geq)*
 $(\lambda(A::('e, 'a, 's) exn-monad) . refines C A s t (rel-prod ((rel-xval (rel-Nonlocal) (=))) (=)))$
by (*rule admissible-refines-spec-funp*) (*intro funp-intros*)

lemma *gen-admissible-refines-gets-the[corres-admissible]*:
shows *ccpo.admissible option.lub-fun option.le-fun* $(\lambda A. refines C (gets-the A) s t (rel-prod (rel-liftE) (=)))$
apply (*clarsimp simp add: refines-def-old relcompp.simps rel-liftE-def rel-map-def*)

split: option.splits)
by (*smt (verit) ccpo.admissibleI chain-fun flat-lub-in-chain fun-lub-def mem-Collect-eq option.discI*)

theorem *refines-option-top [corres-top]: refines f (gets-the Map.empty) s t R*
by (*auto simp add: refines-def-old*)

23.1 Synthesize Rules Setup

Canonical format for the currently supported monads: *refines C (lift-to-spec A) s t (rel-prod rel-res (=))* where *lift-spec*, *rel-res* is

- pure (Pure function): *return, rel-liftE*
- gets (Reader monad): *gets, rel-liftE*
- option (Option monad): *gets-the, rel-liftE*
- nondet: *id (ommitted), rel-liftE*
- exit:
 - *id (ommitted), rel-xval rel-Nonlocal (=)*
 - *id (ommitted), rel-xval (=) (=)* for those function that were lifted by the IO phase.

synthesize-rules *pure and reader and option and nondet and exit*

add-synthesize-pattern *pure and reader and option and nondet and exit where*
refines-pure-synth = ⟨Trueprop (refines ?f ((return ?f')::(?'a, ?'s) res-monad) - - ?R)⟩ (f')

add-synthesize-pattern *reader and option and nondet and exit where*
refines-reader-synth = ⟨Trueprop (refines ?f (gets ?f') - - ?R)⟩ (f')

add-synthesize-pattern *option where*
refines-option-synth = ⟨Trueprop (refines ?f (gets-the ?f') - - ?R)⟩ (f')

add-synthesize-pattern *nondet and exit where*
refines-nondet-synth = ⟨Trueprop (refines ?f ?f' - - ?R)⟩ (f') and

rel-liftE-nondet-synth = ⟨Trueprop (rel-liftE ?x ?x')⟩ (x') and
rel-sum-nondet-synth = ⟨Trueprop (rel-sum - - ?x ?x')⟩ (x') and
rel-sum-nondet-synth = ⟨Trueprop (rel-xval - - ?x ?x')⟩ (x') and
rel-compp-nondet-synth = ⟨Trueprop ((- OO -) ?x ?x')⟩ (x') and
rel-map-synth = ⟨Trueprop (rel-map ?f ?x ?x')⟩ (x') and
rel-eq-nondet-synth = ⟨Trueprop (?x = ?x')⟩ (x')

add-synthesize-pattern *exit where*

rel-Nondet-exit-synth = ⟨Trueprop (rel-Nonlocal ?x ?x')⟩ (x')

Recursive functions are defined using **fixed-point**. The option, nondet and exit monad are setup to handle these definitions. Hence the minimal monad for recursive functions is the option monad.

23.2 Pure Monad

lemma *refines-L2-call-embed-pure:*

assumes *f: refines f (return f') s s (rel-prod rel-liftE (=))*

shows *refines (L2-call f emb ns) (return (L2-VARS f' ns)::('a,'s) res-monad) s s (rel-prod rel-liftE (=))*

unfolding *L2-VARS-def L2-call-def*

using *f*

apply (*clarsimp simp add: refines-def-old reaches-map-value map-exn-def rel-liftE-def split: xval-splits*)

apply (*metis Exn-def default-option-def exception-or-result-cases not-Some-eq*)+
done

lemma *refines-L2-gets-pure:*

refines (L2-gets (λ-. v) ns) (return (L2-VARS v ns)::('a,'s) res-monad) s s (rel-prod (rel-liftE) (=))

unfolding *L2-VARS-def L2-defs gets-return*

by (*auto intro: refines-yield*)

lemma *refines-L2-seq-pure:*

assumes *g [unfolded THIN-def, rule-format]: PROP THIN (∧v t. refines (g v) (return (g' v)::('a,'s) res-monad) t t (rel-prod rel-liftE (=)))*

assumes *f [unfolded THIN-def, rule-format]: PROP THIN (Trueprop (refines f ((return f')::('b,'s) res-monad) s s (rel-prod rel-liftE (=))))*

shows *refines (L2-seq f g) (return (let v = f' in (g' v))::('a,'s) res-monad) s s (rel-prod rel-liftE (=))*

using *g f unfolding L2-defs*

apply (*clarsimp simp add: refines-def-old reaches-bind succeeds-bind rel-liftE-def default-option-def split: exception-or-result-splits*)

apply (*metis default-option-def exception-or-result-cases option.exhaust-sel*)+
done

lemma *refines-L2-condition-pure:*

assumes *g: refines g (return g')::('a,'s) res-monad) s s (rel-prod rel-liftE (=))*

assumes *f: refines f (return f')::('a,'s) res-monad) s s (rel-prod rel-liftE (=))*

shows *refines (L2-condition (λ-. c) f g) (return (if c then f' else g')::('a,'s) res-monad) s s (rel-prod rel-liftE (=))*

using *g f unfolding L2-defs*

by (*auto intro: refines-condition*)

lemma *refines-L2-try-L2-throw-pure:*

shows *refines* (*L2-try* (*L2-throw* (*Inr r ns*)) (*return* (*L2-VARS r ns*))::('a,'s) *res-monad*)
s s (*rel-prod rel-liftE* (=))
unfolding *L2-defs L2-VARS-def*
by (*auto simp add: refines-def-old reaches-try rel-liftE-def unnest-exn-def*)

lemma *refines-L2-try-L2-seq-pure*:

assumes *g*: $\bigwedge v t. \text{refines } (L2\text{-try } (g v)) \text{ (return } (g' v)::('a,'s) \text{ res-monad) } t t$
(*rel-prod rel-liftE* (=))
assumes *f*: *refines* *f* (*return* *f'*::('b,'s) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try* (*L2-seq f g*)) (*return* (*let v = f' in (g' v)*))::('a,'s) *res-monad*)
s s (*rel-prod rel-liftE* (=))
unfolding *L2-defs try-def refines-map-value-left-iff return-let-bind*
apply (*rule refines-bind [OF f]*)
subgoal by *auto*
subgoal by (*auto simp add: default-option-def Exn-def[symmetric] rel-liftE-def*)
subgoal by *auto*
subgoal using *g [simplified L2-defs try-def refines-map-value-left-iff]* **by** *auto*
done

lemma *refines-L2-try-L2-condition-pure*:

assumes *f*: *refines* (*L2-try f*) (*return* *f'*::('a,'s) *res-monad*) *s s* (*rel-prod rel-liftE*
(=))
assumes *g*: *refines* (*L2-try g*) (*return* *g'*::('a,'s) *res-monad*) *s s* (*rel-prod rel-liftE*
(=))
shows *refines* (*L2-try* (*L2-condition* ($\lambda-. c$) *f g*)) (*return* (*if c then f' else*
g'))::('a,'s) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
using *f g unfolding L2-defs*
apply (*auto simp add: refines-def-old reaches-try reaches-bind succeeds-bind rel-liftE-def*
unnest-exn-def
default-option-def Exn-def [symmetric]
split: xval-split exception-or-result-splits sum.splits)
done

lemma *refines-L2-try-pure*:

assumes *f*: *refines* *f* (*return* *f'*::('a,'s) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try f*) (*return* *f'*::('a,'s) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs L2-guarded-def try-def using f*
apply (*clarsimp simp add: refines-def-old reaches-try reaches-map-value rel-liftE-def*
unnest-exn-def
default-option-def Exn-def [symmetric]
split: xval-split exception-or-result-splits sum.splits)
by (*metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel*
sum-all-ex(2))

print-synthesize-rules *pure*

lemmas *refines-monad-pure =*

refines-L2-call-embed-pure [synthesize-rule pure and reader and option and non-
det and exit priority:510]

refines-L2-gets-pure [*synthesize-rule pure and reader and option and nondet and exit priority:510*]
refines-L2-seq-pure [*synthesize-rule pure and reader and option and nondet and exit priority:510 split: g and g'*]
refines-L2-condition-pure [*synthesize-rule pure and reader and option and nondet and exit priority:510*]
refines-L2-try-L2-throw-pure [*synthesize-rule pure and reader and option and nondet and exit priority:520*]
refines-L2-try-L2-seq-pure [*synthesize-rule pure and reader and option and nondet and exit priority:520 split: g and g'*]
refines-L2-try-L2-condition-pure [*synthesize-rule pure and reader and option and nondet and exit priority:520*]
refines-L2-try-pure [*synthesize-rule pure and reader and option and nondet and exit priority:510*]

print-synthesize-rules *pure*

23.3 Reader Monad (Gets)

lemma *refines-L2-call-embed-reader:*

assumes *f: refines f (gets f') s s (rel-prod rel-liftE (=))*
shows *refines (L2-call f emb ns) (gets (L2-VARS f' ns)) s s (rel-prod rel-liftE (=))*
unfolding *L2-defs L2-call-def liftE-def L2-VARS-def using f*
apply (*clarsimp simp add: refines-def-old reaches-try reaches-map-value rel-liftE-def map-exn-def*
default-option-def Exn-def [symmetric]
split: xval-split exception-or-result-splits sum.splits)
by (*metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel*)

lemma *refines-L2-gets-reader:*

refines (L2-gets f ns) (gets (L2-VARS f ns)) s s (rel-prod rel-liftE (=))
unfolding *L2-defs L2-VARS-def liftE-def*
by (*auto intro: refines-gets*)

lemma *refines-lift-pure-reader:*

assumes *f: refines f (return f') s s (rel-prod rel-liftE (=))*
shows *refines f (gets (λ-. f')) s s (rel-prod rel-liftE (=))*
using *f unfolding liftE-def*
by (*simp add: gets-return*)

lemma *refines-L2-seq-reader:*

assumes *g: ∧ v t. refines (g v) (gets (g' v)) t t (rel-prod rel-liftE (=))*
assumes *f: refines f (gets f') s s (rel-prod rel-liftE (=))*
shows *refines (L2-seq f g) (gets (λs. let v = f' s in (g' v s))) s s (rel-prod rel-liftE (=))*
using *g f unfolding L2-defs liftE-def*
apply (*clarsimp simp add: refines-def-old reaches-bind succeeds-bind rel-liftE-def*)

```

    default-option-def Exn-def [symmetric]
    split: xval-split exception-or-result-splits sum.splits)
apply (metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel)+
done

```

lemma *refines-L2-condition-reader:*

```

assumes g [unfolded THIN-def]: PROP THIN (Trueprop (refines g (gets g') s s
(rel-prod rel-liftE (=))))
assumes f [unfolded THIN-def]: PROP THIN (Trueprop (refines f (gets f') s s
(rel-prod rel-liftE (=))))
shows refines (L2-condition c f g) (gets (λs. if c s then f' s else g' s)) s s (rel-prod
rel-liftE (=))
using g f unfolding L2-defs liftE-def
apply (auto simp add: refines-def-old reaches-bind succeeds-bind rel-liftE-def
    default-option-def Exn-def [symmetric]
    split: xval-split exception-or-result-splits sum.splits)
done

```

lemma *refines-L2-try-L2-throw-reader:*

```

shows refines (L2-try (L2-throw (Inr r) ns)) (gets (L2-VARS (λ-. r) ns)) s s
(rel-prod rel-liftE (=))
unfolding L2-defs L2-VARS-def liftE-def
apply (auto simp add: refines-def-old reaches-try reaches-map-value rel-liftE-def
map-exn-def
    default-option-def Exn-def [symmetric]
    split: xval-split exception-or-result-splits sum.splits)
done

```

lemma *refines-L2-try-L2-seq-reader:*

```

assumes g [unfolded THIN-def, rule-format]: PROP THIN (Λv t. refines (L2-try
(g v)) (gets (g' v)) t t (rel-prod rel-liftE (=)))
assumes f [unfolded THIN-def]: PROP THIN (Trueprop (refines f (gets f') s s
(rel-prod rel-liftE (=))))
shows refines (L2-try (L2-seq f g)) (gets (λs. let v = f' s in (g' v s))) s s (rel-prod
rel-liftE (=))
unfolding L2-defs try-def refines-map-value-left-iff gets-let-bind
apply (rule refines-bind [OF f])
subgoal by auto
subgoal by (auto simp add: default-option-def Exn-def [symmetric] rel-liftE-def)
subgoal by auto
subgoal using g [unfolded L2-defs try-def refines-map-value-left-iff ] by auto
done

```

lemma *refines-L2-try-L2-condition-reader:*

```

assumes f [unfolded THIN-def]: PROP THIN (Trueprop (refines (L2-try f) (gets
f') s s (rel-prod rel-liftE (=))))
assumes g [unfolded THIN-def]: PROP THIN (Trueprop (refines (L2-try g) (gets
g') s s (rel-prod rel-liftE (=))))

```

shows *refines* (*L2-try* (*L2-condition* (*c*) *f g*)) (*gets* ($\lambda s. \text{if } c \text{ s then } f' \text{ s else } g' \text{ s}$))
s s (*rel-prod rel-liftE* (=))
using *f g unfolding* *L2-defs liftE-def*
apply (*auto simp add: refines-def-old reaches-try*)
done

lemma *refines-L2-try-reader:*

assumes *f: refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try f*) (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs L2-guarded-def try-def liftE-def* **using** *f*
by (*auto simp add: refines-def-old reaches-map-value rel-liftE-def unnest-exn-def*
split: xval-splits)
(metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel)+

lemmas *refines-monad-reader =*

refines-L2-call-embed-reader [*synthesize-rule reader and option and nondet and*
exit priority:410]
refines-L2-gets-reader [*synthesize-rule reader and option and nondet and exit*
priority:410]
refines-lift-pure-reader [*synthesize-rule reader and option and nondet and exit*
priority:440]
refines-L2-seq-reader [*synthesize-rule reader and option and nondet and exit*
priority:410 split: g and g']
refines-L2-condition-reader [*synthesize-rule reader and option and nondet and*
exit priority:410]
refines-L2-try-L2-throw-reader [*synthesize-rule reader and option and nondet*
and *exit priority:420*]
refines-L2-try-L2-seq-reader [*synthesize-rule reader and option and nondet and*
exit priority:420 split: g and g']
refines-L2-try-L2-condition-reader [*synthesize-rule reader and option and nondet*
and *exit priority:420*]
refines-L2-try-reader [*synthesize-rule reader and option and nondet and exit*
priority:410]

23.4 Option (Reader) Monad

lemma *refines-lift-reader-option:*

assumes *f: refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines f* (*gets-the* (*ogets f'*)) *s s* (*rel-prod rel-liftE* (=))
using *f*
apply (*auto simp add: refines-def-old rel-liftE-def ogets-def*)
done

lemma *refines-L2-call-embed-option:*

assumes *f: refines f* (*gets-the f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-call f emb ns*) (*gets-the* (*L2-VARS f' ns*)) *s s* (*rel-prod rel-liftE*
(=))
unfolding *L2-VARS-def L2-call-def L2-defs* **using** *f*
apply (*clarsimp simp add: refines-def-old reaches-map-value map-exn-def rel-liftE-def*)

split: xval-splits)
apply (*metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel*) +
done

lemma *refines-L2-gets-option:*

refines (L2-gets v ns) (gets-the (L2-VARS (ogets v) ns)) s s (rel-prod rel-liftE (=))
unfolding *L2-VARS-def L2-defs liftE-def*
apply (*auto simp add: refines-def-old rel-liftE-def ogets-def*)
done

lemma *refines-L2-seq-option:*

assumes *g [unfolded THIN-def, rule-format]: PROP THIN ($\bigwedge v t.$ refines (g v) (gets-the (g' v)) t t (rel-prod rel-liftE (=)))*
assumes *f [unfolded THIN-def]: PROP THIN (Trueprop (refines f (gets-the f') s s (rel-prod rel-liftE (=))))*
shows *refines (L2-seq f g) (gets-the (f' |>> g')) s s (rel-prod rel-liftE (=))*
using *g f unfolding L2-defs obind-def [abs-def]*
by (*fastforce simp add: refines-def-old reaches-bind succeeds-bind rel-liftE-def split: xval-splits option.splits*)

lemma *refines-L2-condition-option:*

assumes *g [unfolded THIN-def]: PROP THIN (Trueprop (refines g (gets-the g') s s (rel-prod rel-liftE (=))))*
assumes *f [unfolded THIN-def]: PROP THIN (Trueprop (refines f (gets-the f') s s (rel-prod rel-liftE (=))))*
shows *refines (L2-condition c f g) (gets-the (ocondition c f' g')) s s (rel-prod rel-liftE (=))*
using *g f unfolding L2-defs ocondition-def*
by (*auto simp add: refines-def-old rel-liftE-def split: xval-splits option.splits*)

lemma *refines-L2-try-L2-throw-option:*

shows *refines (L2-try (L2-throw (Inr r) ns)) (gets-the (L2-VARS (oreturn r) ns)) s s (rel-prod rel-liftE (=))*
unfolding *L2-defs L2-VARS-def oreturn-def K-def*
by (*auto simp add: refines-def-old reaches-try unnest-exn-def rel-liftE-def split: xval-splits*)

lemma *refines-L2-try-L2-seq-option:*

assumes *g [unfolded THIN-def, rule-format]: PROP THIN ($\bigwedge v t.$ refines (L2-try (g v)) (gets-the (g' v)) t t (rel-prod rel-liftE (=)))*
assumes *f [unfolded THIN-def]: PROP THIN (Trueprop (refines f (gets-the f') s s (rel-prod rel-liftE (=))))*
shows *refines (L2-try (L2-seq f g)) (gets-the (f' |>> g')) s s (rel-prod rel-liftE (=))*
unfolding *L2-defs try-def refines-map-value-left-iff*
apply (*simp add: gets-the-obind*)
apply (*rule refines-bind [OF f]*)

subgoal by *auto*
subgoal by (*auto simp add: default-option-def Exn-def [symmetric] rel-liftE-def*)
subgoal by *auto*
subgoal using *g [simplified L2-defs try-def refines-map-value-left-iff]* **by** *auto*
done

lemma *refines-L2-try-L2-condition-option:*
assumes *f [unfolded THIN-def]: PROP THIN (Trueprop (refines (L2-try f) (gets-the f') s s (rel-prod rel-liftE (=))))*
assumes *g [unfolded THIN-def]: PROP THIN (Trueprop (refines (L2-try g) (gets-the g') s s (rel-prod rel-liftE (=))))*
shows *refines (L2-try (L2-condition c f g)) (gets-the (ocondition c f' g')) s s (rel-prod rel-liftE (=))*
using *f g unfolding L2-defs ocondition-def*
apply (*auto simp add: refines-def-old reaches-try*)
done

lemma *refines-L2-try-option:*
assumes *f: refines f (gets-the f') s s (rel-prod rel-liftE (=))*
shows *refines (L2-try f) (gets-the f') s s (rel-prod rel-liftE (=))*
unfolding *L2-defs using f*
apply (*clarsimp simp add: refines-def-old reaches-try unnest-exn-def rel-liftE-def split: xval-splits*)
apply (*metis Exn-def default-option-def exception-or-result-cases not-None-eq*)
done

lemma *le-whileLoop-succeeds-terminatesI:*
assumes $\bigwedge s. \text{run } (\text{whileLoop } C \ B \ r) \ s \neq \top \implies \text{run } f \ s \leq \text{run } (\text{whileLoop } C \ B \ r) \ s$
shows $f \leq \text{whileLoop } C \ B \ r$
proof (*rule le-spec-monad-runI*)
fix *s*
show $\text{run } f \ s \leq \text{run } (\text{whileLoop } C \ B \ r) \ s$
proof (*cases run (whileLoop C B r) s = \top*)
case False then show ?thesis by (*rule assms*)
qed *simp*
qed

lemma *gets-the-whileLoop:*
fixes *C :: 'a \Rightarrow 's \Rightarrow bool*
shows *whileLoop C ($\lambda a. \text{gets-the } (B \ a) \ r = ((\text{gets-the } (\text{owhile } C \ B \ r))::('e::\text{default}, 'a, 's) \text{spec-monad}$)*
proof (*rule antisym*)
show (*whileLoop C ($\lambda a. \text{Spec-Monad.gets-the } (B \ a) \ r::('e, 'a, 's) \text{spec-monad}$)*
 \leq
Spec-Monad.gets-the (owhile C B r)

proof –
{

```

fix s v
assume (Some r, Some v) ∈ option-while' (λa. C a s) (λa. B a s)
then have run ((whileLoop C (λa. Spec-Monad.gets-the (B a)) r)::('e, 'a, 's)
spec-monad) s ≤ Success {(Result v, s)}
proof (induct Some r Some v arbitrary: r )
  assume ¬ C v s
  then
    show run (whileLoop C (λa. Spec-Monad.gets-the (B a)) v) s ≤ Success
    {(Result v, s)}
    by (subst whileLoop-unroll) simp
  next
    fix r r'
    assume C r s
    and B r s = Some r'
    and (Some r', Some v) ∈ option-while' (λa. C a s) (λa. B a s)
    and run ((whileLoop C (λa. Spec-Monad.gets-the (B a)) r)::('e, 'a, 's)
spec-monad) s ≤ Success {(Result v, s)}
    thus run ((whileLoop C (λa. Spec-Monad.gets-the (B a)) r)::('e, 'a, 's)
spec-monad) s ≤ Success {(Result v, s)}
    by (subst whileLoop-unroll) (simp add: run-bind)
  qed
} note * = this
show ?thesis
apply (rule le-spec-monad-runI)
apply simp
apply (clarsimp simp add: owhile-def option-while-def option-while'-THE
pure-post-state-def
split: option.splits)
by (rule *)
qed
next
show Spec-Monad.gets-the (owhile C B r) ≤ ((whileLoop C (λa. Spec-Monad.gets-the
(B a)) r)::('e, 'a, 's) spec-monad)
proof (rule le-whileLoop-succeeds-terminatesI)
  fix s
  assume run (whileLoop C (λa. Spec-Monad.gets-the (B a)) r) s ≠ ⊥
  then show run (Spec-Monad.gets-the (owhile C B r)) s
    ≤ run ((whileLoop C (λa. Spec-Monad.gets-the (B a)) r)::('e, 'a, 's)
spec-monad) s
  proof (induction rule: whileLoop-ne-top-induct)
    case (step a s) then show ?case
    apply (subst owhile-unroll)
    apply (subst whileLoop-unroll)
    apply (auto simp add: gets-the-obind gets-the-ocondition gets-the-oreturn
run-condition
run-bind runs-to-holds-def
split: option.split)
  done
qed

```


qed
qed

lemma *refines-L2-while-option*:

assumes f [*unfolded THIN-def, rule-format*]: *PROP THIN* ($\bigwedge v t. \text{refines } (b v)$
(*gets-the* ($b' v$)) $t t$ (*rel-prod rel-liftE* (=)))
shows *refines* (*L2-while* $c b i ns$) (*gets-the* (*L2-VARS* (*owhile* $c b' i ns$))) $s s$
(*rel-prod rel-liftE* (=))
unfolding *L2-while-def L2-VARS-def gets-the-whileLoop*[*symmetric*]
apply (*rule refines-whileLoop*)
using f
apply (*simp-all add: rel-liftE-def*)
done

lemma *refines-L2-fail-option*:

shows *refines* *L2-fail* (*gets-the ofail*) $s s$ (*rel-prod rel-liftE* (=))
unfolding *L2-defs*
by (*auto simp add: refines-def-old*)

lemma *refines-L2-guard-option*:

shows *refines* (*L2-guard* g) (*gets-the* (*oguard* g)) $s s$ (*rel-prod rel-liftE* (=))
unfolding *L2-defs*
by (*auto simp add: refines-def-old oguard-def*)

lemma *refines-L2-guarded*:

assumes $f: g s \implies \text{refines } f$ (*gets-the* f') $s s$ (*rel-prod rel-liftE* (=))
shows *refines* (*L2-guarded* $g f$) (*gets-the* (*oguard* $g |>>$ ($\lambda-. f'$))) $s s$ (*rel-prod*
rel-liftE (=))
unfolding *L2-defs L2-guarded-def* **using** f
by (*auto simp add: refines-def-old reaches-bind succeeds-bind obind-def oguard-def*
split: option.splits)

lemmas *refines-monad-option =*

refines-lift-reader-option [*synthesize-rule option priority:350*]
refines-L2-call-embed-option [*synthesize-rule option priority:310*]
refines-L2-gets-option [*synthesize-rule option priority:310*]

refines-L2-seq-option [*synthesize-rule option priority:310 split: g and g'*]
refines-L2-condition-option [*synthesize-rule option priority:310*]
refines-L2-try-L2-throw-option [*synthesize-rule option priority:320*]
refines-L2-try-L2-seq-option [*synthesize-rule option priority:320 split: g and g'*]
refines-L2-try-L2-condition-option [*synthesize-rule option priority:320*]

refines-L2-try-option [*synthesize-rule option priority:310*]
refines-L2-while-option [*synthesize-rule option priority:310 split: b and b'*]
refines-L2-fail-option [*synthesize-rule option priority:310*]
refines-L2-guard-option [*synthesize-rule option priority:310*]
refines-L2-guarded [*synthesize-rule option priority:310*]

23.5 Nondet Monad

Note that *L2-catch* is already replaced by *L2-try* during exception rewriting in phase L2.

lemma *refines-L2-call-embed-nondet*:

assumes *f*: *refines f f' s s (rel-prod rel-liftE (=))*
shows *refines (L2-call f emb ns) (L2-VARS f' ns) s s (rel-prod rel-liftE (=))*
using *f unfolding L2-call-def L2-VARS-def*
apply (*clarsimp simp add: refines-def-old reaches-map-value map-exn-def rel-liftE-def split: xval-splits*)
by (*metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel*)

lemma *refines-L2-call-embed-exn*:

assumes *f*: *refines f f' s s (rel-prod rel-liftE (=))*
shows *refines (L2-call f emb ns) (liftE (L2-VARS f' ns)) s s (rel-prod (rel-xval L (=)) (=))*
using *f unfolding L2-call-def L2-VARS-def*
apply (*clarsimp simp add: refines-def-old reaches-map-value map-exn-def reaches-liftE rel-liftE-def rel-xval.simps split: xval-splits*)
by (*metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel*)

lemma *refines-liftE-exn*:

assumes *f*: *refines f f' s s (rel-prod rel-liftE (=))*
shows *refines f (liftE f') s s (rel-prod (rel-xval L (=)) (=))*
using *f*
by (*fastforce simp add: refines-def-old reaches-map-value map-exn-def reaches-liftE rel-liftE-def rel-xval.simps split: xval-splits*)

lemma *refines-L2-seq-nondet-polymorphic*:

assumes *r* [*unfolded THIN-def, rule-format*]: *PROP THIN ($\bigwedge v t. \text{refines } (g v) (g' v) t t \text{ (rel-prod rel-liftE (=))}$)*
assumes *f* [*unfolded THIN-def, rule-format*]: *PROP THIN (Trueprop (refines f f' s s (rel-prod rel-liftE (=))))*
shows *refines (L2-seq f g) (bind f' g') s s (rel-prod rel-liftE (=))*
unfolding *L2-seq-def*
apply (*rule refines-bind [OF f]*)
apply (*auto simp add: r rel-liftE-def*)
done

lemma *refines-L2-seq-nondet*:

fixes *f'*:: (*'a, 's*) *res-monad*
assumes *r* [*unfolded THIN-def, rule-format*]: *PROP THIN ($\bigwedge v t. \text{refines } (g v) (g' v) t t \text{ (rel-prod rel-liftE (=))}$)*
assumes *f* [*unfolded THIN-def, rule-format*]: *PROP THIN (Trueprop (refines f f' s s (rel-prod rel-liftE (=))))*
shows *refines (L2-seq f g) ((bind f' g')::('b, 's) res-monad) s s (rel-prod rel-liftE (=))*
using *assms*

by (rule refines-L2-seq-nondet-polymorphic)

lemma *refines-L2-seq-exn*:

assumes g [*unfolded THIN-def*, *rule-format*]: *PROP THIN* ($\bigwedge v t.$ *refines* ($g v$) ($g' v$) $t t$ (*rel-prod* (*rel-xval* $L (=)$) ($=$))))
assumes f [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* $f f' s s$ (*rel-prod* (*rel-xval* $L (=)$) ($=$))))
shows *refines* (*L2-seq* $f g$) (*bind* $f' g'$) $s s$ (*rel-prod* (*rel-xval* $L (=)$) ($=$))
unfolding *L2-seq-def*
apply (*rule refines-bind* [*OF f*])
apply (*auto simp add: g rel-xval.simps default-option-def Exn-def*)
done

lemma *refines-try-bind-rel-liftE'*:

assumes $g : \bigwedge v s t. S s t \implies \text{refines } (\text{try } (g v)) (g' v) s t$ (*rel-prod rel-liftE S*)
assumes $f : \text{refines } f f' s t$ (*rel-prod rel-liftE S*)
shows *refines* (*try* (*bind* $f g$)) (*bind* $f' g'$) $s t$ (*rel-prod rel-liftE S*)
unfolding *try-def refines-map-value-left-iff*
apply (*rule refines-bind* [*OF f*])
subgoal by *auto*
subgoal by (*auto simp add: default-option-def Exn-def [symmetric] rel-liftE-def*)
subgoal by *auto*
subgoal
using g [*simplified L2-defs try-def refines-map-value-left-iff*]
by *auto*
done

lemma *refines-try-bind-rel-liftE*:

assumes $g : \bigwedge v t. \text{refines } (\text{try } (g v)) (g' v) t t$ (*rel-prod rel-liftE (=)*)
assumes $f : \text{refines } f f' s s$ (*rel-prod rel-liftE (=)*)
shows *refines* (*try* (*bind* $f g$)) (*bind* $f' g'$) $s s$ (*rel-prod rel-liftE (=)*)
apply (*rule refines-try-bind-rel-liftE'* [*OF - f*])
using g **by** *simp*

lemma *refines-L2-try-L2-seq-nondet*:

assumes g [*unfolded THIN-def L2-defs*, *rule-format*]: *PROP THIN* ($\bigwedge v t.$ *refines* (*L2-try* ($g v$)) ($g' v$) $t t$ (*rel-prod rel-liftE (=)*))
assumes f [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* $f f' s s$ (*rel-prod rel-liftE (=)*))))
shows *refines* (*L2-try* (*L2-seq* $f g$)) (*bind* $f' g'$) $s s$ (*rel-prod rel-liftE (=)*)
unfolding *L2-defs using g f*
by (*rule refines-try-bind-rel-liftE*)

lemma *refines-L2-try-L2-condition-nondet*:

assumes f [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try* f) $f' s s$ (*rel-prod rel-liftE (=)*))))
assumes g [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try* g) g'

$s s (rel\text{-}prod\ rel\text{-}liftE (=))$
shows $refines (L2\text{-}try (L2\text{-}condition\ c\ f\ g)) (condition\ c\ f'\ g') s s (rel\text{-}prod\ rel\text{-}liftE (=))$
using $f\ g\ unfolding\ L2\text{-}defs$
apply ($auto\ simp\ add: refines\text{-}def\text{-}old\ succeeds\text{-}condition\text{-}iff\ reaches\text{-}condition\text{-}iff\ reaches\text{-}try$)
done

lemma $refines\text{-}L2\text{-}try\text{-}rel\text{-}LiftE\text{-}nondet$:
assumes $f: refines\ f\ f'\ s\ s (rel\text{-}prod\ rel\text{-}liftE (=))$
shows $refines (L2\text{-}try\ f)\ f'\ s\ s (rel\text{-}prod\ rel\text{-}liftE (=))$
using $f\ unfolding\ L2\text{-}defs$
apply ($clarsimp\ simp\ add: refines\text{-}def\text{-}old\ reaches\text{-}try\ rel\text{-}liftE\text{-}def\ unnest\text{-}exn\text{-}def\ split: xval\text{-}splits$)
by ($metis\ Exn\text{-}def\ default\text{-}option\text{-}def\ exception\text{-}or\text{-}result\text{-}cases\ option.\text{exhaust}\text{-}sel$)

lemma $refines\text{-}try\text{-}finally\text{-}rel\text{-}liftE$:
assumes $f: refines\ f\ f'\ s\ s (rel\text{-}prod (rel\text{-}xval\ rel\text{-}liftE' (=)) (=))$
shows $refines (try\ f) (finally\ f') s s (rel\text{-}prod\ rel\text{-}liftE (=))$
using $f\ unfolding\ try\text{-}def\ finally\text{-}def$
apply ($rule\ refines\text{-}map\text{-}value$)
apply ($auto\ simp\ add: unnest\text{-}exn\text{-}def\ unite\text{-}def\ rel\text{-}liftE'\text{-}def\ split: xval\text{-}splits\ sum.\text{splits}$)
done

lemma $refines\text{-}L2\text{-}try\text{-}finally\text{-}nondet$:
assumes $f: refines\ f\ f'\ s\ s (rel\text{-}prod (rel\text{-}xval\ rel\text{-}liftE' (=)) (=))$
shows $refines (L2\text{-}try\ f) (finally\ f') s s (rel\text{-}prod\ rel\text{-}liftE (=))$
using $f\ unfolding\ L2\text{-}defs$
by ($rule\ refines\text{-}try\text{-}finally\text{-}rel\text{-}liftE$)

lemma $refines\text{-}L2\text{-}try\text{-}exn$:
assumes $f: refines\ f\ f'\ s\ s (rel\text{-}prod (rel\text{-}xval (rel\text{-}sum\ L (=)) (=)) (=))$
shows $refines (L2\text{-}try\ f) (try\ f') s s (rel\text{-}prod (rel\text{-}xval\ L (=)) (=))$
using $f\ unfolding\ L2\text{-}defs\ try\text{-}def$
apply ($rule\ refines\text{-}map\text{-}value$)
apply ($auto\ simp\ add: unnest\text{-}exn\text{-}def\ split: xval\text{-}splits\ sum.\text{splits}$)
done

lemma $refines\text{-}L2\text{-}condition\text{-}nondet$:
assumes $g [unfolded\ THIN\text{-}def]: PROP\ THIN (Trueprop (refines\ g\ g'\ s\ s (rel\text{-}prod\ V (=))))$
assumes $f [unfolded\ THIN\text{-}def]: PROP\ THIN (Trueprop (refines\ f\ f'\ s\ s (rel\text{-}prod\ V (=))))$
shows $refines (L2\text{-}condition\ c\ f\ g) (condition\ c\ f'\ g') s s (rel\text{-}prod\ V (=))$
unfolding $L2\text{-}condition\text{-}def$ **using** $g\ f$
apply ($auto\ intro: refines\text{-}condition$)
done

lemma *refines-L2-while-nondet*:
assumes b [*unfolded THIN-def, rule-format*]: *PROP THIN* ($\bigwedge v t. \text{refines } (b v)$
 $(b' v) t t (\text{rel-prod rel-liftE } (=))$)
shows *refines* (*L2-while* $c b i ns$) (*L2-VARS* (*whileLoop* $c b' i ns$) $s s$) (*rel-prod*
rel-liftE $(=)$)
unfolding *L2-while-def L2-VARS-def*
apply (*rule refines-whileLoop'*)
subgoal
by (*auto*)
subgoal
by (*auto intro: b*)
subgoal
by *auto*
subgoal
by (*auto simp add: rel-liftE-def rel-exception-or-result.simps*)
done

lemma *refines-L2-while-exn*:
assumes b [*unfolded THIN-def, rule-format*]: *PROP THIN* ($\bigwedge v t. \text{refines } (b v)$
 $(b' v) t t (\text{rel-prod } (\text{rel-xval } L (=)) (=))$)
shows *refines* (*L2-while* $c b i ns$) (*L2-VARS* (*whileLoop* $c b' i ns$) $s s$) (*rel-prod*
 $((\text{rel-xval } L (=)) (=))$)
unfolding *L2-while-def L2-VARS-def*
apply (*rule refines-whileLoop-exn*)
apply (*auto simp add: b*)
done

lemma *refines-L2-unknown-nondet*:
shows *refines* (*L2-unknown* ns) (*L2-VARS* (*select UNIV*) ns) $s s$ (*rel-prod*
rel-liftE $(=)$)
unfolding *L2-defs L2-VARS-def*
apply (*auto intro: refines-select*)
done

lemma *refines-L2-unknown-exn*:
shows *refines* (*L2-unknown* ns) (*L2-VARS* (*select UNIV*) ns) $s s$ (*rel-prod*
 $(\text{rel-xval } L (=)) (=)$)
unfolding *L2-defs L2-VARS-def*
apply (*auto intro: refines-select*)
done

lemma *refines-L2-modify-nondet*:
shows *refines* (*L2-modify* f) (*modify* f) $s s$ (*rel-prod rel-liftE* $(=)$)
unfolding *L2-defs*
apply (*auto intro: refines-modify*)
done

lemma *refines-L2-gets-nondet*:

shows *refines* (*L2-gets* *f ns*) (*L2-VARS* (*gets* *f*) *ns*) *s s* (*rel-prod* *rel-liftE* (=))
unfolding *L2-defs* *L2-VARS-def*
apply (*auto intro: refines-gets*)
done

lemma *refines-L2-throw-exn*:

L x x' \implies
shows *refines* (*L2-throw* *x ns*) (*L2-VARS* (*throw* *x'*) *ns*) *s s* (*rel-prod* (*rel-xval* *L* (=))
(=))
unfolding *L2-throw-def* *L2-VARS-def*
apply (*auto simp add: refines-def-old rel-xval.simps Exn-def*)
done

lemma *refines-L2-spec-nondet*:

shows *refines* (*L2-spec* *r*) (*assert-result-and-state* ($\lambda s. \{(v, t). (s, t) \in r\}$)) *s s*
(*rel-prod* *rel-liftE* (=))
unfolding *L2-defs*
apply (*auto simp add: refines-def-old reaches-bind succeeds-bind*)
done

lemma *refines-L2-assume-nondet*:

shows *refines* (*L2-assume* *r*) (*assume-result-and-state* *r*) *s s* (*rel-prod* *rel-liftE*
(=))
unfolding *L2-defs*
apply (*auto simp add: refines-def-old reaches-bind succeeds-bind*)
done

lemma *refines-L2-guard-nondet*:

shows *refines* (*L2-guard* *c*) (*guard* *c*) *s s* (*rel-prod* *rel-liftE* (=))
unfolding *L2-defs*
by (*auto intro: refines-guard*)

lemma *refines-L2-guarded-nondet*:

assumes *f: c s \implies refines f f' s s* (*rel-prod* *rel-liftE* (=))
shows *refines* (*L2-guarded* *c f*) (*bind* (*guard* *c*) ($\lambda-. f'$)) *s s* (*rel-prod* *rel-liftE*
(=))
unfolding *L2-guarded-def* *L2-seq-def* *L2-guard-def* **using** *f*
apply (*auto simp add: refines-def-old succeeds-bind reaches-bind*)
done

lemma *refines-L2-guarded-exn*:

assumes *f: c s \implies refines f f' s s* (*rel-prod* (*rel-xval* *L* (=)) (=))
shows *refines* (*L2-guarded* *c f*) (*bind* (*guard* *c*) ($\lambda-. f'$)) *s s* (*rel-prod* (*rel-xval* *L*
(=)) (=))
unfolding *L2-guarded-def* *L2-seq-def* *L2-guard-def* **using** *f*
apply (*auto simp add: refines-def-old succeeds-bind reaches-bind*)

done

lemma *refines-L2-fail-nondet*:

shows *refines L2-fail fail s s (rel-prod R (=))*

by *simp*

lemma *refines-exec-concrete-gen-nondet*:

assumes *f: $\bigwedge s. \text{refines } f f' s s \text{ (rel-prod } R \text{ (=))}$*

shows *refines (exec-concrete st f) (exec-concrete st f') s s (rel-prod R (=))*

using *f*

by (*fastforce simp add: refines-def-old reaches-exec-concrete succeeds-exec-concrete-iff*)

lemmas *refines-exec-concrete-nondet = refines-exec-concrete-gen-nondet [where R=rel-liftE]*

lemmas *refines-exec-concrete-exit = refines-exec-concrete-gen-nondet [where R=rel-xval L (=)] for L*

lemma *refines-exec-abstract-gen-nondet*:

assumes *f: $\bigwedge s. \text{refines } f f' s s \text{ (rel-prod } R \text{ (=))}$*

shows *refines (exec-abstract st f) (exec-abstract st f') s s (rel-prod R (=))*

using *f*

by (*fastforce simp add: refines-def-old reaches-exec-abstract*)

lemmas *refines-exec-abstract-nondet = refines-exec-abstract-gen-nondet [where R=rel-liftE]*

lemmas *refines-exec-abstract-exit = refines-exec-abstract-gen-nondet [where R=rel-xval L (=)] for L*

lemma *rel-map-ResultI*:

rel-map Result x (Result x)

by (*simp add: rel-map-def*)

lemma *rel-map-to-xval-InlI*:

rel-map to-xval (Inl l) (Exn l)

by (*simp add: rel-map-def*)

lemma *rel-map-to-xval-InrI*:

rel-map to-xval (Inr r) (Result r)

by (*simp add: rel-map-def*)

lemma *refines-rel-prod-eq-guard-on-exit*:

assumes *f: refines f_c f_a s s (rel-prod Q (=))*

shows *refines*

(guard-on-exit f_c grd cleanup)

(guard-on-exit f_a grd cleanup) s s

(rel-prod Q (=))

```

unfolding guard-on-exit-bind-exception-or-result-conv
apply (rule refines-bind-exception-or-result-strong [OF f])
apply clarsimp
apply (rule refines-bind-no-throw [where  $Q=rel\text{-}prod \top (=)$ ])
  apply simp
  apply simp
apply (rule refines-guard)
  apply simp
apply simp
apply (rule refines-bind-no-throw)
  apply simp
apply simp
  apply (clarsimp)
  apply (rule refines-state-select-same)
apply clarsimp
done

```

```

lemma refines-rel-prod-eq-assume-on-exit:
assumes  $f: refines\ f_c\ f_a\ s\ s\ (rel\text{-}prod\ Q\ (=))$ 
shows refines
  (assume-on-exit  $f_c\ grd\ cleanup$ )
  (assume-on-exit  $f_a\ grd\ cleanup$ )  $s\ s$ 
  (rel-prod  $Q\ (=)$ )
unfolding assume-on-exit-bind-exception-or-result-conv
apply (rule refines-bind-exception-or-result-strong [OF f])
apply clarsimp
apply (rule refines-bind-no-throw [where  $Q=rel\text{-}prod \top (=)$ ])
  apply simp
  apply simp
subgoal
  by (auto simp add: refines-def-old)
apply (rule refines-bind-no-throw)
  apply simp
apply simp
  apply (clarsimp)
  apply (rule refines-state-select-same)
apply clarsimp
done

```

```

context stack-heap-state
begin

```

```

thm refines-rel-prod-with-fresh-stack-ptr
lemma refines-rel-prod-L2-try-with-fresh-stack-ptr:
assumes  $init\text{-}eq: init_c\ s = init_a\ s$ 
assumes  $f: \bigwedge s\ p. refines\ (L2\text{-}try\ (f_c\ p))\ (f_a\ p)\ s\ s\ (rel\text{-}prod\ Q\ (=))$ 
shows
  refines
  (L2-try (with-fresh-stack-ptr  $n\ init_c\ (L2\text{-}VARS\ f_c\ nm)$ ))

```



```

      (with-fresh-stack-ptr n inita (L2-VARS fa nm)) s s
      (rel-prod Q (=))
apply (simp add: L2-try-with-fresh-stack-ptr L2-VARS-def)
apply (rule refines-rel-prod-with-fresh-stack-ptr [unfolded L2-VARS-def])
apply (rule init-eq)
apply (rule f)
done
end

context typ-heap-typing
begin

lemma refines-rel-prod-guard-with-fresh-stack-ptr:
assumes init-eq: initc s = inita s
assumes f:  $\bigwedge s p.$  refines (fc p) (fa p) s s (rel-prod L (=))
shows
  refines
    (guard-with-fresh-stack-ptr n initc (L2-VARS fc nm))
    (guard-with-fresh-stack-ptr n inita (L2-VARS fa nm)) s s
    (rel-prod L (=))
apply (simp add: guard-with-fresh-stack-ptr-def L2-VARS-def assume-stack-alloc-def)
apply (rule refines-bind')
apply (subst refines-assume-result-and-state-iff)
apply (subst init-eq)
apply (intro sim-set-refl)
apply clarsimp
apply (rule refines-on-exit')
apply (rule refines-weaken[OF f])
apply simp
apply (rule refines-bind'[OF refines-guard])
apply simp-all
apply (rule refines-assert-result-and-state)
apply auto
done

lemma refines-rel-prod-assume-with-fresh-stack-ptr:
assumes init-eq: initc s = inita s
assumes f:  $\bigwedge s p.$  refines (fc p) (fa p) s s (rel-prod L (=))
shows
  refines
    (assume-with-fresh-stack-ptr n initc (L2-VARS fc nm))
    (assume-with-fresh-stack-ptr n inita (L2-VARS fa nm)) s s
    (rel-prod L (=))
apply (simp add: assume-with-fresh-stack-ptr-def L2-VARS-def assume-stack-alloc-def)
apply (rule refines-bind')
apply (subst refines-assume-result-and-state-iff)
apply (subst init-eq)
apply (intro sim-set-refl)
apply clarsimp

```

```

apply (rule refines-on-exit')
apply (rule refines-weaken[OF f])
apply simp
apply (rule refines-bind'[OF refines-assuming])
apply simp-all
apply (rule refines-assert-result-and-state)
apply auto
done

```

```

lemma refines-rel-prod-with-fresh-stack-ptr:
assumes init-eq:  $init_c\ s = init_a\ s$ 
assumes f:  $\bigwedge s\ p.$  refines ( $f_c\ p$ ) ( $f_a\ p$ )  $s\ s$  (rel-prod L (=))
shows
  refines
    (with-fresh-stack-ptr  $n\ init_c$  (L2-VARS  $f_c\ nm$ ))
    (with-fresh-stack-ptr  $n\ init_a$  (L2-VARS  $f_a\ nm$ ))  $s\ s$ 
    (rel-prod L (=))
apply (simp add: with-fresh-stack-ptr-def L2-VARS-def assume-stack-alloc-def)
apply (rule refines-bind-no-throw )
  apply simp
  apply simp
  apply (rule refines-assume-result-and-state-same')
  apply (simp only: init-eq)
apply clarsimp
apply (rule refines-rel-prod-eq-on-exit)
apply (rule f)
done

```

```

lemma refines-rel-prod-L2-try-guard-with-fresh-stack-ptr:
assumes init-eq:  $init_c\ s = init_a\ s$ 
assumes f:  $\bigwedge s\ p.$  refines (L2-try ( $f_c\ p$ )) ( $f_a\ p$ )  $s\ s$  (rel-prod L (=))
shows
  refines
    (L2-try (guard-with-fresh-stack-ptr  $n\ init_c$  (L2-VARS  $f_c\ nm$ )))
    (guard-with-fresh-stack-ptr  $n\ init_a$  (L2-VARS  $f_a\ nm$ ))  $s\ s$ 
    (rel-prod L (=))
apply (simp add: L2-try-guard-with-fresh-stack-ptr L2-VARS-def)
apply (rule refines-rel-prod-guard-with-fresh-stack-ptr [unfolded L2-VARS-def])
  apply (rule init-eq)
apply (rule f)
done

```

```

lemma refines-rel-prod-L2-try-assume-with-fresh-stack-ptr:
assumes init-eq:  $init_c\ s = init_a\ s$ 
assumes f:  $\bigwedge s\ p.$  refines (L2-try ( $f_c\ p$ )) ( $f_a\ p$ )  $s\ s$  (rel-prod L (=))
shows
  refines
    (L2-try (assume-with-fresh-stack-ptr  $n\ init_c$  (L2-VARS  $f_c\ nm$ )))

```

```

      (assume-with-fresh-stack-ptr n inita (L2-VARS fa nm)) s s
      (rel-prod L (=))
apply (simp add: L2-try-assume-with-fresh-stack-ptr L2-VARS-def)
apply (rule refines-rel-prod-assume-with-fresh-stack-ptr [unfolded L2-VARS-def])
  apply (rule init-eq)
apply (rule f)
done

```

lemma *refines-rel-prod-L2-try-with-fresh-stack-ptr*:

```

assumes init-eq: initc s = inita s
assumes f:  $\bigwedge s p.$  refines (L2-try (fc p)) (fa p) s s (rel-prod L (=))
shows
  refines
    (L2-try (with-fresh-stack-ptr n initc (L2-VARS fc nm)))
    (with-fresh-stack-ptr n inita (L2-VARS fa nm)) s s
    (rel-prod L (=))
apply (simp add: L2-try-with-fresh-stack-ptr L2-VARS-def)
apply (rule refines-rel-prod-with-fresh-stack-ptr [unfolded L2-VARS-def])
  apply (rule init-eq)
apply (rule f)
done

```

end

lemma *relcomppI-swapped*: $s b c \implies r a b \implies (r \text{ OO } s) a c$
by (rule relcomppI)

lemmas *refines-monad-nondet* =

```

  refines-L2-call-embed-nondet [synthesize-rule nondet and exit priority:210]
  refines-L2-call-embed-exn [synthesize-rule nondet and exit priority:210]

  refines-liftE-exn [synthesize-rule nondet and exit priority:250]
  refines-L2-seq-nondet [synthesize-rule nondet and exit priority:210 split: g and
g']
  refines-L2-seq-exn [synthesize-rule nondet and exit priority:210 split: g and g']

  refines-L2-try-L2-seq-nondet [synthesize-rule nondet and exit priority:230 split:
g and g']
  refines-L2-try-L2-condition-nondet [synthesize-rule nondet and exit priority:230]
  refines-L2-try-rel-LiftE-nondet [synthesize-rule nondet and exit priority:220]
  refines-L2-try-finally-nondet [synthesize-rule nondet and exit priority:210]
  refines-L2-try-exn [synthesize-rule nondet and exit priority:210]

  refines-L2-condition-nondet [synthesize-rule nondet and exit priority:210]
  refines-L2-while-nondet [synthesize-rule nondet and exit priority:210 split: b and
b']
  refines-L2-while-exn [synthesize-rule nondet and exit priority:210 split: b and b']

```

refines-L2-unknown-nondet [*synthesize-rule nondet and exit priority:210*]
refines-L2-unknown-exn [*synthesize-rule nondet and exit priority:210*]

refines-L2-modify-nondet [*synthesize-rule nondet and exit priority:210*]

refines-L2-gets-nondet [*synthesize-rule nondet and exit priority:210*]

refines-L2-throw-exn [*synthesize-rule nondet and exit priority:210*]

refines-L2-spec-nondet [*synthesize-rule nondet and exit priority:210*]

refines-L2-assume-nondet [*synthesize-rule nondet and exit priority:210*]

refines-L2-guard-nondet [*synthesize-rule nondet and exit priority:210*]

refines-L2-guarded-nondet [*synthesize-rule nondet and exit priority:210*]
refines-L2-guarded-exn [*synthesize-rule nondet and exit priority:210*]

refines-L2-fail-nondet [*synthesize-rule nondet and exit priority:210*]

refines-exec-concrete-nondet [*synthesize-rule nondet and exit priority:210*]
refines-exec-concrete-exit [*synthesize-rule nondet and exit priority:210*]

refines-exec-abstract-nondet [*synthesize-rule nondet and exit priority:210*]
refines-exec-abstract-exit [*synthesize-rule nondet and exit priority:210*]

rel-liftE-trivial [*synthesize-rule nondet and exit priority:210*]
rel-liftE'-Inr [*synthesize-rule nondet and exit priority:210*]
rel-sum-Inl [*synthesize-rule nondet and exit priority:210*]
rel-sum-Inr [*synthesize-rule nondet and exit priority:210*]
rel-xval-Exn [*synthesize-rule nondet and exit priority:210*]
rel-xval-Result [*synthesize-rule nondet and exit priority:210*]

relcomppI-swapped [*synthesize-rule nondet and exit priority:210*]
rel-map-ResultI [*synthesize-rule nondet and exit priority:210*]
rel-map-to-xval-InlI [*synthesize-rule nondet and exit priority:210*]
rel-map-to-xval-InrI [*synthesize-rule nondet and exit priority:210*]

refl [*synthesize-rule nondet and exit priority: 210*]

context *typ-heap-typing*
begin
lemmas *refines-monad-nondet* =
refines-rel-prod-L2-try-guard-with-fresh-stack-ptr [*synthesize-rule nondet and exit priority:232*]
refines-rel-prod-L2-try-assume-with-fresh-stack-ptr [*synthesize-rule nondet and exit priority:232*]
refines-rel-prod-L2-try-with-fresh-stack-ptr [*synthesize-rule nondet and exit pri-*

```

ority:232]
  refines-rel-prod-guard-with-fresh-stack-ptr [synthesize-rule nondet and exit priority:210]
  refines-rel-prod-assume-with-fresh-stack-ptr [synthesize-rule nondet and exit priority:210]
  refines-rel-prod-with-fresh-stack-ptr [synthesize-rule nondet and exit priority:210]
  refines-rel-xval-guard-with-fresh-stack-ptr [synthesize-rule nondet and exit priority:210]
  refines-rel-xval-assume-with-fresh-stack-ptr [synthesize-rule nondet and exit priority:210]
  refines-rel-xval-with-fresh-stack-ptr [synthesize-rule nondet and exit priority:210]
end

```

```

context stack-heap-state
begin
lemmas refines-nondet-monad =
  refines-rel-prod-L2-try-with-fresh-stack-ptr [synthesize-rule nondet and exit priority:232]
  refines-rel-prod-with-fresh-stack-ptr [synthesize-rule nondet and exit priority:210]
end

```

```

add-synthesize-pattern nondet and exit where
  rel-throwE-nondet-synth = ⟨Trueprop (rel-throwE - ?x ?x')⟩ (x')

```

23.5.1 Elimination of $L2$ -try in the Error Monad

Eliminate Inner Exception

rules for elimination of $L2$ -try when the inner exception layer is not needed, i.e., $rel\text{-}sum\ (rel\text{-}throwE\ L)\ (=)$

```

lemma bind-bind-exception-or-result-conv:
  (f ≫ g) = (bind-exception-or-result f (λException e ⇒ yield (Exception e) |
Result v ⇒ g v))
  apply (rule spec-monad-eq1)
  apply (clarsimp simp add: runs-to-iff)
  apply (auto simp add: runs-to-def-old split: exception-or-result-splits)
done

```

```

lemma rel-throwE'-rel-throwE-conv: rel-throwE' L = (rel-map to-xval OO rel-throwE L)
  apply (rule ext)+
  apply (auto simp add: rel-throwE-def rel-throwE'-def rel-xval.simps relcompp.simps

    rel-map-def to-xval-def split: sum.splits)
done

```

```

lemma rel-throwE (rel-throwE' L) = (rel-throwE (rel-map to-xval OO rel-throwE L))
  apply (rule ext)+

```

apply (*auto simp add: rel-throwE-def rel-throwE'-def rel-xval.simps relcompp.simps*
rel-map-def to-xval-def split: sum.splits)
done

lemma *refines-L2-try-L2-seq-exn*:
fixes $f:: ('b + 'c), 'f, 'a$ *exn-monad*
and $g:: 'f \Rightarrow ('b + 'c), 'c, 'a$ *exn-monad*
assumes g [*unfolded THIN-def L2-defs, rule-format*]:
PROP THIN ($\bigwedge v t.$ *refines* (*L2-try* ($g\ v$)) ($g'\ v$) $t\ t$ (*rel-prod* ((*rel-xval* $L\ R$))
(=)))
assumes f [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines* $f\ f'\ s\ s$ (*rel-prod* (*rel-xval* (*rel-throwE'* L) (=))
(=))))
shows *refines* (*L2-try* (*L2-seq* $f\ g$)) (*bind* $f'\ g'$) $s\ s$ (*rel-prod* (*rel-xval* $L\ R$) (=))
unfolding *L2-try-def L2-seq-def try-nested-bind-exception-or-result-conv*
apply (*simp add: bind-bind-exception-or-result-conv*)
apply (*rule refines-bind-exception-or-result*)
apply (*rule refines-weaken [OF f]*)
apply (*auto split: xval-splits sum.splits simp: rel-xval.simps rel-prod.simps g*)
done

lemma *refines-L2-try-L2-condition-exit*:
assumes f [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try* f) f'
 $s\ s$ (*rel-prod* R (=))))
assumes g [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try* g) g'
 $s\ s$ (*rel-prod* R (=))))
shows *refines* (*L2-try* (*L2-condition* $c\ f\ g$)) (*condition* $c\ f'\ g'$) $s\ s$ (*rel-prod* R
(=))
using $f\ g$ **unfolding** *L2-defs L2-try-def*
apply (*auto simp add: refines-def-old succeeds-condition-iff reaches-condition-iff*
reaches-try)
done

lemma *refines-try-rel-xval-rel-throwE'*:
assumes *refines* $f\ f'\ s\ s$ (*rel-prod* (*rel-xval* (*rel-throwE'* A) B) S)
shows *refines* (*try* f) $f'\ s\ s$ (*rel-prod* (*rel-xval* $A\ B$) S)
unfolding *try-def refines-map-value-left-iff*
apply (*rule refines-weaken [OF assms]*)
apply (*auto simp add: unnest-exn-def rel-xval.simps split: xval-splits sum.splits*)
done

lemma *refines-L2-try-rel-sum-rel-throwE-nondet*:
assumes *refines* $f\ f'\ s\ s$ (*rel-prod* (*rel-xval* (*rel-throwE'* A) B) (=))
shows *refines* (*L2-try* f) $f'\ s\ s$ (*rel-prod* (*rel-xval* $A\ B$) (=))
using *assms* **unfolding** *L2-defs*
by (*rule refines-try-rel-xval-rel-throwE'*)

lemma *refines-try-rel-throwE*:

```

assumes refines f f' s s (rel-prod (rel-throwE (rel-throwE' L)) S)
shows refines (try f) f' s s (rel-prod (rel-throwE L) S)
unfolding try-def
apply (simp add: refines-map-value-left-iff)
apply (rule refines-weaken [OF assms])
apply (auto simp add: unnest-exn-def split: xval-splits sum.splits)
done

```

```

lemma refines-L2-try-rel-throwE-nondet:
assumes refines f f' s s (rel-prod (rel-throwE (rel-throwE' L)) (=))
shows refines (L2-try f) f' s s (rel-prod (rel-throwE L) (=))
using assms unfolding L2-defs
by (rule refines-try-rel-throwE)

```

```

lemmas ts-L2-try-inner-exception =
  rel-throwE'-Inl[synthesize-rule nondet and exit]
  rel-throwE-Exn[synthesize-rule nondet and exit]
  refines-L2-try-L2-seq-exn [synthesize-rule nondet and exit priority: 218 split:
  g and g']
  refines-L2-try-L2-condition-exit [synthesize-rule nondet and exit priority: 218]
  refines-L2-try-rel-throwE-nondet [synthesize-rule nondet and exit priority: 216]
  refines-L2-try-rel-sum-rel-throwE-nondet [synthesize-rule nondet and exit prior-
  ity: 215]
bundle del-ts-L2-try-inner-exception =
  rel-throwE'-Inl[synthesize-rule nondet and exit del]
  rel-throwE-Exn[synthesize-rule nondet and exit del]
  refines-L2-try-L2-seq-exn [synthesize-rule nondet and exit priority: 218 split:
  g and g' del]
  refines-L2-try-L2-condition-exit [synthesize-rule nondet and exit priority: 218 del]
  refines-L2-try-rel-throwE-nondet [synthesize-rule nondet and exit priority: 216
  del]
  refines-L2-try-rel-sum-rel-throwE-nondet [synthesize-rule nondet and exit prior-
  ity: 215 del]

```

```

print-synthesize-rules exit < Trueprop (refines (L2-seq - -) - - - (rel-prod rel-liftE
  (=))) >

```

Eliminate *L2-try* over exiting branches

Eliminate *L2-try* over a condition, when one branch always exits (*rel-throwE* (*rel-sum L R*))

```

lemma rel-map-to-xval-rel-xval-rel-sum-conv:
  rel-map to-xval OO rel-xval L R = rel-sum L R OO rel-map to-xval
apply (rule ext)+
apply (auto simp add: rel-sum.simps rel-xval.simps relcompp.simps
  rel-map-def to-xval-def split: sum.splits)
done

```

```

lemma refines-L2-try-L2-seq-L2-condition-exit1:

```

```

assumes  $g$  [unfolded THIN-def]:
  PROP THIN (Trueprop (refines (L2-try (L2-seq  $g$   $h$ ))  $gh'$   $s$   $s$  (rel-prod LR (=)))))
assumes  $f$  [unfolded THIN-def]:
  PROP THIN (Trueprop (refines  $f f' s s$  (rel-prod (rel-throwE (rel-map to-xval OO LR)) (=)))))
shows refines (L2-try (L2-seq (L2-condition  $c f g$ )  $h$ )) (condition  $c f' gh'$ )  $s s$  (rel-prod LR (=))
using  $g f$ 
apply (simp add: L2-defs try-def refines-map-value-left-iff refines-condition-bind-left refines-condition-false)
apply (intro impI refines-condition-true)
apply assumption
subgoal premises  $prems$ 
proof –
  have refines ( $f >>= h$ ) ( $f' \gg= return$ )  $s s$  ( $\lambda(x, s) (y, t). LR (unnest-exn x) y \wedge s = t$ )
    apply (intro refines-bind[OF  $prems(2)$ ])
    apply (auto simp: rel-throwE-def rel-map-def unnest-exn-def split: xval-splits sum.splits)
  done
then show ?thesis by simp
qed
done

```

lemma *refines-L2-try-L2-seq-L2-condition-exit2*:

```

assumes  $f$  [unfolded THIN-def]:
  PROP THIN (Trueprop (refines (L2-try (L2-seq  $f h$ ))  $fh'$   $s s$  (rel-prod LR (=)))))
assumes  $g$  [unfolded THIN-def]:
  PROP THIN (Trueprop (refines  $g g' s s$  (rel-prod (rel-throwE (rel-map to-xval OO LR)) (=)))))
shows refines (L2-try (L2-seq (L2-condition  $c f g$ )  $h$ )) (condition  $c fh' g'$ )  $s s$  (rel-prod LR (=))
using  $assms$ 
unfolding  $L2-defs$ 
apply (subst (1 2) condition-swap)
apply (erule (1) refines-L2-try-L2-seq-L2-condition-exit1[unfolded L2-defs])
done

```

lemma *refines-L2-seq-L2-condition-rel-throwE1*:

```

assumes  $g$  [unfolded THIN-def]:
  PROP THIN (Trueprop (refines (L2-seq  $g h$ )  $gh'$   $s s$  (rel-prod (rel-throwE LR)(=)))))
assumes  $f$  [unfolded THIN-def]:
  PROP THIN (Trueprop (refines  $f f' s s$  (rel-prod (rel-throwE LR) (=)))))
shows refines (L2-seq (L2-condition  $c f g$ )  $h$ ) (condition  $c f' gh'$ )  $s s$  (rel-prod (rel-throwE LR) (=))
proof cases
  assume  $c s$ 

```


from *this* *f*
show *?thesis*
unfolding *L2-defs*
by (*auto simp add: refines-def-old reaches-condition-iff succeeds-condition-iff succeeds-bind reaches-bind rel-throwE-def default-option-def Exn-def split: xval-splits exception-or-result-splits*)
(metis default-option-def the-Exception-simp exception-or-result-cases option.exhaust-sel)+
next
assume $\neg c$ *s*
then have *:
run ((condition c f g) >>= h) s = run (g >>= h) s
run (condition c f' gh') s = run gh' s
by (*auto simp add: run-condition run-bind*)

from *g* **show** *?thesis*
unfolding *refines-def-old succeeds-def reaches-def L2-defs * .*
qed

lemma *refines-L2-seq-L2-condition-rel-throwE2*:
assumes *f* [*unfolded THIN-def*]:
PROP THIN (Trueprop (refines (L2-seq f h) fh' s s (rel-prod (rel-throwE LR) (=))))
assumes *g* [*unfolded THIN-def*]:
PROP THIN (Trueprop (refines g g' s s (rel-prod (rel-throwE LR) (=))))
shows *refines (L2-seq (L2-condition c f g) h) (condition c fh' g') s s (rel-prod (rel-throwE LR) (=))*
using *assms*
unfolding *L2-defs*
by (*subst (1 2) condition-swap*)
(erule (1) refines-L2-seq-L2-condition-rel-throwE1[unfolded L2-defs])

lemma *refines-bind-rel-throwE-first*:
assumes *f*: *refines f f' s s (rel-prod (rel-throwE LR) S)*
shows *refines (f >>= g) f' s s (rel-prod (rel-throwE LR) S)*
using *f* **unfolding** *L2-defs*
by (*auto simp add: refines-def-old succeeds-bind reaches-bind rel-throwE-def default-option-def Exn-def split: xval-splits exception-or-result-splits*)
(metis default-option-def the-Exception-simp exception-or-result-cases option.exhaust-sel)+

lemma *refines-L2-seq-rel-throwE-throwE*:
assumes *f* [*unfolded THIN-def*]:
PROP THIN (Trueprop (refines f f' s s (rel-prod (rel-throwE LR) (=))))
shows *refines (L2-seq f g) f' s s (rel-prod (rel-throwE LR) (=))*

unfolding *L2-defs* **using** *f*
by (*rule refines-bind-rel-throwE-first*)

lemma *refines-L2-seq-rel-throwE-throwE1*:
assumes *g* [*unfolded THIN-def, rule-format*]:
PROP THIN ($\bigwedge v t. \text{refines } (g v) (g' v) t t \text{ (rel-prod (rel-throwE (rel-map to-xval OO rel-xval L R)) (=))}$)
assumes *f* [*unfolded THIN-def, rule-format*]:
PROP THIN (*Trueprop* (*refines f f' s s* (*rel-prod* (*rel-xval* (*rel-throwE' L*) (=)) (=))))
shows *refines* (*L2-seq f g*) (*bind f' g'*) *s s* (*rel-prod* (*rel-throwE* (*rel-map to-xval OO rel-xval L R*)) (=))
unfolding *L2-defs*
apply (*rule refines-bind-bind-ern* [*OF f*])
apply (*auto simp: g rel-throwE'-iff*)
done

lemma *refines-L2-seq-rel-throwE-liftE*:
assumes *g* [*unfolded THIN-def, rule-format*]:
PROP THIN ($\bigwedge v t. \text{refines } (g v) (g' v) t t \text{ (rel-prod (rel-throwE LR) (=))}$)
assumes *f* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines f f' s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-seq f g*) (*bind f' g'*) *s s* (*rel-prod* (*rel-throwE LR*) (=))
using *g f* **unfolding** *L2-defs*
apply (*clarsimp simp add: refines-def-old succeeds-bind reaches-bind reaches-try succeeds-condition-iff reaches-condition-iff rel-throwE-def rel-throwE'-def relcompp.simps rel-liftE-def rel-xval.simps split: xval-splits sum.splits, safe*)
apply (*metis the-Result-simp*)
apply (*smt (verit) case-exception-or-result-Result case-xval-simps(1)*)
by (*metis (no-types, lifting) case-exception-or-result-Result case-xval-simps(2)*)

lemma *refines-L2-throw-rel-throwE-Inl*:
assumes *L l l'*
shows *refines* (*L2-throw* (*Inl l ns*) (*throw* (*L2-VARS l' ns*)) *s s* (*rel-prod* (*rel-throwE* (*rel-map to-xval OO rel-xval L R*)) (=))
using *assms* **unfolding** *L2-defs L2-VARS-def*
by (*auto simp add: refines-def-old rel-throwE-def rel-xval.simps relcompp.simps rel-map-def to-xval-def Ern-def split: sum.splits*)

lemma *refines-L2-throw-rel-throwE-Inr*:
assumes *R r r'*
shows *refines* (*L2-throw* (*Inr r ns*) (*return* (*L2-VARS r' ns*)) *s s* (*rel-prod* (*rel-throwE* ((*rel-map to-xval OO rel-xval L R*))) (=))
using *assms* **unfolding** *L2-defs L2-VARS-def*
by (*auto simp add: refines-def-old rel-throwE-def rel-xval.simps relcompp.simps rel-map-def to-xval-def Ern-def split: sum.splits*)

lemma *refines-L2-throw-rel-throwE*:
assumes $R\ r$ (*Result* r)
shows *refines* ($L2$ -*throw* $r\ ns$) ($L2$ -*VARS* (*return* r) ns) $s\ s$ (*rel-prod* (*rel-throwE* R) (=))
using *assms* **unfolding** $L2$ -*defs* $L2$ -*VARS-def*
by (*auto simp add: refines-def-old rel-throwE-def rel-sum.simps relcompp.simps rel-map-def to-xval-def Exn-def*
split: sum.splits)

lemmas *ts-L2-try-condition-exit* =

refines-L2-try-L2-seq-L2-condition-exit1 [*synthesize-rule nondet and exit priority:217*]

refines-L2-try-L2-seq-L2-condition-exit2 [*synthesize-rule nondet and exit priority:217*]

refines-L2-seq-L2-condition-rel-throwE1 [*synthesize-rule nondet and exit priority:216*]

— Note that 1. this together with *refines* $?f\ ?f'\ ?s\ ?s$ (*rel-prod* (*rel-throwE* (*rel-throwE'* $?L$)) (=)) \implies *refines* ($L2$ -*try* $?f$) $?f'\ ?s\ ?s$ (*rel-prod* (*rel-throwE* $?L$) (=)) does not replace 2. \llbracket *THIN* (*refines* ($L2$ -*try* ($L2$ -*seq* $?g\ ?h$)) $?gh'\ ?s\ ?s$ (*rel-prod* $?LR$ (=))) ; *THIN* (*refines* $?f\ ?f'\ ?s\ ?s$ (*rel-prod* (*rel-throwE* (*rel-project to-xval OO* $?LR$)) (=))) $\rrbracket \implies$ *refines* ($L2$ -*try* ($L2$ -*seq* ($L2$ -*condition* $?c\ ?f\ ?g$) $?h$)) (*condition* $?c\ ?f'\ ?gh'$) $?s\ ?s$ (*rel-prod* $?LR$ (=)) and vice versa. 1 is more compositional and also works deeply nested but only handles the case *rel-throwE*, whereas 2 also handles *rel-liftE* and *rel-sum* by pushing down $L2$ -*try* into the branch of the conditional.

refines-L2-seq-L2-condition-rel-throwE2 [*synthesize-rule nondet and exit priority:216*]

refines-L2-seq-rel-throwE-throwE [*synthesize-rule nondet and exit priority:213*]

refines-L2-seq-rel-throwE-throwE1 [*synthesize-rule nondet and exit priority:212*
split: g and g']

refines-L2-seq-rel-throwE-liftE [*synthesize-rule nondet and exit priority:212*
split: g and g']

refines-L2-throw-rel-throwE-Inl [*synthesize-rule nondet and exit priority:212*]

refines-L2-throw-rel-throwE-Inr [*synthesize-rule nondet and exit priority:212*]

refines-L2-throw-rel-throwE [*synthesize-rule nondet and exit priority:212*]

bundle *del-ts-L2-try-condition-exit* =

refines-L2-try-L2-seq-L2-condition-exit1 [*synthesize-rule nondet and exit priority:217 del*]

refines-L2-try-L2-seq-L2-condition-exit2 [*synthesize-rule nondet and exit priority:217 del*]

refines-L2-seq-L2-condition-rel-throwE1 [*synthesize-rule nondet and exit priority:216 del*]

refines-L2-seq-L2-condition-rel-throwE2 [*synthesize-rule nondet and exit priority:216 del*]

refines-L2-seq-rel-throwE-throwE [*synthesize-rule nondet and exit priority:213 del*]

refines-L2-seq-rel-throwE-throwE1 [*synthesize-rule nondet and exit priority:212*]

split: g and g' del
refines-L2-seq-rel-throwE-liftE [synthesize-rule nondet and exit priority:212 split:
g and g' del
refines-L2-throw-rel-throwE-Inl [synthesize-rule nondet and exit priority:212 del]
refines-L2-throw-rel-throwE-Inr [synthesize-rule nondet and exit priority:212 del]
refines-L2-throw-rel-throwE [synthesize-rule nondet and exit priority:212 del]

23.6 Error Monad (exit)

lemma *refines-L2-call-embed-exit:*

assumes *f: refines f f' s s (rel-prod (rel-xval rel-Nonlocal (=)) (=))*
assumes *emb: $\bigwedge e'. L (emb (Nonlocal e')) (emb' e')$*
shows *refines (L2-call f emb ns) (L2-VARS (map-value (map-exn emb') f') ns)*
s s (rel-prod (rel-xval L (=)) (=))
unfolding *L2-defs L2-VARS-def L2-call-def*
apply (*rule refines-map-value [OF f]*)
using *emb*
by (*auto simp add: rel-xval.simps map-exn-def rel-Nonlocal-conv*)

lemma *refines-L2-call-embed-exit-in-out:*

assumes *emb: $\bigwedge e'. L (emb e') (emb' e')$*
assumes *f: refines f f' s s (rel-prod (rel-xval (=) (=)) (=))*
shows *refines (L2-call f emb ns) (L2-VARS (map-value (map-exn emb') f') ns)*
s s (rel-prod (rel-xval L (=)) (=))
unfolding *L2-defs L2-VARS-def L2-call-def*
apply (*rule refines-map-value [OF f]*)
using *emb*
by (*auto simp add: rel-xval.simps map-exn-def rel-Nonlocal-conv*)

lemma *refines-L2-catch-exit:*

assumes *f: refines f f' s s (rel-prod (rel-xval (=) R) (=))*
assumes *h: $\bigwedge s' v. refines (h v) (h' v) s' s' (rel-prod (rel-xval L R) (=))$*
shows *refines (L2-catch f h) (catch f' h') s s (rel-prod (rel-xval L R) (=))*
unfolding *L2-catch-def*
using *f h*
apply (*auto intro!: refines-catch*)
done

lemma *rel-xval-eq-refl: rel-xval (=) (=) x x*

by (*auto simp add: rel-xval-eq*)

lemmas *refines-monad-exit =*

refines-L2-call-embed-exit [synthesize-rule exit priority:110]
refines-L2-call-embed-exit-in-out [synthesize-rule exit priority:109 split: emb and
emb]
refines-L2-catch-exit [synthesize-rule exit priority:110 split: h and h']
rel-Nonlocal-Nonlocal [synthesize-rule exit priority:110]
rel-sum-eq [synthesize-rule exit priority:210]

```

    rel-xval-eq-refl [synthesize-rule exit priority:210]

print-synthesize-rules exit

end

theory TypeStrengthen
imports
  Refines-Spec
begin

ML-file monad-types.ML

lemma gets-the-ogets-return-conv [fun-ptr-simps]: gets-the (ogets ( $\lambda$ -. f)) = return
f
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff ogets-def)
  done

lemma gets-the-ogets-gets-conv [fun-ptr-simps]: gets-the (ogets f) = gets f
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff ogets-def)
  done

lemma gets-the-ogets: gets-the (ogets f) = gets f
  apply (simp add: gets-the-def assert-opt-def[abs-def] ogets-def gets-def)
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff)
  done

lemma gets-the-obind:
  gets-the (f |>> g) = gets-the f >>= ( $\lambda$ x. gets-the (g x))
  apply (simp add: obind-def)
  apply (rule spec-monad-eqI)
  apply (clarsimp simp add: runs-to-iff)
  apply (auto split: option.splits)
  done

lemma gets-the-oguard: gets-the (oguard P) = guard P
  apply (simp add: oguard-def)
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff)
  done

```

```

lemma gets-the-ocondition:
  gets-the (ocondition P f g) = condition P (gets-the f) (gets-the g)
apply (simp add: ocondition-def)
apply (rule spec-monad-eqI)
apply (auto simp add: runs-to-iff)
done

```

A best-effort approach to determine the simplest possible 'monad' for the final definition is implemented. We first try to define a function into the most restrictive monad and if that fails successively try more expressive monads until we finally hit the most expressive monad. In the original autocorres version this phase was based on equations and not on a simulation relation as all the other autocorres phases. With the switch to model recursive functions with a CCPO **fixed-point** instead of **function** with an explicit measure parameter this did no longer work, as equations are not *ccpo.admissible*. Fortunately simulation is admissible, so we changed this phase to *refines*, cf: `Refines_Spec.thy`. So the main purpose of this theory is to set up the available target monads by applying some meta information: `monad_types.ML`.

The correspondence equations have the format:

(*) $p = L2\text{-call-lift } p'$

where *L2-call-lift* depends on the (simpler) target monad and lifts the program p' from that simpler monad to the fully fledged monad we start with:

The program p is the definition we have from the last layer of autocorres (WA). The final definition will refer to p' .

For the code to work *L2-call-lift* has to be a distinct constant, as some matching is performed on that assumption. That is why some new definitions are introduced below.

Note that the final (most expressive) monad is characterised by the lifting function is *lift-exit-status* which merely removes the exception handling artefact from SIMPL by extracting the exception value $'a$ from $'a\ c\text{-exntype}$. So it should be sufficiently expressive for any input C program.

When the proof for a certain monad fails it can either have a good reason (as the input function is just not expressible in that particular monad) or it can fail because the implementation is missing some rules.

Note some peculiarities on the current state of affairs:

- When a guard remains (e.g. bounds for an integer) you end up at least in the option monad (to model the possible failure).
- As recursive functions are currently implemented with **fixed-point** they are at least in the option monad.

setup <

```

Monad-Types.new-monad-type
  pure
  Pure function
  (* unused ccpo-name for recursive definitions *)
  100
  #resT
  (fn - => fn - => error monad-type pure: there is no previous monad to lift from)
  {rules-name = @{synthesize-rules-name pure},
   relator = @{term rel-liftE::('a, 'b) xval => 'b val => bool},
   relator-from-c-exntype = NONE, lift = @{term return},
   dest-lift = (fn @{term-pat return ?x} => SOME x | - => NONE),
   lift-prev = []}
|> Context.theory-map
>

```

```

setup <
Monad-Types.new-monad-type
  gets (* reader monad *)
  Read-only function
  (* unused ccpo-name for recursive definitions *)
  80
  (fn {stateT, resT, exT} => stateT --> resT)
  (fn - => fn stateT => let fun lift t = Abs (-, stateT, t) in Utils.lift-result-with-arity
  0 lift end)
  {rules-name = @{synthesize-rules-name reader},
   relator = @{term rel-liftE::('a, 'b) xval => 'b val => bool},
   relator-from-c-exntype = NONE, lift = @{term gets},
   dest-lift = (fn @{term-pat gets ?x} => SOME x | - => NONE),
   lift-prev = @{thms refines-lift-pure-reader}}
|> Context.theory-map
>

```

lemma *monotone-L2-VARS* [*partial-function-mono*]:
 $monotone\ R\ X\ a \implies monotone\ R\ X\ (\lambda f. L2-VARS\ (a\ f)\ ns)$
by (*simp add: L2-VARS-def*)

lemma *monotone-ocondition* [*partial-function-mono*]:
assumes *mono-X*: $monotone\ R\ (fun\ ord\ Q)\ X$
assumes *mono-Y*: $monotone\ R\ (fun\ ord\ Q)\ Y$
shows $monotone\ R\ (fun\ ord\ Q)$
 $(\lambda f. (ocondition\ C\ (X\ f)\ (Y\ f)))$
using *mono-X mono-Y unfolding ocondition-def monotone-def fun-ord-def*

```

by auto

declare Complete-Partial-Order2.option.preorder [partial-function-mono]

lemma monotone-obind[partial-function-mono]:
  monotone R option.le-fun a  $\implies$  ( $\bigwedge x$ . monotone R option.le-fun ( $\lambda f$ . b f x))  $\implies$ 
    monotone R option.le-fun ( $\lambda f$ . obind (a f) (b f))
  unfolding monotone-def obind-def
  apply (clarsimp simp add: flat-ord-def fun-ord-def split: option.splits)
  by (metis option.sel option.simps(3))

lemma monotone-option-fun-const [partial-function-mono]:
  monotone R option.le-fun ( $\lambda f$ . c)
  by (auto simp add: monotone-def flat-ord-def fun-ord-def)

lemma option-while-eq-Some:
  option-while C B I = Some F  $\longleftrightarrow$  (Some I, Some F)  $\in$  option-while' C B
  using option-while'-THE by (force simp: option-while-def)

lemma option-while'-monotone:
  assumes B:  $\bigwedge r$ . flat-ord None (B r) (B' r)
  assumes b: (a, b)  $\in$  option-while' C B b  $\neq$  None shows (a, b)  $\in$  option-while'
    C B'
  using b
proof induction
  case (step r1 r2 s) then show ?case
    by (metis B flat-ord-def option.simps(2) option-while'.intros(3))
qed (auto intro: option-while'.intros)

lemma monotone-option-while[partial-function-mono]:
  assumes B:  $\bigwedge a$ . monotone R (flat-ord None) ( $\lambda f$ . B f a)
  shows monotone R (flat-ord None) ( $\lambda f$ . option-while C (B f) I)
proof
  fix x y assume R x y
  show option-ord (option-while C (B x) I) (option-while C (B y) I)
    unfolding flat-ord-def
  proof (intro disjCI2)
    assume option-while C (B x) I  $\neq$  None
    then obtain F where option-while C (B x) I = Some F by auto
    then have x: (Some I, Some F)  $\in$  option-while' C (B x)
      by (auto simp: option-while-eq-Some)
    have (Some I, Some F)  $\in$  option-while' C (B y)
      using B  $\langle R x y \rangle$  by (intro option-while'-monotone[OF - x]) (auto simp:
        monotone-def)
    with x show option-while C (B x) I = option-while C (B y) I
      unfolding option-while-eq-Some[symmetric] by simp
  qed
qed

```



```

lemma monotone-owhile[partial-function-mono]:
  ( $\wedge a$ . monotone  $R$  option.le-fun ( $\lambda f$ .  $B f a$ ))  $\implies$ 
    monotone  $R$  option.le-fun ( $\lambda f$ . owhile  $C (B f) I$ )
unfolding owhile-def monotone-fun-ord-apply
by (intro allI monotone-option-while) simp

```

```

setup <
  let open Mutual-CCPO-Rec in
    add-ccpo option-state-monad (fn ctxt => fn T =>
      let
        val oT = range-type T
      in
        synth-fun ctxt (domain-type T) (synth-option ctxt oT)
      end)
  |> Context.theory-map
end
>

```

```

lemma refines-lift-pure-option:
assumes  $f$ : refines  $f$  (return  $f'$ )  $s$   $s$  (rel-prod rel-liftE (=))
shows refines  $f$  (gets-the (oreturn  $f'$ ))  $s$   $s$  (rel-prod rel-liftE (=))
using  $f$ 
apply (auto simp add: refines-def-old)
done

```

```

setup <
  Monad-Types.new-monad-type
    option
    Option monad
    option-state-monad
    60
    (fn {stateT, resT, exT} =>
      stateT --> Term.map-atyps (fn T => if T = @{typ 'a} then resT else T)
    @{typ 'a option})
    (fn ctxt => fn - => let fun lift t = infer-instantiate <t = t in term <ogets t>>
      ctxt
        in Utils.lift-result-with-arity 1 lift end)
    {rules-name = @{synthesize-rules-name option},
      relator = @{term rel-liftE::('a, 'b) xval  $\Rightarrow$  'b val  $\Rightarrow$  bool},
      relator-from-c-exntype = NONE, lift = @{term gets-the},
      dest-lift = (fn @{term-pat gets-the ?x} => SOME x | - => NONE),
      lift-prev = @{thms refines-lift-pure-option refines-lift-reader-option}}
  |> Context.theory-map
>

```

```

setup <
  Monad-Types.new-monad-type
  nondet
  Nondeterministic state monad
  spec-monad-gfp
  20
  (fn {stateT, resT, exT} =>
    Term.map-atyps (fn T => if T = @{typ 'a} then resT
                          else if T = @{typ 's} then stateT else T)
    @{typ ('a, 's) res-monad})
  (fn ctxt => fn - => let fun lift t = infer-instantiate <t = t in term <gets-the
t::('a, 's) res-monad>> ctxt in Utils.lift-result-with-arity 1 lift end)
  {rules-name = @{synthesize-rules-name nondet},
    relator = @{term rel-liftE::('a, 'b) xval => 'b val => bool},
    relator-from-c-exntype = NONE, lift = @{term <λx. x>},
    dest-lift = (fn - => NONE),
    lift-prev = []}
|> Context.theory-map
>

```

```

setup <
  Monad-Types.new-monad-type
  exit
  Nondeterministic state monad with exit (default)
  spec-monad-gfp
  10
  (fn {stateT, resT, exT} =>
    Term.map-atyps (fn T => if T = @{typ 'a} then resT
                          else if T = @{typ 's} then stateT
                          else if T = @{typ 'e} then HP-TermsTypes.strip-c-exntype
                          else T)
    @{typ ('e, 'a, 's) exn-monad})
  (fn ctxt => fn - => let fun lift t = infer-instantiate <t = t in term <liftE t::
(exit-status, 'a, 's) exn-monad>> ctxt in Utils.lift-result-with-arity 0 lift end)
  {rules-name = @{synthesize-rules-name exit},
    relator = @{term <rel-xval (=) (=)>},
    relator-from-c-exntype = SOME @{term <rel-xval rel-Nonlocal (=)>}, lift =
@{term <λx. x>},
    dest-lift = (fn - => NONE),
    lift-prev = []}
|> Context.theory-map
>

```

```
lemma id-comps:  
  id o f = f  
  ((λs. s) o f) = f  
  by (simp-all add: comp-def)  
  
lemma gets-bind-ign: gets f >>= (λx. m) = m  
  apply (rule spec-monad-eqI)  
  apply (auto simp add: runs-to-iff)  
  done  
  
end
```

Chapter 24

Polishing the Final Outcome

```
theory Polish
imports HeapLift WordPolish TypeStrengthen
begin

context heap-typing-state
begin

lemma unchanged-typing-bind[unchanged-typing]:
  assumes [runs-to-vcg]:  $f \cdot s \text{ ?}\{\lambda r. \text{unchanged-typing-on } S \ s \}$   $(\bigwedge r \ s. g \ r \cdot s \text{ ?}\{\lambda r. \text{unchanged-typing-on } S \ s \})$ 
  shows  $(\text{bind } f \ g) \cdot s \text{ ?}\{\lambda r. \text{unchanged-typing-on } S \ s \}$ 
  apply (runs-to-vcg)
  using unchanged-typing-on-trans by blast

lemma unchanged-typing-while[unchanged-typing]:
  assumes  $B: \bigwedge r \ s. C \ r \ s \implies (B \ r) \cdot s \text{ ?}\{\lambda r. \text{unchanged-typing-on } S \ s \}$ 
  shows  $(\text{whileLoop } C \ B \ i) \cdot s \text{ ?}\{\lambda r. \text{unchanged-typing-on } S \ s \}$ 
  apply (rule runs-to-partial-whileLoop [where  $I = \lambda-. \text{unchanged-typing-on } S \ s$ ])
  subgoal by simp
  subgoal by simp
  subgoal by simp
  subgoal
    apply (rule runs-to-partial-weaken [OF B])
    apply (simp)
    by (rule unchanged-typing-on-trans)
  done

lemma unchanged-typing-finally[unchanged-typing]:
  assumes [runs-to-vcg]:  $f \cdot s \text{ ?}\{\lambda r. \text{unchanged-typing-on } S \ s \}$ 
  shows  $\text{finally } f \cdot s \text{ ?}\{\lambda r. \text{unchanged-typing-on } S \ s \}$ 
  by (runs-to-vcg)

lemma unchanged-typing-try[unchanged-typing]:
  assumes [runs-to-vcg]:  $f \cdot s \text{ ?}\{\lambda r. \text{unchanged-typing-on } S \ s \}$ 
```

shows $try\ f \cdot s \ ?\{\lambda r. \text{unchanged-typing-on } S\ s\}$
by (*runs-to-vcg*)

lemma *runs-to-partial-catch*[*unchanged-typing*]:
assumes [*runs-to-vcg*]: $f \cdot s \ ?\{\lambda r. \text{unchanged-typing-on } S\ s\}$
assumes [*runs-to-vcg*]: $\bigwedge r\ s. (g\ r) \cdot s \ ?\{\lambda r. \text{unchanged-typing-on } S\ s\}$
shows $(catch\ f\ g) \cdot s \ ?\{\lambda r. \text{unchanged-typing-on } S\ s\}$
by (*runs-to-vcg*) (*auto simp add: unchanged-typing-on-simps*)

lemma *runs-to-partial-bind-handle*[*unchanged-typing*]:
assumes [*runs-to-vcg*]: $f \cdot s \ ?\{\lambda r. \text{unchanged-typing-on } S\ s\}$
assumes [*runs-to-vcg*]: $\bigwedge r\ s. (g\ r) \cdot s \ ?\{\lambda r. \text{unchanged-typing-on } S\ s\}$
assumes [*runs-to-vcg*]: $\bigwedge r\ s. (h\ r) \cdot s \ ?\{\lambda r. \text{unchanged-typing-on } S\ s\}$
shows $(bind\ handle\ f\ g\ h) \cdot s \ ?\{\lambda r. \text{unchanged-typing-on } S\ s\}$
by (*runs-to-vcg*) (*auto simp add: unchanged-typing-on-simps*)

lemma *unchanged-typing-liftE*[*unchanged-typing*]:
assumes [*runs-to-vcg*]: $f \cdot s \ ?\{\lambda r. \text{unchanged-typing-on } S\ s\}$
shows $liftE\ f \cdot s \ ?\{\lambda r. \text{unchanged-typing-on } S\ s\}$
by (*runs-to-vcg*)

end

context *open-types-heap-typing-state*
begin

lemma *unchanged-typing-ptr-valid-preserved*:
 $ptr\ valid\ (heap\ typing\ t1)\ p \implies \text{unchanged-typing-on } UNIV\ t1\ t2 \implies ptr\ valid\ (heap\ typing\ t2)\ p$
by (*simp add: unchanged-typing-on-UNIV-iff*)

lemma *reaches-unchanged-typing-ptr-valid-preserved*:
 $reaches\ f\ s\ r\ t \implies ptr\ valid\ (heap\ typing\ s)\ p \implies f \cdot s \ ?\{\lambda t. \text{unchanged-typing-on } UNIV\ s\ t\} \implies ptr\ valid\ (heap\ typing\ t)\ p$
using *unchanged-typing-ptr-valid-preserved*
by (*simp add: runs-to-partial-def-old reaches-succeeds unchanged-typing-on-UNIV-iff*)

thm *unchanged-typing* [*no-vars*]
end

locale *typ-heap-simulation-open-types-stack* =
typ-heap-simulation-open-types \mathcal{T} *st* *r* *w* *t*-*hrs* *t*-*hrs*-*update* *heap-typing* *heap-typing-upd*
for
 \mathcal{T} **and**
 $st:: 's \Rightarrow 't$ **and**
 $r:: 't \Rightarrow ('a::\{xmem\text{-type}, stack\text{-type}\})\ ptr \Rightarrow 'a$ **and**
 $w:: 'a\ ptr \Rightarrow ('a \Rightarrow 'a) \Rightarrow 't \Rightarrow 't$ **and**

```

v:: 't ⇒ 'a ptr ⇒ bool and
t-hrs :: 's ⇒ heap-raw-state and
t-hrs-update:: (heap-raw-state ⇒ heap-raw-state) ⇒ 's ⇒ 's and
heap-typing :: 't ⇒ heap-typ-desc and
heap-typing-upd :: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 't ⇒ 't +
fixes S:: addr set
begin
  sublocale typ-heap-typing r w heap-typing heap-typing-upd S
  by intro-locales

lemma unchanged-typing-on-with-fresh-stack-ptr[unchanged-typing]:
  fixes f::'a ptr ⇒ ('e::default, 'b, 't) spec-monad
  assumes [runs-to-vcg]: $\bigwedge p t. (f p) \cdot t \text{ ?}\{\lambda- t'. \text{unchanged-typing-on } A t t'\}$ 
  shows (with-fresh-stack-ptr n init f) \cdot t \text{ ?}\{\lambda- t'. \text{unchanged-typing-on } A t t'\}
  unfolding with-fresh-stack-ptr-def on-exit-def stack-ptr-acquire-def assume-stack-alloc-def
  apply runs-to-vcg
  subgoal for x d vs t'
    apply (subst (asm) typing-eq-left-unchanged-typing-on [of - (heap-typing-upd
(\lambda-. d) t)])
      apply (simp add: heap-typing-fold-upd-write)
      apply (simp add: unchanged-typing-on-def stack-ptr-release-def heap-typing-fold-upd-write
stack-allocs-releases-equal-on-typing)
    done
  done

lemma unchanged-typing-on-assume-with-fresh-stack-ptr[unchanged-typing]:
  fixes f::'a ptr ⇒ ('e::default, 'b, 't) spec-monad
  assumes [runs-to-vcg]: $\bigwedge p t. (f p) \cdot t \text{ ?}\{\lambda- t'. \text{unchanged-typing-on } A t t'\}$ 
  shows (assume-with-fresh-stack-ptr n init f) \cdot t \text{ ?}\{\lambda- t'. \text{unchanged-typing-on } A
t t'\}
  unfolding assume-with-fresh-stack-ptr-def stack-ptr-acquire-def assume-stack-alloc-def
  apply runs-to-vcg
  subgoal for x d vs t'
    apply (subst (asm) typing-eq-left-unchanged-typing-on [of - (heap-typing-upd
(\lambda-. d) t)])
      apply (simp add: heap-typing-fold-upd-write)
      apply (simp add: unchanged-typing-on-def stack-ptr-release-def heap-typing-fold-upd-write
stack-allocs-releases-equal-on-typing)
    done
  done

lemma unchanged-typing-on-guard-with-fresh-stack-ptr[unchanged-typing]:
  fixes f::'a ptr ⇒ ('e::default, 'b, 't) spec-monad
  assumes [runs-to-vcg]: $\bigwedge p t. (f p) \cdot t \text{ ?}\{\lambda- t'. \text{unchanged-typing-on } A t t'\}$ 
  shows (guard-with-fresh-stack-ptr n init f) \cdot t \text{ ?}\{\lambda- t'. \text{unchanged-typing-on } A t
t'\}
  unfolding guard-with-fresh-stack-ptr-def stack-ptr-acquire-def assume-stack-alloc-def
  apply runs-to-vcg
  subgoal for x d vs t'

```

```

apply (subst (asm) typing-eq-left-unchanged-typing-on [of - (heap-typing-upd
( $\lambda$ -. d) t))
apply (simp add: heap-typing-fold-upd-write)
apply (simp add: unchanged-typing-on-def stack-ptr-release-def heap-typing-fold-upd-write
stack-allocs-releases-equal-on-typing)
done
done

```

end

```

named-theorems polish
named-theorems polish-cong

```

```

declare map-value-id [polish]

```

```

lemmas [polish] = mex-def meq-def

```

```

declare WORD-values-fold [polish]

```

```

lemmas WORD-bound-simps [polish] =
  WORD-MAX-simps
  WORD-MIN-simps
  UWORD-MAX-simps
  WORD-signed-to-unsigned
  INT-MIN-MAX-lemmas

```

```

declare singleton-iff [polish]
declare the-Nonlocal.simps [polish]
declare map-exn-id [polish]
declare prod.case [polish]

```

```

declare mem-Collect-eq [polish]

```

```

lemma L2-VARS-polish [polish]: L2-VARS x ns = x
by (simp add: L2-VARS-def)

```

```

lemmas liftE-atomic [polish] =
  liftE-unknown
  liftE-top
  liftE-bot
  liftE-fail
  liftE-throw-Exception
  liftE-return
  liftE-throw-exception-or-result
  liftE-get-state

```

liftE-set-state
liftE-select
liftE-assert
liftE-assume
liftE-gets
liftE-guard
liftE-assert-opt
liftE-gets-the
liftE-modify

lemma *map-value-map-exn-id* [*simp, polish*]:
 $map\ value\ (map\ exn\ (\lambda e'.\ e'))\ f = f$
by (*rule spec-monad-ext*) (*simp add: run-map-value*)

lemma *gets-to-return* [*polish*]:
 $gets\ (\lambda x.\ a) = return\ a$
by (*rule spec-monad-ext*) *simp*

lemma *select-UNIV-unknown* [*polish*]: *select UNIV = unknown*
by (*clarsimp simp: unknown-def*)

lemma *unknown-unit* [*polish, simp*]: (*unknown :: (unit,'s) res-monad*) = *skip*
apply (*rule spec-monad-ext*)
apply (*auto*)
done

lemma *condition-to-when* [*polish*]:
 $condition\ (\lambda s.\ C)\ A\ skip = when\ C\ A$
by (*simp add: when-def*)

lemma *condition-to-unless* [*polish*]:
 $condition\ (\lambda s.\ C)\ skip\ A = unless\ C\ A$
apply (*simp add: when-def*)
apply (*rule condition-swap*)
done

lemma *bind-skip* [*simp, polish*]:
 $(x\ >>= (\lambda -. skip)) = x$
by *simp*

lemma *skip-bind* [*simp, polish*]:
 $(skip\ >>= P) = (P\ ())$
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind*)
done

lemma *catch-skip* [*simp, polish*]: *catch skip f = skip*
by (*rule spec-monad-ext*) (*simp add: run-catch*)

lemma *ogets-to-oreturn* [*polish*]: $ogets (\lambda s. P) = oreturn P$
apply (*clarsimp simp: ogets-def oreturn-def K-def*)
done

lemma *ocondition-ret-ret* [*polish*]:
 $ocondition P (oreturn A) (oreturn B) = ogets (\lambda s. \text{if } P \text{ s then } A \text{ else } B)$
by (*auto simp: ocondition-def ogets-def*)

lemma *ocondition-gets-ret* [*polish*]:
 $ocondition P (ogets A) (oreturn B) = ogets (\lambda s. \text{if } P \text{ s then } A \text{ s else } B)$
by (*auto simp: ocondition-def ogets-def*)

lemma *ocondition-ret-gets* [*polish*]:
 $ocondition P (oreturn A) (ogets B) = ogets (\lambda s. \text{if } P \text{ s then } A \text{ else } B \text{ s})$
by (*auto simp: ocondition-def ogets-def*)

lemma *ocondition-gets-gets* [*polish*]:
 $ocondition P (ogets A) (ogets B) = ogets (\lambda s. \text{if } P \text{ s then } A \text{ s else } B \text{ s})$
by (*auto simp: ocondition-def ogets-def*)

lemma *case-prod-trivial*[: *NO-MATCH* ($g::('e::\text{default}, 'a, 's) \text{spec-monad}$) $f \implies$
 $(\lambda(x,y). f) = (\lambda-. f)$

— We avoid this rule during polish to keep brittle tuple structure (*case-prod*) and bound variable names in eg. ($\gg=$). With the *NO-MATCH* setup the simplification might still trigger e.g. on conditions of while loops

by *auto*

lemma *bind-case-prod-trivial*[*polish*]: $bind f (\lambda(x, y). g) = bind f (\lambda-. g)$
by (*rule spec-monad-ext*) (*simp add: run-bind*)

lemma *condition-to-if* [*polish*]:
 $condition (\lambda s. C) (return a) (return b) = return (\text{if } C \text{ then } a \text{ else } b)$
by (*rule spec-monad-ext*) (*simp add: run-condition*)

lemma *guard-merge-bind*:
 $guard P \gg= (\lambda-. guard Q \gg= M) = guard (P \text{ and } Q) \gg= M$
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind run-guard*)
done

lemma *guard-merge-bind'*:
 $guard P \gg= (\lambda-. guard Q \gg= M) = guard (\lambda s. P \text{ s} \wedge Q \text{ s}) \gg= M$
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind run-guard*)
done

lemma *guard-merge*:

```

guard P >>= (λ-. guard Q) = guard (P and Q)
  apply (rule spec-monad-ext)
  apply (simp add: run-bind run-guard)
  done

```

```

lemma guard-merge':
guard P >>= (λ-. guard Q) = guard (λs. P s ∧ Q s)
  apply (rule spec-monad-ext)
  apply (simp add: run-bind run-guard)
  done

```

```

lemma pred-conj-commute: (P and Q) = (Q and P)
  by (auto simp add: pred-conj-def)

```

```

lemma guard-True-skip[polish, simp]: guard (λ-. True) = skip
  apply (rule spec-monad-ext)
  apply (simp add: run-guard)
  done

```

```

lemma guard-True-bind: — subsumed by skip >>= ?P = ?P () and guard (λ-.
True) = skip
  (guard (λ-. True) >>= M) = M ()
  apply (rule spec-monad-ext)
  apply (simp add: run-bind run-guard)
  done

```

```

declare assert-simps(1) [polish]
declare assume-simps(1) [polish]

```

```

lemma simple-bind-fail [simp]:
  (guard P >>= (λ-. fail)) = fail
  (modify f >>= (λ-. fail)) = fail
  (return x >>= (λ-. fail)) = fail
  (gets f >>= (λ-. fail)) = fail
  apply (rule spec-monad-ext, simp add: run-bind run-guard)+
  done

```

```

declare condition-fail-rhs [polish]
declare condition-fail-lhs [polish]
declare simple-bind-fail [polish]
declare condition-bind-fail [polish]

```

```

lemma whileLoop-fail:
  (whileLoop C (λ-. fail) i) = (guard (λs. ¬ C i s) >>= (λ-. return i))
  apply (subst whileLoop-unroll)
  apply (rule spec-monad-ext)
  apply (simp add: run-condition run-bind run-guard)
  done

```

lemma *owhile-fail*:
 (*owhile* *C* (λ -. *ofail*) *i*) = (*oguard* (λ s. \neg *C* *i* *s*) |>> (λ -. *oreturn* *i*))
apply (*rule ext*)
apply (*subst owhile-unroll*)
apply (*clarsimp simp: obind-def oguard-def ocondition-def split: option.splits*)
done

declare *whileLoop-fail* [*polish*]
declare *owhile-fail* [*polish*]

lemma *oguard-True* [*simp, polish*]: *oguard* (λ -. *True*) = *oreturn* ()
by (*clarsimp simp: oreturn-def oguard-def K-def*)

lemma *oguard-False* [*simp, polish*]: *oguard* (λ -. *False*) = *ofail*
by (*clarsimp simp: ofail-def oguard-def K-def*)

declare *oreturn-bind* [*polish*]
declare *obind-return* [*polish*]
lemma *infinite-option-while'*: (*Some* *x*, *Some* *y*) \notin *option-while'* (λ -. *True*) *B*
apply (*rule notI*)
apply (*induct Some* *x* :: '*a* *option* *Some* *y* :: '*a* *option*
 arbitrary: *x* *y* rule: *option-while'.induct*)
apply *auto*
done

lemma *expand-guard-conj* [*polish*]:
guard (λ s. *A* *s* \wedge *B* *s*) = (*do* {*guard* (λ s. *A* *s*); *guard* (λ s. *B* *s*) })
apply (*rule spec-monad-ext*)
apply (*simp add: run-guard run-bind*)
done

lemma *oguard-K-bind-cong* [*polish-cong*]: $g = g' \implies (\bigwedge s. g' s \implies c s = c' s) \implies$
 \implies (*oguard* *g* >>= (λ -. *c*)) = (*oguard* *g'* >>= (λ -. *c'*))
apply (*rule ext*)
apply (*auto simp add: oguard-def obind-def*)
done

lemma *oguard-obind-cong*: $g = g' \implies (\bigwedge s. g' s \implies c s = c' s) \implies$
 $do \{- <- oguard g ; c\} = do \{- <- oguard g' ; c'\}$
by (*auto simp add: oguard-def obind-def*)

lemma *guard-bind-cong*:
 $g = g' \implies (\bigwedge s. g' s \implies run c s = run c' s) \implies$
 $do \{- <- guard g ;$
 $\quad c$
 $\} =$

```

do { - <- guard g';
    c'
}
apply (rule spec-monad-ext)
apply (simp add: run-bind run-guard)
done

```

lemma *modify-guard-swap*:

```

( $\bigwedge s. P (f s) = P s$ )  $\implies$ 
do {
  - <- modify f;
  guard P
} =
do {
  - <- guard P;
  modify f
}
apply (rule spec-monad-ext)
apply (simp add: run-bind run-guard)
done

```

lemma *modify-guard-bind-swap*:

```

( $\bigwedge s. P (f s) = P s$ )  $\implies$ 
do {
  - <- modify f;
  - <- guard P;
  X
} =
do {
  - <- guard P;
  - <- modify f;
  X
}
apply (rule spec-monad-ext)
apply (simp add: run-bind run-guard)
done

```

lemma *when-throw-cong*: $P = P' \implies (\bigwedge s. P' \implies \text{run } Y s = \text{run } Y' s) \implies$

```

do { - <- when ( $\neg P$ ) (throw x); Y } =
do { - <- when ( $\neg P'$ ) (throw x); Y' }
apply (rule spec-monad-ext)
apply (auto simp add: run-bind run-when)
done

```

lemma *condition-throw*:

```

condition ( $\lambda s. P$ ) (throw e) B = do { when P (throw e); B }
apply (auto simp add: spec-monad-eq-iff runs-to-iff Exn-def default-option-def)

```

done

lemma *map-value-map-lift-result-bind*[simp]:
map-value (map-exception-or-result ($\lambda x::unit. undefined$) f) (bind m g) =
bind m ($\lambda x. map-value (map-exception-or-result (\lambda x. undefined) f) (g x)$)
by (simp add: spec-monad-eq-iff runs-to-iff)

lemma *map-value-map-result-bind*[simp]:
map-value (map-exception-or-result id f) (bind m g) =
bind m ($\lambda x. map-value (map-exception-or-result id f) (g x)$)
apply (auto simp add: spec-monad-eq-iff runs-to-iff)
done

lemma *map-value-throw*[simp]: map-value f (yield x) = yield (f x)
apply (rule spec-monad-ext)
apply (simp add: run-map-value)
done

lemma *map-result-return*[simp]: map-value (map-exception-or-result id f) (return x) = return (f x)
apply (rule spec-monad-ext)
apply (simp add: run-map-value)
done

lemma *map-result-throw*[simp]: map-value (map-exception-or-result id f) (throw x) = throw x
apply (rule spec-monad-ext)
apply (simp add: run-map-value)
done

lemma *map-value-compose*[simp]: map-value f (map-value g m) = (map-value (f \circ g) m)
apply (auto simp add: runs-to-iff spec-monad-eq-iff)
done

lemma *map-result-compose*[simp]:
map-exception-or-result id f \circ map-exception-or-result id g = map-exception-or-result id (f \circ g)
by (simp add: map-exception-or-result-comp fun-eq-iff)

lemma *map-value-condition*:
map-value f (condition c g h) = condition c (map-value f g) (map-value f h)
apply (auto simp add: runs-to-iff spec-monad-eq-iff)
done

lemma *oguard-True-K-bind* [polish]: (oguard ($\lambda-. True$) >>= ($\lambda-. c$)) = c
apply (rule ext)
apply (auto simp add: oguard-def obind-def)
done

lemma *oguard-False-bind* [*polish*]: (*oguard* ($\lambda\cdot$. *False*) \gg = *c*) = *ofail*
apply (*rule ext*)
apply (*auto simp add: oguard-def obind-def*)
done

lemma *guard-False-bind* [*polish*]: (*guard* ($\lambda\cdot$. *False*) \gg = *c*) = *fail*
apply (*rule spec-monad-ext*)
apply (*simp add: run-guard run-bind*)
done

lemma *expand-oguard-conj* [*polish*]:
oguard (λs . *A s* \wedge *B s*) = (*obind* (*oguard* (λs . *A s*)) ($\lambda\cdot$. *oguard* (λs . *B s*)))
by (*rule ext*) (*clarsimp simp: oguard-def obind-def split: option.splits*)

lemma *owhile-infinite-loop* [*simp, polish*]:
owhile (λr *s*. *C*) *B x* = (*oguard* (λs . \neg *C*) $|>>$ ($\lambda\cdot$. *oreturn x*))
apply (*cases C*)
apply (*rule ext*)
apply (*clarsimp simp: owhile-def option-while-def obind-def split: option.splits*)
apply (*metis infinite-option-while' None-not-eq option-while'-THE*)
apply (*subst owhile-unroll*)
apply (*clarsimp simp: obind-def oreturn-def K-def split: option.splits*)
done

declare *obind-return* [*polish*]
declare *bind-return* [*polish*]

lemma *fail-bind* [*simp*]:
fail \gg = *f* = *fail*
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind*)
done

declare *fail-bind* [*polish*]

declare *ofail-bind* [*polish*]
declare *obind-fail* [*polish*]

declare *singleton-iff* [*polish*]
declare *when-True* [*polish*]
declare *when-False* [*polish*]

lemma *let-triv* [*polish*]: (*let* *x = y in x*) = *y*
apply (*clarsimp simp: Let-def*)
done

lemma *ucast-scast-same* [*polish, L2opt, simp*]:
ucast ((*scast x* :: ('*a*::*len*) *word*)) = (*x* :: '*a* *word*)

apply (*clarsimp simp: ucast-def scast-def*)
done

lemma *word-of-int-of-nat* [*polish, L2opt, simp*]:
 $\text{word-of-int } (\text{int } x) = \text{of-nat } x$
by (*rule of-int-of-nat-eq*)

lemma *return-if-P-1-0-bind* [*polish*]:
 $(\text{return } (\text{if } P \text{ then } 1 \text{ else } 0)) \gg = (\lambda x. Q \ x)$
 $= (\text{return } P \gg = (\lambda x. Q \ (\text{if } x \text{ then } 1 \text{ else } 0)))$
apply *simp*
done

lemma *gets-if-P-1-0-bind* [*polish*]:
 $(\text{gets } (\lambda s. \text{if } P \ s \text{ then } 1 \text{ else } 0)) \gg = (\lambda x. Q \ x)$
 $= (\text{gets } P \gg = (\lambda x. Q \ (\text{if } x \text{ then } 1 \text{ else } 0)))$
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind*)
done

lemma *if-P-1-0-neq-0* [*polish, simp*]:
 $((\text{if } P \text{ then } 1 \text{ else } (0::('a::\{\text{zero-neq-one}\})))) \neq 0 = P$
apply *simp*
done

lemma *if-P-1-0-eq-0* [*polish, simp*]:
 $((\text{if } P \text{ then } 1 \text{ else } (0::('a::\{\text{zero-neq-one}\})))) = 0 = (\neg P)$
apply *simp*
done

lemma *if-if-same-output* [*polish*]:
 $(\text{if } a \text{ then if } b \text{ then } x \text{ else } y \text{ else } y) = (\text{if } a \wedge b \text{ then } x \text{ else } y)$
 $(\text{if } a \text{ then } x \text{ else if } b \text{ then } x \text{ else } y) = (\text{if } a \vee b \text{ then } x \text{ else } y)$
by *auto*

lemma *collect-guarded-conj*[*polish*]:
 $\text{condition } C1 \ (\text{do } \{ \text{guard } G; \text{gets } (\lambda s. \text{if } C2 \ s \text{ then } 1 \text{ else } 0) \})$
 $(\text{return } 0)$
 $= \text{do } \{ \text{guard } (\lambda s. C1 \ s \longrightarrow G \ s);$
 $\text{gets } (\lambda s. \text{if } C1 \ s \wedge C2 \ s \text{ then } 1 \text{ else } 0) \}$
apply (*rule spec-monad-ext*)
apply (*simp add: run-bind run-guard run-condition*)
done

```

lemma collect-guarded-disj[polish]:
  condition C1 (return 1)
    (do {guard G; gets ( $\lambda s$ . if C2 s then 1 else 0) })
  = do {guard ( $\lambda s$ .  $\neg$  C1 s  $\longrightarrow$  G s);
        gets ( $\lambda s$ . if C1 s  $\vee$  C2 s then 1 else 0) }
apply (rule spec-monad-ext)
apply (simp add: run-bind run-guard run-condition)
done

```

```

lemmas [polish] = bind-assoc obind-assoc

```

```

declare not-not [polish]

```

```

lemma collect-then-cond-1-0[polish]:
  do {cond  $\leftarrow$  gets ( $\lambda s$ . if P s then (1::('a::{zero-neq-one})) else 0);
      condition ( $\lambda$ -. cond  $\neq$  0) L R }
  = condition P L R
apply (rule spec-monad-ext)
apply (simp add: run-bind run-guard run-condition)
done

```

```

lemma collect-then-cond-1-0-assoc[polish]:
  (do {cond  $\leftarrow$  gets ( $\lambda s$ . if P s then (1::('a::{zero-neq-one})) else 0);
      condition ( $\lambda$ -. cond  $\neq$  0) L R
       $\gg=$  f })
  = (condition P L R  $\gg=$  f)
apply (rule spec-monad-ext)
apply (simp add: run-bind run-guard run-condition)
done

```

```

lemma bind-return-bind [polish]:
  (A  $\gg=$  ( $\lambda x$ . (return x  $\gg=$  ( $\lambda y$ . B x y)))) = (A  $\gg=$  ( $\lambda x$ . B x x))
by simp

```

```

lemma obind-oreturn-obind [polish]:
  (A  $|>>$  ( $\lambda x$ . (oreturn x  $|>>$  ( $\lambda y$ . B x y)))) = (A  $|>>$  ( $\lambda x$ . B x x))
by simp

```

```

declare obind-assoc [polish]

```

```

declare if-distrib [where f=scast, polish, simp]
declare if-distrib [where f=ucast, polish, simp]
declare if-distrib [where f=unat, polish, simp]

```


declare *if-distrib* [**where** $f=uint, polish, simp$]
declare *if-distrib* [**where** $f=sint, polish, simp$]

declare *cast-simps* [*polish*]

lemma *Suc-0-eq-1* [*polish*]: $Suc\ 0 = 1$
by *simp*

lemma *bind-return-case-prod* [*polish, simp*]:
 $(do\ \{\} \leftarrow A0; return\ \{\}) = A0$
 $(do\ \{a\} \leftarrow A1; return\ (a)) = A1$
 $(do\ \{a, b\} \leftarrow A2; return\ (a, b)) = A2$
 $(do\ \{a, b, c\} \leftarrow A3; return\ (a, b, c)) = A3$
 $(do\ \{a, b, c, d\} \leftarrow A4; return\ (a, b, c, d)) = A4$
 $(do\ \{a, b, c, d, e\} \leftarrow A5; return\ (a, b, c, d, e)) = A5$
 $(do\ \{a, b, c, d, e, f\} \leftarrow A6; return\ (a, b, c, d, e, f)) = A6$
by (*auto simp: split-beta' case-unit-Unity[abs-def]*)

lemma *bind-return-case-prod'* [*polish, simp*]:
 $(A1 \gg return \gg g1) = (A1 \gg g1)$
 $(A2 \gg (\lambda(a, b). (return\ (a, b) \gg g2))) = (A2 \gg g2)$
 $(A3 \gg (\lambda(a, b, c). (return\ (a, b, c) \gg g3))) = (A3 \gg g3)$
 $(A4 \gg (\lambda(a, b, c, d). (return\ (a, b, c, d) \gg g4))) = (A4 \gg g4)$
 $(A5 \gg (\lambda(a, b, c, d, e). (return\ (a, b, c, d, e) \gg g5))) = (A5 \gg g5)$
 $(A6 \gg (\lambda(a, b, c, d, e, f). (return\ (a, b, c, d, e, f) \gg g6))) = (A6 \gg g6)$
by (*auto simp: split-beta' case-unit-Unity[abs-def]*)

lemma *obind-returnOk-prodE-case* [*polish, simp*]:
 $(A1 |>> (\lambda a. oreturn\ (a))) = A1$
 $(A2 |>> (\lambda(a, b). oreturn\ (a, b))) = A2$
 $(A3 |>> (\lambda(a, b, c). oreturn\ (a, b, c))) = A3$
 $(A4 |>> (\lambda(a, b, c, d). oreturn\ (a, b, c, d))) = A4$
 $(A5 |>> (\lambda(a, b, c, d, e). oreturn\ (a, b, c, d, e))) = A5$
 $(A6 |>> (\lambda(a, b, c, d, e, f). oreturn\ (a, b, c, d, e, f))) = A6$
by *auto*

lemma *obind-returnOk-prodE-case'* [*polish, simp*]:
 $(A1 |>> (\lambda a. (oreturn\ (a) |>> g1))) = A1 |>> g1$
 $(A2 |>> (\lambda(a, b). (oreturn\ (a, b) |>> g2))) = A2 |>> g2$
 $(A3 |>> (\lambda(a, b, c). (oreturn\ (a, b, c) |>> g3))) = A3 |>> g3$
 $(A4 |>> (\lambda(a, b, c, d). (oreturn\ (a, b, c, d) |>> g4))) = A4 |>> g4$
 $(A5 |>> (\lambda(a, b, c, d, e). (oreturn\ (a, b, c, d, e) |>> g5))) = A5 |>> g5$
 $(A6 |>> (\lambda(a, b, c, d, e, f). (oreturn\ (a, b, c, d, e, f) |>> g6))) = A6 |>> g6$
by *auto*

lemma *bind-fixup-1*:
 $(bind\ A\ (\lambda x. case\ y\ of\ (a, b) \Rightarrow B\ a\ b\ x)) =$

$(\text{case } y \text{ of } (a, b) \Rightarrow \text{bind } A (\lambda x. B a b x))$
by (*auto simp: split-def*)

lemma *bind-fixup-2*:
 $(\text{bind } (\text{case } y \text{ of } (a, b) \Rightarrow B a b) A) =$
 $(\text{case } y \text{ of } (a, b) \Rightarrow \text{bind } (B a b) A)$
by (*auto simp: split-def*)

lemma *finally-fixup*: $(\lambda x. \text{finally } (\text{case } x \text{ of } (a, b) \Rightarrow f a b)) = (\lambda(a,b). \text{finally } (f a b))$
by (*simp add: fun-eq-iff split: prod.splits*)

lemma *try-fixup*: $(\lambda x. \text{try } (\text{case } x \text{ of } (a, b) \Rightarrow f a b)) = (\lambda(a,b). \text{try } (f a b))$
by (*simp add: fun-eq-iff split: prod.splits*)

lemma *liftE-fixup*: $(\lambda x. \text{liftE } (\text{case } x \text{ of } (a, b) \Rightarrow f a b)) = (\lambda(a,b). \text{liftE } (f a b))$
by (*simp add: fun-eq-iff split: prod.splits*)

lemmas *spec-monad-split-fixup* [*polish*]= — should we even take more from *L2-seq*
 $?A (\lambda x. \text{case } ?y \text{ of } (a, b) \Rightarrow ?B a b x) = (\text{case } ?y \text{ of } (a, b) \Rightarrow \text{L2-seq } ?A (?B a b))$
 $\text{L2-seq } (\text{case } ?y \text{ of } (a, b) \Rightarrow ?B a b) ?A = (\text{case } ?y \text{ of } (a, b) \Rightarrow \text{L2-seq } (?B a b) ?A)$
 $(\text{case } (?x, ?y) \text{ of } (a, b) \Rightarrow ?P a b) = ?P ?x ?y$
 $(\lambda(a, b). ?P a) = (\lambda(a, x, y). ?P a)$
 $(\lambda x. \text{liftE } (\text{case } x \text{ of } (a, b) \Rightarrow ?f a b)) = (\lambda(a, b). \text{liftE } (?f a b))$
 $(\lambda x. \text{finally } (\text{case } x \text{ of } (a, b) \Rightarrow ?f a b)) = (\lambda(a, b). \text{finally } (?f a b))$
 $(\lambda x. \text{try } (\text{case } x \text{ of } (a, b) \Rightarrow ?f a b)) = (\lambda(a, b). \text{try } (?f a b))$
 $\text{L2-guard } (\text{case } ?y \text{ of } (a, b) \Rightarrow ?G a b) = (\text{case } ?y \text{ of } (a, b) \Rightarrow \text{L2-guard } (?G a b))$
 $\text{L2-gets } (\text{case } ?y \text{ of } (a, b) \Rightarrow ?G a b) = (\text{case } ?y \text{ of } (a, b) \Rightarrow \text{L2-gets } (?G a b))$
 $\text{L2-modify } (\text{case } ?y \text{ of } (a, b) \Rightarrow ?G a b) = (\text{case } ?y \text{ of } (a, b) \Rightarrow \text{L2-modify } (?G a b))$
 $(\lambda(a, b). \text{L2-gets } (?P a b) ?n) = (\lambda(a, x, y). \text{L2-gets } (?P a (x, y)) ?n)$
 $(\lambda(a, b). \text{L2-guard } (?P a b)) = (\lambda(a, x, y). \text{L2-guard } (?P a (x, y)))$
 $(\lambda(a, b). \text{L2-modify } (?P a b)) = (\lambda(a, x, y). \text{L2-modify } (?P a (x, y)))$
 $(\lambda(a, b). \text{L2-spec } (?P a b)) = (\lambda(a, x, y). \text{L2-spec } (?P a (x, y)))$
 $(\lambda(a, b). \text{L2-assume } (?P a b)) = (\lambda(a, x, y). \text{L2-assume } (?P a (x, y)))$
 $(\lambda(a, b). \text{L2-throw } (?P a b) ?n) = (\lambda(a, x, y). \text{L2-throw } (?P a (x, y)) ?n)$
 $(\lambda(a, b). \text{L2-seq } (?L a b) (?R a b)) = (\lambda(a, x, y). \text{L2-seq } (?L a (x, y)) (?R a (x, y)))$
 $(\lambda(a, b). \text{L2-while } (?C a b) (?B a b) (?I a b) ?n) = (\lambda(a, x, y). \text{L2-while } (?C a (x, y)) (?B a (x, y)) (?I a (x, y)) ?n)$
 $(\lambda(a, b). \text{L2-unknown } ?n) = (\lambda(a, x, y). \text{L2-unknown } ?n)$
 $(\lambda(a, b). \text{L2-catch } (?L a b) (?R a b)) = (\lambda(a, x, y). \text{L2-catch } (?L a (x, y)) (?R a (x, y)))$
 $(\lambda(a, b). \text{L2-condition } (?C a b) (?L a b) (?R a b)) = (\lambda(a, x, y). \text{L2-condition } (?C a (x, y)) (?L a (x, y)) (?R a (x, y)))$
 $(\lambda(a, b). \text{L2-call } (?M a b)) = (\lambda(a, x, y). \text{L2-call } (?M a (x, y)))$
 $(\lambda(a, b). \text{liftE } (?M a b)) = (\lambda(a, x, y). \text{liftE } (?M a (x, y)))$
 $(\lambda(a, b). \text{finally } (?M a b)) = (\lambda(a, x, y). \text{finally } (?M a (x, y)))$
 $(\lambda(a, b). \text{try } (?M a b)) = (\lambda(a, x, y). \text{try } (?M a (x, y))) ?$
bind-fixup-1

bind-fixup-2
finally-fixup
try-fixup
liftE-fixup

lemma *scast-1-simps* [*simp*, *L2opt*, *polish*]:

scast (1 :: ('a::len) bit1 word) = 1
scast (1 :: ('a::len) bit0 word) = 1
scast (1 :: ('a::len) bit1 signed word) = 1
scast (1 :: ('a::len) bit0 signed word) = 1
by *auto*

lemma *scast-1-simps-direct* [*simp*, *L2opt*, *polish*]:

scast (1 :: sword64) = (1 :: word64)
scast (1 :: sword64) = (1 :: word32)
scast (1 :: sword64) = (1 :: word16)
scast (1 :: sword64) = (1 :: word8)
scast (1 :: sword32) = (1 :: word32)
scast (1 :: sword32) = (1 :: word16)
scast (1 :: sword32) = (1 :: word8)
scast (1 :: sword16) = (1 :: word16)
scast (1 :: sword16) = (1 :: word8)
scast (1 :: sword8) = (1 :: word8)
by *auto*

declare *scast-0* [*L2opt*, *polish*]

declare *Word.sint-0* [*polish*]

declare *More-Word.sint-0* [*polish*]

lemma *sint-1-eq-1-x* [*polish*, *simp*]:

sint (1 :: (('a::len) bit0) word) = 1
sint (1 :: (('a::len) bit1) word) = 1
sint (1 :: (('a::len) bit0) signed word) = 1
sint (1 :: (('a::len) bit1) signed word) = 1
by *auto*

lemma *if-P-then-t-else-f-eq-t* [*L2opt*, *polish*]:

((if P then t else f) = t) = (P ∨ t = f)
by *auto*

lemma *if-P-then-t-else-f-eq-f* [*L2opt*, *polish*]:

((if P then t else f) = f) = (¬ P ∨ t = f)
by *auto*

lemma *sint-1-ne-sint-0*: *sint* 1 ≠ *sint* 0

by *simp*

lemmas *if-P-then-t-else-f-eq-f-simps* [*L2opt*, *polish*] =
if-P-then-t-else-f-eq-f [**where** $t = \text{ sint } 1$ **and** $f = \text{ sint } 0$, *simplified sint-1-ne-sint-0*
simp-thms]
if-P-then-t-else-f-eq-t [**where** $t = \text{ sint } 1$ **and** $f = \text{ sint } 0$, *simplified sint-1-ne-sint-0*
simp-thms]
if-P-then-t-else-f-eq-f [**where** $t = 1 :: \text{ int}$ **and** $f = 0 :: \text{ int}$, *simplified zero-neq-one-class.one-neq-zero*
simp-thms]
if-P-then-t-else-f-eq-t [**where** $t = 1 :: \text{ int}$ **and** $f = 0 :: \text{ int}$, *simplified zero-neq-one-class.one-neq-zero*
simp-thms]

lemma *borring-bind-K-bind* [*simp*, *polish*]:
(gets X >>= (λ-. M)) = M
(return Y >>= (λ-. M)) = M
apply (*rule spec-monad-ext*, *simp add: run-bind*)
done

lemma *pred-and-true-var*[*simp*]: $((\lambda-. \text{ True}) \text{ and } P) = P$
by(*simp add:pred-conj-def*)
lemma *pred-and-true*[*simp*]: $(P \text{ and } (\lambda-. \text{ True})) = P$
by(*simp add:pred-conj-def*)

declare *pred-and-true-var* [*L2opt*, *polish*]
declare *pred-and-true* [*L2opt*, *polish*]

lemmas [*polish*] = *rel-simps eq-numeral-extra*

declare *ptr-add-0-id*[*polish*]
declare *ptr-coerce.simps*[*polish*]

declare *uint-nat*[*symmetric*, *polish*]

lemma *finally-throw*: *finally (throw x) = return x*
apply (*rule spec-monad-ext*)
apply (*simp add: run-finally*)
done

lemma *finally-liftE*: *finally (liftE m) = m*
apply (*auto simp add: runs-to-iff spec-monad-eq-iff intro!: runs-to-cong-pred-only*)
done

lemma *bind-post-state-map-post-state*:
bind-post-state (map-post-state f g) h = bind-post-state g (λx. h (f x))

by (*simp flip: bind-post-state-pure-post-state2*)

lemma *map-post-state-case-exception-or-result-distrib:*

map-post-state f (case-exception-or-result g h v) =
(case-exception-or-result (λx. map-post-state f (g x)) (λx. map-post-state f (h x))
v)
by (*auto simp add: map-post-state-def split: exception-or-result-splits*)

lemma *map-post-state-case-exception-or-result-prod-distrib:*

map-post-state f
((case v of (Exception e, t) ⇒ g e t
| (Result v, t) ⇒ h v t)) =
((case v of (Exception e, t) ⇒ map-post-state f (g e t)
| (Result v, t) ⇒ map-post-state f (h v t)))
by (*auto simp add: map-post-state-def split: prod.splits exception-or-result-splits*)

lemma *finally-liftE-bind:* (*finally ((liftE L) >>= (λr. R r)) = (L >>= (λr.*
finally (R r))))

by (*auto simp add: spec-monad-eq-iff runs-to-iff intro!: runs-to-cong-pred-only*)

lemma *finally-guardE:* *finally (do { r <- guard X; Y }) = do { r <- guard X;*
finally Y }

apply (*rule spec-monad-ext*)
apply (*simp add: run-bind run-finally run-guard*)
done

lemma *finally-condition-distrib:* *finally (condition c X Y) = condition c (finally*
X) (finally Y)

apply (*rule spec-monad-ext*)
apply (*simp add: run-bind run-finally run-liftE run-guard run-condition map-post-state-bind-post-state*)
done

lemma *unite-Exception-option:*

unite (Exception (v::'a option)) = Result (the v)
apply (*cases v*)
apply (*simp add: unite-def case-xval-def default-option-def option.the-def*)
by (*metis Exn-def option.sel unite-simps(1)*)

lemma *finally-bindE-liftE-throw:* *finally (X >>= (λv. L2-VARS (throw v) ns)) =*
finally X

by (*auto simp add: spec-monad-eq-iff runs-to-iff L2-VARS-def intro!: runs-to-cong-cases*)

lemmas *finally-simps =*
finally-throw finally-liftE
finally-liftE-bind
finally-guardE

```

finally-condition-distrib
condition-fail-rhs condition-fail-lhs

bind-assoc

thm finally-simps
declare finally-simps [polish]

declare finally-bindE-liftE-throw [simplified L2-VARS-def, polish]

lemma nested-bind[polish, simp]:
  do { y <- f;
      return (g y)
    } >>= h =
  do { y <- f;
      h (g y)
    }
apply (auto simp add: spec-monad-eq-iff runs-to-iff)
done

lemma select-UNIV-bind-const[simp]: select UNIV >>= (λ-. g) = g
apply (auto simp add: spec-monad-eq-iff runs-to-iff)
done

lemma do-bind-assoc:
  ∧f fa fb. do { u <- f::(unit, -) res-monad;
                fa::(unit, -) res-monad } >>= fb = do { u <- f; fa >>=
                fb::(unit, -) res-monad }
using bind-assoc by blast

```

24.1 Support to normalise guards and array index expressions

```

method simp-guards =
  (simp
   add:
     guard-merge' guard-merge-bind' conj-commute
     array-ptr-index-field-lvalue-conv
     addressable-field-exec find-array-fields-Some
     unat-less-helper
   cong:
     HOL.conj-cong)

simproc-setup field-lvalue-unfold (⟨&(p::'a::mem-type ptr→f#g#gs)⟩) = ⟨
  let
    fun check @{term-pat ⟨&(-→?fs)⟩} = is-some (try UMM-Proofs.dest-fields fs)
  orelse raise Pattern.MATCH
  | check - = raise Pattern.MATCH

```

```

in
  fn - => fn ctxt => fn ct =>
  let
    val - = check (Thm.term-of ct)
    val {p, f, g, ...} = @{cterm-match ‹&(?p→?f#?g)›} ct
    val thm0 = Drule.infer-instantiate ctxt [((p, 0), p), ((f, 0), f), ((g, 0), g)]
      @{thm field-lvalue-cons-unfold}
    val thm1 = Utils.solve-sideconditions ctxt thm0 (
      ALLGOALS (asm-full-simp-tac ctxt))
    val - = Utils.verbose-msg 6 ctxt (fn - => field-lvalue-unfold:\n ^ Thm.string-of-thm
  ctxt thm1)
  in
    SOME thm1
  end
  handle THM - =>
    (Utils.verbose-msg 6 ctxt (fn - => field-lvalue-unfold proof failed:\n ^
string-of-cterm ctxt ct);
    NONE)
  | Pattern.MATCH => NONE
end›
declare [[simproc del: field-lvalue-unfold]] — loops with [[field-ti TYPE(?'a) ?f =
Some ?t; export-uinfo ?t = typ-uinfo-t TYPE(?'b); field-ti TYPE(?'b) ?g = Some
?k]] ==> &(PTR(?'b) &( ?p→?f)→?g) = &( ?p→?f @ ?g)

attribute-setup array-bound-mksimps = ‹
  Scan.succeed (Thm.declaration-attribute
    (fn - => (Context.map-proof (UMM-Proofs.set-array-bound-mksimps))))›

```

```

lemmas whileLoop-congs-tupled =
  whileLoop-cong
  whileLoop-cong [split-tuple C and C' and B and B' arity: 2]
  whileLoop-cong [split-tuple C and C' and B and B' arity: 3]
  whileLoop-cong [split-tuple C and C' and B and B' arity: 4]
  whileLoop-cong [split-tuple C and C' and B and B' arity: 5]
  whileLoop-cong [split-tuple C and C' and B and B' arity: 6]
  whileLoop-cong [split-tuple C and C' and B and B' arity: 7]
  whileLoop-cong [split-tuple C and C' and B and B' arity: 8]
  whileLoop-cong [split-tuple C and C' and B and B' arity: 9]
  whileLoop-cong [split-tuple C and C' and B and B' arity: 10]
  whileLoop-cong [split-tuple C and C' and B and B' arity: 11]
  whileLoop-cong [split-tuple C and C' and B and B' arity: 12]
  whileLoop-cong [split-tuple C and C' and B and B' arity: 13]

```

```

lemma finally-condition-throw-conv: finally (condition c (throw x) g) = condition
c (return x) (finally g)
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff)

```

done

lemma *finally-unless-throw-conv*: $\text{finally } (\text{do } \{ \text{unless } c \text{ (throw } x); g \}) = \text{condition } (\lambda-. \neg c) \text{ (return } x) \text{ (finally } g)$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *finally-bind-condition-throw-conv*: $\text{finally } (\text{do } \{ \text{condition } c \text{ (throw } x) \text{ skip; } h \}) = \text{condition } c \text{ (return } x) \text{ (finally } h)$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *finally-return-conv*: $\text{finally } (\text{return } x) = \text{return } x$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *finally-bind-guard-conv*: $\text{finally } (\text{do } \{ \text{guard } g; f \}) = \text{do } \{ \text{guard } g; \text{finally } f \}$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *finally-bind-modify-conv*: $\text{finally } (\text{do } \{ \text{modify } g; f \}) = \text{do } \{ \text{modify } g; \text{finally } f \}$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *finally-bind-gets-conv*: $\text{finally } ((\text{gets } g) \ggg f) = \text{do } \{ r \leftarrow \text{gets } g; \text{finally } (f r) \}$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *finally-bind-gets-the-conv*: $\text{finally } (\text{gets-the } g \ggg f) = \text{do } \{ r \leftarrow \text{gets-the } g; \text{finally } (f r) \}$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *finally-bind-unknown-conv*: $\text{finally } (\text{unknown} \ggg f) = \text{do } \{ r \leftarrow \text{unknown}; \text{finally } (f r) \}$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *finally-bind-get-state-conv*: $\text{finally } (\text{get-state} \ggg f) = \text{do } \{ r \leftarrow \text{get-state};$


```

finally (f r)}
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff)
  done

```

```

lemma finally-bind-set-state-conv: finally (set-state s  $\gg$  f) = do {r ← set-state
s; finally (f r)}
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff)
  done

```

```

lemma finally-bind-select-conv: finally (select S  $\gg$  f) = do {r ← select S; finally
(f r)}
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff)
  done

```

```

lemma finally-bind-assert-conv: finally (assert P  $\gg$  f) = do {r ← assert P;
finally (f r)}
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff)
  done

```

```

lemma finally-bind-assume-conv: finally (assume P  $\gg$  f) = do {r ← assume P;
finally (f r)}
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff)
  done

```

```

lemma finally-bind-assert-opt-conv: finally (assert-opt v  $\gg$  f) = do {r ← as-
sert-opt v; finally (f r)}
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff)
  done

```

```

lemmas finally-convs =
  finally-liftE finally-liftE-bind
  finally-condition-throw-conv
  finally-unless-throw-conv
  finally-bind-condition-throw-conv
  finally-return-conv
  finally-bind-guard-conv
  finally-bind-modify-conv
  finally-bind-gets-conv
  finally-bind-gets-the-conv
  finally-bind-unknown-conv
  finally-bind-get-state-conv
  finally-bind-set-state-conv
  finally-bind-select-conv

```

finally-bind-assert-conv
finally-bind-assume-conv
finally-bind-assert-opt-conv
finally-throw

lemma *try-bind-liftE-conv*: $(\text{try } (\text{liftE } f \ggg g)) = (\text{liftE } f \ggg (\lambda x. \text{try } (g x)))$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *try-bind-guard-conv*: $\text{try } (\text{do } \{\text{guard } g; f\}) = \text{do } \{\text{guard } g; \text{try } f\}$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff*)
done

lemma *try-unless-throw-Inr-conv*:
 $\text{try } (\text{do } \{\text{unless } c (\text{throw } (\text{Inr } x)); g\}) = \text{condition } (\lambda-. \neg c) (\text{return } x) (\text{try } g)$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff unnest-exn-def*)
done

lemma *try-unless-throw-Inl-conv*:
 $\text{try } (\text{do } \{\text{unless } c (\text{throw } (\text{Inl } x)); g\}) = \text{condition } (\lambda-. \neg c) (\text{throw } x) (\text{try } g)$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff unnest-exn-def*)
done

lemma *try-when-throw-Inr-conv*:
 $\text{try } (\text{do } \{\text{when } c (\text{throw } (\text{Inr } x)); g\}) = \text{condition } (\lambda-. c) (\text{return } x) (\text{try } g)$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff unnest-exn-def*)
done

lemma *try-when-throw-Inl-conv*:
 $\text{try } (\text{do } \{\text{when } c (\text{throw } (\text{Inl } x)); g\}) = \text{condition } (\lambda-. c) (\text{throw } x) (\text{try } g)$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff unnest-exn-def*)
done

lemma *try-bind-condition-throw-Inr-conv*: $\text{try } (\text{do } \{\text{condition } c (\text{throw } (\text{Inr } x)) \text{ skip}; h\}) = \text{condition } c (\text{return } x) (\text{try } h)$
apply (*rule spec-monad-eqI*)
apply (*auto simp add: runs-to-iff unnest-exn-def*)
done

lemma *try-bind-condition-throw-Inl-conv*: $\text{try } (\text{do } \{\text{condition } c (\text{throw } (\text{Inl } x)) \text{ skip}; h\}) = \text{condition } c (\text{throw } x) (\text{try } h)$

```

apply (rule spec-monad-eqI)
apply (auto simp add: runs-to-iff unnest-exn-def)
done

```

```

lemma try-condition-conv: try (condition c f g) = condition c (try f) (try g)
apply (rule spec-monad-eqI)
apply (auto simp add: runs-to-iff)
done

```

```

lemmas try-convs =
  try-bind-liftE-conv
  try-bind-guard-conv
  try-unless-throw-Inr-conv
  try-unless-throw-Inl-conv
  try-when-throw-Inr-conv
  try-when-throw-Inl-conv
  try-bind-condition-throw-Inr-conv
  try-bind-condition-throw-Inl-conv
  try-condition-conv

```

```

lemma runs-to-partial-case-prod : (∧ a b. (f a b) · s ?{Q}) ⇒
  (case x of (a, b) ⇒ f a b) · s ?{Q}
by (cases x) simp

```

24.2 Monad simplification with custom congruence rules

```

context open-types
begin

```

We supply a custom simplification method for spec monad expressions that gathers and propagates information from conditions and guards while descending into the term. The core motivation is to clean up repeated occurrences of $PTR-VALID('a)$ (*heap-typing* s) p . Although this is a state dependent predicate it stays invariant as long as the typing does not change. So the goal of the simplification method is to prove preservation of such predicates while descending into the term. Supplying congruence rules to the simplifier is unfortunately not enough as we want to keep control over what kind of invariants are propagated. Unfortunately the simplifier has no concept of a *congps* that are triggered when descending into the term only the *simprocs* which are triggered by the usual bottom up simplifier strategy. To work around this limitation we implement *congps* by *simprocs*:

- We block the simplifier to descend into compound terms by adding trivial congruence rules to the simplifier, like $f \gg = g \equiv f \gg = g$
- We add a simproc that then triggers on the $f \gg = g$ and manually extends the context and invokes the simplifier on the subterms.

end

definition *ADD-FACT*:: ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool **where**

ADD-FACT P s = P s

definition *PRESERVED-FACTS*:: ('e::default, 'a, 's) *spec-monad* \Rightarrow 's \Rightarrow ('e, 'a) *exception-or-result* \Rightarrow 's \Rightarrow bool **where**

PRESERVED-FACTS f s r t = (reaches f s r t)

definition *PRESERVED-FACTS-WHILE* :: ('a \Rightarrow 's \Rightarrow bool) \Rightarrow ('a \Rightarrow ('e::default, 'a, 's) *spec-monad*) \Rightarrow 'a \Rightarrow 's \Rightarrow 'a \Rightarrow 's \Rightarrow bool **where**

PRESERVED-FACTS-WHILE C B i s r t = whileLoop-unroll-reachable C B i s r t

definition *RUN-CASE-PROD* ::

('a \Rightarrow 'b \Rightarrow ('c::default, 'd, 'e) *spec-monad*) \Rightarrow 'a \times 'b \Rightarrow 'e \Rightarrow

('f \Rightarrow 'g \Rightarrow ('c, 'd, 'e) *spec-monad*) \Rightarrow 'f \times 'g \Rightarrow bool **where**

RUN-CASE-PROD f x s f' x' \longleftrightarrow run (case-prod f x) s = run (case-prod f' x')

s

lemma *ADD-FACT-D*: *ADD-FACT* P s \Longrightarrow P s

by (simp add: *ADD-FACT-def*)

lemma *RUN-CASE-PROD-I*: x \equiv x' \Longrightarrow run (case-prod f x') s \equiv run (case-prod f' x') s \Longrightarrow *RUN-CASE-PROD* f x s f' x'

by (auto simp add: *RUN-CASE-PROD-def*)

named-theorems

monad-simp-simp simplification rules to derive facts **and**

monad-simp-split-tuple-cong congruence rules to control tuple expansion

lemma *PRESERVED-FACTS-runs-to-partial*: *PRESERVED-FACTS* f s r t \Longrightarrow f \cdot s $\{Q\} \Longrightarrow$ Q r t

by (auto simp add: *runs-to-partial-def-old* *PRESERVED-FACTS-def* reaches-succeeds)

lemma *STOP-I*: x \equiv y \Longrightarrow x \equiv *STOP* y **by** (simp add: *STOP-def*)

lemma *eq-TrueD*: P \equiv *True* \Longrightarrow P

by blast

lemma *eq-FalseD*: P \equiv *False* \Longrightarrow \neg P

by blast

lemma *expand-unused-case-prod*:

($\lambda(x,y::('c * 'd)). f x$) = ($\lambda(x, y1, y2). f x$)

by auto

ML ‹

```
structure Monad-Cong-Simp =
struct
type rule = {apply-thm: thm, stop-thm: thm, split-vars: string list, name: binding}
type rules = rule Net.net

type cond-rule = {thm: thm, name: binding}
type cond-rules = cond-rule Net.net

fun rule-eq ({apply-thm = thm1, ...}:rule, {apply-thm = thm2, ...}:rule) = Thm.eq-thm-prop
(thm1, thm2)
fun cond-rule-eq ({thm = thm1, ...}:cond-rule, {thm = thm2, ...}:cond-rule) =
Thm.eq-thm-prop (thm1, thm2)

fun transfer-rule ctxt {apply-thm, stop-thm, split-vars, name}:rule =
{apply-thm = Thm.transfer' ctxt apply-thm,
 stop-thm = Thm.transfer' ctxt stop-thm,
 split-vars = split-vars, name = name}

fun merge-rules (n1, n2) =
if pointer-eq (n1, n2) then n1
else if Net.is-empty n2 then n1
else if Net.is-empty n1 then n2
else Net.merge rule-eq (n1, n2)

fun merge-cond-rules (n1, n2) =
if pointer-eq (n1, n2) then n1
else if Net.is-empty n2 then n1
else if Net.is-empty n1 then n2
else Net.merge cond-rule-eq (n1, n2)

type monad-cong-data = {rules: rules, derive-rules: cond-rules,
 stop-congs: thm list, simprocs: string list}

structure Data = Generic-Data (
type T = monad-cong-data;
val empty = {rules = Net.empty, derive-rules = Net.empty, stop-congs = [],
simprocs = []}
fun merge ({rules = rules1, derive-rules = d1, stop-congs = congs1, simprocs =
simprocs1},
{rules = rules2, derive-rules = d2, stop-congs = congs2, simprocs =
simprocs2}) =
{rules = merge-rules (rules1, rules2), derive-rules = merge-cond-rules (d1,
d2),
 stop-congs = Library.merge (Thm.eq-thm-prop) (congs1, congs2),
 simprocs = Ord-List.merge fast-string-ord (simprocs1, simprocs2)}
)
```

```

fun map-rules f = Data.map (fn {rules, derive-rules, stop-congs, simprocs}:monad-cong-data
=>
  {rules = f rules, derive-rules = derive-rules, stop-congs = stop-congs, simprocs
= simprocs})
fun map-derive-rules f = Data.map (fn {rules, derive-rules, stop-congs, simprocs}:monad-cong-data
=>
  {rules = rules, derive-rules = f derive-rules, stop-congs = stop-congs, simprocs
= simprocs})
fun map-stop-congs f = Data.map (fn {rules, derive-rules, stop-congs, simprocs}:monad-cong-data
=>
  {rules = rules, derive-rules = derive-rules, stop-congs = f stop-congs, simprocs
= simprocs})
fun map-simprocs f = Data.map (fn {rules, derive-rules, stop-congs, simprocs}:monad-cong-data
=>
  {rules = rules, derive-rules = derive-rules, stop-congs = stop-congs, simprocs =
f simprocs})

fun add-simproc p = map-simprocs (Ord-List.insert fast-string-ord p)

type fact = {pred: cterm, thms: thm list}

type proof-data = {states: cterm list, facts: fact list,
  start: (Timing.start * string) option,
  cache: thm Net.net Unsynchronized.ref }

structure Prf-Data = Proof-Data (
  type T = proof-data;
  val init = (K {states = [], facts = [], start = NONE, cache = Unsynchronized.ref
Net.empty});
)

fun map-states f = Prf-Data.map (fn {states, facts, start, cache}:proof-data =>
  {states = f states, facts = facts, start = start, cache = cache})
fun map-facts f = Prf-Data.map (fn {states, facts, start, cache}:proof-data =>
  {states = states, facts = f facts, start = start, cache = cache})
fun map-start f = Prf-Data.map (fn {states, facts, start, cache}:proof-data =>
  {states = states, facts = facts, start = f start, cache = cache})
fun map-cache f = Prf-Data.map (fn {states, facts, start, cache}:proof-data =>
  {states = states, facts = facts, start = start, cache = f cache})

val facts-of = #facts o Prf-Data.get
val states-of = #states o Prf-Data.get
val start-of = #start o Prf-Data.get
val cache-of = #cache o Prf-Data.get

val reset-cache = map-cache (fn - => Unsynchronized.ref Net.empty)

```

```

fun add-to-cache ctxt thm =
  let
    val cache-ref = cache-of ctxt
    val cache' = Net.insert-term (Thm.eq-thm-prop) (Thm.prop-of thm, thm)
  (!cache-ref)
    val - = cache-ref := cache'
    val - = Utils.verbose-msg 4 ctxt (fn - => cache add: ^ Thm.string-of-thm ctxt
thm)
  in
    ctxt
  end

fun lookup-cache ctxt ct =
  Net.match-term (!cache-of ctxt) (Thm.term-of ct)

fun start str = map-start (K (SOME (Timing.start (), str)))

fun stop str ctxt = ctxt |> map-start (Option.mapPartial (fn (start, str0) =>
  let
    val - = Utils.timing-msg' 3 ctxt (fn - => str0 ^ -> ^ str) start
  in
    NONE
  end))

fun monad-simp-active ctxt = not (null (states-of ctxt))

val lhs-of = Thm.dest-equals-lhs o Thm.cconcl-of
val rhs-of = Thm.dest-equals-rhs o Thm.cconcl-of

val lhs-of-rule = #apply-thm #> lhs-of

fun cong-lhs rule =
  let
    val lhs = lhs-of rule
    val {f, ...} = @{cterm-match (fo) run ?f ?s} lhs
  in
    f
  end

fun stop-cong rule =
  Thm.reflexive (cong-lhs rule)

fun mk-rule ctxt name split-vars thm =
  let
    val eq = Simpdata.mk-meta-cong ctxt thm
    val apply-thm = eq |> Drule.zero-var-indexes
    val lhs = lhs-of apply-thm
  end

```

```

    val stop-thm = stop-cong apply-thm
    val rule = {apply-thm = Thm.trim-context apply-thm, stop-thm = Thm.trim-context
stop-thm, split-vars = split-vars, name = name}
    in
      (Thm.term-of lhs, rule)
    end

fun mk-cond-rule name thm =
  let
    val prem = Thm.major-prem-of thm
    val rule = {thm = Thm.trim-context thm, name = name}
  in
    (prem, rule)
  end

fun compare-congs (thm1, thm2) =
  let
    val maxidx1 = Thm.maxidx-of thm1
    val lhs1 = lhs-of thm1
    val lhs2 = lhs-of thm2 |> Thm.incr-indexes-cterm (maxidx1 + 1)
    val result =
      if is-some (try Thm.match (lhs1, lhs2)) then
        SOME GREATER
      else if is-some (try Thm.match (lhs2, lhs1)) then
        SOME LESS
      else NONE
  in
    result
  end

fun equiv-congs (thm1, thm2) =
  compare-congs (thm1, thm2) = SOME GREATER andalso compare-congs (thm2,
thm1) = SOME GREATER

fun more-general thm1 thm2 =
  (case compare-congs (thm1, thm2) of
    SOME GREATER => SOME thm1
  | SOME LESS => SOME thm2
  | NONE => NONE)

fun more-specific thm1 thm2 =
  (case compare-congs (thm1, thm2) of
    SOME GREATER => SOME thm2
  | SOME LESS => SOME thm1
  | NONE => NONE)

fun insert-cong thm [] = [Thm.trim-context thm]
  | insert-cong thm (thms as (thm1::thms')) =
    (case compare-congs (thm1, thm) of

```



```

    SOME GREATER => thms
  | SOME LESS => insert-cong thm thms'
  | NONE => thm1 :: insert-cong thm thms')

fun rules-of ctxt = Data.get (Context.Proof ctxt)

fun build-congs stop-thms =
  [] |> fold (insert-cong) stop-thms

fun rebuild-stop-congs context =
  let
    val stop-thms = Data.get context |> #rules |> Net.entries |> map (Thm.transfer''
context o #stop-thm)
    val congs = build-congs stop-thms |> map Thm.trim-context
  in context |> (map-stop-congs (K congs)) end

val rebuild-stop-congs' = Context.proof-map rebuild-stop-congs

fun add-rule split-vars thm context =
  let
    val concl = Thm.concl-of thm
    val concl-vars = Term.add-vars concl [] |> map (#1 o #1)
    val unused = filter-out (member (op =) concl-vars) split-vars
    val - = null unused orelse error (split variables do not occur in conclusion: ^
@{make-string} unused)
    val ctxt = Context.proof-of context
    val name = Utils.guess-binding-of-thm ctxt thm
    val rule = mk-rule ctxt name split-vars thm
    val cong = #stop-thm (snd rule) |> Thm.transfer' ctxt
  in
    context
  |> map-rules (Net.insert-term-safe rule-eq rule)
  |> map-stop-congs (insert-cong cong)
  end

fun add-rule' split-vars thm =
  Context.proof-map (add-rule split-vars thm)

fun del-rule thm context =
  let
    val ctxt = Context.proof-of context
  in
    context
  |> map-rules (Net.delete-term-safe rule-eq (mk-rule ctxt Binding.empty [] thm))
  |> rebuild-stop-congs
  end

fun del-rule' thm = Context.proof-map (del-rule thm)

```

```

fun add-derive-rule thm0 context =
  let
    val ctxt = Context.proof-of context
    val thm = Simplifier.simplify ctxt thm0
    val name = Utils.guess-binding-of-thm ctxt thm
    val rule = mk-cond-rule name thm
  in
    context |> map-derive-rules (Net.insert-term-safe (* (K true) *) cond-rule-eq
rule)
  end

fun del-derive-rule thm0 context =
  let
    val ctxt = Context.proof-of context
    val thm = Simplifier.simplify ctxt thm0
  in
    context
    |> map-derive-rules (Net.delete-term-safe cond-rule-eq (mk-cond-rule Bind-
ing.empty thm))
  end

fun add-global-stop-cong thm =
  map-stop-congs (fn congs => (build-congs (thm :: congs)))

fun del-global-stop-cong thm =
  map-stop-congs (fn congs => (build-congs (remove Thm.eq-thm-prop thm congs)))

fun pretty-binding b = Pretty.str (More-Binding.here b)
fun pretty-entry label p = Pretty.block [(Pretty.str (suffix : label)), p]

fun pretty-rule ctxt (rule:rule) =
  Pretty.item (
    [pretty-entry rule (pretty-binding (#name rule)), Pretty.brk 1,
    Pretty.indent 2 (Thm.pretty-thm ctxt (#apply-thm rule) |> Pretty.cartouche),
    Pretty.brk 1])

fun pretty-rules ctxt (rules: rule list) = Pretty.big-list (rules) (map (pretty-rule ctxt)
rules)

fun print-rules context =
  Data.get context |> #rules |> Net.entries
  |> pretty-rules (Context.proof-of context)
  |> Pretty.string-of |> writeln

fun pretty-cond-rule ctxt (rule:cond-rule) =
  Pretty.item (
    [pretty-entry cond-rule (pretty-binding (#name rule)), Pretty.brk 1,
    Pretty.indent 2 (Thm.pretty-thm ctxt (#thm rule) |> Pretty.cartouche), Pretty.brk
1])

```

```

fun pretty-derive-rules ctxt (rules: cond-rule list) =
  Pretty.big-list (derive rules) (map (pretty-cond-rule ctxt) rules)

fun print-derive-rules context =
  Data.get context |> #derive-rules |> Net.entries
  |> pretty-derive-rules (Context.proof-of context)
  |> Pretty.string-of |> writeln

fun pretty-stop-cong ctxt thm =
  Pretty.item [Thm.pretty-thm ctxt thm]

fun pretty-stop-congs ctxt thms = Pretty.big-list (stop-congs) (map (pretty-stop-cong
  ctxt) thms)

fun print-stop-congs context =
  Data.get context |> #stop-congs
  |> pretty-stop-congs (Context.proof-of context)
  |> Pretty.string-of |> writeln

fun map-theory-rules f thy =
  let
    val ctxt' = f (Proof-Context.init-global thy);
    val thy' = Proof-Context.theory-of ctxt';
  in
    Context.theory-map (Data.map (K (rules-of ctxt'))) thy'
  end

fun mk-fact thm =
  let
    val {P, ...} = @ {cterm-match < Trueprop (ADD-FACT ?P ?s)} (Thm.concl-of
    thm)
  in
    {pred = P, thms = [@ {thm ADD-FACT-D} OF [thm]]}:fact
  end

fun derive-facts ctxt thm =
  let
    val derive-rules = Net.match-term (#derive-rules (Data.get (Context.Proof
    ctxt))) (Thm.concl-of thm)
    val derived = map-filter (try (fn rule => rule OF [thm])) (map (Thm.transfer'
    ctxt o #thm) derive-rules)
  in
    map mk-fact derived
  end

fun get-more-specific [] = NONE
  | get-more-specific [rule] = SOME rule
  | get-more-specific (rule1::rules) =

```

```

let
  fun get-more-specific' (lhs0, rule0) [] = SOME rule0
  | get-more-specific' (lhs0, rule0) (rule1::rules) =
    let
      val lhs1 = lhs-of-rule rule1
    in
      if is-some (try Thm.match (lhs0, lhs1)) then
        get-more-specific' (lhs1, rule1) rules
      else
        get-more-specific' (lhs0, rule0) rules
      end
    in
      get-more-specific' (lhs-of-rule rule1, rule1) rules
    end

fun matching-rule ctxt trm =
  let
    val rules = Data.get (Context.Proof ctxt) |> #rules
    val matches = Net.match-term rules trm
  in get-more-specific matches |> Option.map (transfer-rule ctxt) end

fun lookup-by-name tvars x =
  let
    val xs = Vars.dest tvars |> map (apfst fst)
  in
    AList.lookup (op =) xs (x, 0)
  end

fun lhs-concl-conv cv thm =
  let
    val n = Thm.nprems-of thm
  in
    Conv.fconv-rule
      (Conv.concl-conv n
       (Conv.arg1-conv cv)) thm
  end

(* Plain Conv.rewr-conv did not work in our setting. The matching of lhs and ct
fails.
   But Simplifier.simplify would work.
   So Conv.rewr-conv seems to do too much for our exact match but not as much
as simplifier.
   (Maybe some kind of weired eta-conversion stuff?)
   But as we have an exact match anyway there is no need to match again.
*)
fun rewr-exact-conv eq ct =
  let
    val eq2 =
      if Thm.lhs-of eq aconvc ct then eq

```

```

else
  let val ceq = Thm.dest-fun2 (Thm.cprop-of eq)
      val eq1 = Thm.trivial (Thm.mk-binop ceq ct (Thm.rhs-of eq))
      in eq COMP eq1 end;
in Thm.transitive eq2 (Thm.beta-conversion true (Thm.rhs-of eq2)) end;

fun inst-split-rule ctxt ({apply-thm, split-vars, ...}:rule) ct =
  let
    val lhs-rule = lhs-of apply-thm
    val env as (Tvars, tvars) = Thm.match (lhs-rule, ct)
  in
    if null split-vars then
      (Thm.instantiate env apply-thm, 0, [])
    else
      let
        val orig-var = hd split-vars
        val bdy = the (lookup-by-name tvars orig-var)
        val domT = Thm.typ-of-cterm bdy |> domain-type
        val arity = domT |> HOLogic.flatten-tupleT |> length
        val names = Tuple-Tools.strip-case-prod' (Thm.term-of bdy)
        val split-ctxt = Proof-Context.init-global (Proof-Context.theory-of ctxt)
        val (no-split, splitted-thm) =
          if length names = 1 andalso arity > 1 then
            (true, apply-thm) (* corner case of trivial <math>\lambda (::('a * ...)). f > *)
          else
            (false, Tuple-Tools.split-rule split-ctxt split-vars apply-thm arity)

        val tvars-eta = Vars.map (fn - => Tuple-Tools.eta-expand-tuple ctxt) tvars
        val eta-inst-rule = Thm.instantiate (Tvars, tvars-eta) apply-thm
        val lhs-eta-inst-rule = lhs-of eta-inst-rule
        val (inst-rule, names') =
          if arity <= 1 then (eta-inst-rule, names)
          else
            let
              val env1 = Thm.match (lhs-of splitted-thm, lhs-eta-inst-rule)
              val inst-thm1 = Thm.instantiate env1 splitted-thm
              val inst-thm2 =
                if Utils.cterm-eq (lhs-eta-inst-rule, ct) then inst-thm1
              else
                let
                  val - = warning (eta expanding case prod from:\n ^ string-of-cterm
                                ctxt ct ^ to:\n ^
                                string-of-cterm ctxt lhs-eta-inst-rule)
                  val eta-eq-prop = infer-instantiate <math>\langle \text{eta-lhs} = \text{lhs-eta-inst-rule} >
                and orig-lhs = ct
                in cterm <math>\langle \text{eta-lhs} \equiv \text{orig-lhs} > > ctxt
              val eta-eq = Goal.prove-internal ctxt [] eta-eq-prop
              (fn - => asm-full-simp-tac ((Simplifier.clear-simpset ctxt)
addsimps

```

```

      @{thms case-prod-eta bind-case-prod-trivial case-prod-beta
        case-prod-beta' prod.sel prod.collapse} 1)
    val obj = Thm.term-of (Thm.lhs-of eta-eq)
    val pat = Thm.term-of (Utils.clhs-of (Thm.cconcl-of inst-thm1))
    val inst-thm' = lhs-concl-conv (rewr-exact-conv eta-eq) inst-thm1
    (* Preserve bound names *)
    val inst-thm'' = Thm.rename-boundvars pat obj inst-thm'
  in inst-thm'' end
  val bdy' = the (lookup-by-name tvars-eta orig-var)
  val names' = if no-split then names else Tuple-Tools.strip-case-prod'
(Thm.term-of bdy')
  in
    (inst-thm2, names')
  end

  val names'' = if arity = 1 andalso null names' then
    [(r, domT)] (* make up name for eta-contracted corner case*)
  else names'
  in
    (inst-rule, arity, names'')
  end
end

end

fun strip-all-fix names ct ctxt =
  let
    val n = length names
    val (bounds, -) = Utils.strip-all-open [] (Thm.term-of ct) |> apfst rev
    val - = if n <> length bounds then raise CTERM (strip-all-fix: unequal length:
  ^ @ {make-string} (names, bounds), [ct]) else ()
    val vars = names ^^ (map snd (take n bounds))
    val (fixes, ctxt') = Utils.fix-variant-cfrees vars ctxt
    val thm = Goal.prove-internal ctxt [] ct (fn - => ALLGOALS (Skip-Proof.cheat-tac
  ctxt))
    val thm' = thm |> fold Thm.forall-elim fixes
    val ct' = Thm.cprop-of thm'
  in
    ((fixes, ct'), ctxt')
  end

fun fix-import-state-and-names ctxt names ct =
  let
    val states = Prf-Data.get ctxt |> #states
    val nstates = length states
    val ((fixes, ct'), ctxt') = strip-all-fix ((s ^ string-of-int nstates)::names) ct ctxt
  in ((fixes, ct'), ctxt') end

fun match-preserved-facts ctxt names ct =
  let
    val ((fixes, ct'), ctxt') = fix-import-state-and-names ctxt names ct
  end

```

```

    handle CTERM - => raise Match
    val {f, s, r, t, lhs, rhs, ... } = @ {cterm-match <PRESERVED-FACTS ?f ?s ?r
?t ==> ?lhs == ?rhs>} ct'
    val {g', ... } = @ {cterm-match <run ?g' ?s>} rhs
  in
    {f = f, s = s, t = t, r = r, results = tl fixes, lhs = lhs, g' = g', ctxt' = ctxt'}
  end

```

```

fun match-preserved-facts-while ctxt names ct =
  let
    val ((fixes, ct'), ctxt') = fix-import-state-and-names ctxt names ct
    handle CTERM - => raise Match
    val {C, B, i, s, r, t, C-lhs, C-rhs, B-lhs, B-rhs, ... } =
      @ {cterm-match <PRESERVED-FACTS-WHILE ?C ?B ?i ?s ?r ?t ==>
        ((?C-lhs == ?C-rhs) &&& (?C-rhs ==> (?B-lhs == ?B-rhs)))>} ct'
    val C' = fst (Utils.strip-comb-cterm C-rhs)
    val {B', ... } = @ {cterm-match <run ?B' ?t>} B-rhs
  in
    {C = C, C' = C', B = B, B' = B', i = i, s = s, r = r, t = t, results = tl
  fixes,
    C-lhs = C-lhs, B-lhs = B-lhs, ctxt' = ctxt'}
  end

```

```

fun rule-by-tactic-schematic ctxt thm tac =
  Utils.check-solve-sideconditions (K true) ctxt tac thm

```

```

fun rhs-conv conv = Conv.fconv-rule (Conv.arg-conv conv)
fun STOP-conv ctxt eq =
  let
    in
      Utils.timeit-msg 4 ctxt (fn - => STOP-conv: )
      (fn - => rhs-conv (Conv.try-conv (Conv.rewr-conv @ {thm STOP-def})) eq)
    end

```

```

fun rewr-conv rules ct =
  let
    val thms = Net.match-term rules (Thm.term-of ct)
  in
    Conv.rewrs-conv thms ct
  end

```

```

fun mk-rewr-conv eqs =
  let
    fun prep eq =
      let val eq' = safe-mk-meta-eq eq
        in (Utils.rhs-of-eq (Thm.prop-of eq'), eq') end
  in
    val rules = Net.empty |> fold (Net.insert-term (Thm.eq-thm) o prep) eqs
  end

```

```

in
  rewr-conv rules
end

fun dest-bool-eq eq =
  (try (fn eq => (@{thm eq-TrueD} OF [eq])) eq |> the-list) @
  (try (fn eq => (@{thm eq-FalseD} OF [eq])) eq |> the-list)

val dest-bool-eqs = maps dest-bool-eq

val is-iarith = Match-Cterm.switch [
  @{cterm-match Trueprop ((?m::'a::linordered-idom) < ?n) } #> (fn - => true),
  @{cterm-match Trueprop ((?m::'a::linordered-idom) ≤ ?n) } #> (fn - => true),
  @{cterm-match Trueprop ((?m::'a::linordered-idom) = ?n) } #> (fn - => true),
  @{cterm-match Trueprop (¬ (?m::'a::linordered-idom) < ?n) } #> (fn - =>
true),
  @{cterm-match Trueprop (¬ (?m::'a::linordered-idom) ≤ ?n) } #> (fn - =>
true),
  @{cterm-match Trueprop (¬ (?m::'a::linordered-idom) = ?n) } #> (fn - =>
true),
  fn - => false
]

val iariths-of-eqs = dest-bool-eqs #> filter (is-iarith o Thm.cprop-of)

fun add-ariths thms ctxt =
  let
    val - = Utils.verbose-msg 5 ctxt (fn - => (adding ariths:\n ^ string-of-thms
ctxt thms))
  in
    ctxt |> Context.proof-map (
      fold (Named-Theorems.add-thm @{named-theorems arith}) thms)
  end

fun dest-abs-fresh (n, x) f =
  snd (Thm.dest-abs-fresh n f) handle CTERM - => Thm.apply f x

fun strip-case-prod-cterm ctxt ct =
  let
    val args = Tuple-Tools.strip-case-prod' (Thm.term-of ct)
    val (fixes, ctxt1) = Utils.fix-variant-cfrees args ctxt
    val orig-names = map fst args
    val names = map (fst o dest-Free o Thm.term-of) fixes
    val named-fixes = names ~~~ fixes
    fun strip [x, y] ct =
      let
        val {f, ...} = @{cterm-match (fo) case-prod ?f} ct
        in dest-abs-fresh y (dest-abs-fresh x f) end
      | strip [x] ct = dest-abs-fresh x ct
  end

```



```

| strip [] ct = ct
| strip (x::xs) ct =
  let
    val {f, ...} = @{cterm-match (fo) case-prod ?f} ct
    val ct' = dest-abs-fresh x f
  in strip xs ct' end
val bdy = strip named-fixes ct
in
  ((names ~~ fixes, orig-names, bdy), ctxt1)
end

fun case-prod-bdy-conv ctxt ct =
  let
    val ((fixes, orig-names, bdy), ctxt') = strip-case-prod-cterm ctxt ct
    val bdy-eq = Simplifier.rewrite ctxt' bdy
    val lhs = bdy-eq |> Thm.lhs-of
    val rhs = bdy-eq |> Thm.rhs-of
    val fixes' = map2 (fn (-, x) => fn n => (n, x)) fixes orig-names
    val f = Utils.lambdas-tupled fixes' lhs
    val g = Utils.lambdas-tupled fixes' rhs
    val eq-prop = infer-instantiate <f = f and g = g in cprop f ≡ g> ctxt
    val_simps = if (Thm.term-of f) aconv (Thm.term-of g) then [] else (Proof-Context.export
  ctxt' ctxt [bdy-eq])
    val simp-ctxt = Simplifier.clear-simpset ctxt addsimps (@{thms fun-eq-iff} @
 _simps)
    val eq = Goal.prove-internal simp-ctxt [] eq-prop
      (fn - => simp-tac simp-ctxt 1)
  in
    eq
  end

val case-prod-cong = @{lemma x ≡ x' => case-prod f ≡ g => case-prod f x = g
  x' for x x' f g by auto }
fun case-prod-apply-conv ctxt = Match-Cterm.switch [
  @{cterm-match case-prod ?f ?x} #> (fn {f, x, ct-, ...} =>
  let
    val x-eq = Simplifier.rewrite ctxt x
    val cp = infer-instantiate <f = f in cterm <case-prod f>> ctxt
    val cp-eq = case-prod-bdy-conv ctxt cp
  in case-prod-cong OF [x-eq, cp-eq] end)]

(* e.g. avoid substitution with <n = global-variable s> *)
fun state-dependent-rhs ctxt thm =
  case states-of ctxt of
  [] => false
| (s::-) =>
  let val s' = Thm.term-of s
  in exists-subterm (fn t => t = s') (Utils.rhs-of (Thm.concl-of thm)) end

```

```

fun gen-mksimps do-simp ctxt thm =
  let
    val thm' = if do-simp then Simplifier.asm-full-simplify ctxt thm else thm
  in
    thm' |> Simplifier.mksimps ctxt |> filter-out ( Uutils.trivial-meta-eq-thm orf
(state-dependent-rhs ctxt)) (* avoid loops *)
  end

val mksimps = gen-mksimps false

fun params-of prem = prem |> Thm.term-of |> Uutils.strip-all-open [] |> fst

fun timeit-export str inner outer thms =
  Uutils.timeit-msg 3 outer (fn - => export ^ str) (fn - => Proof-Context.export
inner outer thms)

fun monad-state-conv (rule as {split-vars, ...}) ctxt ct =
  let
    val - = Uutils.verbose-msg 4 ctxt (fn - => (rule:\n ^ Thm.string-of-thm ctxt
(#apply-thm rule)))

    val (inst-rule, arity, names) = Uutils.timeit-msg 3 ctxt (fn - => inst-split-rule:
)
    (fn - => inst-split-rule ctxt rule ct)
    val - = Uutils.verbose-msg 7 ctxt (fn - => (inst-rule:\n ^ Thm.string-of-thm
ctxt inst-rule))

    fun simp-rule rule =
      let
        in
          (case Thm.cprems-of rule of
            [] => rule
          | (prem:-) =>
            let val - = Uutils.verbose-msg 7 ctxt (fn - => prem: ^ string-of-cterm ctxt
prem)
              in prem
                |> Match-Cterm.switch [
                  @{cterm-match (fo) Trueprop (RUN-CASE-PROD ?f ?x ?s ?f' ?x')} #>
                (fn {f, x, s, f', x',...} =>
                  let
                    val ctxt = stop RUN-CASE-PROD ctxt
                    val x-eq = Uutils.timeit-msg 4 ctxt (fn - => eq (0) x: )
                    (fn - => Simplifier.asm-full-rewrite ctxt x)
                    val - = Uutils.verbose-msg 5 ctxt (fn - => (eq (0) x:\n ^
Thm.string-of-thm ctxt x-eq))
                    val start0 = Timing.start ();
                    val x'-inst = rhs-of x-eq

```

```

val f1 = infer-instantiate ⟨f = f in cterm ⟨case-prod f⟩⟩ ctxt
val ((fixes, orig-names, f-bdy), ctxt') = strip-case-prod-cterm ctxt f1
val run-f = infer-instantiate ⟨f = f-bdy and s = s in cterm⟨run f
s⟩⟩ ctxt'

val - = Utils.timing-msg' 3 ctxt (fn - => case-prod (0)) start0

val run-f-eq = Utils.timeit-msg 4 ctxt (fn - => eq (1) x: )
(fn - => Simplifier.asm-full-rewrite (start eq (1) x ctxt') run-f |>
STOP-conv ctxt')

val - = Utils.verbose-msg 5 ctxt (fn - => (eq (1) f:\n ^
Thm.string-of-thm ctxt run-f-eq))
val start1 = Timing.start ();
val {f = f1', ...} = @ {cterm-match ⟨run ?f ?s⟩} (rhs-of run-f-eq)
val fixes' = map2 (fn (-, x) => fn n => (n, x)) fixes orig-names
val f-tupled = Utils.lambdas-tupled fixes' f-bdy
val f'-tupled = Utils.lambdas-tupled fixes' f1'
val eq-prop = infer-instantiate ⟨f = f-tupled and f' = f'-tupled and
x = x'-inst and s = s in
cprop ⟨run (f x) s ≡ run (f' x) s⟩⟩ ctxt
val_simps = if Utils.trivial-meta-eq-thm run-f-eq then [] else (timeit-export
RUN-CASE-PROD ctxt' ctxt [run-f-eq])
@ @ {thms Spec-Monad.run-case-prod-distrib}
val simp-ctxt = Simplifier.clear-simpset ctxt add_simps (@ {thms
fun-eq-iff} @_simps)
val eq = Goal.prove-internal simp-ctxt [] eq-prop
(fn - => simp-tac simp-ctxt 1)
val thm = @ {thm RUN-CASE-PROD-I} OF [x-eq, eq]
val rule1 = rule OF [thm]
val - = Utils.timing-msg' 3 ctxt (fn - => case-prod (1)) start1
in simp-rule rule1 end),
@ {cterm-match (fo) ?lhs ≡ run ?f ?s} #> (fn {lhs, f, s, ...} =>
let
val ctxt = stop lhs = run f s ctxt
val fN = fst (dest-Var (Thm.term-of f))
val eq = Utils.timeit-msg 4 ctxt (fn - => eq (1): )
(fn - => Simplifier.asm-full-rewrite (start eq (1) ctxt) lhs |>
STOP-conv ctxt)
val - = Utils.verbose-msg 5 ctxt (fn - => (eq (1):\n ^ Thm.string-of-thm
ctxt eq))

val start0 = Timing.start ();
val {f', ...} = @ {cterm-match ⟨run ?f' ?s⟩} (rhs-of eq)
val rule1 = Drule.infer-instantiate ctxt [(fN, f')] rule
val rule2 = rule1 OF [eq]
val - = Utils.timing-msg' 3 ctxt (fn - => lhs = run f s (0)) start0
in simp-rule rule2 end),
@ {cterm-match (fo) ?lhs ≡ ?f ?s} #> (fn {lhs, f, s, ct-,...} =>
let
val ctxt = stop lhs = f s ctxt
val fN = fst (dest-Var (Thm.term-of f))

```

```

    val eq = Utils.timeit-msg 4 ctxt (fn - => eq (2): )
      (fn - => Simplifier.asm-full-rewrite (start eq (2) ctxt) lhs)
    val - = Utils.verbose-msg 5 ctxt (fn - => (eq (2):\n ^ Thm.string-of-thm
    ctxt eq))

    val start0 = Timing.start ();
    val rhs-eq = rhs-of eq
    val f' = Thm.lambda-name (s, s) rhs-eq
    val rule1 = Drule.infer-instantiate ctxt [(fN, f')] rule
    val rule2 = rule1 OF [eq]
    val - = Utils.timing-msg' 3 ctxt (fn - => lhs = f s (0)) start0
    in simp-rule rule2 end),
  @{cterm-match (fo) ?lhs ≡ ?f} #> (fn {lhs, f,...} =>
  let
    val ctxt = stop lhs = f ctxt
    val fN = fst (dest-Var (Thm.term-of f))
    val eq = Utils.timeit-msg 4 ctxt (fn - => eq (3): )
      (fn - => Simplifier.asm-full-rewrite (start eq (3) ctxt) lhs)
    val - = Utils.verbose-msg 5 ctxt (fn - => (eq (3):\n ^ Thm.string-of-thm
    ctxt eq))

    val start0 = Timing.start ();
    val rhs-eq = rhs-of eq
    val f' = rhs-eq
    val rule1 = Drule.infer-instantiate ctxt [(fN, f')] rule
    val rule2 = rule1 OF [eq]
    val - = Utils.timing-msg' 3 ctxt (fn - => lhs = f (0)) start0
    in simp-rule (rule2) end),
  @{cterm-match (fo) ADD-FACT ?P ?s ==> ?lhs ≡ run ?f ?s} #> (fn
  {P, s, lhs, f, ...} =>
  let
    val ctxt = stop ADD-FACT ctxt
    val start0 = Timing.start ();
    val fN = fst (dest-Var (Thm.term-of f))
    val fact-prop = infer-instantiate ⟨P = P and s = s in cprop
  ⟨ADD-FACT P s⟩⟩ ctxt
    val ([fact], ctxt1) = Assumption.add-assumes [fact-prop] ctxt
    val fact0 = Utils.apply-beta-conv P s
      |> Utils.Trueprop-cterm
    val fact1 = Goal.prove-internal ctxt1 [] fact0 (fn - =>
      Method.insert-tac ctxt1 [fact] 1 THEN
      asm-full-simp-tac (ctxt1 addsimps @{thms ADD-FACT-def}) 1)
    val fact-eqs = gen-mksimps true ctxt1 fact1
    val fact-ariths = iariths-of-eqs fact-eqs
    val ctxt2 = (ctxt1 |> map-facts (cons {pred = P, thms = fact-eqs}))
      addsimps fact-eqs
      |> add-ariths fact-ariths
      |> Simplifier.add-prems fact-eqs
    val - = Utils.timing-msg' 3 ctxt (fn - => ADD-FACT (0)) start0
    val - = Utils.verbose-msg 4 ctxt (fn - => (adding:\n ^ string-of-thms
    ctxt1 fact-eqs))

```

```

    val eq = Utils.timeit-msg 4 ctxt (fn => eq (4):)
      (fn => Simplifier.asm-full-rewrite (start eq (4) ctxt2) lhs
        |> STOP-conv ctxt2 |> singleton (timeit-export (eq (4):) ctxt2
    ctxt))
  val - = Utils.verbose-msg 5 ctxt (fn => (eq (4):\n ^ Thm.string-of-thm
    ctxt eq))

    val start1 = Timing.start ();
    val {f', ...} = @ {cterm-match <run ?f' ?s> } (rhs-of eq)
    val rule1 = Utils.timeit-msg 4 ctxt (fn => ADD-FACT rule1: )
      (fn => Drule.infer-instantiate ctxt [(fN, f')] rule)
    val rule2 = Utils.timeit-msg 4 ctxt (fn => ADD-FACT rule2: )
      (fn => eq COMP rule1)
    val - = Utils.timing-msg' 3 ctxt (fn => ADD-FACT (1)) start1
      in simp-rule rule2 end),
    match-preserved-facts ctxt (map fst names) #> (fn {f, s, t, r, results,
    lhs, g'=g'0, ctxt'} =>
      let
        val ctxt' = stop preserved ctxt' |> reset-cache
        val facts = facts-of ctxt'
        val - = Utils.verbose-msg 5 ctxt (fn => facts (1):\n ^ string-of-thms
    ctxt' (maps #thms facts))
        val start0 = Timing.start ();
        val s-eqs = maps #thms facts
        val gN = fst (dest-Var (Thm.term-of (fst (Utils.strip-comb-cterm
    g'0))))
        val preserved-prop = infer-instantiate <f = f and s = s and r = r
    and t = t in
          cprop PRESERVED-FACTS f s r t> ctxt'
        val ([preserved], ctxt1) = Assumption.add-assumes [preserved-prop]
    ctxt'

        val derived-props = map #pred facts |> map (fn pred =>
          (pred, Thm.apply pred t |> Utils.Trueprop-cterm))
        val derived-facts0 = derive-facts ctxt' preserved
        fun prover ctxt =
          let
            val ctxt = ctxt (*|> Config.put Simplifier.simp-trace true*)
          in
            Method.insert-tac ctxt [preserved] 1 THEN
              (SOLVED-DETERM-verbose preserved ctxt
                (asm-full-simp-tac (ctxt
                  addsimps @ {thms PRESERVED-FACTS-def} @ s-eqs))) 1
            end
          val derived-facts1 = derived-props |> map-filter (fn (pred, prop) =>
            try (fn prop => Goal.prove-internal ctxt1 [] prop (fn => prover
    ctxt1)) prop |>
              Option.map (fn thm => ({pred = pred, thms = mksimps ctxt1
    thm})))
          val derived-facts = derived-facts0 @ derived-facts1
          val derived-eqs = maps #thms derived-facts

```

```

val derived-ariths = iariths-of-eqs (maps #thms derived-facts)
val ctxt2 = ctxt1
  addsimps derived-eqs
  |> Simplifier.add-prems derived-eqs
  |> map-states (cons t)
  |> map-facts (K derived-facts)
  |> add-ariths derived-ariths
val - = Utils.timing-msg' 3 ctxt (fn - => preserved (0)) start0
  val - = Utils.verbose-msg 4 ctxt (fn - => derived-facts (1):\n ^
string-of-thms ctxt1 (maps #thms derived-facts))
val eq1 = Utils.timeit-msg 4 ctxt (fn - => eq (5):)
  (fn - => Simplifier.asm-full-rewrite (start eq (5) ctxt2) lhs
  |> STOP-conv ctxt2 |> singleton (timeit-export eq (5): ctxt2
ctxt'))
  val - = Utils.verbose-msg 5 ctxt (fn - => (eq (5):\n ^ Thm.string-of-thm
ctxt' eq1))
  val start1 = Timing.start ();
  val eq = eq1 |> singleton (Proof-Context.export ctxt' ctxt)
  val {g', ...} = @ {cterm-match <run ?g' ?s>} (rhs-of eq1)
  val - = if length names <> length results then
    error (match-preserved-facts: unequal length ^ @ {make-string}
(names, results)) else ()
  val results' = map fst names ~ results
  val g' = Utils.timeit-msg 4 ctxt (fn - => preserved lambdas: )
    (fn - => Utils.lambdas (results') g')
  val rule1 = Utils.timeit-msg 4 ctxt (fn - => preserved rule1: )
    (fn - => Drule.infer-instantiate ctxt [(gN, g')] rule)
  val rule2 = Utils.timeit-msg 4 ctxt (fn - => preserved rule2: )
    (fn - => (rule1 OF [eq])
  |> rule-by-tactic-schematic ctxt (Method.assm-tac ctxt 1))
  val - = Utils.timing-msg' 3 ctxt (fn - => preserved (1)) start1
in
  simp-rule rule2
end),
match-preserved-facts-while ctxt (map fst names) #> (fn {C, C'=C'0,
B, B'=B'0, i, s, t, r, results, C-lhs, B-lhs, ctxt'} =>
let
  val ctxt' = stop while ctxt' |> reset-cache
  val start0 = Timing.start ();
  (* Generalize and merge code with previous case *)
  val facts = facts-of ctxt'
  val s-eqs = maps #thms facts
  val CN = fst (dest-Var (Thm.term-of (fst (Utils.strip-comb-cterm
C'0))))
  val BN = fst (dest-Var (Thm.term-of (fst (Utils.strip-comb-cterm
B'0))))
  val preserved-prop = infer-instantiate <C = C and B = B and i
= i and s = s and r = r and t = t in
  cprop PRESERVED-FACTS-WHILE C B i s r t> ctxt'

```

```

    val ([preserved], ctxt1) = Assumption.add-assumes [preserved-prop]
  ctxt'
    val derived-props = map #pred facts |> map (fn pred =>
      (pred, Thm.apply pred t |> Utils.Trueprop-cterm))
    fun prover ctxt =
      Method.insert-tac ctxt [preserved] 1 THEN
    asm-full-simp-tac (ctxt addsimps @{thms PRESERVED-FACTS-WHILE-def}
  @ s-eqs) 1
    val derived-facts = derived-props |> map-filter (fn (pred, prop) =>
      try (fn prop => Goal.prove-internal ctxt1 [] prop (fn - => prover
  ctxt1)) prop |>
      Option.map (fn thm => ({pred = pred, thms = mksimps ctxt1
  thm})))
    val derived-eqs = maps #thms derived-facts
    val derived-ariths = iariths-of-eqs (maps #thms derived-facts)

    val ctxt2 = ctxt1
      addsimps derived-eqs
      |> Simplifier.add-prems derived-eqs
      |> map-states (cons t)
      |> map-facts (K derived-facts)
      |> add-ariths derived-ariths
    val - = Utils.timing-msg' 3 ctxt (fn - => while (0)) start0
      val - = Utils.verbose-msg 4 ctxt (fn - => derived-facts (2):\n ^
  string-of-thms ctxt1 (maps #thms derived-facts))
    val C-eq1 = Utils.timeit-msg 3 ctxt (fn - => eq (6):)
      (fn - => Simplifier.asm-full-rewrite (start eq (6) ctxt2) C-lhs
      |> STOP-conv ctxt2|> singleton (timeit-export eq (6): ctxt2
  ctxt'))
    val start1 = Timing.start ();
    val C'1 = (rhs-of C-eq1)
    val - = if length names <> length results then
      error (match-preserved-facts-while: unequal length ^@{make-string}
  (names, results)) else ()
    val results' = map fst names ^^ results
    val C' = Utils.lambdas (results' @ [(s, t)]) C'1
    val C'-pred = Utils.lambdas [(s, t)] C'1
    val ([C'-thm], ctxt3) = Assumption.add-assumes [Utils.Trueprop-cterm
  C'1] ctxt2
    val C'-eqs = mksimps ctxt3 C'-thm
    val C'-ariths = iariths-of-eqs C'-eqs
    val ctxt4 = (ctxt3 |> map-facts (cons {pred = C'-pred, thms =
  C'-eqs}))
      addsimps C'-eqs
      |> Simplifier.add-prems C'-eqs
      |> add-ariths C'-ariths
    val - = Utils.timing-msg' 3 ctxt (fn - => while (1)) start1
    val - = Utils.verbose-msg 5 ctxt (fn - => (adding:\n ^ string-of-thms
  ctxt1 C'-eqs))

```

```

    val B-eq1 = Utils.timeit-msg 4 ctxt (fn => eq (7): )
      (fn => Simplifier.asm-full-rewrite (start eq (7) ctxt4) B-lhs
        |> STOP-conv ctxt4 |> singleton (timeit-export eq (7): ctxt4
      ctxt'))
    val start2 = Timing.start ();
    val {B', ...} = @{cterm-match <run ?B' ?t>} (rhs-of B-eq1)
    val B' = Utils.lambdas (results') B'
    val rule1 = Drule.infer-instantiate ctxt [(CN, C'), (BN, B')] rule
    val [C-eq, B-eq] = [C-eq1, B-eq1] |> Proof-Context.export ctxt' ctxt
    val - = Utils.verbose-msg 5 ctxt (fn => (eq (6):\n ^ Thm.string-of-thm
      ctxt C-eq))
    val - = Utils.verbose-msg 5 ctxt (fn => (eq (7):\n ^ Thm.string-of-thm
      ctxt B-eq))
    val rule2 = Utils.solve-sideconditions ctxt rule1 (
      resolve-tac ctxt [Conjunction.conjunctionI] 1 THEN
      resolve-tac ctxt [C-eq] 1 THEN
      Method.assm-tac ctxt 1 THEN
      resolve-tac ctxt [B-eq] 1 THEN
      Method.assm-tac ctxt 1 THEN
      Method.assm-tac ctxt 1
    )
    val - = Utils.timing-msg' 3 ctxt (fn => while (2)) start2
    in simp-rule rule2
  end),
  fn ct => raise TERM (monad-state-conv: unexpected: , [Thm.term-of
ct])) end )
  end
in
  simp-rule inst-rule
end
handle Option.Option => Thm.reflexive ct

fun monad-state-conv' (rules: rules) ctxt ct =
  let
    val matches0 = Net.match-term rules (Thm.term-of ct)
  in
    case get-more-specific matches0 of
      NONE => Thm.reflexive ct
    | SOME rule => monad-state-conv rule ctxt ct
  end

fun monad-run-simproc rule ctxt ct =
  let
    val eq = monad-state-conv rule ctxt ct
  in
    if Utils.is-reflexive ctxt eq then
      NONE
    else
      SOME (Utils.timeit-msg 3 ctxt (fn => STOPI: ) (fn => @{thm STOPI}

```



```

OF [eq]))
end

```

```

end
>

```

```

attribute-setup monad-simp-global-stop-cong = <
  Scan.lift (Scan.option Args.add) >> (fn - => Thm.declaration-attribute (Monad-Cong-Simp.add-global-stop-c
  || Scan.lift Args.del >> (fn - => Thm.declaration-attribute (Monad-Cong-Simp.del-global-stop-cong))
> Global 'stop' congruence rule for monad simplification

```

```

attribute-setup monad-simp-cong = <
let
  val split = Args.*** split |-- Args.colon |-- Parse.and-list Parse.short-ident
  val opt-split = Scan.optional (Scan.lift (split)) []
  fun pos-here [] = (Position.none, [])
    | pos-here ts = (Token.pos-of (hd ts), ts)
in
  Scan.lift Args.add |-- (Scan.lift pos-here -- opt-split) >> (fn (pos, splits) =>
    Thm.declaration-attribute (Monad-Cong-Simp.add-rule splits))
  || Scan.lift Args.del >> (K (Thm.declaration-attribute (Monad-Cong-Simp.del-rule)))
  || opt-split >> (fn splits =>
    Thm.declaration-attribute (Monad-Cong-Simp.add-rule splits))
end> Monad congruence theorems

```

```

attribute-setup monad-simp-derive-rule = <
  Scan.lift (Scan.option Args.add) >> (fn - => Thm.declaration-attribute (Monad-Cong-Simp.add-derive-rule))
  || Scan.lift Args.del >> (fn - => Thm.declaration-attribute (Monad-Cong-Simp.del-derive-rule))
> Conditional rules to derive facts in monad simplification

```

```

simproc-setup monad-run-simproc (⟨run f s⟩) = ⟨fn - => fn ctxt => fn ct =>
  Monad-Cong-Simp.matching-rule ctxt (Thm.term-of ct) |> Option.mapPartial
(fn rule =>
  let
    val - = Utils.verbose-msg 7 ctxt (fn - => monad-run-simproc:\n ^ string-of-cterm
  ctxt ct)
  in
    Monad-Cong-Simp.monad-run-simproc rule ctxt ct
  end)⟩

```

```

declare [[simproc del: monad-run-simproc]]

```

```

lemma spec-monad-ext': (∧s. run f s ≡ run f' s) ⇒ f ≡ f'
  by (smt (verit) spec-monad-ext)

```

```

ML <
structure Monad-Cong-Simp =
struct
  open Monad-Cong-Simp

```

```

fun dest-spec-monadT Type ⟨spec-monad E A S⟩ = {exnT = E, regT = A, stateT
= S}
| dest-spec-monadT T = raise TYPE(dest-spec-monadT: , [T], [])

fun gen-monad-conv {stop} ctxt ct =
  let
    val cT = Thm.ctyp-of-cterm ct
    val {stateT, exnT, regT} = Thm.typ-of-cterm ct |> dest-spec-monadT
    val {rules, ...} = Data.get (Context.Proof ctxt)
    val ([s], ctxt1) = Utils.fix-variant-cfrees [(s, stateT)] ctxt
    val ctxt2 = map-states (K [s]) ctxt1
    val lhs = infer-instantiate ⟨f = ct and s = ⟨s⟩
      in cterm ⟨run f s⟩ for f::⟨('e::default, 'a, 's) spec-monad⟩⟩ ctxt
    val eq = monad-state-conv' rules ctxt2 lhs
    val eq' = singleton (Proof-Context.export ctxt2 ctxt) eq
    val eq' = @{thm spec-monad-ext'} OF [eq']
  in
    if stop then (@{thm STOPI} OF [eq']) |> Drule.zero-var-indexes else eq'
  end

end
>

```

We make a global setup here, as the locale one within *heap-typing-state* fails. The reason is that the pattern is not taken into account in *simproc* equality when inserted into the net. Hence multiple instances (e.g. for *globals* and *lifted-globals*) are considered a duplicate when adding them both to the net but the simplifier does not match the pattern for *globals* on a *lifted-globals* instance. In the upcoming Isabelle2024 this issue should be resolved.

```

simproc-setup unchanged-typing-spec-monad (⟨c · s ?λr. heap-typing-state.unchanged-typing-on
htd S s⟩ ) = ⟨
  let
    val is-assume-stack-alloc = Match-Cterm.switch [
      @{cterm-match (fo) typ-heap-typing.assume-stack-alloc ?r ?w ?ht ?ht-upd
?S ?n ?X · ?s ?λ?Q} #>
      (fn - => true),
      (fn - => false)]

    val active = Config.declare-bool (active, Position.none) (K false);
    val prop-to-meta-eq = @{lemma ⟨P ==> (P ≡ True)⟩ for P by auto}
    fun try-prove ctxt prop =
      case Monad-Cong-Simp.lookup-cache ctxt prop of
      [] =>
        let
          val ctxt' = Context.proof-map (Named-Rules.add-thm @{named-rules
runs-to-vcg} @{thm runs-to-partial-case-prod}) ctxt
        in

```

```

      case try (Goal.prove-internal ctxt' [] prop) (fn - => Unchanged-Typing.unchanged-typing-tac
NONE ctxt') of
        NONE => (warning (unchanged-typing proof failed: ^ string-of-cterm
ctxt prop); NONE)
        | SOME thm => (Monad-Cong-Simp.add-to-cache ctxt thm; SOME thm)
      end
    | (thm::-) => (Utils.verbose-msg 4 ctxt (fn - => cache hit: ^ Thm.string-of-thm
ctxt thm); SOME thm)
  in
    fn phi => fn ctxt => fn ct =>
      if Config.get ctxt active orelse (is-assume-stack-alloc ct) then NONE else
      let
        val - = Utils.verbose-msg 4 ctxt (fn - => unchanged-typing-spec-monad
invoked on:\n ^ @{make-string} ct)
      in
        try-prove (Config.put active true ctxt) instantiate <P = ct in cterm <Trueprop
P> for P>
          |> Option.map (fn thm => prop-to-meta-eq OF [thm])
        end
      end

    end
  >
declare [[simproc del: unchanged-typing-spec-monad]]

setup <
  let
    val (unchanged-typing-simproc-name, -) =
      Simplifier.check-simproc @ {context} (unchanged-typing-spec-monad, here)
  in
    Context.theory-map (Monad-Cong-Simp.add-simproc unchanged-typing-simproc-name)
  end
  >

simproc-setup spec-monad-simproc (<f::('a, 'b, 'c) spec-monad>) = <fn - => fn
ctxt =>
  if Monad-Cong-Simp.monad-simp-active ctxt then K NONE else
  Match-Cterm.switch [
    @ {cterm-match STOP -} #> (fn - => NONE),
    fn ct =>
      let
        val - = Utils.verbose-msg 7 ctxt (fn - => (spec-monad-simproc:\n ^
string-of-cterm ctxt ct))
        val eq = Monad-Cong-Simp.gen-monad-conv {stop=true} ctxt ct
        val - = Utils.verbose-msg 7 ctxt (fn - => (spec-monad-simproc eq:\n ^
Thm.string-of-thm ctxt eq))
      in

```

```

        SOME eq
      end]›
declare [[simproc del: spec-monad-simproc]]

ML ‹
  structure Monad-Cong-Simp =
  struct
    open Monad-Cong-Simp

  fun prepare-simpset max-depth ctxt =
    let
      val {stop-congs, simprocs, ...} = Monad-Cong-Simp.Data.get (Context.Proof
        ctxt)
      val simprocs' = map (snd o Simplifier.check-simproc ctxt o rpair Position.none)
        simprocs
      val visible = Context-Position.is-visible ctxt
      val monad-cong-simps = Named-Theorems.get ctxt @{named-theorems monad-simp-simp}
      val run-spec-monad = Named-Theorems.get ctxt @{named-theorems run-spec-monad}

      val ctxt' = (ctxt |> Context-Position.set-visible false
        |> Simplifier.add-cong @{thm STOP-cong}
        |> fold Simplifier.add-cong stop-congs)
      addsimprocs [ @{simproc spec-monad-simproc}, @{simproc monad-run-simproc} ]
    @ simprocs'
      delsimps @{thms return-bind bind-return Spec-Monad.run-case-prod-distrib}
    @ run-spec-monad
      addsimps monad-cong-simps
      |> Context-Position.set-visible visible
      |> Config.map simp-depth-limit (K (max-depth + 20))
    in ctxt' end

  fun monad-simp-tac ctxt = SUBGOAL (fn (goal, i) =>
    let
      val depth = strip-comb-depth-of-term goal
      val ctxt' = prepare-simpset depth ctxt
    in
      (asm-full-simp-tac ctxt' THEN-ALL-NEW
        full-simp-tac (Simplifier.clear-simpset ctxt addsimps @{thms STOP-def polish}))
      i
    end)

  fun monad-asm-full-simplify ctxt thm =
    let
      val depth = strip-comb-depth-of-term (Thm.prop-of thm)
      val ctxt' = prepare-simpset depth ctxt
    in
      thm |> asm-full-simplify ctxt' |>
      full-simplify (Simplifier.clear-simpset ctxt addsimps @{thms STOP-def polish})
    end

```

```

fun monad-asm-full-rewrite ctxt ct =
  let
    val depth = strip-comb-depth-of-term (Thm.term-of ct)
    val ctxt' = prepare-simpset depth ctxt
  in
    ct |> (Simplifier.asm-full-rewrite ctxt' then-conv
    Simplifier.full-rewrite (Simplifier.clear-simpset ctxt addsimps @ {thms STOP-def
    polish}))
  end

fun monad-simplify-def ctxt def-eq =
  let
    val ctxt' = Variable.declare-thm def-eq ctxt
    val ((-, [eq']), ctxt'') = Variable.import true [Thm.transfer' ctxt def-eq] ctxt'
  in
    eq' |> monad-asm-full-simplify ctxt'' |> singleton (Variable.export ctxt'' ctxt')
    |> zero-var-indexes
  end

end
>

method-setup monad-simp = <Method.sections Simplifier.simp-modifiers >>
  (K (fn ctxt => METHOD (fn facts => HEADGOAL (
    Method.insert-tac ctxt facts THEN' (CHANGED-PROP oo Monad-Cong-Simp.monad-simp-tac)
    ctxt))))>
  Simplification for Spec Monad with custom congruence rules

context open-types-heap-typing-state
begin
lemma PRESERVED-FACTS-unchanged-typing-ptr-valid-preserved[monad-simp-simp]:
  reaches fs r t  $\implies$  ptr-valid (heap-typing s) p  $\implies$  f  $\cdot$  s ?{\lambda-. unchanged-typing-on
  UNIV s t}  $\implies$ 
  ptr-valid (heap-typing t) p
  by (rule reaches-unchanged-typing-ptr-valid-preserved)

lemma PRESERVED-FACTS-WHILE-unchanged-typing-ptr-valid-preserved[monad-simp-simp]:
  assumes reach: whileLoop-unroll-reachable C B i s r t
  assumes valid: ptr-valid (heap-typing s) p
  assumes B:  $\bigwedge r t. C r t \implies (B r) \cdot t ?{\lambda-. unchanged-typing-on UNIV t}$ 
  shows ptr-valid (heap-typing t) p
proof -
  have unchanged-typing-on UNIV s t
  using reach
proof induction
  case (step b t X c u)

```

with B [of b t] *unchanged-typing-on-trans*[of $UNIV$ s t u]
show $?case$
by (*auto simp add: runs-to-partial-holds-partial-def*)
qed *simp*

with *valid* **show** $?thesis$
by (*simp add: unchanged-typing-on-UNIV-iff*)
qed

end

lemma *PRESERVED-FACTS-run-bind-cong*[*monad-simp-cong split: g and g'*]:
 $run\ f\ s = run\ f'\ s \implies (\bigwedge t\ r.\ PRESERVED-FACTS\ f'\ s\ (Result\ r)\ t \implies run\ (g\ r)\ t = run\ (g'\ r)\ t) \implies$
 $(run\ (bind\ f\ g)\ s) = (run\ (bind\ f'\ g')\ s)$
apply (*rule Spec-Monad.run-bind-cong*)
apply (*auto simp add: PRESERVED-FACTS-def runs-to-partial-def-old reaches-def*)
done

lemma *ADD-FACT-run-bind-guard-cong*[*monad-simp-cong*]:
 $P\ s = P'\ s \implies (ADD-FACT\ P'\ s \implies run\ f\ s = run\ f'\ s) \implies$
 $(run\ (bind\ (guard\ P)\ (\lambda-. f))\ s) = (run\ (bind\ (guard\ P')\ (\lambda-. f'))\ s)$
apply (*simp add: ADD-FACT-def*)
apply (*rule run-bind-cong*)
apply (*simp add: run-guard*)
apply (*auto simp add: run-guard if-split-asm runs-to-partial-def-old*)
done

lemma *PRESERVED-FACTS-WHILE-run-whileLoop-cong*[*monad-simp-cong split: B and B' and C and C'*]:
assumes $i: i = i'$
assumes $C-B$ [*unfolded PRESERVED-FACTS-WHILE-def*]:
 $\bigwedge t\ r.\ PRESERVED-FACTS-WHILE\ C\ B\ i\ s\ r\ t \implies (C\ r\ t \equiv C'\ r\ t) \ \&\&\&$
 $(C'\ r\ t \implies (run\ (B\ r)\ t \equiv run\ (B'\ r)\ t))$
shows $run\ (whileLoop\ C\ B\ i)\ s = run\ (whileLoop\ C'\ B'\ i')\ s$
proof –
from $C-B$
have $C: \bigwedge r\ s'. whileLoop-unroll-reachable\ C\ B\ i\ s\ r\ s' \implies C\ r\ s' = C'\ r\ s'$ **by**
blast
from $C-B$
have $B: \bigwedge r\ s'. whileLoop-unroll-reachable\ C\ B\ i\ s\ r\ s' \implies C'\ r\ s' \implies run\ (B\ r)\ s' = run\ (B'\ r)\ s'$
by *blast*
with C **have** $\bigwedge r\ s'. whileLoop-unroll-reachable\ C\ B\ i\ s\ r\ s' \implies C\ r\ s' \implies run\ (B\ r)\ s' = run\ (B'\ r)\ s'$
by *simp*
from *run-whileLoop-unroll-reachable-cong* [*OF C this*]
show $?thesis$ **by** (*simp add: i*)
qed

lemma *whileLoop-condition-only-cong*[*monad-simp-split-tuple-cong*]:
 $C = C' \implies \text{whileLoop } C \ B \ I = \text{whileLoop } C' \ B \ I$
by *simp*

lemma *PRESERVED-FACTS-run-bind-exception-or-result-cong*[*monad-simp-cong split: g and g'*]:
 $\text{run } f \ s = \text{run } f' \ s \implies (\bigwedge t \ r. \text{PRESERVED-FACTS } f' \ s \ r \ t \implies \text{run } (g \ r) \ t = \text{run } (g' \ r) \ t) \implies$
 $(\text{run } (\text{bind-exception-or-result } f \ g) \ s) = (\text{run } (\text{bind-exception-or-result } f' \ g') \ s)$
apply (*rule Spec-Monad.run-bind-exception-or-result-cong*)
apply (*auto simp add: PRESERVED-FACTS-def runs-to-partial-def-old reaches-def*)
done

lemma *PRESERVED-FACTS-run-on-exit'-cong*[*monad-simp-cong split: g and g'*]:
 $\text{run } f \ s = \text{run } f' \ s \implies (\bigwedge t \ r. \text{PRESERVED-FACTS } f' \ s \ r \ t \implies \text{run } g \ t = \text{run } g' \ t) \implies$
 $(\text{run } (\text{on-exit}' \ f \ g) \ s) = (\text{run } (\text{on-exit}' \ f' \ g') \ s)$
unfolding *on-exit'-def*
apply (*rule PRESERVED-FACTS-run-bind-exception-or-result-cong*)
apply *assumption*
apply (*rule PRESERVED-FACTS-run-bind-cong*)
apply *assumption*
apply *simp*
done

lemma *PRESERVED-FACTS-run-on-exit-cong*[*monad-simp-cong split: g and g'*]:
 $\text{run } f \ s = \text{run } f' \ s \implies$
 $(\bigwedge t \ r. \text{PRESERVED-FACTS } f' \ s \ r \ t \implies \text{run } ((\text{state-select } g::('e::\text{default}, \text{unit}, 's) \text{ spec-monad})) \ t = \text{run } (\text{state-select } g') \ t) \implies$
 $(\text{run } (\text{on-exit } f \ g::('e::\text{default}, 'a, 's) \text{ spec-monad}) \ s) = (\text{run } (\text{on-exit } f' \ g') \ s)$
unfolding *on-exit-def*
apply (*rule PRESERVED-FACTS-run-on-exit'-cong*)
apply *assumption*
apply *simp*
done

lemma *ADD-FACT-run-condition-cong* [*monad-simp-cong*]:
assumes $c \ s = c' \ s$
assumes *ADD-FACT* $c' \ s \implies \text{run } f \ s = \text{run } f' \ s$
assumes *ADD-FACT* $(\lambda s. \neg c' \ s) \ s \implies \text{run } g \ s = \text{run } g' \ s$
shows $\text{run } (\text{condition } c \ f \ g) \ s = \text{run } (\text{condition } c' \ f' \ g') \ s$
using *assms*
by (*auto simp add: ADD-FACT-def run-condition*)

lemma *ADD-FACT-run-when* [*monad-simp-cong*]:
assumes $c = c'$
assumes *ADD-FACT* $(\lambda-. \ c') \ s \implies \text{run } f \ s = \text{run } f' \ s$
shows $\text{run } (\text{when } c \ f) \ s = \text{run } (\text{when } c' \ f') \ s$

using *assms*
by (*auto simp add: ADD-FACT-def when-def run-condition*)

lemma *PRESERVED-FACTS-run-catch-cong* [*monad-simp-cong split: g and g'*]:
 $run\ f\ s = run\ f'\ s \implies (\bigwedge t\ e.\ PRESERVED-FACTS\ f'\ s\ (Exn\ e)\ t \implies run\ (g\ e)\ t = run\ (g'\ e)\ t) \implies$
 $(run\ (catch\ f\ g)\ s) = (run\ (catch\ f'\ g')\ s)$
apply (*simp add: run-catch PRESERVED-FACTS-def reaches-def*)
apply (*rule bind-post-state-cong*)
apply (*auto split: xval-splits*)
done

lemma *run-finally-cong* [*monad-simp-cong*]:
assumes $run\ f\ s = run\ f'\ s$
shows $run\ (finally\ f)\ s = run\ (finally\ f')\ s$
using *assms*
by (*simp add: run-finally*)

lemma *run-liftE-cong* [*monad-simp-cong*]:
assumes $run\ f\ s = run\ f'\ s$
shows $run\ (liftE\ f)\ s = run\ (liftE\ f')\ s$
using *assms*
by (*simp add: run-liftE*)

lemma *run-guard-cong* [*monad-simp-cong*]:
assumes $P\ s = P'\ s$
shows $run\ (guard\ P)\ s = run\ (guard\ P')\ s$
using *assms*
by (*simp add: run-guard*)

lemma *run-try-cong* [*monad-simp-cong*]:
assumes $run\ f\ s = run\ f'\ s$
shows $run\ (try\ f)\ s = run\ (try\ f')\ s$
using *assms*
by (*simp add: run-try*)

lemma *run-case-prod-cong*[*monad-simp-cong*]: *RUN-CASE-PROD* $f\ x\ s\ f'\ x' \implies$

$(run\ (case-prod\ f\ x)\ s) = (run\ (case-prod\ f'\ x')\ s)$
by (*cases x*) (*auto simp add: RUN-CASE-PROD-def*)

lemma *run-bind-when-throw-cong* [*monad-simp-cong*]:
 $c = c' \implies e = e' \implies (ADD-FACT\ (\lambda-. \neg c')\ s \implies run\ f\ s = run\ f'\ s) \implies$
 $run\ (bind\ (when\ c\ (throw\ e))\ (\lambda-. f))\ s = run\ (bind\ (when\ c'\ (throw\ e'))\ (\lambda-. f'))\ s$
by (*auto simp add: ADD-FACT-def*)

ML \langle
Monad-Cong-Simp.print-rules (*Context.Proof* @*{context}*)

>

lemma *with-fresh-stack-ptr-stop-cong* [*monad-simp-global-stop-cong*]:
 typ-heap-typing.with-fresh-stack-ptr r w heap-typing heap-typing-upd S n init f \equiv
 typ-heap-typing.with-fresh-stack-ptr r w heap-typing heap-typing-upd S n init f
 by (*simp*)

lemma *assume-with-fresh-stack-ptr-stop-cong* [*monad-simp-global-stop-cong*]:
 typ-heap-typing.assume-with-fresh-stack-ptr r w heap-typing heap-typing-upd S n
 init f \equiv
 typ-heap-typing.assume-with-fresh-stack-ptr r w heap-typing heap-typing-upd S
 n init f
 by (*simp*)

lemma *guard-with-fresh-stack-ptr-stop-cong* [*monad-simp-global-stop-cong*]:
 typ-heap-typing.guard-with-fresh-stack-ptr r w heap-typing heap-typing-upd S n
 init f \equiv
 typ-heap-typing.guard-with-fresh-stack-ptr r w heap-typing heap-typing-upd S
 n init f
 by (*simp*)

context *typ-heap-simulation-open-types-stack*
begin

lemma *PRESERVED-FACTS-assume-stack-alloc-ptr-valid-preserved*[*monad-simp-simp*]:
 reaches (assume-stack-alloc n init) s x t \implies *ptr-valid (heap-typing s) (p::'b::mem-type*
 ptr) \implies
 typ-uinfo-t TYPE('b) \neq typ-uinfo-t TYPE(stack-byte) \implies
 ptr-valid (heap-typing t) p
 apply (*clarsimp simp add: stack-ptr-acquire-def heap-typing-fold-upd-write reaches-def*

 assume-stack-alloc-def)
 using *stack-allocs-ptr-valid-cases2* **by** *blast*

lemma *PRESERVED-FACTS-with-fresh-stack-ptr-cong*[*monad-simp-cong split: f*
and *f'*]:
 fixes *f::'a ptr \Rightarrow ('e::default, 'd, 't) spec-monad*
 assumes $\bigwedge t p.$ *PRESERVED-FACTS (assume-stack-alloc n init::('e::default, 'a*
 ptr, 't) spec-monad) s (Result p) t \implies
 run (f p) t = run (f' p) t
 shows *run (with-fresh-stack-ptr n init f) s = run (with-fresh-stack-ptr n init f')*
 s
 apply (*simp add: with-fresh-stack-ptr-def PRESERVED-FACTS-def ADD-FACT-def*)
 apply (*rule run-bind-cong*)
 subgoal by *simp*
 subgoal
 apply (*clarsimp simp add: runs-to-partial-def-old on-exit-def on-exit'-def*)
 apply (*rule run-bind-exception-or-result-cong*)

```

subgoal
  apply (rule assms)
  apply (simp add: PRESERVED-FACTS-def ADD-FACT-def)
  done
subgoal by simp
done
done

```

```

lemma PRESERVED-FACTS-assume-with-fresh-stack-ptr-cong[monad-simp-cong
split: f and f']:
  fixes f::'a ptr  $\Rightarrow$  ('e::default, 'd, 't) spec-monad
  assumes  $\bigwedge t p$ . PRESERVED-FACTS (assume-stack-alloc n init::('e::default, 'a
ptr, 't) spec-monad) s (Result p) t  $\Longrightarrow$ 
    run (f p) t = run (f' p) t
  shows run (assume-with-fresh-stack-ptr n init f) s = run (assume-with-fresh-stack-ptr
n init f') s
  apply (simp add: assume-with-fresh-stack-ptr-def PRESERVED-FACTS-def ADD-FACT-def)
  apply (rule run-bind-cong)
  subgoal by simp
  subgoal
    apply (clarsimp simp add: runs-to-partial-def-old on-exit-def on-exit'-def)
    apply (rule run-bind-exception-or-result-cong)
  subgoal
    apply (rule assms)
    apply (simp add: PRESERVED-FACTS-def ADD-FACT-def)
    done
  subgoal by simp
  done
done

```

```

lemma PRESERVED-FACTS-guard-with-fresh-stack-ptr-cong[monad-simp-cong split:
f and f']:
  fixes f::'a ptr  $\Rightarrow$  ('e::default, 'd, 't) spec-monad
  assumes  $\bigwedge t p$ . PRESERVED-FACTS (assume-stack-alloc n init::('e::default, 'a
ptr, 't) spec-monad) s (Result p) t  $\Longrightarrow$ 
    run (f p) t = run (f' p) t
  shows run (guard-with-fresh-stack-ptr n init f) s = run (guard-with-fresh-stack-ptr
n init f') s
  apply (simp add: guard-with-fresh-stack-ptr-def PRESERVED-FACTS-def ADD-FACT-def)
  apply (rule run-bind-cong)
  subgoal by simp
  subgoal
    apply (clarsimp simp add: runs-to-partial-def-old on-exit-def on-exit'-def)
    apply (rule run-bind-exception-or-result-cong)
  subgoal
    apply (rule assms)
    apply (simp add: PRESERVED-FACTS-def ADD-FACT-def)
    done
  subgoal by simp

```

```

done
done

lemma ptr-valid-assume-stack-alloc [monad-simp-derive-rule]:
  PRESERVED-FACTS (assume-stack-alloc n init::('e::default, 'a ptr, 't) spec-monad)
  s (Result p) t  $\implies$ 
    ADD-FACT ( $\lambda t$ . PTR-VALID('a) (heap-typing t) p) t
      apply (simp add: PRESERVED-FACTS-def ADD-FACT-def)
      apply (auto simp add: stack-ptr-acquire-def heap-typing-fold-upd-write
reaches-def
assume-stack-alloc-def root-ptr-valid-ptr-valid elim!: stack-allocs-cases)
    done

end

method array-index-to-ptr-arith-simp uses simp cong =
  (use TrueI[array-bound-mksimps, simproc add: field-lvalue-unfold]
  in ⟨monad-simp
    simp add:field-lvalue-array-index' simp
    simp del: field-lvalue-append
    cong: if-cong cong⟩)

end

theory Runs-To-VCG-StackPointer
imports
  HeapLift
  Refines-Spec
begin

definition distinct-span xs  $\equiv$ 
  distinct-prop ( $\lambda(v1, s1) (v2, s2)$ . disjnt {v1 ..+ s1} {v2 ..+ s2}) xs

definition distinct-spans xs  $\equiv$ 
  distinct-prop ( $\lambda(v1, n1, s1) (v2, n2, s2)$ . disjnt {v1 ..+ n1 * s1} {v2 ..+ n2 *
s2}) xs

lemma distinct-span-spans-conv:
  distinct-span xs  $\longleftrightarrow$  distinct-spans (map ( $\lambda(v, s)$ . (v, 1, s)) xs)
  unfolding distinct-span-def distinct-spans-def
  by (clarsimp simp: distinct-prop-map case-prod-beta intro!: distinct-prop-iff)

definition lvp-distinct xs  $\equiv$ 
  distinct-spans (map ( $\lambda(d, v, n, s)$ . (v, n, s)) xs)  $\wedge$ 
  distinct (map ( $\lambda(d, v, n, s)$ . (d, v)) xs)  $\wedge$ 
  sorted-wrt ( $\lambda(d1, -) (d2, -)$ . stack-free d1  $\subseteq$  stack-free d2) xs  $\wedge$ 
  ( $\forall (d, v, n, s) \in \text{set } xs$ .
    disjnt {v ..+ n * s} (stack-free d)  $\wedge$  s > 0)

```

lemma *lvp-distinct-pairwise-disjnt*:

$\llbracket \text{lvp-distinct } xs; xs ! m1 = (d1, v1, n1, s1); xs ! m2 = (d2, v2, n2, s2); m1 < m2; m2 < \text{length } xs \rrbracket \implies$

$\text{disjnt } \{v1 \text{ ..+ } n1 * s1\} \{v2 \text{ ..+ } n2 * s2\}$

unfolding *lvp-distinct-def*

apply (*clarsimp*)⁺

apply (*simp add: distinct-spans-def distinct-prop-map case-prod-beta*)

apply (*drule(2) distinct-prop-nth*)

by *auto*

lemma *lvp-distinct-spansD*: *lvp-distinct* *xs* \implies

distinct-spans (*map* ($\lambda(d, v, n, s). (v, n, s)$) *xs*)

by (*simp add: lvp-distinct-def*)

named-theorems *stack-ptr-simps*

context *stack-simulation-heap-typing* **begin**

lemmas [*stack-ptr-simps*] =

stack-ptr-acquire-def

stack-ptr-release-def

write-same-ZERO

stack-releases-stack-allocs-inverse

root-ptr-valid-ptr-valid

lemma *runs-to-guard-with-fresh-stack-ptr*[*runs-to-vcg*]:

assumes *f*[*runs-to-vcg*]: $\bigwedge d \text{ ptr } vs.$

$(\text{ptr}, d) \in \text{stack-allocs } n \mathcal{S} \text{ TYPE}'a (\text{heap-typing } s) \implies$

$vs \in \text{init } s \implies$

$\text{length } vs = n \implies$

$(\forall i \in \{0..<n\}. r \text{ s } (\text{ptr} +_p \text{int } i) = \text{ZERO}'a) \implies$

$(\forall i \in \{0..<n\}. \text{root-ptr-valid } d (\text{ptr} +_p \text{int } i)) \implies$

$(f \text{ ptr}) \cdot (\text{stack-ptr-acquire } n \text{ vs } \text{ptr } d \text{ s})$

$\{\lambda r u. (\forall i < n. \text{root-ptr-valid } (\text{heap-typing } u) (\text{ptr} +_p \text{int } i)) \wedge Q \text{ r } (\text{stack-ptr-release } n \text{ ptr } u)\}$

shows

$(\text{guard-with-fresh-stack-ptr } n \text{ init } f) \cdot s \{\! \{ Q \}\!$

apply (*unfold guard-with-fresh-stack-ptr-def assume-stack-alloc-def*)

apply *runs-to-vcg*

apply (*auto elim: stack-allocs-cases*)

done

lemma *runs-to-with-fresh-stack-ptr*[*runs-to-vcg*]:

assumes *f*[*runs-to-vcg*]: $\bigwedge d \text{ ptr } vs.$

$(\text{ptr}, d) \in \text{stack-allocs } n \mathcal{S} \text{ TYPE}'a (\text{heap-typing } s) \implies$

$vs \in \text{init } s \implies$

```

length vs = n  $\implies$ 
 $(\forall i \in \{0..<n\}. r\ s\ (ptr\ +_p\ int\ i) = ZERO('a)) \implies$ 
 $(\forall i \in \{0..<n\}. root\ ptr\ valid\ d\ (ptr\ +_p\ int\ i)) \implies$ 
 $(f\ ptr) \cdot (stack\ ptr\ acquire\ n\ vs\ ptr\ d\ s) \{\lambda r\ u. Q\ r\ (stack\ ptr\ release\ n\ ptr\ u)\}$ 
shows
   $(assume\ with\ fresh\ stack\ ptr\ n\ init\ f) \cdot s \{\ Q \}$ 
   $(with\ fresh\ stack\ ptr\ n\ init\ f) \cdot s \{\ Q \}$ 
subgoal
  apply  $(unfold\ assume\ with\ fresh\ stack\ ptr\ def\ assume\ stack\ alloc\ def)$ 
  apply  $(runs\ to\ vcg)$ 
  apply  $(auto\ elim: stack\ allocs\ cases\ simp\ add: singleton\ iff)$ 
  done
subgoal
  apply  $(unfold\ with\ fresh\ stack\ ptr\ def\ on\ exit\ def\ assume\ stack\ alloc\ def)$ 
  apply  $(runs\ to\ vcg)$ 
  apply  $(auto\ elim: stack\ allocs\ cases\ simp\ add: singleton\ iff)$ 
  done
done

```

lemma *lvp-distinct-singleton-local-with-params:*

```

fixes  $p::'a\ ptr$ 
assumes  $(p, d') \in stack\ allocs\ n\ \mathcal{S}\ TYPE('a)\ d$ 
assumes  $root\ ptr\ valid\ d'\ p$ 
assumes  $distinct\ span\ params$ 
assumes  $\forall (v, s) \in set\ params. disjoint\ \{v\ ..+ s\}\ (stack\ free\ d)$ 
assumes  $\forall (v, s) \in set\ params. s > 0$ 
shows  $lvp\ distinct\ ((d', ptr\ val\ p, n, size\ td\ (typ\ uinfo\ t\ TYPE('a)))\#(map\ (\lambda(v, s). (d, v, 1, s))\ params))$ 
unfolding  $lvp\ distinct\ def$ 
apply  $(clarsimp\ simp: case\ prod\ beta\ distinct\ spans\ def\ assume\ stack\ alloc\ def)$ 
apply  $(intro\ conjI, clarsimp\ simp: distinct\ prop\ map)$ 
subgoal for  $v\ t$ 
  using  $assms(4)$  apply  $clarsimp$ 
  apply  $(rule\ disjoint\ subset1\ [where\ X=stack\ free\ d])$ 
  apply  $(auto\ simp: disjoint\ comm\ [simplified\ disjoint\ def]\ disjoint\ def)[1]$ 
  apply  $(simp\ add: size\ of\ fold)$ 
  by  $(rule\ stack\ allocs\ stack\ subset\ stack\ free\ [OF\ assms(1), simplified])$ 
subgoal
  apply  $(clarsimp\ simp: distinct\ prop\ map)$ 
  apply  $(rule\ distinct\ prop\ iff\ [THEN\ iffD1, OF\ -\ assms(3)\ [simplified\ distinct\ span\ def]])$ 
  by  $clarsimp$ 
subgoal
  apply  $(clarsimp)$ 
  by  $(contradiction, rule\ stack\ allocs\ neq\ [OF\ assms(1)], erule\ sym)$ 
subgoal
  apply  $(rule\ distinct\ iff\ distinct\ prop\ ne\ [THEN\ iffD2])$ 
  apply  $(clarsimp\ simp: distinct\ prop\ map)$ 
  apply  $(rule\ distinct\ prop\ impl\ [OF\ -\ assms(3)\ [simplified\ distinct\ span\ def]])$ 

```

```

using assms(5) apply clarsimp
subgoal for t v t'
  apply (frule(1) bspec [where  $x=(v, t)$ ], drule(1) bspec [where  $x=(v, t')$ ])
  by (auto simp: intvl-start-inter disjnt-def)
done
subgoal
  apply (rule ssubst [OF stack-free-stack-allocs [OF assms(1)]]])
  by clarsimp
subgoal apply (clarsimp simp: sorted-wrt-map)
  apply (rule sorted-wrt-mono-rel [where  $P=\lambda x y. \text{stack-free } d \subseteq \text{stack-free } d$ ])
  apply (clarsimp)
  by (simp)
subgoal
  apply (simp add: size-of-fold disjnt-def)
  by (rule fresh-ptrs-stack-free-disjunct [OF assms(1), simplified])
subgoal
  by (simp add: size-of-fold)
subgoal
  using assms(4,5) by (auto simp: disjnt-def dest!: bspec)
done

```

lemma *runs-to-init-local-variables-and-parameters*:

```

assumes
   $n > 0$ 
  distinct-span params
   $\forall (v, sz) \in \text{set } \text{params}. \text{disjnt } \{v..+sz\} (\text{stack-free } (\text{heap-typing } s))$ 
   $\forall (v, sz) \in \text{set } \text{params}. 0 < sz$ 
   $\wedge d \text{ ptr } \text{vs.}$ 
   $(\text{ptr}, d) \in \text{stack-allocs } n \mathcal{S} \text{ TYPE}'a (\text{heap-typing } s) \implies$ 
   $\text{vs} \in \text{init } s \implies$ 
   $\text{length } \text{vs} = n \implies$ 
   $\forall i \in \{0..<n\}. r \ s \ (\text{ptr} +_p \text{int } i) = \text{ZERO}'a \implies$ 
   $\forall i \in \{0..<n\}. \text{root-ptr-valid } d \ (\text{ptr} +_p \text{int } i) \implies$ 
   $\text{lvp-distinct } ((d, \text{ptr-val } \text{ptr}, n, \text{size-td } (\text{typ-uinfo-t } \text{TYPE}'a))) \# (\text{map } (\lambda (v,$ 
sz). (heap-typing s, v, 1, sz)) params)  $\implies$ 
   $f \ \text{ptr} \cdot (\text{stack-ptr-acquire } n \ \text{vs } \text{ptr } d \ s) \ \{\!| \ \lambda r \ u. Q \ r \ (\text{stack-ptr-release } n \ \text{ptr } u)$ 
 $\}\!|$ 
shows assume-with-fresh-stack-ptr n init f · s  $\{\!| \ Q \!\}$ 
apply (rule runs-to-with-fresh-stack-ptr)
apply (rule assms(5))
apply (assumption)+
apply (rule lvp-distinct-singleton-local-with-params)
apply (simp add: assms)
by (drule bspec [where  $x=0$ ], simp, rule assms(1), simp add: ptr-add-def) (rule
assms) $\+$ 

```

lemma *lvp-distinct-add-stack-allocs*:

```

fixes  $p::'a \ \text{ptr}$ 
assumes  $(p, d') \in \text{stack-allocs } n \mathcal{S} \text{ TYPE}'a (\text{fst } (\text{hd } xs))$ 

```

```

assumes  $n > 0$ 
assumes root-ptr-valid  $d' p$ 
assumes lvp-distinct  $xs$ 
shows lvp-distinct  $((d', ptr-val\ p, n, size-td\ (typ-uinfo-t\ TYPE('a)))\#xs)$ 
unfolding lvp-distinct-def
apply (clarsimp simp: case-prod-beta distinct-spans-def)
apply (intro conjI, clarsimp)
subgoal for  $d\ v\ n\ t$ 
  using assms(4) apply (clarsimp simp: lvp-distinct-def)
  apply (rule disjnt-subset1 [where X=stack-free d])
  subgoal by (auto simp: disjnt-comm [simplified disjnt-def] disjnt-def)[1]
  apply (rule subset-trans)
  apply (simp add: size-of-fold)
  apply (rule stack-allocs-stack-subset-stack-free [OF assms(1), simplified])
  apply (drule sorted-wrt-hd-min [rotated], clarsimp)
  by (drule(1) bspec, auto)
subgoal using assms(4) by (auto simp: lvp-distinct-def distinct-spans-def)
subgoal
  using assms(4) apply (clarsimp simp: lvp-distinct-def)
  apply (drule(1) bspec, clarsimp)
  apply (rule disjoint-no-subset)
  apply (rule fresh-ptrs-stack-free-disjunct [OF assms(1)], simp add: assms(2))
intvl-non-zero-non-empty
  apply (rule subset-trans)
  apply (rule stack-allocs-stack-subset-stack-free [OF assms(1)])
  apply (drule sorted-wrt-hd-min [rotated], clarsimp)
  by (drule (1) bspec, auto)
subgoal
  using assms(4) by (clarsimp simp: lvp-distinct-def)
apply (rule ballI)
subgoal for  $x$ 
  apply (cases x; simp)
  subgoal for  $d\ v'\ t'$ 
    apply (rule subset-trans)
    apply (rule ssubst [OF stack-free-stack-allocs [OF assms(1)]])
    apply (rule Diff-subset)
    using assms(4) apply (clarsimp simp: lvp-distinct-def)
    apply (drule sorted-wrt-hd-min [rotated], clarsimp)
    by (drule(1) bspec, auto)
  done
subgoal using assms(4) by (auto simp: lvp-distinct-def)
subgoal
  apply (simp add: size-of-fold disjnt-def)
  by (rule fresh-ptrs-stack-free-disjunct [OF assms(1), simplified])
subgoal by (simp add: size-of-fold)
subgoal using assms(4) by (auto simp: lvp-distinct-def case-prod-beta)
done

```

lemma *runs-to-add-local-variable*:

```

assumes
  lwp-distinct xs
  heap-typing s = fst (hd xs)
  n > 0
   $\bigwedge d \text{ ptr } vs.$ 
    (ptr, d)  $\in$  stack-allocs n  $\mathcal{S}$  TYPE('a) (heap-typing s)  $\implies$ 
    vs  $\in$  init s  $\implies$ 
    length vs = n  $\implies$ 
     $\forall i \in \{0..<n\}. r \ s \ (\text{ptr} +_p \ \text{int } i) = \text{ZERO}('a) \implies$ 
     $\forall i \in \{0..<n\}. \text{root-ptr-valid } d \ (\text{ptr} +_p \ \text{int } i) \implies$ 
    lwp-distinct ((d, ptr-val ptr, n, size-td (typ-uinfo-t TYPE('a)))#xs)  $\implies$ 
    (f ptr)  $\cdot$  (stack-ptr-acquire n vs ptr d s)  $\{\!|$   $\lambda r \ u. Q \ r \ (\text{stack-ptr-release } n \ \text{ptr}$ 
u)  $\!\}$ 
shows (assume-with-fresh-stack-ptr n init f)  $\cdot$  s  $\{\!|$  Q  $\!\}$ 
apply (rule runs-to-with-fresh-stack-ptr)
apply (rule assms(4))
apply (assumption)+
apply (rule lwp-distinct-add-stack-allocs)
apply (simp add: assms)
apply (rule assms(3))
by (drule bspec [where x=0], simp, rule assms(3), simp add: ptr-add-def) (rule
assms(1))

end

end

```

```

theory AutoCorres
imports
  LocalVarExtract
  AutoCorresSimpset
  Polish
  Runs-To-VCG-StackPointer

```

```

keywords
  autocorres
  init-autocorres
  final-autocorres:: thy-decl
begin

```

```

no-syntax -Lab:: 'a bexp  $\Rightarrow$  ('a,'p,'f) com  $\Rightarrow$  bdy
  (-./- [1000,71] 81) — avoid syntax conflict with f  $\cdot$  s  $\{\!|$  Q  $\!\}$ 

```

```

declare word-neq-0-conv [simp del]
declare neq0-conv [simp del]

```


declare *fun-upd-apply*[*simp del*]
declare *fun-upd-same* [*simp add*]
lemma *o-const-simp*[*simp*]: $(\lambda x. C) \circ f = (\lambda x. C)$
by (*simp add: o-def*)

lemma *corresTA-trivial*: *corresTA* $(\lambda-. True)$ $(\lambda x. x)$ $(\lambda x. x)$ *A A*
apply (*auto intro: corresXF-I simp add: corresTA-def*)
done

lemma *L2Tcorres-trivial-from-in-out-parameters*:
IOcorres P Q st rx ex A C \implies *L2Tcorres id A A*
by (*rule L2Tcorres-id*)

lemma *L2Tcorres-trivial-from-local-var-extract*:
L2corres st rx ex P A C \implies *L2Tcorres id A A*
by (*rule L2Tcorres-id*)

lemma *corresTA-trivial-from-heap-lift*:
L2Tcorres st A C \implies *corresTA* $(\lambda-. True)$ $(\lambda x. x)$ $(\lambda x. x)$ *A A*
by (*rule corresTA-trivial*)

lemma *corresXF-from-L2-call*:
L2-call c-WA emb ns = A \implies *corresXF* $(\lambda s. s)$ $(\lambda rv s. rv)$ $(\lambda r t. emb r)$ $(\lambda-. True)$ *A c-WA*
unfolding *L2-call-def* *corresXF-refines-conv*
apply (*auto simp add: refines-def-old reaches-map-value map-exn-def split: xval-splits*)[1]
by (*smt (z3) Exn-neq-Result Result-eq-Result the-Exn-Exn(1)*)

definition *ac-corres' exn st check-termination AF* Γ *rx ex G* \equiv
 $\lambda A B. \forall s. (G s \wedge \text{succeeds } A (st s)) \longrightarrow$
 $(\forall t. \Gamma \vdash \langle B, Normal s \rangle \Rightarrow t \longrightarrow$
 (case *t* of
 Normal s' $\Rightarrow (Result (rx s'), st s') \in \text{outcomes } (run A (st s))$
 | *Abrupt s'* $\Rightarrow (exn (ex s'), st s') \in \text{outcomes } (run A (st s))$
 | *Fault e* $\Rightarrow e \in AF$
 | - $\Rightarrow False$))
 $\wedge (\text{check-termination} \longrightarrow \Gamma \vdash B \downarrow Normal s)$

lemma *ac-corres'-nd-monad*:
assumes *ac: ac-corres st check-termination AF* Γ *rx ex G B C*
assumes *refines: $\bigwedge s. refines B A s s$* (*rel-prod rel-liftE (=)*)
shows *ac-corres'* $(\lambda-. Exception (default::unit))$ *st check-termination AF* Γ *rx ex G A C*
apply (*simp add: ac-corres'-def*)[1]

```

apply (intro conjI allI impI)
subgoal
  using assms
  apply (auto simp add: ac-corres-def refines-def-old split: xstate.splits) [1]
  apply (metis reaches-def rel-liftE-Result-Result-iff)
  by (metis Exn-neq-Result rel-liftE-def)
apply (elim conjE)
subgoal premises prems for s
proof –
  from refines [simplified refines-def-old, rule-format, OF prems (3)] have succeeds B (st s) by blast
  with prems(2) have G s ∧ succeeds B (st s)
  by (auto simp add: succeeds-def)
  from ac [simplified ac-corres-def, rule-format, OF this] prems(1)
  show ?thesis
  by blast
qed
done

```

lemma *refines-spec-rel-Nonlocal-conv*:

```

shows refines f g s t (rel-prod (rel-xval rel-Nonlocal (=)) (=))
   $\longleftrightarrow$  refines f (map-value (map-exn Nonlocal) g) s t (rel-prod (=) (=))
apply (simp add: refines-def-old reaches-map-value rel-xval.simps map-exn-def
  split: xval-splits)
apply (intro iffI)
  apply (metis Exn-eq-Exn Result-eq-Result Result-neq-Exn rel-Nonlocal-conv)
apply (simp add: rel-Nonlocal-def)
apply clarsimp
subgoal for r s
  apply (erule-tac x=r in allE)
  apply (erule-tac x=s in allE)
  by (smt (verit, best) Exn-def c-exntype.case(5) default-option-def
  exception-or-result-cases not-Some-eq)
done

```

lemmas *refines-eq-convs = refines-spec-rel-Nonlocal-conv sum.rel-eq rel-xval-eq Relation.eq-OO*

lemma *ac-corres'-exception-monad*:

```

assumes ac: ac-corres st check-termination AF  $\Gamma$  rx ex G B C
assumes refines:  $\bigwedge s. refines B A s s (rel-prod (=) (=))$ 
shows ac-corres' Exn st check-termination AF  $\Gamma$  rx ex G A C
term map-value (map-exn Nonlocal) A
apply (simp add: ac-corres'-def, intro allI impI conjI)
subgoal
  using assms
  by (auto simp add: refines-def-old reaches-map-value ac-corres-def
  map-exn-def rel-sum.simps rel-Nonlocal-def split: xstate.splits c-exntype.splits)
  (metis reaches-def)+

```

```

apply (elim conjE)
subgoal premises prems for s
proof –
  from refines [simplified refines-def-old, rule-format, OF prems (3)] have succeeds B (st s) by blast
  with prems(2) have G s ∧ succeeds B (st s)
  by (auto simp add: succeeds-def)
  from ac [simplified ac-corres-def, rule-format, OF this] prems(1)
  show ?thesis
  by blast
qed
done

```

lemma *ac-corres-chain*:

```

[[ L1corres check-termination Gamma c-L1 c;
   L2corres st-L2 rx-L2 ex-L2 P-L2 c-L2 c-L1;
   L2Tcorres st-HL c-HL c-L2;
   corresTA P-WA rx-WA ex-WA c-WA c-HL;
   L2-call c-WA emb ns = A
]] ⇒
ac-corres (st-HL o st-L2) check-termination {AssumeError, StackOverflow} Gamma
(rx-WA o rx-L2) (emb o ex-WA o ex-L2) (P-L2 and (P-WA o st-HL o st-L2)) A c

```

```

apply (rule ccorresE-corresXF-merge)
  apply (unfold L1corres-alt-def)
  apply assumption
  apply (unfold L2corres-def L2Tcorres-def corresTA-def)
  apply (drule corresXF-from-L2-call)

  apply ((drule (1) corresXF-corresXF-merge)+, assumption)
  apply (clarsimp simp: L2-call-def L2-defs)

  apply simp
  apply clarsimp
  apply clarsimp
done

```

lemma *ac-corres-chain-sim-nd-monad*:

```

[[ L1corres check-termination Gamma c-L1 c;
   L2corres st-L2 rx-L2 ex-L2 P-L2 c-L2 c-L1;
   IOcorres P-IO Q-IO st-IO rx-IO ex-IO c-IO c-L2;
   L2Tcorres st-HL c-HL c-IO;
   corresTA P-WA rx-WA ex-WA c-WA c-HL;
    $\bigwedge s. \text{refines } c\text{-WA } A \text{ } s \text{ } s \text{ } (\text{rel-prod rel-liftE } (=))$ 
]] ⇒
ac-corres' (λ-. Exception (default::unit)) (st-HL o st-IO o st-L2) check-termination
{AssumeError, StackOverflow} Gamma
(λs. (rx-WA o (λv. rx-IO v (st-L2 s)) o rx-L2) s)
(λs. (ex-WA o (λe. ex-IO e (st-L2 s)) o ex-L2) s)

```

```

(P-L2 and (P-IO o st-L2) and (P-WA o st-HL o st-IO o st-L2)) A c
apply (rule ac-corres'-nd-monad)
apply (rule ccorresE-corresXF-merge)
  apply (unfold L1corres-alt-def)
  apply assumption
  apply (unfold L2corres-def L2Tcorres-def corresTA-def IOcorres-def)
  apply (drule corresXF-post-to-corresXF)
  apply ((drule (1) corresXF-corresXF-merge)+, assumption)
  apply (clarsimp simp: L2-call-def L2-defs)
apply simp
apply clarsimp
apply clarsimp
apply assumption
done

```

lemma *ac-corres-chain-sim-exception-monad*:

```

[[ L1corres check-termination Gamma c-L1 c;
   L2corres st-L2 rx-L2 ex-L2 P-L2 c-L2 c-L1;
   IOcorres P-IO Q-IO st-IO rx-IO ex-IO c-IO c-L2;
   L2Tcorres st-HL c-HL c-IO;
   corresTA P-WA rx-WA ex-WA c-WA c-HL;
    $\wedge$ s. refines c-WA A s s (rel-prod (=) (=))
]]  $\implies$ 
ac-corres' Exn (st-HL o st-IO o st-L2) check-termination {AssumeError, Stack-
Overflow} Gamma
  ( $\lambda$ s. (rx-WA o ( $\lambda$ v. rx-IO v (st-L2 s)) o rx-L2) s)
  ( $\lambda$ s. (ex-WA o ( $\lambda$ e. ex-IO e (st-L2 s)) o ex-L2) s)
  (P-L2 and (P-IO o st-L2) and (P-WA o st-HL o st-IO o st-L2)) A c
apply (rule ac-corres'-exception-monad)
apply (rule ccorresE-corresXF-merge)
  apply (unfold L1corres-alt-def)
  apply assumption
  apply (unfold L2corres-def L2Tcorres-def corresTA-def IOcorres-def)
  apply (drule corresXF-post-to-corresXF)
  apply ((drule (1) corresXF-corresXF-merge)+, assumption)
  apply (clarsimp simp: L2-call-def L2-defs)
apply simp
apply clarsimp
apply clarsimp
apply assumption
done

```

lemmas *ac-corres-chain-sims = ac-corres-chain-sim-nd-monad ac-corres-chain-sim-exception-monad*

definition *FUNCTION-BODY-NOT-IN-INPUT-C-FILE* \equiv fail

lemma [*polish*]:

guard (λ s. UNDEFINED-FUNCTION) = FUNCTION-BODY-NOT-IN-INPUT-C-FILE

$(FUNCTION-BODY-NOT-IN-INPUT-C-FILE \gg= m) = FUNCTION-BODY-NOT-IN-INPUT-C-FILE$
 $unknown \gg= (\lambda x. FUNCTION-BODY-NOT-IN-INPUT-C-FILE) = FUNCTION-$
 $TION-BODY-NOT-IN-INPUT-C-FILE$
 $liftE FUNCTION-BODY-NOT-IN-INPUT-C-FILE = FUNCTION-BODY-NOT-IN-INPUT-C-FILE$
by (rule spec-monad-ext,
 simp add: run-bind run-guard UNDEFINED-FUNCTION-def FUNCTION-BODY-NOT-IN-INPUT-C-FIL

lemmas *ac-statistics-rewrites* =

L1-seq-def

L2-defs'

There might be unexpected simplification 'unfolding' of *id* due to eta-expansion: *id* might be expanded (e.g. by resolution to) *id*. Now the simp rule *id ?x = ?x* triggers and rewrites it $\lambda x. x$. Folding this back to *id* might help in those cases.

named-theorems

l1-corres **and** *l2-corres* **and** *io-corres* **and** *hl-corres* **and** *wa-corres* **and** *ts-corres*
and *ac-corres* **and**
known-function-corres **and** *known-function*

lazy-named-theorems

l1-def **and** *l2-def* **and** *io-def* **and** *hl-def* **and** *wa-def* **and** *ts-def* **and** *ac-def*
named-theorems

heap-update-syntax

lemma *fold-id*: $(\lambda x. x) = id$
by (simp add: *id-def*)

lemma *fold-id-unit*: $(\lambda-. ()) = id$
by (simp add: *id-def*)

ML-file *lib/set.ML*

ML-file *trace-antiquote.ML*

ML-file *function-info.ML*

ML-file *program-info.ML*

ML-file *autocorres-trace.ML*

ML-file *autocorres-options.ML*

ML-file *autocorres-data.ML*

ML-file *autocorres-util.ML*

ML-file *exception-rewrite.ML*
ML-file *simpl-conv.ML*

ML-file *prog.ML*
ML-file *l2-opt.ML*
ML-file *local-var-extract.ML*

context *globals-stack-heap-state*
begin
ML-file *in-out-parameters.ML*

declaration *⟨fn phi =>*
 In-Out-Parameters.Data.add (In-Out-Parameters.operations phi) phi⟩
end

ML-file *record-utils.ML*
ML-file *heap-lift-base.ML*
ML-file *heap-lift.ML*

ML-file *word-abstract.ML*
ML-file *pretty-bound-var-names.ML*
ML-file *monad-convert.ML*

ML-file *type-strengthen.ML*

ML-file *autocorres.ML*

ML *⟨*
 Outer-Syntax.command @{command-keyword init-autocorres}
 Initialise AutoCorres
 (AutoCorres.init-autocorres-parser >>
 (Toplevel.theory o (fn (opt, filename) => AutoCorres.do-init-autocorres opt
 filename true #> snd)))
⟩

ML *⟨*
 Outer-Syntax.command @{command-keyword autocorres}
 Abstract the output of the C parser into a monadic representation.
 (AutoCorres.autocorres-parser >>
 (Toplevel.theory o (fn (opt, filename) => AutoCorres.parallel-autocorres opt
 filename)))
⟩

>

ML <

```
Outer-Syntax.command @{command-keyword final-autocorres}
  Finalise Autocorres
  (AutoCorres.final-autocorres-parser >>
   (Toplevel.theory o (fn filename => AutoCorres.final-autocorres-cmd filename)))
>
```

setup <

```
let
  fun fresh-var maxidx (n, T) = Var ((- ^ n, maxidx + 1), T)
  fun head-type t = t |> HOLogic.dest-Trueprop |> head-of |> fastype-of
  fun get-maxidx maxidx ts =
    if maxidx < 0
    then fold (curry Int.max) (map maxidx-of-term ts) 0
    else maxidx
```

— Note that the *mk-pattern* functions serve two purposes. When adding a rule into the term net we insert Var's for the positions that we want to synthesise. The concrete program '?C' serves as index into the net and is unmodified. Note that `Utils.open_beta_norm_eta` should actually be the identity (with respect to matching) when applied to the rules.

Before querying the term-net with a concrete subgoal we also use *mk-pattern*. Here `Utils.open_beta_norm_eta` is essential to make term-net-retrieval for matching (instead of 'unification') work, as it gets rid of beta-eta artefacts in the goal that are generated during recursive application of the rules.

```
fun mk-corresTA-pattern maxidx (t as @{term-pat Trueprop (corresTA ?P ?rx
?ex ?A ?C)}) =
```

```
  let
    val mi = get-maxidx maxidx [P, rx, ex, A, C]
    val T = head-type t
    val [PT, rxT, exT, AT, -] = binder-types T
    val [P', rx', ex', A'] = map (fresh-var mi) [(P, PT), (rx, rxT), (ex, exT), (A,
AT)]
    val pat = HOLogic.mk-Trueprop (Const (@{const-name corresTA}, T) $
      P' $ rx' $ ex' $ A' $ Utils.open-beta-norm-eta C)
```

```
  in pat end
```

```
fun mk-abstract-val-pattern maxidx (t as (@{term-pat Trueprop (abstract-val ?P
?A ?f ?C)})) =
```

```
  let
    val mi = get-maxidx maxidx [P, A, f, C]
    val T = head-type t
    val [PT, AT, fT, -] = binder-types T
    val [P', A', f'] = map (fresh-var mi) [(P, PT), (A, AT), (f, fT)]
    val pat = HOLogic.mk-Trueprop (Const (@{const-name abstract-val}, T) $
```

```

      P' $ A' $ f' $ Utils.open-beta-norm-eta C)
in pat end

fun mk-valid-typ-abs-fn-pattern maxidx (t as (@{term-pat Trueprop (valid-typ-abs-fn
?P ?Q ?A ?C)})) =
let
  val mi = get-maxidx maxidx [P, Q, A, C]
  val T = head-type t
  val [PT, QT, AT, CT] = binder-types T
  val [P', Q', A', C'] = map (fresh-var mi) [(P, PT), (Q, QT), (A, AT), (C,
CT)]
  val pat = HOLogic.mk-Trueprop (Const (@{const-name valid-typ-abs-fn}, T)
$
      P' $ Q' $ A' $ C')
in pat end

fun mk-introduce-typ-abs-fn-pattern maxidx (t as (@{term-pat Trueprop (introduce-typ-abs-fn
?f)})) =
let
  val mi = get-maxidx maxidx [f]
  val T = head-type t
  val [fT] = binder-types T
  val [f'] = map (fresh-var mi) [(f, fT)]
  val pat = HOLogic.mk-Trueprop (Const (@{const-name introduce-typ-abs-fn},
T) $ f')
in pat end

fun mk-id-pattern - t = t

fun mk-abs-expr-pattern maxidx (t as (@{term-pat Trueprop (abs-expr ?st ?P ?A
?C)})) =
let
  val mi = get-maxidx maxidx [st, P, A, C]
  val T = head-type t
  val [stT, PT, AT, -] = binder-types T
  val [st', P', A'] = map (fresh-var mi) [(st, stT), (P, PT), (A, AT)]
  val pat = HOLogic.mk-Trueprop (Const (@{const-name abs-expr}, T) $
      st' $ P' $ A' $ Utils.open-beta-norm-eta C)
in pat end

fun mk-abs-guard-pattern maxidx (t as (@{term-pat Trueprop (abs-guard ?st ?A
?C)})) =
let
  val mi = get-maxidx maxidx [st, A, C]
  val T = head-type t
  val [stT, AT, -] = binder-types T
  val [st', A'] = map (fresh-var mi) [(st, stT), (A, AT)]
  val pat = HOLogic.mk-Trueprop (Const (@{const-name abs-guard}, T) $
      st' $ A' $ Utils.open-beta-norm-eta C)

```



```

in pat end

fun mk-L2Tcorres-pattern maxidx (t as (@{term-pat Trueprop (L2Tcorres ?st ?A
?C)})) =
let
  val mi = get-maxidx maxidx[st, A, C]
  val T = head-type t
  val [stT, AT, -] = binder-types T
  val [st', A'] = map (fresh-var mi) [(st, stT), (A, AT)]
  val pat = HOLogic.mk-Trueprop (Const (@{const-name L2Tcorres}, T) $
    st' $ A' $ Utils.open-beta-norm-eta C)
in pat end

fun mk-abs-modifies-pattern maxidx (t as (@{term-pat Trueprop (abs-modifies ?st
?P ?A ?C)})) =
let
  val mi = get-maxidx maxidx [st, P, A, C]
  val T = head-type t
  val [stT, PT, AT, -] = binder-types T
  val [st', P', A'] = map (fresh-var mi) [(st, stT), (P, PT), (A, AT)]
  val pat = HOLogic.mk-Trueprop (Const (@{const-name abs-modifies}, T) $
    st' $ P' $ A' $ Utils.open-beta-norm-eta C)
in pat end

fun mk-struct-rewrite-guard-pattern maxidx (t as (@{term-pat Trueprop (struct-rewrite-guard
?A ?C)})) =
let
  val mi = get-maxidx maxidx [A, C]
  val T = head-type t
  val [AT, -] = binder-types T
  val [A'] = map (fresh-var mi) [(A, AT)]
  val pat = HOLogic.mk-Trueprop (Const (@{const-name struct-rewrite-guard},
T) $
    A' $ Utils.open-beta-norm-eta C)
in pat end

fun mk-struct-rewrite-expr-pattern maxidx (t as (@{term-pat Trueprop (struct-rewrite-expr
?P ?A ?C)})) =
let
  val mi = get-maxidx maxidx[P, A, C]
  val T = head-type t
  val [PT, AT, -] = binder-types T
  val [P', A'] = map (fresh-var mi) [(P, PT), (A, AT)]
  val pat = HOLogic.mk-Trueprop (Const (@{const-name struct-rewrite-expr},
T) $
    P' $ A' $ Utils.open-beta-norm-eta C)
in pat end

fun mk-struct-rewrite-modifies-pattern maxidx (t as (@{term-pat Trueprop (struct-rewrite-modifies

```

```

?P ?A ?C))) =
  let
    val mi = get-maxidx maxidx [P, A, C]
    val T = head-type t
    val [PT, AT, -] = binder-types T
    val [P', A'] = map (fresh-var mi) [(P, PT), (A, AT)]
    val pat = HOLogic.mk-Trueprop (Const (@{const-name struct-rewrite-modifies},
T) $
      P' $ A' $ Utils.open-beta-norm-eta C)
  in pat end

fun mk-abs-spec-pattern maxidx (t as (@{term-pat Trueprop (abs-spec ?st ?P ?A
?C)})) =
  let
    val mi = get-maxidx maxidx [st, P, A, C]
    val T = head-type t
    val [stT, PT, AT, -] = binder-types T
    val [st', P', A'] = map (fresh-var mi) [(st, stT), (P, PT), (A, AT)]
    val pat = HOLogic.mk-Trueprop (Const (@{const-name abs-spec}, T) $
      st' $ P' $ A' $ Utils.open-beta-norm-eta C)
  in pat end

fun mk-heap-lift--wrap-h-val-pattern maxidx (t as (@{term-pat Trueprop (heap-lift--wrap-h-val
?X ?Y)})) =
  let
    val mi = get-maxidx maxidx [X, Y]
    val T = head-type t
    val [XT, YT] = binder-types T
    val [X', Y'] = map (fresh-var mi) [(X, XT), (Y, YT)]
    val pat = HOLogic.mk-Trueprop (Const (@{const-name heap-lift--wrap-h-val},
T) $ X' $ Y' )
  in pat end

in
Context.theory-map (fold WordAbstract.add-pattern [
  mk-corresTA-pattern,
  mk-abstract-val-pattern,
  mk-valid-typ-abs-fn-pattern,
  mk-introduce-typ-abs-fn-pattern,
  mk-id-pattern
]) #>
Context.theory-map (fold HeapLift.add-pattern [
  mk-abs-expr-pattern,
  mk-abs-guard-pattern,
  mk-L2Tcorres-pattern,
  mk-abs-modifies-pattern,
  mk-struct-rewrite-guard-pattern,
  mk-struct-rewrite-expr-pattern,
  mk-struct-rewrite-modifies-pattern,

```

```
    mk-abs-spec-pattern,  
    mk-heap-lift--wrap-h-val-pattern,  
    mk-id-pattern  
  )  
end  
  
>  
  
end
```

Part IV

Documentation

Chapter 25

Quickstart

JAPHETH LIM, ROHAN JACOB-RAO, DAVID GREENAWAY AND NORBERT SCHIRMER

25.1 Introduction

AutoCorres is a tool that attempts to simplify the formal verification of C programs in the Isabelle/HOL theorem prover. It allows C code to be automatically abstracted to produce a higher-level functional specification.

AutoCorres relies on the C-Parser [7] developed by Michael Norrish at NICTA. This tool takes raw C code as input and produces a translation in SIMPL [8], an imperative language written by Norbert Schirmer on top of Isabelle. AutoCorres takes this SIMPL code to produce a "monadic" specification, which is intended to be simpler to reason about in Isabelle. The composition of these two tools (AutoCorres applied after the C-Parser) can then be used to reason about C programs.

This guide is written for users of Isabelle/HOL, with some knowledge of C, to get started proving properties of C programs. Using AutoCorres in conjunction with the verification condition generator (VCG) *runs-to-vcg*, one should be able to do this without an understanding of SIMPL nor all the details of the monadic representation produced by AutoCorres. We will see how this is possible in the next chapter.

25.2 A First Proof with AutoCorres

We will now show how to use these tools to prove correctness of some very simple C functions.

25.2.1 Two simple functions: min and max

Consider the following two functions, defined in a file `minmax.c`, which (we expect) return the minimum and maximum respectively of two unsigned integers.

```
unsigned min(unsigned a, unsigned b)
{
    if (a <= b) {
        return a;
    } else {
        return b;
    }
}

unsigned max(unsigned a, unsigned b)
{
    return UINT_MAX - (
        min(UINT_MAX - a, UINT_MAX - b));
}
```

It is easy to see that `min` is correct, but perhaps less obvious why `max` is correct. AutoCorres will hopefully allow us to prove these claims without too much effort.

25.2.2 Invoking the C-parser

As mentioned earlier, AutoCorres does not handle C code directly. The first step is to apply the C-Parser by using **install-C-file** to obtain a SIMPL translation. We do this using the `install-C-file` command in Isabelle, as shown.

```
install-C-file sources/minmax.c
```

For every function in the C source file, the C-Parser generates a corresponding Isabelle definition. These definitions are placed in an Isabelle "locale", whose name matches the input filename. For our file `minmax.c`, the C-Parser will place definitions in the locale `minmax`.¹

For our purposes, we just have to remember to enter the appropriate locale before writing our proofs. This is done using the **context** keyword in Isabelle.

Let's look at the C-Parser's outputs for `min` and `max`, which are contained in the theorems `min_body_def` and `max_body_def`. These are simply definitions of the generated names `min_body` and `max_body`. We can also see here how our work is wrapped within the `minmax` context.

```
context minmax-simpl begin
```

¹The C-parser uses locales to avoid having to make certain assumptions about the behaviour of the linker, such as the concrete addresses of symbols in your program.

thm *min-body-def*

min-body \equiv

TRY

lvar-nondet-init ($\lambda upd. (min.ret'::32\ word):=_{\mathcal{L}}\ upd$);;

IF $\text{'}(min.a::32\ word) \leq \text{'}(min.b::32\ word)$ *THEN*

creturn global-exn-var'-'-update ($\lambda upd. (min.ret'::32\ word):=_{\mathcal{L}}\ upd$)
($\lambda s. s \cdot_{\mathcal{L}} (min.a::32\ word)$)

ELSE

creturn global-exn-var'-'-update ($\lambda upd. (min.ret'::32\ word):=_{\mathcal{L}}\ upd$)
($\lambda s. s \cdot_{\mathcal{L}} (min.b::32\ word)$)

FI;;

Guard DontReach {} *SKIP*

CATCH ccatchreturn global-exn-var'-'-

END

thm *max-body-def*

max-body \equiv

TRY

lvar-nondet-init ($\lambda upd. (max.ret'::32\ word):=_{\mathcal{L}}\ upd$);;

$\text{'}(max.ret'unsigned'1::32\ word) := CALL_e\ minmax.min(-\ 1 -$
 $\text{'}(max.a::32\ word),$

$-\ 1 - \text{'}(max.b::32\ word))$);;

creturn global-exn-var'-'-update ($\lambda upd. (max.ret'::32\ word):=_{\mathcal{L}}\ upd$)
($\lambda s. -\ 1 - (s \cdot_{\mathcal{L}} (max.ret'unsigned'1::32\ word))$);;

Guard DontReach {} *SKIP*

CATCH ccatchreturn global-exn-var'-'-

END

end

The definitions above show us the SIMPL generated for each of the functions; we can see that C-parser has translated **min** and **max** very literally and no detail of the C language has been omitted. For example:

- C **return** statements have been translated into exceptions which are caught at the outside of the function's body;
- *Guard* statements are used to ensure that behaviour deemed 'undefined' by the C standard does not occur. In the above functions, we see that a guard statement is emitted that ensures that program execution does not hit the end of the function, ensuring that we always return a value (as is required by all non-void functions).
- Function parameters are modelled as local variables, which are setup prior to a function being called. Return variables are also modelled as local variables, which are then read by the caller.

While a literal translation of C helps to improve confidence that the translation is sound, it does tend to make formal reasoning an arduous task.

25.2.3 Invoking AutoCorres

Now let's use AutoCorres to simplify our functions. This is done using the **autocorres** command, in a similar manner to the **install-C-file** command:

```
autocorres minmax.c
```

AutoCorres produces a definition in the **minmax** locale for each function body produced by the C parser. For example, our **min** function is defined as follows:

```
context minmax-all-corres begin
thm min'-def
```

```
min' ?a ?b ≡ if ?a ≤ ?b then ?a else ?b
```

Each function's definition is named identically to its name in C, but with a prime mark (') appended. For example, our functions **min** above was named *min'*, while the function **foo_Bar** would be named *foo-Bar'*.

AutoCorres does not require you to trust its translation is sound, but also emits a *correspondence* or *refinement* proof, as follows:

```
thm min'-ac-corres
```

```
ac-corres' (λ-. Exception ()) (lift-global-heap ∘ (λs. s) ∘ globals) True
{AssumeError, StackOverflow} Γ (λs. s ·ℒ (min.ret'::32 word))
global-exn-var'-'
((λs. s ·ℒ (min.a::32 word) = ?a') and
 (λs. s ·ℒ (min.b::32 word) = ?b') and
 (λ-. True) ∘ globals and
 (λx. abs-var ?a id ?a' ∧ abs-var ?b id ?b') ∘ lift-global-heap ∘ (λs. s) ∘
 globals)
(return (min' ?a ?b)) (Call minmax.min)
```

Informally, this theorem states that, assuming the abstract function *min'* can be proven to not fail for a particular input, then for the associated input, the concrete C SIMPL program also will not fault, will always terminate, and will have a corresponding end state to the generated abstract program.

For more technical details, see [2] and [3] or [chapter 26](#).

25.2.4 Verifying `min`

In the abstracted version of `min'`, we can see that AutoCorres has simplified away the local variable reads and writes in the C-parser translation of `min`, simplified away the exception throwing and handling code, and also simplified away the unreachable guard statement at the end of the function. In fact, `min'` has been simplified to the point that it exactly matches Isabelle's built-in function `min`:

thm `min-def`

`min ?a ?b = (if ?a ≤ ?b then ?a else ?b)`

So, verifying `min'` (and by extension, the C function `min`) should be easy:

lemma `min'-is-min`: `min' a b = min a b`
unfolding `min-def min'-def`
by `(rule refl)`

25.2.5 Verifying `max`

Now we also wish to verify that `max'` implements the built-in function `max`. `min'` was nearly too simple to bother verifying, but `max'` is a bit more complicated. Let's look at AutoCorres' output for `max`:

thm `max'-def`

`max' ?a ?b ≡ - 1 - min' (- 1 - ?a) (- 1 - ?b)`

At this point, you might still doubt that `max'` is indeed correct, so perhaps a proof is in order. The basic idea is that subtracting from `UINT_MAX` flips the ordering of unsigned ints. We can then use `min'` on the flipped numbers to compute the maximum.

The next lemma proves that subtracting from `UINT_MAX` flips the ordering. To prove it, we convert all words to `int`'s, which does not change the meaning of the statement.

lemma `n1-minus-flips-ord`:
`((a :: word32) ≤ b) = ((-1 - a) ≥ (-1 - b))`
apply `(subst word-le-def)+`
apply `(subst word-n1-ge [simplified uint-minus-simple-alt])+`

Now that our statement uses `int`, we can apply Isabelle's built-in `arith` method.

apply `arith`
done

And now for the main proof:

lemma `max'-is-max`: `max' a b = max a b`

```

unfolding max'-def min'-def max-def
using n1-minus-flips-ord
by force

```

end

In the next section, we will see how to use AutoCorres to simplify larger, more realistic C programs.

25.3 More Complex Functions with AutoCorres

In the previous section we saw how to use the C-Parser and AutoCorres to prove properties about some very simple C programs.

Real life C programs tend to be far more complex however; they read and write to local variables, have loops and use pointers to access the heap. In this section we will look at some simple programs which use loops and access the heap to show how AutoCorres can allow such constructs to be reasoned about.

25.3.1 A simple loop: `mult_by_add`

Our C function `mult_by_add` implements a multiply operation by successive additions:

```

unsigned mult_by_add(unsigned a, unsigned b)
{
    unsigned result = 0;
    while (a > 0) {
        result += b;
        a--;
    }
    return result;
}

```

We start by translating the program using the C parser and AutoCorres, and entering the generated locale `mult_by_add`.

```

install-C-file sources/mult-by-add.c
autocorres [ts-rules = nondet] mult-by-add.c

```

The C parser produces the SIMPL output as follows:

```

thm mult-by-add-body-def

mult-by-add-body ≡
TRY
  lvar-nondet-init ( $\lambda upd. (ret'::32\ word) :=_{\mathcal{L}} upd$ );;

```

```

('result::32 word) ::= SCAST(32 signed → 32) 0;;
(WHILE SCAST(32 signed → 32) 0 < '(a::32 word) DO
  '(result::32 word) ::= '(result::32 word) + '(b::32 word);;
  '(a::32 word) ::= '(a::32 word) - SCAST(32 signed → 32) 1
OD);;
creturn global-exn-var'-!-update (λupd. (ret'::32 word):=ℒ upd)
(λs. s ·ℒ (result::32 word)));;
Guard DontReach {} SKIP
CATCH ccatchreturn global-exn-var'-!-
END

```

Which is abstracted by AutoCorres to the following:

thm *mult-by-add'-def*

```

mult-by-add' ?a ?b ≡ do {
  (a, result) ← whileLoop (λ(a, result) s. 0 < a)
    (λ(a, result). return (a - 1, result + ?b))
  (?a, 0);
  return result
}

```

In this case AutoCorres has abstracted `mult_by_add` into a *monadic* form. Monads are a pattern frequently used in functional programming to represent imperative-style control-flow; an in-depth introduction to monads can be found elsewhere.

The monads used by AutoCorres in this example is a *non-deterministic state monad*; program state is implicitly passed between each statement, and results of computations may produce more than one (or possibly zero) results².

The HOL type is called specification monad: $(\prime e, \prime a, \prime s)$ *spec-monad*, where

- $\prime e$ type for exception / error results,
- $\prime r$ type of the result and
- $\prime s$ type of the state.

When $\prime e$ is instantiated with the unit type *unit*, there is an abbreviation $(\prime a, \prime s)$ *res-monad*, for *result monad*. When it is instantiated with a proper type we have an abbreviation $(\prime e, \prime a, \prime s)$ *exn-monad*, for *exception monad*.

To uniformly model results and exceptions as values the type $(\prime e, \prime a)$ *exception-or-result* is used. It is constructed as a sum type where

²Non-determinism becomes useful when modelling hardware, for example, where the exact results of the hardware cannot be determined ahead of time.

we exclude a default value from the exception type $'e$. So when $'e$ is instantiated with *unit* the type is isomorphic to the result type $'a$. Type *unit* is only inhabited by the unique unit value $()$ which is also default value. This instance is abbreviated with $'a$ *val*. For proper exceptions we instantiate $'e$ with an option type $'x$ *option*. This instance is abbreviated with $('x, 'a)$ *xval*.

Constructors and pattern matching:

- $('e, 'a)$ *exception-or-result*: *Exception e, Result v*, and pattern matching $\lambda x. \text{case } x \text{ of } \text{Exception } e \Rightarrow f \ e \mid \text{Result } v \Rightarrow g \ v$
- $'a$ *val*: *Result v*, with pattern matching $\lambda Res \ v. \ g \ v$
- $('e, 'a)$ *xval*: *Exn e, Result v*, and pattern matching $\lambda x. \text{case } x \text{ of } \text{Exn } e \Rightarrow f \ e \mid \text{Result } v \Rightarrow g \ v$

This encoding allows us to give a uniform definition of the monadic $(>>=)$ which works for the generic specification monad and thus also for its instances the result and the exception monad.

'Hoare Triples'

The bulk of *mult-by-add'* is wrapped inside the *whileLoop monad combinator*, which is really just a fancy way of describing the method used by *AutoCorres* to encode (potentially non-terminating) loops using monads.

If we want to describe the behaviour of this program, we can use Hoare logic as follows:

$$P \ s \Longrightarrow \text{mult-by-add}' \ a \ b \cdot s \ \{\!\! \{ \} \!\!\} \ Q$$

This predicate states that, assuming P holds on the initial program state, if we execute *mult-by-add'* $a \ b$, then Q will hold on the final state of the program.

There are a few details: while P has type $'s \Rightarrow \text{bool}$ (i.e., takes a state and returns true if it satisfies the precondition), Q has an additional parameter $'r \Rightarrow 's \Rightarrow \text{bool}$. The additional parameter $'r$ is the *return value* of the function; so, in our *mult-by-add'* example, it will be the result of the multiplication. Note that the precondition is not part of the 'hoare-triple' but is an ordinary Isabelle/Isar assumption on the initial state s . That way we can directly use Isabelle/Isar to decompose the precondition and can also refer to the initial state within the postcondition. The foundational constant for this hoare triple is *runs-to* which means total correctness: we have to prove that the program terminates and that there is no undefined behaviour (all guards must hold). There is also a weaker notion for partial correctness *runs-to-partial* with syntax $f \cdot s \ ?\!\! \{ \} \ Q$.

For example one useful property to prove on our program would be:

mult-by-add' $a \cdot b \cdot s \{ \lambda Res \ r \ -. \ r = a * b \}$

That is, for all possible input states, our `mult_by_add'` function returns the product of a and b .

Our proof of *mult-by-add'* could then proceed as follows:

lemma *mult-by-add-correct*:

mult-by-add' $a \cdot b \cdot s \{ \lambda Res \ r \ -. \ r = a * b \}$

Unfold abstracted definition

apply (*unfold mult-by-add'-def*)

Annotate the loop with an invariant and a measure, by adding a specialised rule to the verification condition generator.

supply *runs-to-whileLoop-res* [**where**

$I = \lambda(a', \text{result}) \ s. \ \text{result} = (a - a') * b$ **and**

$R = \text{measure}' (\lambda((a', \text{result}), s). \ a'), \ \text{runs-to-vcg}]$

Run the “verification condition generator”.

apply (*runs-to-vcg*)

Solve the program correctness goals.

apply (*simp add: field-simps*)

apply *unat-arith*

apply (*auto simp: field-simps not-less*)

done

The main tool is the proof method *runs-to-vcg*, a verification condition generator for monadic programs. It uses weakest precondition style rules which are collected in named theorems collection *runs-to-vcg*. In that collection there is no rule for *whileLoop*. The verification generator will stop there unless you specify a rule to use. This was done in the proof above by instantiating the rule *runs-to-whileLoop-res* with an invariant and a measure and by adding it to the verification condition generator via attribute *runs-to-vcg*. We finally discharge the remaining subgoals left from *runs-to-vcg* with standard proof tools of Isabelle.

In the next section, we will look at how we can use AutoCorres to verify a C program that reads and writes to the heap.

25.3.2 swap

Here, we use AutoCorres to verify a C program that reads and writes to the heap. Our C function, `swap`, swaps two words in memory:

```
void swap(unsigned *a, unsigned *b)
{
    unsigned t = *a;
```

```

    *a = *b;
    *b = t;
}

```

Again, we translate the program using the C parser and AutoCorres.

```

install-C-file sources/swap.c
autocorres [heap-abs-syntax, ts-rules = nondet] swap.c

```

Most heap operations in C programs consist of accessing a pointer. AutoCorres abstracts the global C heap by creating one heap for each type. (In our simple `swap` example, it creates only a `Word-32.word32` heap.) This makes verification easier as long as the program does not access the same memory as two different types.

There are other operations that are relevant to program verification, such as changing the type that occupies a given region of memory. AutoCorres will not abstract any functions that use these operations, so verifying them will be more complicated (but still possible).

The C parser expresses `swap` like this:

```

thm swap-body-def

```

```

swap-body ≡
TRY
  Guard C-Guard {c-guard '(a::32 word ptr)}
  ('(t::32 word) := h-val (hrs-mem 't-hrs) '(a::32 word ptr));;
  (Guard C-Guard {c-guard '(a::32 word ptr)}
   (Guard C-Guard {c-guard '(b::32 word ptr)}
    ('globals :=
     t-hrs-'update
     (hrs-mem-update
      (heap-update '(a::32 word ptr)
                   (h-val (hrs-mem 't-hrs) '(b::32 word ptr))))));;
   Guard C-Guard {c-guard '(b::32 word ptr)}
   ('globals :=
    t-hrs-'update
    (hrs-mem-update (heap-update '(b::32 word ptr) '(t::32 word))))))
CATCH ccatchreturn global-exn-var'-'
END

```

AutoCorres abstracts the function to this:

```

thm swap'-def

```

```

swap' ?a ?b ≡ do {
  guard (λs. IS-VALID(32 word) s ?a);
  guard (λs. IS-VALID(32 word) s ?b);
  t ← gets (λs. s[?a]);
  modify (λs. s[?a] := s[?b]);
  modify (λs. s[?b] := t)
}

```

There are some things to note:

The function contains guards (assertions) that the pointers `a` and `b` are valid. We need to prove that these guards never fail when verifying `swap`. Conversely, when verifying any function that calls `swap`, we need to show that the arguments are valid pointers.

We saw a monadic program in the previous section, but here the monad is actually being used to carry the program heap.

Now we prove that `swap` is correct. We use x and y to “remember” the initial values so that we can talk about them in the post-condition. The heap access syntax $s[a]$ is used to select the split heap *heap-w32* from state s at pointer a .³

```

lemma  $[[IS-VALID(32\ word)\ s\ a; s[a] = x; IS-VALID(32\ word)\ s\ b; s[b] = y]] \implies$ 
   $(swap'\ a\ b) \cdot s$ 
   $\{ \lambda\ s.\ s[a] = y \wedge s[b] = x \}$ 
apply (unfold swap'-def)
apply runs-to-vcg
apply clarsimp

```

The C parser and AutoCorres both model the C heap using functions, which takes a pointer to some object in memory. Heap updates are modelled using the functional update *fun-upd*:

```
?f(?a := ?b) = ( $\lambda x.\ if\ x = ?a\ then\ ?b\ else\ ?f\ x$ )
```

To reason about functional updates, we use the rule `fun_upd_apply`.

```

apply (simp add: fun-upd-apply)
done

```

Note that we have “only” proved that the function swaps its arguments. We have not proved that it does *not* change any other state. This is a typical *frame problem* with pointer reasoning. We can prove a more complete specification of `swap`:

```

lemma  $[[(\wedge x\ y\ s.\ P\ (s[a := x][b := y]) = P\ s);$ 
   $IS-VALID(32\ word)\ s\ a; s[a] = x; IS-VALID(32\ word)\ s\ b; s[b] = y; P\ s]]$ 
 $\implies$ 
   $(swap'\ a\ b) \cdot s$ 
   $\{ \lambda\ s.\ s[a] = y \wedge s[b] = x \wedge P\ s \}$ 
apply (unfold swap'-def)
apply runs-to-vcg
apply (clarsimp simp: fun-upd-apply)+
done

```

In other words, if predicate P does not depend on the inputs to `swap`, it will continue to hold.

Separation logic provides a more structured approach to this problem.

³For more details on the split heap model in autocorres see [section 26.21](#).

25.4 Command Options and Invocation

25.4.1 Session Structure

The supplied session structure has some peculiarities to accommodate the AFP. The AFP presents the documentation of the session that is named like the AFP-entry. This is `AutoCorres2`. This session also includes this documentation and other example application of `AutoCorres`. When you want to use `AutoCorres` for your own projects you do not have to import these examples. That's why we also supply the leaner `AutoCorres2_Main` as entry point, which we recommend as parent session for your applications.

25.4.2 C-Parser

Options for `install-C-file`,

syntax `install_C_file <filename> [<option> = value, ...]`:

- `memsafe` (default `false`): add additional guards to ensure that well-typedness of pointers
- `c_types` (default `true`): import types to UMM model
- `c_defs` (default `true`): import function to SIMPL
- `roots`: (default all functions): List of C functions that are the root functions to import
- `prune_types` (default `true`): only import those types that are actually used in the program
- `machinety`: HOL type for ghost state modelling the machine
- `gostty`: HOL type for some additional ghost state
- `skip_asm` (default `false`): Skip inline assembler

Moreover there are some Isabelle options for more feedback / tracing:

- `c_parser_feedback_level` (default 0), higher number means more tracing
- option `c_parser_verbose` (default `false`), enable for verbose messages

Multiple C Files

Command **install-C-file** only has a single C file as argument. When your project consists of several `.c` and `.h` you can register those dependencies first, with command **include-C-file**. The main C file, which is the argument to **install-C-file**, can then include all these files. If there is no such C file in your project you can create one to accomodate **install-C-file**.

Target Architecture Selection L4V_ARCH

The Environment variable `L4V_ARCH` can be used to determine the target architecture which influences the sizes of C integral types and pointer types. Supported platforms are `ARM` (default), `ARM64`, `ARM_HYP`, `RISCV64` and `X64`.

For `ARM`, the sizes are:

- 128 bits: `int128`
- 64 bits: `long long`
- 32 bits: `pointers`, `long`, `int`
- 16 bits: `short`

For `X64`, `ARM64`, `ARM_HYP`:

- 128 bits: `int128`
- 64 bits: `pointers`, `long long`, `long`
- 32 bits: `int`
- 16 bits: `short`

For `ARM64` `char` is signed.

For example to build `AutoCorres` for `ARM64`:

```
L4V_ARCH=ARM64 isabelle build -d . AutoCorres_Main
```

25.4.3 AutoCorres

Options for **init-autocorres** / **autocorres**,

syntax `autocorres [option = value, ...] <filename>`:

- **phase**: perform autocorres up to specified phase only (`L1`, `L2`, `HL`, `WA`, `IO`, `TS`)
- **scope**: space separated list of functions to perform autocorres on. (Default all).

- `scope_depth`: depth of callees to also include
- `single_threaded`: flag to disable parallel processing (e.g. to make more sense out of tracing messages)
- `no_heap_abs`: space separated list of functions that should be excluded from heap abstraction
- `in_out_parameters`: 'and' separated list of function specs e.g. `inc(y:in_out)`, cf. `../In_Out_Parameters_Ex.thy`
- `in_out_globals`: 'and' separated list of functions which modify global variables via pointers
- `skip_io_abs`: flag to disable IO abstraction
- `addressable_fields`: space separated list of paths to struct fields that should be addressable in the split heap, cf. `../open_struct.thy`
- `ignore_addressable_fields_error`: option to downgrade error to a warning
- `skip_heap_abs`: flag to disable heap abstraction (into split heap)
- `unsigned_word_abs`: space separated list of functions where unsigned words should be abstracted to *nat*.
- `unsigned_word_abs_known_functions`: assume unsigned word abstraction for function pointers
- `no_signed_word_abs`: space separated list of functions where abstraction of unsigned word to *int* should be disabled.
- `no_signed_word_abs_known_functions`: don't assume unsigned word abstraction for function pointers
- `skip_word_abs`: flag to disable word abstraction
- `ts_rules`: space separated list of rules to consider during TS phase (pure, gets, option, nondet, exit).
- `ts_force <rule-name>`: space separated list of function names to put in the specified monad (pure, gets, option, nondet, exit).
- `ts_force_known_functions`: assume function pointers live in the specified monad (pure, gets, option, nondet, exit)
- `heap_abs_syntax`: enable some additional syntax for heap accesses
- `do_polish` (default true): flag for polish phase

- `L1_opt` (RAW | PEEP (default)): level for L1 optimisation
- `L2_opt` (RAW | PEEP (default)): level for L2 optimisation
- `HL_opt` (RAW | PEEP (default)): level for HL optimisation
- `WA_opt` (RAW | PEEP (default)): level for WA optimisation
- `trace_opt`: flag to enable some tracing
- `gen_word_heaps`: flag to generate word heaps even if the program does not use them
- `keep_going`: continue despite some errors
- `lifted_globals_field_prefix`: custom prefix for split heap naming
- `lifted_globals_field_suffix`: custom suffix for split heap naming
- `function_name_prefix`: custom prefix for generated function names
- `function_name_suffix`: custom suffix for generated function names
- `no_c_termination`: flag to disable termination precondition for correspondence proofs
- `unfold_constructor_bind`: (Selectors (default) | Always | Never) to give some user level control to unfold certain "simple" binds. cf.: `../tests/proof-tests/unfold_bind_options.thy`
- `base_locale`: custom base locale for all autocorres locales

Moreover there are some Isabelle options for more feedback / tracing:

- `option verbose` (default 0), higher value means more verbosity
- `verbose_timing` (default 0), higher value means more timing messages
- `timing_threshold` (default 3), threshold in milliseconds for timing messages

Chapter 26

Overview of AutoCorres

```
theory AutoCorresInfrastructure  
imports AutoCorres  
begin
```

This theory elaborates on some of the (internal) AutoCorres infrastructure and is also supposed to act as a testbench for this infrastructure, e.g. simplifier setup.

Following the Isabelle tradition user relevant changes to AutoCorres are described in the file `../NEWS`.

26.1 Building Blocks

'AutoCorres' has the following major building blocks and contributions.

- The 'C-parser' (by Michael Norrish) takes C-Input files and parses them to 'SIMPL' (by Norbert Schirmer) programs within Isabelle/HOL:
 - C-parser: [chapter 28](#)
 - SIMPL: https://www-wjp.cs.uni-saarland.de/leute/private_homepages/nschirmer/pub/schirmer_phd.pdf
- The unified memory model (UMM) (by Harvey Tuch) models C-types as HOL-types with conversions between the typed view and the raw-byte-level view. Moreover it provides a separation logic framework.
 - UMM: <https://trustworthy.systems/publications/papers/Tuch%3Aphd.pdf>
- AutoCorres (by David Greenaway): https://trustworthy.systems/publications/nicta_full_text/8758.pdf

Some historic remarks, that might give some insight why things are as they are. The motivation for AutoCorres was the C-verification tasks coming from the sel4 Projects <https://sel4.systems>, a verified microkernel, written in C.

The project started with high level HOL-specifications, refining them to monadic functions (within HOL and Haskell) and then refining them to C-code. The nondeterministic state-monad goes back to Cook. et al http://www.cse.unsw.edu.au/~kleing/papers/Cock_KS_08.pdf. To link the C-code with the monadic functions within Isabelle / HOL the C-parser was written. The C-parser produces SIMPL programs, and SIMPL is equipped with various semantics (big and small-step) and a Hoare-logic framework including a verification condition generator. The correspondence of the C-functions represented in SIMPL and the monadic functions was manually expressed within this framework.

AutoCorres was then built after (or in parallel) to this verification effort in order to speed up the process for future projects.

While the main concepts described in the publications above are still valid, a lot of things have evolved and were extended. Especially the implementation details have changed. In this document we also give some notes on these differences. Also see `../NEWS`.

26.2 C Parser

Some remarks on the C-Parser

- Lexical and syntactical analysis is implemented using ML-Lex / ML-Yacc coming from the MLton project <http://mlton.org/>. Originally mlton was used to generate the ML files from the grammars. Meanwhile this is all integrated into Isabelle/ML.
- The C parser first generates an ML-data structure representing the program. It then does some transformations on that data structure (e.g. storing results of nested function calls into temporary variables), before translating it to SIMPL. In roughly the following steps:
 1. Generating HOL-types for the C-types.
 - The C parser performs a complete program analysis to determine the types used.
 - Defines the types and performs the UMM proofs, in particular conversions and properties to and from raw byte lists.
 2. Defining the state-space type: global variables, local variables, heap variables. The C parser does a complete program analysis to determine how global and heap variables are used to decide

whether to model them as part of the globals-record or the heap. When no 'address-of' a variable is calculated it is stored as part of the global variable record, otherwise it is stored within the heap. Global variables are more abstract to deal with compared to heap variables.

3. Defining the SIMPL procedures
4. Proving 'modifies' specifications for the procedures, i.e. frame conditions on global variables.

26.3 AutoCorres

AutoCorres abstracts the SIMPL program to a monadic HOL function in various phases, producing correspondence (a.k.a. simulation, a.k.a. refinement) theorems along the way. Note that the translation (and the correspondence theorems) might only be partial in the following sense: A phase might introduce additional guards into the code. Simulation only holds for programs that do not fail on the abstract execution. Note that non-termination is currently also modelled as failure. So simulation only holds for terminating abstract programs:

- Correspondence relation: *ac-corres*

Each phase is processed in similar stages:

- Raw translation and refinement-proof
- Optimisation of the output program, sometimes distinguished between a more lightweight "peephole" optimisation and a more heavyweight "flow" optimisation
- Define a constant for the output of the translation for that phase.

The phases are:

- From SIMPL (deep embedding of statements, shallow embedding of expressions) to L1 (shallow embedding of statements and expressions). The transformation preserves the state-space representation, and merely focuses on translating SIMPL statements to monadic functions:

- Definition of L1: `../L1Defs.thy`
- Correspondence relation: *L1corres*
- Peephole optimisation: `../L1Peephole.thy`
- Flow optimisation: `../ExceptionRewrite.thy`

- Main ML file: `../simpl_conv.ML`
- Local variable abstraction, from L1 to L2. In L2, local variables are represented as lambda abstractions. So the underlying state-space type changes, getting rid of the local variable record. As local variables are now treated as lambda abstractions, and thus names become meaningless, autocorres keeps the original names around as part of (logically redundant) name annotations within the L2 constants, e.g. *L2-gets* $(\lambda-. x) [c\text{-source-name-hint}]$. Lists of names are used, as variables might pile up in tuples. These annotations are removed in the final polishing phase of autocorres, attempting to rename the bound variables accordingly on the fly.
 - Definition of L2: `../L2Defs.thy`
 - Correspondence relation: *L2corres*
 - Exception rewriting, introducing nested exceptions: `../L2ExceptionRewrite.thy`
 - Peephole optimisation: `../L2Peephole.thy`
 - Main ML files: `../local_var_extract.ML`, `../l2_opt.ML` From now on we stay in the language L2, and also the optimisation stages are reused.
- In/Out parameter abstraction (IO). Pointer parameters are replaced by passing value parameters. This step may eliminate pointers to local variables.
 - Theory: `../In_Out_Parameters.thy`
 - Correspondence relation: *IOcorres*
 - Documentation: `In_Out_Parameters_Ex.thy`
- Heap abstraction, referred to as heap-lifting (HL). The monolithic-byte-oriented heap is abstracted to a typed-split-heap model. For that the new type *lifted-globals* is introduced. Note, that the separation logic of Tuch also attempts to give a typed view on top of the byte-level-heap. In case of nested structure types it even is capable to express the various levels of typed-heap views from bytes to individual structure fields to (nested) structures. AutoCorres takes a simpler, more pragmatic approach here and only provides a single split-heap abstraction on the level of complete types. This means there is a heap for every compound type and for every primitive type but not for nested structures. However, autocorres allows to mix between the abstractions layers (split-heap vs. byte-heap) on the granularity of functions. So you can model low-level functions like memory-allocators on the

untyped byte-heap and then still use these functions within other functions in the split-heap abstraction. Meanwhile, genuine support for a split-heap approach with addressable nested structures was added. It is described in `open_struct.thy`.

- Main theory: `../HeapLift.thy`
 - Correspondence relation: `L2Tcorres`
 - Main ML files: `../heap_lift_base.ML`, `../heap_lift.ML`
- Word abstraction (WA): (un)signed n-bit words are converted to *int* or *nat*. This conversion only affects local variable (lambda abstraction), so the underlying state-space type is maintained. When reading and storing to memory the values are converted accordingly.
 - Main theory: `../WordAbstract.thy`
 - Correspondence relation: `corresTA`
 - Main ML file: `../word_abstract.ML`
 - Type strengthening (TS), sometimes also referred to as type lifting or type specialisation: Best effort approach to simplify the structure of the monad as far as possible:
 - Pure functions
 - Reader monad, read-only state-dependant functions.
 - Option (reader) monad, in particular for potential non-terminating functions (recursion / while). Nontermination is treated as failure in autocorres and failure of guards are represented as a result of *None*
 - Nondeterministic state monad (exceptional control flow confined within function boundaries). Failure (of guards) and nontermination are represented by *Failure* as outcome.
 - Nondeterministic state monad with exit (exceptional control flow crossing function boundaries). New monad types can be registered within Isabelle / ML.
 - Main theories: `../TypeStrengthen.thy`, `../Refines_Spec.thy`
 - Correspondence relation: `refines`. Originally this was actually based on equality. As we now implement (mutually) recursive functions by a CCPO **fixed-point** instead of introducing an explicit measure parameter we changed to simulation. Equality is not `ccpo.admissible`, whereas simulation is.
 - Main ML files: `../type_strengthen.ML`, `../monad_types.ML`, `../monad_convert.ML`

- Polish. Performed as a part of type strengthening. Here remaining special and limited L2 definitions are expanded to 'ordinary' monadic functions. Also annotations of original `c` variable names are removed and bound variables are named accordingly.

26.4 AutoCorres Flow

The original AutoCorres implementation of David Greenaway was refined in several ways. In particular from a high level perspective the concepts of incremental build and parallel processing were unified:

- Parallel processing is moved to the outermost invocation of `autocorres`. All tasks are calculated respecting the dependencies of the call graph and the various `autocorres` phases, cf. `AutoCorres.parallel_autocorres`.
- Every task then invokes a distinct call to `AutoCorres.do_autocorres` exactly specifying the scope (which function clique) and which phase via the options.
- The results of an invocation are maintained in generic data, such that they are available to subsequent dependent calls.

The common parts of each phase are implemented in `AutoCorresUtil`. The last phase, Type strengthening (TS), historically played a special role and still does not make much use of `AutoCorresUtil`, but conceptually it follows the same idea:

Functions are processed in a sequence of cliques, from the bottom of the call graph to the top. Here a clique is either a single function or a group of strongly connected recursive calls. After processing of a phase the clique might get splitted due to dead code elimination. This general idea is implemented in `AutoCorresUtil.convert_and_define_cliques`.

26.4.1 General Remarks

- The main `autocorres` files, putting everything together: `../AutoCorres.thy`, `../autocorres.ML`
- There are two main approaches in a single phase to come up with an abstract program (output of the phase) and the correspondence proof to the concrete program (input of the phase):
 - Implicit synthesis of the abstract program from the concrete program by applying "intro" rules, where the abstract program is initialised with a schematic variable. Prototypical examples are the heap abstraction, word abstraction and type strengthening.

- First explicitly calculate the abstract program (fragment) within ML from the concrete program (fragment), and then perform the correspondence proof. An prototypical example is the local variable abstraction.
- The common aspects of the various stages within a single phase is (partially) generalised into library functions: `../autocorres_util.ML`

26.4.2 Links to more documentation / examples

- Pointers to local variables: `pointers_to_locals.thy`
- In-Out parameters: `In_Out_Parameters_Ex.thy`
- Addressable fields and open types: `open_struct.thy`
- Function pointers: `fnptr.thy`
- Unions: `union_ac.thy`

26.5 Overview on the Locales

The representation of local variables in SIMPL and L1 where changed from records to 'Statespaces' as implemented by `statespace` and finally to positional variables as in `locals` in `../c-parser/CLocals.thy`. This allows for an uniform representation of local variables as a function from 'nat to byte list', Typing is achieved by ML data organised in locales and bundles, cf. `../c-parser/CTranslationInfrastructure.thy`. These are also used to do the L1 and the L2 correspondence proofs. Starting from L2 on the local variables are represented as lambda bindings.

We describe the various locales (and bundles) later on with our running examples.

To support function pointers even more locales where introduced. See `fnptr.thy`.

26.6 Example Program

```
install-C-file autocorres-infrastructure-ex.c
print-theorems
thm upd-lift-simps
thm fl-ti-simps
```

The variables consist of parameters (input and output) and the local variables. Note that this is merely a ML-declaration of the scope which can be accessed via a bundle, cf. `../c-parser/CTranslationInfrastructure.thy`.

```
context includes add-variables
begin
term 'n ::= 42
end
```

Addresses of global variables or function pointers are kept in the *global-addresses* locale.

```
print-locale autocorres-infrastructure-ex-global-addresses
```

This is everything necessary to define the body of a function. All bodies are defined in that locale.

```
context autocorres-infrastructure-ex-global-addresses
  opening add-variables
begin
term add-body
thm add-body-def
end
```

The implementation locale holds the defining equation of the function in SIMPL and is closed under callees.

```
context call-add-impl
begin
thm add-impl
thm call-add-impl
end
```

In case of a clique there is a clique locale and aliases for each function.

```
print-locale even-odd-impl
print-locale even-impl
print-locale odd-impl
```

The locale importing all implementations is named like the file with suffix *-simpl*.

```
print-locale autocorres-infrastructure-ex-simpl
```

26.6.1 Incremental Build

```
autocorres [phase=L1, scope=add] autocorres-infrastructure-ex.c
autocorres [phase=L2, scope=add] autocorres-infrastructure-ex.c
autocorres [phase=HL, scope=add] autocorres-infrastructure-ex.c
autocorres [phase=WA, scope=add] autocorres-infrastructure-ex.c
autocorres [phase=TS, scope=add] autocorres-infrastructure-ex.c
```

26.6.2 All the rest

autocorres *autocorres-infrastructure-ex.c*

```
context unsigned-to-signed-impl
begin
thm unsigned-to-signed-body-def
end
context autocorres-infrastructure-ex-all-corres
begin
thm unsigned-to-signed'-def
```

The SIMPL versions produced by c-parser

```
declare [[locals-short-names]]
thm max-body-def
thm add-body-def
term add-body::((globals, locals, 32 signed word) CProof.state, unit ptr, strictc-errortype)
com
thm call-add-body-def
thm seq-assign-body-def
thm call-seq-assign-body-def
thm inc-g-body-def
thm deref-body-def
thm deref-g-body-def
thm factorial-body-def
thm call-factorial-body-def
thm odd-body-def
thm even-body-def
thm dead-f-body-def
thm dead-h-body-def
thm dead-g-body-def
```

L1 versions. Monadic with same state. Note that recursive procedures are defined with the **fixed-point** package and hence the *.simps* instead of *-def* makes more sense to look at. In this layer a first exception elimination optimisation takes places. The optimized definition has the extension "opt".

```
declare [[show-types=false]]
thm l1-max'-def
thm l1-opt-max'-def

term l1-add'::(unit, unit, (globals, locals, 32 signed word) CProof.state) exn-monad
thm l1-add'-def
thm l1-opt-add'-def

thm l1-call-add'-def
thm l1-opt-call-add'-def

thm l1-seq-assign'-def
thm l1-opt-seq-assign'-def
```

thm *l1-call-seq-assign'-def*
thm *l1-opt-call-seq-assign'-def*

thm *l1-inc-g'-def*
thm *l1-opt-inc-g'-def*

thm *l1-deref'-def*
thm *l1-opt-deref'-def*

thm *l1-deref-g'-def*
thm *l1-opt-deref-g'-def*

thm *l1-factorial'.simps*
thm *l1-opt-factorial'-def*

thm *l1-call-factorial'-def*
thm *l1-opt-call-factorial'-def*

thm *l1-odd'.simps*
thm *l1-opt-odd'-def*

thm *l1-even'.simps*
thm *l1-opt-even'-def*

thm *l1-dead-f'.simps*
thm *l1-opt-dead-f'-def*

thm *l1-dead-h'.simps*
thm *l1-opt-dead-h'-def*

thm *l1-dead-g'.simps*
thm *l1-opt-dead-g'-def*

L2 version: lambda bindings for local variables

thm *l2-max'-def*
term *l2-add'::32 signed word \Rightarrow 32 signed word*
 \Rightarrow (32 signed word c-exntype, 32 signed word, globals) exn-monad
thm *l2-add'-def*
thm *l2-call-add'-def*
thm *l2-seq-assign'-def*
thm *l2-call-seq-assign'-def*
thm *l2-inc-g'-def*
thm *l2-deref'-def*
thm *l2-deref-g'-def*
thm *l2-factorial'.simps*
thm *l2-call-factorial'-def*
thm *l2-odd'.simps*
thm *l2-even'.simps*
thm *l2-dead-f'.simps*

thm *l2-dead-h'.simps*

thm *l2-dead-g'-def* — Note that *l2-dead-g'* is no longer part of the clique, because of dead code elimination.

Heap abstraction. Note that the change in the heap component of the global variables. The type changes to *lifted-globals* instead of *globals*.

print-record *globals* — contains byte level tagged heap: *t-hrs'*

print-record *lifted-globals* — contains split heap with single component *heap-w32*, as we only have a pointers to unsigend.

thm *hl-max'-def*

term *hl-add'::32 signed word* \Rightarrow *32 signed word*

\Rightarrow (*32 signed word c-exntype*, *32 signed word*, *lifted-globals*) *exn-monad*

thm *hl-add'-def*

thm *hl-call-add'-def*

thm *hl-seq-assign'-def*

thm *hl-call-seq-assign'-def*

thm *hl-inc-g'-def*

thm *hl-deref'-def*

thm *hl-deref-g'-def*

thm *hl-factorial'.simps*

thm *hl-call-factorial'-def*

thm *hl-odd'.simps*

thm *hl-even'.simps*

thm *hl-dead-f'.simps*

thm *hl-dead-h'.simps*

thm *hl-dead-g'-def*

Word abstraction.

thm *wa-max'-def*

term *wa-add'::int* \Rightarrow *int* \Rightarrow (*32 signed word c-exntype*, *int*, *lifted-globals*) *exn-monad*

thm *wa-add'-def*

thm *wa-call-add'-def*

thm *wa-seq-assign'-def*

thm *wa-call-seq-assign'-def*

thm *wa-inc-g'-def*

thm *wa-deref'-def*

thm *wa-deref-g'-def*

thm *wa-factorial'.simps*

thm *wa-call-factorial'-def*

thm *wa-odd'.simps*

thm *wa-even'.simps*

thm *wa-dead-f'.simps*

thm *wa-dead-h'.simps*

thm *wa-dead-g'-def*

Final definition.

term *max'::int* \Rightarrow *int* \Rightarrow *int*

thm *max'-def*

```

term add':: int  $\Rightarrow$  int  $\Rightarrow$  lifted-globals  $\Rightarrow$  int option
thm add'-def
thm wa-call-add'-def
thm wa-seq-assign'-def
thm wa-call-seq-assign'-def
thm wa-inc-g'-def
thm wa-deref'-def
thm wa-deref-g'-def
thm wa-factorial'.simps
thm wa-call-factorial'-def
thm wa-odd'.simps
thm wa-even'.simps
thm wa-dead-f'.simps
thm wa-dead-h'.simps
thm wa-dead-g'-def

```

Correspondence theorem.

```

thm factorial'-ac-corres
thm call-factorial'-ac-corres

```

end

26.7 Simplification strategy and dealing with tuples in L2-optimization phases

Consider a congruence rule from AutoCorres

lemma

```

assumes c-eq:  $\bigwedge r s. c r s = c' r s$ 
assumes bdy-eq:  $\bigwedge r s. c' r s \Longrightarrow \text{run } (A r) s = \text{run } (A' r) s$ 
shows L2-while c A = L2-while c' A'
using c-eq bdy-eq
by (rule L2-while-cong)

```

Note that r could be a tuple and we would like to "split" it. The condition c will already be formulated with *case-prod*, e.g. $\lambda(x, y, z) s. x < y$.

When the congruence rule above is applied, variable $?c'$ has to be instantiated at some point. First the congruence rule is applied as is. Then we use *split-paired-all* to split up the r . For the example we will end up with the subgoal:

$$\bigwedge a b c s. (a < b) = ?c' (a, b, c) s$$

Mere Unification will fail here as it does not work "modulo" *case-prod*. To find the proper instantiation we developed `Tuple_Tools.tuple_inst_tac` which can be added to the simplifier as a looper. It will instantiate $?c'$ with the respective *case-prod* instance introducing a new schematic variable: $?c' = (\lambda(a, b, c). ?f a b c)$. Now the unification will kick in and do the rest of the work.

However, although an instantiation is now found, it does not have the expected effect on `body_eq`. The problem is, that the premise of the implication is not simplified as it is added to the "prems" of the simplifier so it will be $(\text{case } (a, b, c) \text{ of } (a, b, c) \Rightarrow \lambda s. a < b) s \equiv \text{True}$ and not just $a < b \equiv \text{True}$ ". The former format is unfortunately ineffective in the simplification of the while body.

Fortunately the simpset can be instructed to apply a function to the premise before it is added. So adding `Tuple_Tools.mk_simps` with `Simplifier.set_mk_simps` helps with that respect. As the simplifier descends into the term it calls the function to do "beta-reduction" of tuple application before adding the premise.

```

lemma L2-while ( $\lambda(x,y,z) s. 0 < (y::nat)$ ) ( $\lambda(x,y,z). L2\text{-seq } (L2\text{-guard } (\lambda-. 0 < y)) (\lambda-. X)) x ns
=
  L2-while ( $\lambda(x,y,z) s. 0 < y$ ) ( $\lambda(x,y,z). L2\text{-seq } (L2\text{-guard } (\lambda-. \text{True})) (\lambda-. X)) x ns
apply (tactic <
simp-tac ( @ {context}
  addloop (tuple-inst-tac, Tuple-Tools(tuple-inst-tac)
  addsimps @ {thms split-paired-all}
  |> Simplifier.add-cong @ {thm L2-seq-guard-cong}
  |> Simplifier.add-cong @ {thm L2-while-cong}
  |> Simplifier.set-mk_simps (Tuple-Tools.mk_simps)
) 1>)
done$$ 
```

Unfortunately there is still an issue. Linear arithmetic is applied as a simproc by the simplifier, e.g. $0 < n \implies \text{Suc } 0 \leq n$

Unfortunately this does not work as expected in the setup above:

```

unbundle clocals-string-embedding

```

```

lemma L2-while ( $\lambda(x,y,z) s. 0 < (y::nat)$ ) ( $\lambda(x,y,z). L2\text{-seq } (L2\text{-guard } (\lambda-. \text{Suc } 0 \leq y \wedge z = a)) (\lambda-. X)) x ns
=
  L2-while ( $\lambda(x,y,z) s. 0 < y$ ) ( $\lambda(x,y,z). L2\text{-seq } (L2\text{-guard } (\lambda-. z = a)) (\lambda-. X)) x ns
apply (tactic <
simp-tac ( @ {context}
  addloop (tuple-inst-tac, Tuple-Tools(tuple-inst-tac)
  addsimps @ {thms split-paired-all}
  |> Simplifier.add-cong @ {thm L2-seq-guard-cong}
  |> Simplifier.add-cong @ {thm L2-while-cong}
  |> Simplifier.set-mk_simps (Tuple-Tools.mk_simps)
) 1>)
oops$$ 
```

After some investigation in the interplay between the simplifier and the linear arithmetic simproc it seems that the linear arithmetic somehow ig-

nores the transformed theorem that `Tuple_Tools.mk_simps` yields. Without understanding all the details I guess the reason is the mismatch between the term in the hyps and the prop of the resulting theorem. As the simplifier adds descends into an implication $P \implies Q$ it generates a theorem from P , by also adding P to the hyps, so we have $P [P]$ as a theorem. When `Tuple_Tools.mk_simps` kicks in it only simplifies the prop not the hyps. So we end up with $a < b \equiv True [(\lambda(a, b, c) s. a < b) (a, b, c) s \equiv True]$ instead of $a < b \equiv True [a < b \equiv True]$. This seems to irritate the linear arithmetic. One solution could be to also simplify the hyps. We did not explore this path, as I would expect some issues when the hyps are eventually discharged. Nevertheless there could be a solution following that path.

We came up with a different solution. We get rid of the dependency of `Tuple_Tools.mk_simps` by using $P =simp=> Q$ to trigger simplification of P .

Note the various options to trace the simplification process.

```
declare [[linarith-trace=false]]
declare [[simp-trace=false, simp-trace-depth-limit=100]]
declare [[simp-debug=false]]
declare [[show-hyps=false]]
```

lemma

```
assumes c-eq:  $\bigwedge r s. c r s = c' r s$ 
assumes bdy-eq:  $\bigwedge r s. c' r s =simp=> run (A r) s = run (A' r) s$ 
shows  $L2\text{-while } c A = L2\text{-while } c' A'$ 
using c-eq bdy-eq by (rule L2-while-simp-cong)
```

lemma $L2\text{-while } (\lambda(x,y,z) s. 0 < (y::nat)) (\lambda(x,y,z). L2\text{-seq } (L2\text{-guard } (\lambda-. Suc 0 \leq y \wedge z = a)) (\lambda-. X)) x ns$

=
 $L2\text{-while } (\lambda(x,y,z) s. 0 < y) (\lambda(x,y,z). L2\text{-seq } (L2\text{-guard } (\lambda-. z = a)) (\lambda-. X)) x ns$

```
apply (tactic <
simp-tac ( @{context}
addloop (tuple-inst-tac, Tuple-Tools(tuple-inst-tac)
addsimps @{thms split-paired-all}
|> Simplifier.add-cong @{thm L2-seq-guard-cong}
|> Simplifier.add-cong @{thm L2-while-simp-cong}
) 1>)
done
```

The more standard congruence rule also works fine.

lemma

```
assumes c-eq:  $c = c'$ 
assumes bdy-eq:  $\bigwedge r s. c' r s =simp=> run (A r) s = run (A' r) s$ 
shows  $L2\text{-while } c A = L2\text{-while } c' A'$ 
using c-eq bdy-eq by (rule L2-while-simp-cong')
```

```

lemma L2-while ( $\lambda(x,y,z) s. 0 < (y::nat)$ ) ( $\lambda(x,y,z). L2\text{-seq} (L2\text{-guard} (\lambda-. Suc\ 0 \leq y \wedge z = a)) (\lambda-. X))$ )  $x\ ns$ 
=
  L2-while ( $\lambda(x,y,z) s. 0 < y$ ) ( $\lambda(x,y,z). L2\text{-seq} (L2\text{-guard} (\lambda-. z = a)) (\lambda-. X))$ )  $x\ ns$ 
apply (tactic ◁
  simp-tac ( @ {context}
    addloop (tuple-inst-tac, Tuple-Tools(tuple-inst-tac)
    addsimps @ {thms split-paired-all}
    |> Simplifier.add-cong @ {thm L2-seq-guard-cong}
    |> Simplifier.add-cong @ {thm L2-while-simp-cong'})
  ) 1)
done

```

To avoid repeated splitting (and diving into the subterms) of tuples with $(\bigwedge x. PROP\ ?P\ x) \equiv (\bigwedge a\ b. PROP\ ?P\ (a, b))$ consider the following simproc setup.

```

lemma L2-while ( $\lambda(x,y,z) s. 0 < (y::nat)$ ) ( $\lambda(x,y,z). L2\text{-seq} (L2\text{-guard} (\lambda-. Suc\ 0 \leq y \wedge z = a)) (\lambda-. X))$ )  $x\ ns$ 
=
  L2-while ( $\lambda(x,y,z) s. 0 < y$ ) ( $\lambda(x,y,z). L2\text{-seq} (L2\text{-guard} (\lambda-. z = a)) (\lambda-. X))$ )  $x\ ns$ 
apply (tactic ◁
  simp-tac ( @ {context}
    addloop (tuple-inst-tac, Tuple-Tools(tuple-inst-tac)
    addsimprocs [ Tuple-Tools.split-tupled-all-simproc, Tuple-Tools(tuple-case-simproc)
    delsimps @ {thms Product-Type.prod.case Product-Type.case-prod-conv}
    |> Simplifier.add-cong @ {thm L2-seq-guard-cong}
    |> Simplifier.add-cong @ {thm L2-while-simp-cong'})
  ) 1)
done

```

This is also the setup of `L2Opt.cleanup_ss`

```

lemma L2-while ( $\lambda(x,y,z) s. 0 < (y::nat)$ ) ( $\lambda(x,y,z). L2\text{-seq} (L2\text{-guard} (\lambda-. Suc\ 0 \leq y \wedge z = a)) (\lambda-. X))$ )  $x\ ns$ 
=
  L2-while ( $\lambda(x,y,z) s. 0 < y$ ) ( $\lambda(x,y,z). L2\text{-seq} (L2\text{-guard} (\lambda-. z = a)) (\lambda-. X))$ )  $x\ ns$ 
apply (tactic ◁
  asm-full-simp-tac (L2Opt.cleanup-ss @ {context} [] FunctionInfo.PEEP) 1)
done

```

```

lemma L2-while ( $\lambda(x, y, z) s. y = 0$ )
  ( $\lambda(x, y, z).$ 
    L2-seq (L2-gets ( $\lambda s. y$ ) [ $\mathcal{S}$  "ret'"]) ( $\lambda r. XXX21\ r\ x$ ))
  names =

```

```

    L2-while ( $\lambda(x, y, z) s. y = 0$ ) ( $\lambda(x, y, z). L2\text{-seq} (L2\text{-gets} (\lambda s. 0) [\mathcal{S} \text{ "ret"}])$ )
( $\lambda r. XXX21 r x$ ) names
apply (tactic  $\langle$ 
  asm-full-simp-tac (L2Opt.cleanup-ss @{context} [] FunctionInfo.PEEP) 1  $\rangle$ )
done

```

```

lemma PROP SPLIT ( $\bigwedge r. ((\lambda(x,y,z). y < z \wedge z=s) r) \implies P r$ )
 $\equiv (\bigwedge x y z. y < z \wedge z = s \implies P (x, y, s))$ 
apply (tactic  $\langle$ 
  asm-full-simp-tac (put-simpset HOL-basic-ss @{context})
  addsimprocs [Tuple-Tools.SPLIT-simproc, Tuple-Tools.tuple-case-simproc] |> Simplifier.add-cong @{thm SPLIT-cong}) 1  $\rangle$ )
done

```

```

experiment
begin

```

```

schematic-goal foo:

```

```

 $\bigwedge x1 x2 x3 s. (case (x1, x2, x3) of (x, y, z) \implies \lambda s. 0 < y) s \implies$ 
  ( $case (x1, x2, x3) of (x, y, z) \implies L2\text{-seq} (L2\text{-guard} (\lambda-. Suc\ 0 \leq y \wedge z = a)) X$ )
 $= ?A' (x1, x2, x3) s$ 
apply (tactic  $\langle$ 
  asm-full-simp-tac ( @{context})
  addloop (tuple-inst-tac, Tuple-Tools.tuple-inst-tac)
  addsimprocs [Tuple-Tools.SPLIT-simproc, Tuple-Tools.tuple-case-simproc]
  delsimps @{thms Product-Type.prod.case Product-Type.case-prod-conv}
  ) 1  $\rangle$ )
done

```

?A' should be properly instantiated.

```

thm foo

```

```

end

```

```

declare [[simp-trace=true, simp-trace-depth-limit=100]]

```

```

lemma L2-while ( $\lambda(x,y,z) s. 0 < (y::nat)$ ) ( $\lambda(x,y,z). L2\text{-seq} (L2\text{-guard} (\lambda-. Suc\ 0 \leq y \wedge z = a)) X$ ) x ns
 $=$ 
  L2-while ( $\lambda(x,y,z) s. 0 < y$ ) ( $\lambda(x,y,z). L2\text{-seq} (L2\text{-guard} (\lambda-. z = a)) X$ ) x
ns

```

```

apply (tactic  $\langle$ 
  asm-full-simp-tac ( @{context})
  addloop (tuple-inst-tac, Tuple-Tools.tuple-inst-tac)
  addsimprocs [Tuple-Tools.SPLIT-simproc, Tuple-Tools.tuple-case-simproc]
  delsimps @{thms Product-Type.prod.case Product-Type.case-prod-conv}
  |> Simplifier.add-cong @{thm L2-seq-guard-cong}
  )

```

```

    |> Simplifier.add-cong @{\thm L2-while-cong-simp-split}
    |> Simplifier.add-cong @{\thm SPLIT-cong}
  ) 1)
  done

```

Finally the following setup implements a very controlled tuple esplitting on the While body.

lemma

```

assumes c-eq: c = c'
assumes bdy-eq: PROP SPLIT ( $\bigwedge r s. c' r s \implies \text{run } (A r) s = \text{run } (A' r) s$ )
shows L2-while c A = L2-while c' A'
using c-eq bdy-eq by (rule L2-while-cong-split)

```

With *SPLIT* we mark a position in the term that will trigger `Tuple_Tools.SPLIT_simproc`. By adding *SPLIT PROP ?P* \equiv *SPLIT PROP ?P* as congruence rule we prohibit the simplifier from diving into the loop body before we actually split the result variable. Note that the simplifier usually works bottom up, only congruence rules are applied top down.

lemma *L2-while* ($\lambda(x,y,(z::nat)) s. 0 < (y::nat) \wedge y=x$) ($\lambda(x,y,z). L2\text{-seq } (L2\text{-guard } (\lambda-. \text{Suc } 0 \leq y \wedge z = a)) X$) *x ns*

=
L2-while ($\lambda(x, y, z) s. 0 < y \wedge y = x$) ($\lambda(x, x, z). L2\text{-seq } (L2\text{-guard } (\lambda-. z = a)) X$) *x ns*

apply (tactic <

```

asm-full-simp-tac ( @{\context}
  addloop (tuple-inst-tac, Tuple-Tools(tuple-inst-tac)
  addsimprocs [Tuple-Tools.SPLIT_simproc, Tuple-Tools(tuple-case-simproc]
  delsimps @{\thms Product-Type.prod.case Product-Type.case-prod-conv}
  |> Simplifier.add-cong @{\thm L2-seq-guard-cong}
  |> Simplifier.add-cong @{\thm L2-while-cong-simp-split}
  |> Simplifier.add-cong @{\thm SPLIT-cong}

```

```

) 1)
done

```

26.8 Simplification of conditions (guards, loops, conditionals)

The basic peephole optimization for conditions tries to simplify "trivial" guards / conditions by propagating the information of guards / conditions to subsequent guards / conditions. It is called "peephole" optimisation (in contrast to "flow"-sensitive), because it only propagates the state independent information of guards, i.e. constraints on constant expressions or local variables, not constraints on the state itself.

To facilitate this propagation with congruence rules and simprocs, we introduce two special constants *L2-seq-gets* and *L2-seq-guard* that are introduced as intermediate step by a conversion. With this marking we can

distinguish the cases by congruence rules, without the marking both cases would be subject to a single congruence rule for *L2-seq*. (Note that the simplifier only considers the head term of a congruence rules.)

We add the congruence rules $?c = ?c' \implies L2\text{-seq-gets } ?c \ ?n \ ?A \equiv L2\text{-seq-gets } ?c' \ ?n \ ?A$ and $\llbracket ?P = ?P'; \bigwedge s. ?P' \ s \implies \text{run } (?X \ ()) \ s = \text{run } ?X' \ s \rrbracket \implies L2\text{-seq-guard } ?P \ ?X = L2\text{-seq-guard } ?P' \ (\lambda-. ?X')$. The congruence rules for the guard is a standard congruence rules that adds the gaurd as a precondition for simplifying the second statement in the sequential composition. The congruence rule for $?c = ?c' \implies L2\text{-seq-gets } ?c \ ?n \ ?A \equiv L2\text{-seq-gets } ?c' \ ?n \ ?A$ stops the simplifier from descending into the second term of the sequential composition. Instead the simproc `L2Opt.l2_marked_gets_bind_simproc` is triggered to analyse the situation. It does the following:

- If the returned value in the first statement is simple, only occurs once in second statement, or is a structure-constructor or update that is only applied to structure-selections then the value is directly propagated to the second statement and the sequential composition is removed.
- It peeks into the prems of the simplifier to see if there are already "interesting" facts collected from the congruence rules of guards / conditions. If not, it unfolds the marking and continues with the ordinary sequential composition by simplifying the second statement. "Interesting" means there is at least one premise that has impact on the new return value. E.g. if we have $x + y < (5::'a)$ in the prems, and we now we return $x + y$, than the simproc introduces a new variable r for return value, augments the context with the equation $r = x + y$ and derives the new premise $r < (5::'a)$ which is put to the premises of the simplifier as well as the simpset. Then the simplifier is called recursively on the second statement of the sequential composition with the augmented context. When it is finished it uses the rule $?f \ ?c \equiv ?g \implies L2\text{-seq-gets } ?c \ ?n \ ?f \equiv STOP \ (L2\text{-seq-gets } ?c \ ?n \ (\lambda-. ?g))$ to introduce the constant *STOP*. The congruence rule $STOP \ ?P \equiv STOP \ ?P$ prohibits the simplifier to descend down into the just simplified second statement again.

Note that this setup works around the limitation of the simplifier that does not allow us to modify the context by something like a "congruc", similar to a "simproc". Keep in mind that a congruence rule is applied top-down as the simplifier works its way down into a term, whereas a simpproc (or any other simplification rule) is applied bottom up. So a simproc / simplification rule can build on the fact that the subterms are already normalised. The simplifier makes use of this by doing some bookkeeping with "term-skeletons" to avoid resimplification of subterms. This mechanism fails short in our setup of using a

congruence rule to stop simplification of the second statement and calling the `simplproc` instead. That is why we explicitly introduce `STOP`. It will be removed after simplification by an additional traversal of the term.

declare `[[simp-trace=false, ML-print-depth=1000]]`

Propagation of simple constant by unfolding.

lemma `L2-seq-gets c [S "r'] (λr. (L2-guard (λ-. P r))) =`
`L2-guard (λ-. P c)`
by `(tactic <simp-tac (L2Opt.cleanup-ss @ {context} [] FunctionInfo.PEEP) 1 >)`

Propagation of term, as it only appears once in the second statement.

lemma `L2-seq-gets (c + d) [S "r'] (λr. (L2-guard (λ-. P r))) =`
`L2-guard (λ-. P (c + d))`
by `(tactic <simp-tac (L2Opt.cleanup-ss @ {context} [] FunctionInfo.PEEP) 1 >)`

As nothing is in the prems yet, marking is just removed and second statement is thus simplified

lemma `L2-seq-gets (c + d) [S "r'] (λr. (L2-guard (λ-. P r ∧ (P r → Q r))))`
`=`
`L2-seq (L2-gets (λ-. c + d) [S "r'] (λr. L2-guard (λ-. P r ∧ Q r)))`
by `(tactic <simp-tac (L2Opt.cleanup-ss @ {context} [] FunctionInfo.PEEP) 1 >)`

The first guard condition is propagated to the second guard, via the intermediate assignment `r = a + b`.

lemma `L2-seq-guard (λ-. a + b < 5)`
`(λ-. L2-seq-gets (a + b) [S "r']`
`(λr. L2-guard (λ-. r < 5 ∧ P))) =`
`L2-seq-guard (λ-. a + b < 5)`
`(λ-. L2-guard (λ-. P))`
by `(tactic <simp-tac (L2Opt.cleanup-ss @ {context} [] FunctionInfo.PEEP) 1 >)`

ML <

`fun assert-rhs thm =`

`Tuple-Tools.assert-cterm' @ {context} (Thm.term-of (Thm.rhs-of thm))`

>

ML-val <

`val ct = @ {cterm L2-seq-guard (λ-. (a::int) + b < 5)`

`(λ-. L2-seq-gets (a + b) [S "r']`
`(λr. (L2-guard (λ-. r < 5 ∧ P))))}`

`val test = Simplifier.asm-full-rewrite (L2Opt.cleanup-ss @ {context} [] Function-`
`Info.PEEP) ct`

`val - = assert-rhs test`

`@ {cterm STOP (L2-seq-guard (λ-. (a::int) + b < 5) (λ-. L2-guard (λ-. P)))}`

>

lemma *L2-seq-guard* ($\lambda s. V1 + a - (r::int) \leq 2$)
 $(\lambda-. L2\text{-seq } (L2\text{-condition } (\lambda s. CC) (L2\text{-seq-guard } (\lambda s. V1 + a - r \leq 3)$
 $(\lambda-. X1)) X2) (\lambda-. X3)) =$
 $L2\text{-seq-guard } (\lambda s. V1 + a - r \leq 2)$
 $(\lambda-. L2\text{-seq } (L2\text{-condition } (\lambda s. CC) (L2\text{-seq-guard } (\lambda s. True) (\lambda-. X1)) X2)$
 $(\lambda-. X3))$
by (*tactic* $\langle \text{asm-full-simp-tac } (L2Opt.cleanup\text{-ss } @\{context\} \ [] \text{ FunctionInfo.PEEP}$
 $1 \rangle$)

ML-val \langle
 $val \text{ ct} = @\{cterm \text{ L2-seq-guard } (\lambda s. (V1 + a - (r::int) \leq 2))$
 $(\lambda-. L2\text{-seq } (L2\text{-condition } (\lambda s. CC) (L2\text{-seq-guard } (\lambda s. V1 + a - r \leq 3)$
 $(\lambda-. X1)) X2) (\lambda-. X3))\}$

$val \text{ test} = \text{Simplifier.asm-full-rewrite } (L2Opt.cleanup\text{-ss } @\{context\} \ [] \text{ Function-}$
 $\text{Info.PEEP}) \text{ ct}$
 $val \text{ -} = \text{assert-rhs test}$
 $@\{cterm \text{ STOP } (L2\text{-seq-guard } (\lambda s. V1 + a - r \leq 2)$
 $(\lambda-. L2\text{-seq } (L2\text{-condition } (\lambda s. CC) (\text{STOP } (L2\text{-seq-guard } (\lambda s. True) (\lambda-. X1))))$
 $X2) (\lambda-. X3))\}$
 \rangle

ML \langle
 $val \text{ ct} = @\{cterm \text{ L2-seq-guard}$
 $(\lambda-. ((g \ k)::int) + (a::int) - r \leq n)$
 $(\lambda-.$
 $(L2\text{-seq-guard } (\lambda s. (g \ k) + a - r \leq n + 1) (\lambda-. X1))\}$

$val \text{ test} = \text{Simplifier.asm-full-rewrite } (\text{Variable.set-body true } (L2Opt.cleanup\text{-ss } @\{context\}$
 $\ [] \text{ FunctionInfo.PEEP})) \text{ ct}$
 $val \text{ -} = \text{assert-rhs test } @\{cterm \text{ STOP } (L2\text{-seq-guard } (\lambda-. g \ k + a - r \leq n) (\lambda-.$
 $\text{STOP } (L2\text{-seq-guard } (\lambda s. True) (\lambda-. X1))\}$
 \rangle

declare $[[\text{simp-trace=false, linarith-trace=false}]]$

26.9 Tricks to enforce first-order unification

ML-val \langle
 $val \text{ fo} = \text{Utils.reify-comb-conv } @\{context\} @\{pattern \ ?a \ \$ \ (?g \ \$ \ ?x)\} @\{cterm (f$
 $z) (k \ y) \}$
 $val \text{ orig} = \text{Utils.unreify-comb-conv } @\{context\} @\{pattern \ ?a \ \$ \ (?g \ \$ \ ?x)\} (\text{Thm.rhs-of}$
 $\text{fo})$
 $val \text{ t} = \text{Utils.dest-reified-comb } @\{cterm \ a \ \$ \ b\}$
 \rangle

In heap lifting phase:

thm *heap-abs-fo*

In word abstraction phase:

thm *abstract-val-fun-app*

lemma *abstract-val-fun-app'*:

$\llbracket \text{abstract-val } Q \ x \ \text{id } x'; \ \text{abstract-val } P \ f \ \text{id } f' \rrbracket \implies$
 $\text{abstract-val } (P \wedge Q) \ (g \ (f \ x)) \ g \ (f' \ x')$

by (*simp add: abstract-val-def*)

schematic-goal *abstract-val ?Q ?f g (f' a')*

apply (*rule abstract-val-fun-app'*)

oops

`Utils.fo_arg_resolve_tac` does the trick for us.

schematic-goal *abstract-val ?Q ?f g (f' a')*

apply (*tactic* $\langle \text{Utils.fo-arg-resolve-tac } @\{\text{context}\} \ @\{\text{thm abstract-val-fun-app}\}$
 $@\{\text{context}\} \ 1 \rangle$)

oops

It turned out that `Utils.fo_arg_resolve_tac` and other tactics like simplification can become quite slow when operating under a lot of bound variables. We introduced a `WordAbstract.thin_abs_var_tac` to remove unused premises and bound variables. It is triggered by *THIN* in the rules, e.g. $\llbracket \text{introduce-tyt-abs-fn } ?rx1.0; \ \text{THIN } (\text{corresTA } ?P \ ?rx1.0 \ ?ex \ ?L \ ?L'); \ \text{THIN } (\bigwedge r \ r'. \ \text{abs-var } r \ ?rx1.0 \ r' \implies \ \text{corresTA } (?Q \ r) \ ?rx2.0 \ ?ex \ (?R \ r) \ (?R' \ r')) \rrbracket \implies \ \text{corresTA } ?P \ ?rx2.0 \ ?ex \ (L2\text{-seq } ?L \ (\lambda r. \ L2\text{-seq } (L2\text{-guard } (?Q \ r)) \ (\lambda -. \ ?R \ r))) \ (L2\text{-seq } ?L' \ ?R')$.

schematic-goal

$\bigwedge n' \ s \ r \ r' \ ra \ r'a \ sa.$

$\text{abs-var } r \ \text{id } r' \implies$

$\text{abs-var } ra \ \text{id } r'a \implies \ \text{abstract-val } (?Q57 \ r \ ra \ sa) \ (?b60 \ r \ ra \ sa) \ \text{id } (6$

$* \ r'a)$

apply (*tactic* $\langle \text{WordAbstract.thin-abs-var-tac } @\{\text{context}\} \ 1 \rangle$)

oops

schematic-goal

$\bigwedge n' \ s \ r \ r' \ ra \ r'a \ sa.$

$\text{abs-var } n \ \text{id } n' \implies$

$\text{abs-var } r \ \text{id } r' \implies$

$\text{abs-var } ra \ \text{id } r'a \implies \ \text{abstract-val } (?Q57 \ r \ ra \ sa) \ (?b60 \ r \ ra \ sa) \ \text{id } (6$

$* \ r'a)$


```

apply (tactic < WordAbstract.thin-abs-var-tac @{context} 1 > )
oops

```

26.10 Exception Rewriting

This section highlights some of the ideas to rewrite and optimise exceptions.

In SIMPL and L1 the cause for an exception is stored in the state record in field *global-exn-var'-'*. The handler then inspects the state to decide whether it is responsible or not, and either handles the exception or re-raises it. In L2 this is refined in several steps.

- The cause of the exception is moved out of the state and represented as an error result in the L2 monad.
- The check for the cause in the handler of *L2-catch* is resolved to a static nesting of *L2-try* instead. The static nesting depth is directly reflected in the depth of the sum that reflects the error value *Inl* (*Inl* ...). For local exceptions this sequence of *Inl* ends with an *Inr*, global exception (crossing function boundaries) end in an *Inl*.
- The tuple arity of intermediate bindings in statements like *L2-seq*, *L2-while* is optimised to only propagate values that are actually used later on. As a side effect of this step, unnecessary nondeterministic initialisation steps for local variables (especially the technical return variable) are removed. The initialisation becomes unnecessary if the variable is strictly assigned to before used. This is always the case for the return variable, as *return x* is translated to an assignment to the return variable followed by raising the *Return* exception.

Note that the field *global-exn-var'-'* has a special status, it is neither part of *globals* nor *locals*. In function calls it is treated like a global variable to ensure that the exception passes function boundaries. However, in the local variable abstraction of phase L2 it is treated similar to local variable and abstracted to lambda bindings.

```

declare [[verbose=3]]
declare [[verbose-timing = 3]]

```

26.10.1 Preliminary examples illustrating the usage of *rel-spec-monad*

```

schematic-goal rel-spec-monad (=) (=)
(L2-catch
  (L2-seq
    (L2-catch
      (L2-while ( $\lambda r$  s. True)

```

```

      (λr. L2-condition (λs. 3 < a) (L2-throw (Return, (1::int)) [S
"global-exn-var", S "ret"]))
      (L2-seq
        (L2-condition (λs. 1 < a)
          ((L2-throw (Nonlocal (1::nat), r) [S "global-exn-var", S
"ret"]):(nat c-exntype × int, int, 'a) exn-monad)
          (L2-throw (Break, r) [S "global-exn-var", S "ret"]))
          (λra. L2-gets (λs. r) [S "ret"])))
      r [S "ret"])

(λ(a, b).
  L2-condition (λs. a = Break) (L2-gets (λ-. b) [S "ret"])
  (L2-throw (a, b) [S "global-exn-var", S "ret"]))
(λr. L2-throw (Return, 2) [S "global-exn-var", S "ret"])
(λ(a, b).
  L2-condition (λs. is-local a) (L2-gets (λs. b) [S "ret"]) (L2-throw
(the-Nonlocal a) [S "global-exn-var"])))

```

?A

```

supply c-exntype.case-distrib [where h=from-xval, simp] prod.case-distrib [where
h=from-xval, simp]
  from-xval-simps [simp] c-exntype.case-cong [cong]
apply (simp add: cond-return2 if-c-exntype-cases)
apply (rule rel-spec-monad-rel-xvalI)
apply (rule rel-spec-monad-rel-xval-try-catch [split-tuple f arity: 2])
apply (rule rel-spec-monad-L2-seq-rel-xval-same-split)
apply (rule rel-spec-monad-rel-xval-try-catch [split-tuple f arity: 2])
apply (rule rel-spec-monad-L2-while-rel-xval-same-split)
apply (rule rel-spec-monad-eq-rel-xval-L2-condition)
apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])
apply (rule rel-xval.Exn)
apply simp
apply (rule rel-sum-Inl)
apply simp
apply (rule rel-sum-Inr)
apply (rule refl)
apply (simp add: SANITIZE-NAMES-def)
apply (rule rel-spec-monad-L2-seq-rel-xval-same-split)
apply (rule rel-spec-monad-eq-rel-xval-L2-condition)
apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])
apply (rule rel-xval.Exn)
apply simp
apply (rule rel-sum-Inl)
apply simp
apply (rule rel-sum-Inl)
apply (rule refl)
apply (simp add: SANITIZE-NAMES-def)
apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])

```

```

apply (rule rel-xval.Exn)
apply simp
apply (rule rel-sum-Inr)
apply (rule refl)
apply (simp add: SANITIZE-NAMES-def)
apply (rule rel-spec-monad-rel-xval-L2-gets)
apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])
apply (rule rel-xval.Exn)
apply simp
apply (rule rel-sum-Inr)
apply (rule refl)
apply (simp add: SANITIZE-NAMES-def)
done

```

```

schematic-goal rel-spec-monad (=) (=)
(L2-seq (L2-unknown [S "ret-int"]))
(λr. L2-catch
  (L2-seq
    (L2-catch
      (L2-seq
        (L2-while (λr s. True)
          (λr. L2-condition (λs. 3 < a) (L2-throw (Return, 1) [S "global-exn-var",
S "ret"]))
        (L2-seq
          (L2-condition (λs. 1 < a)
            (L2-throw (Nonlocal 1, r) [S "global-exn-var", S "ret"]))
            (L2-throw (Break, r) [S "global-exn-var", S "ret"]))
          (λra. L2-gets (λs. r) [S "ret"])))
        r [S "ret"])
      (λr. L2-gets (λs. ()) [S "ret"]))
    (λ(a, b).
      L2-condition (λs. a = Break) (L2-gets (λ-. ()) [S "ret"])
      (L2-throw (a, b) [S "global-exn-var", S "ret"])))
    (λr. L2-throw (Return, 2) [S "global-exn-var", S "ret"]))
  (λ(a, b).
    L2-condition (λs. is-local a) (L2-gets (λs. b) [S "ret"])
    (L2-throw a [S "global-exn-var"])))

```

?A

```

supply c-exntype.case-distrib [where h=from-xval, simp] prod.case-distrib [where
h=from-xval, simp]
  from-xval-simps [simp] c-exntype.case-cong [cong]
apply (simp add: cond-return2 if-c-exntype-cases)
apply (rule rel-spec-monad-rel-xvalI)
apply (rule rel-spec-monad-L2-seq-rel-xval-same-split)
apply (rule rel-spec-monad-rel-xval-L2-unknown)
apply (rule rel-spec-monad-rel-xval-try-catch [split-tuple f arity: 2])

```

```

apply (rule rel-spec-monad-L2-seq-rel-xval-same-split)
apply (rule rel-spec-monad-rel-xval-try-catch [split-tuple f arity: 2])
apply (rule rel-spec-monad-L2-seq-rel-xval-same-split)
apply (rule rel-spec-monad-L2-while-rel-xval-same-split)
apply (rule rel-spec-monad-eq-rel-xval-L2-condition)
apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])
  apply (rule rel-xval.Exn)
  apply simp
  apply (rule rel-sum-Inl)
  apply simp
  apply (rule rel-sum-Inr)
  apply (rule refl)
apply (simp add: SANITIZE-NAMES-def)
apply (rule rel-spec-monad-L2-seq-rel-xval-same-split)
apply (rule rel-spec-monad-eq-rel-xval-L2-condition)
apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])
  apply (rule rel-xval.Exn)
  apply simp
  apply (rule rel-sum-Inl)
  apply simp
  apply (rule rel-sum-Inl)
  apply (rule refl)
  apply (simp add: SANITIZE-NAMES-def)
apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])
  apply (rule rel-xval.Exn)
  apply simp
  apply (rule rel-sum-Inr)
  apply (rule refl)
  apply (simp add: SANITIZE-NAMES-def)
apply (rule rel-spec-monad-rel-xval-L2-gets)
apply (rule rel-spec-monad-rel-xval-L2-gets)
apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])
  apply (rule rel-xval.Exn)
  apply simp
  apply (rule rel-sum-Inr)
  apply (rule refl)
apply (simp add: SANITIZE-NAMES-def)
done

```

26.10.2 From $L2$ -catch and flat exceptions to $L2$ -try and nested exceptions

```

schematic-goal rel-spec-monad (=) (=)
(L2-seq (L2-unknown [S "ret-int"]))
(λr. L2-catch
  (L2-seq
    (L2-catch
      (L2-seq
        (L2-while (λr s. True)

```

```

    (λr. L2-condition (λs. 3 < a) (L2-throw (Return, 1) [ $\mathcal{S}$  "global-exn-var",
 $\mathcal{S}$  "ret'"]
      (L2-seq
        (L2-condition (λs. 1 < a)
          (L2-throw (Nonlocal 1, r) [ $\mathcal{S}$  "global-exn-var",  $\mathcal{S}$  "ret'"])
          (L2-throw (Break, r) [ $\mathcal{S}$  "global-exn-var",  $\mathcal{S}$  "ret'"])))
        (λra. L2-gets (λs. r) [ $\mathcal{S}$  "ret'"])))
      r [ $\mathcal{S}$  "ret'"])
    (λr. L2-gets (λs. ()) [ $\mathcal{S}$  "ret'"])
  (λ(a, b).
    L2-condition (λs. a = Break) (L2-gets (λ-. ()) [ $\mathcal{S}$  "ret'"])
      (L2-throw (a, b) [ $\mathcal{S}$  "global-exn-var",  $\mathcal{S}$  "ret'"])))
  (λr. L2-throw (Return, 2) [ $\mathcal{S}$  "global-exn-var",  $\mathcal{S}$  "ret'"])
  (λ(a, b).
    L2-condition (λs. is-local a) (L2-gets (λs. b) [ $\mathcal{S}$  "ret'"])
      (L2-throw a [ $\mathcal{S}$  "global-exn-var'"])))

```

?A

```

apply (simp add: cond-return2 if-c-exntype-cases cong: c-exntype.case-cong)
apply (rule rel-spec-monad-rel-xvalI)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)

```

```

apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
done

```

```

schematic-goal rel-spec-monad (=) (=)
(L2-seq (L2-unknown [S "ret-int"])
  (λr. L2-catch
    (L2-seq
      (L2-catch
        (L2-seq
          (L2-while (λr s. True)
            (λr. L2-condition (λs. 3 < a) (L2-throw (Return, 1) [S "global-exn-var"],
              S "ret"'))
            (L2-seq
              (L2-condition (λs. 1 < a)
                (L2-throw (Nonlocal 1, r) [S "global-exn-var"], S "ret"'))
                (L2-throw (Break, r) [S "global-exn-var"], S "ret"'))
                (λra. L2-gets (λs. r) [S "ret"]))
              r [S "ret"])
            (λr. L2-gets (λs. ()) [S "ret"]))
          (λ(a, b).
            (L2-condition (λs. a = Break) (L2-gets (λ-. ()) [S "ret"])
              (L2-throw (a, b) [S "global-exn-var"], S "ret"'))))
            (λr. L2-throw (Return, 2) [S "global-exn-var"], S "ret"'))
          (λ(a, b).
            (L2-condition (λs. is-local a) (L2-gets (λs. b) [S "ret"])
              (L2-throw a [S "global-exn-var"]))))

```

?A

```

apply (simp add: cond-return2 if-c-exntype-cases cong: c-exntype.case-cong)
apply (rule rel-spec-monad-rel-xvalI)
apply (rel-spec-monad-L2-rewrite)
done

```

```

schematic-goal rel-spec-monad (=) (=)
(L2-seq (L2-unknown [S "ret-int"])
  (λret-int.
    (L2-catch
      (L2-seq
        (L2-condition (λs. p = 0x20 ∨ 2 < s p)
          (L2-condition (λs. p ≠ 0x1E ∧ p ≠ 2) (L2-throw
            (Return, p) [S "global-exn-var"], S "ret"'))
          (L2-gets (λ-. ret-int) [S "ret"]))
        (L2-gets (λ-. ret-int) [S "ret"]))
      (λret-int.
        (L2-seq
          (L2-call (l2-add' undefined p p) (λglobal-exn-var.

```

```

(Nonlocal (the-Nonlocal global-exn-var), ret--int))
      [S "global-exn-var", S "ret'"]
      (λp---int. L2-throw (Return, p) [S
"global-exn-var", S "ret'"])))
      (λ(global-exn-var, ret--int).
L2-condition (λs. is-local global-exn-var) (L2-gets
(λ-. ret--int) [S "ret'"]
      (L2-throw global-exn-var [S "global-exn-var'"]))))
?A

```

```

apply (simp add: cond-return2 if-c-exntype-cases cong: c-exntype.case-cong)
apply (rule rel-spec-monad-rel-xvalI)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
done

```

```

ML <
val thm = L2-Exception-Rewrite.abstract-try-catch @{context} @term (L2-seq (L2-unknown
[S "ret--int'"])
  (λr. L2-catch
    (L2-seq
      (L2-catch
        (L2-seq
          (L2-while (λr s. True)
            (λr. L2-condition (λs. 3 < a) (L2-throw (Return, 1) [S "global-exn-var",
S "ret'"])))
          (L2-seq
            (L2-condition (λs. 1 < a)
              (L2-throw (Nonlocal 1, r) [S "global-exn-var", S "ret'"])
              (L2-throw (Break, r) [S "global-exn-var", S "ret'"])))
          (λra. L2-gets (λs. r) [S "ret'"])))

```

```

    r [S "ret'"]
    (λr. L2-gets (λs. ()) [S "ret'"])
    (λ(a, b).
      L2-condition (λs. a = Break) (L2-gets (λ-. ()) [S "ret'"])
        (L2-throw (a, b) [S "global-exn-var", S "ret'"]))
    (λr. L2-throw (Return, 2) [S "global-exn-var", S "ret'"])
    (λ(a, b).
      L2-condition (λs. is-local a) (L2-gets (λs. b) [S "ret'"])
        (L2-throw a [S "global-exn-var"]))}

```

```

schematic-goal rel-spec-monad (=) (rel-xval (=) (=))
(L2-seq (L2-unknown [S "ret-int'"])
(λr. L2-catch
  (L2-seq
    (L2-catch
      (L2-seq
        (L2-while (λr s. True)
          (λr. L2-condition (λs. 3 < a) (L2-throw (Return, 1) [S "global-exn-var",
S "ret'"])
          (L2-condition (λs. 1 < a) (L2-gets (λs. r) [S "ret'"])
            (L2-throw (Break, r) [S "global-exn-var", S "ret'"])))
          r [S "ret'"])
        (λr. L2-gets (λs. ()) [S "ret'"]))
      (λ(a, b).
        L2-condition (λs. a = Break) (L2-gets (λ-. ()) [S "ret'"])
          (L2-throw (a, b) [S "global-exn-var", S "ret'"]))
        (λr. L2-throw (Return, 2) [S "global-exn-var", S "ret'"]))
      (λ(a, b). L2-condition (λs. is-local a) (L2-gets (λs. b) [S "ret'"]) (L2-throw
a [S "global-exn-var'"])))
  ?A
  apply (simp add: cond-return2 if-c-exntype-cases cong: c-exntype.case-cong)
  apply (rel-spec-monad-L2-rewrite)
  done

```

```

schematic-goal
rel-spec-monad (=) (rel-xval (=) (=))
(L2-seq (L2-unknown [S "ret-int'"])
(λr. L2-catch
  (L2-seq
    (L2-catch
      (L2-seq
        (L2-while (λ(a, b) a. True)
          (λ(a, b).
            L2-seq
              (L2-condition (λs. 3 < a)
                (L2-throw (a, Return, 1) [S "a", S "global-exn-var", S "ret'"])
                (L2-seq
                  (L2-condition (λs. 1 < a) (L2-gets (λs. a + 1) [S "a'"])

```



```

(L2-throw (a, Break, b) [S "a", S "global-exn-var", S "ret"])
  (λr. L2-gets (λs. (r, b)) [S "a", S "ret"])
    (λ(a, b). L2-gets (λs. (a, b)) [S "a", S "ret"])
      (a, r) [S "a", S "ret"]
        (λ(a, b). L2-gets (λs. a) [S "a"])
          (λ(aa, a, b).
            L2-condition (λs. a = Break) (L2-gets (λs. aa) [S "a"])
              (L2-throw (a, b) [S "global-exn-var", S "ret"]))
            (λr. L2-throw (Return, r) [S "global-exn-var", S "ret"]))
          (λ(a, b).
            L2-condition (λs. is-local a) (L2-gets (λs. b) [S "ret"])
              (L2-throw a [S "global-exn-var"])))

```

?A

```

apply (simp add: cond-return2 if-c-exn-type-cases cong: c-exn-type.case-cong)
apply (rel-spec-monad-L2-rewrite)
done

```

```

declare [[show-main-goal=true]]

```

26.10.3 Flatten the error type of calls, aka get rid of constructor *Nonlocal*

```

schematic-goal rel-spec-monad (=) (rel-xval rel-Nonlocal (=))
(L2-seq (L2-unknown [S "ret-int"]))
  (λr. L2-catch
    (L2-seq
      (L2-catch
        (L2-seq
          (L2-while (λr s. True)
            (λr. L2-condition (λs. 3 < a) (L2-throw (Return, 1) [S "global-exn-var",
S "ret"])))
          (L2-seq
            (L2-condition (λs. 1 < a)
              (L2-throw (Nonlocal 1, r) [S "global-exn-var", S "ret"])
              (L2-throw (Break, r) [S "global-exn-var", S "ret"])))
            (λra. L2-gets (λs. r) [S "ret"])))
          r [S "ret"]
        (λr. L2-gets (λs. ()) [S "ret"])))
    (λ(a, b).
      L2-condition (λs. a = Break) (L2-gets (λ-. ()) [S "ret"])
        (L2-throw (a, b) [S "global-exn-var", S "ret"]))
      (λr. L2-throw (Return, 2) [S "global-exn-var", S "ret"]))
  (λ(a, b).
    L2-condition (λs. is-local a) (L2-gets (λs. b) [S "ret"])
      (L2-throw a [S "global-exn-var"])))

```

?A

```

apply (simp add: cond-return2 if-c-exntype-cases cong: c-exntype.case-cong)
apply (rel-spec-monad-L2-rewrite)
done

```

Some statistics on the caches involved.

```

ML <
val statistics = map (apsnd Fun-Cache.get-info) (Fun-Cache.all-handlers ())
>

```

26.11 Tuple optimization by analysing variable use and removing unused variables.

The cases for *L2-seq* are the points where the uses-analysis is invoked: `Rel_Spec_Monad_Synthesize_Ru`. The analysis follows the usage of the bound variable within the term.

schematic-goal

```

rel-spec-monad (=) (rel-xval (=) (rel-project (λv. ())))
(L2-seq (L2-gets (λ-. x) [S "x'"]))
  (λx. L2-seq
    (L2-gets (λ-. (x, y)) [S "x'",S "y'"]))
    (λ(x, y). L2-gets (λ-. y) [S "y'"]))
?X

```

```

apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
done

```

schematic-goal

```

rel-spec-monad (=) (rel-xval (=) (rel-project (λv. v)))
(L2-seq (L2-gets (λ-. x) [S "x'"]))
  (λx. L2-seq
    (L2-gets (λ-. (x, y)) [S "x'",S "y'"]))
    (λ(x, y). L2-gets (λ-. y) [S "y'"]))
?X

```

```

apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)

```

```

apply (rel-spec-monad-L2-step)
done

```

schematic-goal

```

rel-spec-monad (=) (rel-xval (=) (rel-project ( $\lambda v. v$ )))
  (L2-seq (L2-gets ( $\lambda-. (l, m)$ ) [ $\mathcal{S}$  "l",  $\mathcal{S}$  "m"] )
    ( $\lambda(x, y). L2-seq$ 
      (L2-gets ( $\lambda-. (x, k)$ ) [ $\mathcal{S}$  "x",  $\mathcal{S}$  "y"] )

      ( $\lambda(a, b).$ 
        L2-seq
          (L2-while ( $\lambda(x, y) -. x < 2$ )
            ( $\lambda(x, y). (L2-gets$  ( $\lambda-. (y, k)$ ) [ $\mathcal{S}$  "x",  $\mathcal{S}$  "y"] )
              ( $a, b$ ) [ $\mathcal{S}$  "x",  $\mathcal{S}$  "y"] )
            ( $\lambda(x, y). L2-gets$  ( $\lambda-. y$ ) [ $\mathcal{S}$  "y"] ) )))

```

?X

```

apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
done

```

declare [[*show-main-goal=true*]]

schematic-goal

```

rel-spec-monad (=) (rel-xval(=) (rel-project ( $\lambda v. v$ )))
  (L2-seq (L2-gets ( $\lambda-. (l, m)$ ) [ $\mathcal{S}$  "l",  $\mathcal{S}$  "m"] )
    ( $\lambda(x, y). L2-seq$ 
      (L2-gets ( $\lambda-. (x, k)$ ) [ $\mathcal{S}$  "x",  $\mathcal{S}$  "y"] )

      ( $\lambda(a, b).$ 
        L2-seq
          (L2-while ( $\lambda(x, y) -. y < 2$ )
            ( $\lambda(x, y). (L2-gets$  ( $\lambda-. (y, k)$ ) [ $\mathcal{S}$  "x",  $\mathcal{S}$  "y"] )
              ( $a, b$ ) [ $\mathcal{S}$  "x",  $\mathcal{S}$  "y"] )
            ( $\lambda(x, y). L2-gets$  ( $\lambda-. y$ ) [ $\mathcal{S}$  "y"] ) )))

```

?X

```

apply (rel-spec-monad-L2-step)

```

```

apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
done

```

schematic-goal

```

rel-spec-monad (=) (rel-xval (=) (rel-project ( $\lambda v. v$ )))
(L2-seq (L2-unknown [ $\mathcal{S}$  "ret--unsigned"]])
( $\lambda x. L2\text{-try}$ 
  (L2-seq
    (L2-while ( $\lambda(n\text{---int}, \text{ret--unsigned}) s. \text{True}$ )
      ( $\lambda(x1, x2).$ 
        (L2-seq (L2-guard ( $\lambda s. 0 \leq 2147483649 + \text{sint } x1 \wedge \text{sint } x1 \leq$ 
          2147483646))
          ( $\lambda xa. L2\text{-seq}$  (L2-gets ( $\lambda-. x1 + 1$ ) [ $\mathcal{S}$  "n"]])
            ( $\lambda xb. L2\text{-seq}$ 
              (L2-condition ( $\lambda s. xb < s 2$ ) (L2-throw (Inr 1) [ $\mathcal{S}$  "ret'"]])
                (L2-throw (Inr 2) [ $\mathcal{S}$  "ret'"]])
                  ( $\lambda xc. L2\text{-gets}$  ( $\lambda-. (xb, xc)$ ) [ $\mathcal{S}$  "n",  $\mathcal{S}$  "ret'"]))))))
            ( $n, x$ ) [ $\mathcal{S}$  "n",  $\mathcal{S}$  "ret'"]])
          ( $\lambda(x1, x2). L2\text{-fail}$ ))))))

```

?X

```

apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)

```

```

apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
done

```

```

schematic-goal rel-spec-monad (=) (rel-xval (=) (rel-project ( $\lambda v. v$ )))
(L2-seq (L2-call (l2-plus' undefined 1 2) ( $\lambda e. e$ ) []))
( $\lambda x. L2-seq$  (L2-call (l2-plus' undefined 2 3) ( $\lambda e. e$ ) []))
( $\lambda xa. L2-gets$  ( $\lambda s. \text{if } xa = 0 \text{ then } 1 \text{ else } 0$ ) [ $\mathcal{S}$  "ret"]))

```

?XX

```

apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
done

```

```

schematic-goal rel-spec-monad (=) (rel-xval (=) (rel-project ( $\lambda v. v$ )))

```

```

(L2-seq (L2-unknown [ $\mathcal{S}$  "ret-int"]))
( $\lambda x. L2-seq$  (L2-gets ( $\lambda-. 2$ ) [ $\mathcal{S}$  "x"]))
( $\lambda xa. L2-try$ 
(L2-seq
(L2-try
(L2-seq
(L2-while ( $\lambda(a---int, ret---int, y---int) s. True$ )
( $\lambda(x1, x2, x3).$ 
L2-seq (L2-gets ( $\lambda-. 4$ ) [ $\mathcal{S}$  "y"]))
( $\lambda xb. L2-seq$ 
(L2-condition ( $\lambda s. 3 < s x1$ ) (L2-throw (Inl (Inr 1)))
[ $\mathcal{S}$  "ret",  $\mathcal{S}$  "y"])

```

```

(L2-seq
(L2-condition ( $\lambda s. 1 < s x1$ )
(L2-seq (L2-guard ( $\lambda s. 0 \leq 2147483649 + sint$ 
 $x1 \wedge sint x1 \leq 2147483646$ ))
( $\lambda xc. L2-gets$  ( $\lambda s. x1 + 1$ ) [ $\mathcal{S}$  "a"])))
(L2-try
(L2-seq (L2-try (L2-while ( $\lambda- s. True$ ) ( $\lambda xc.$ 
L2-throw (Inr  $xc$ ) [])) ( $\lambda xc. L2-throw (Inl (Inr  $xb$ ) []))))
( $\lambda xc. L2-gets$  ( $\lambda-. (xc, x2)$ ) [ $\mathcal{S}$  "a",  $\mathcal{S}$  "ret"])))$ 
```



```

apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
done

```

```

schematic-goal rel-spec-monad (=) (rel-xval (=) (rel-project ( $\lambda v. v$ )))
  (L2-condition ( $\lambda s. n = 0$ ) (L2-gets ( $\lambda s. 0$ ) [S "ret"])
    (L2-seq (L2-call (l2-even' (recguard-dec rec-measure') ( $n - 1$ )) ( $\lambda e. e$ ) []))
    ( $\lambda x. L2-gets$  ( $\lambda s. SCAST(32 \text{ signed} \rightarrow 32)$  (if  $x = 0$  then  $1$  else  $0$ ))
      [S "ret"])))

```

?X

```

apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
done

```

```

schematic-goal rel-spec-monad (=) (rel-xval (=) (rel-project ( $\lambda v. v$ )))
  (L2-seq (L2-unknown [S "ret--unsigned"])
    ( $\lambda x. L2-condition$  ( $\lambda s. n = 0$ ) (L2-gets ( $\lambda s. 0$ ) [S "ret"])
      (L2-try
        (L2-seq
          (L2-call
            (l2-fac-exit' (recguard-dec rec-measure') ( $n - 1$ ))
            ( $\lambda e. Inl$  (Nonlocal (the-Nonlocal  $e$ ))) [S "ret"])
            ( $\lambda xa. L2-throw$  (Inr  $xa$ ) [S "ret"]))))))

```

?X

```

apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)

```

```

apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
done

```

```

schematic-goal rel-spec-monad (=) (rel-xval (=) (rel-project ( $\lambda v. v$ )))
(L2-try
 (L2-call (l2-just-exit' undefined)
 ( $\lambda e. \text{Inl } (\text{Nonlocal } (\text{the-Nonlocal } e))$ ) [])) []
)
?X
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
done

```

```

declare [[show-main-goal=false]]
declare [[verbose=0]]
declare [[verbose-timing=0]]

```

26.12 Locales, Local-Theories, Named-Targets, Morphisms and Declarations...

We explore some of the concepts of Locales, Local-Theories, Named-Targets, Declarations etc. Besides the corresponding chapters in the Isabelle/Isar Reference Manual as well as the Isabelle/Isar Implementation Manual, the following references give a solid theoretical background:

- Local Theories: http://isabelle.in.tum.de/~haftmann/pdf/local_theory_specifications_haftmann_wenzel.pdf
- Locales: <http://www21.in.tum.de/~ballarin/publications/jar2013.pdf>
- Morphisms and Declarations: <https://www21.in.tum.de/~wenzelm/papers/context-methods.pdf>

Roughly speaking, locales provide the infrastructure to manage the dependency graph of locale declarations. When entering a locale this

dependency graph is traversed and results in a blue-print to build a local context by expanding and evaluating the locale expressions in a canonical order. The result is presented to the user as a local theory. A local theory consists of an axiomatic part (think of fixes and assumes) and a derived part, consisting of definitions and theorems within the axiomatic part.

```
locale Foo =
  fixes N::nat
  fixes M::nat
  assumes N-le-M:  $N < M$ 
```

Locale *Foo* is blue-print to build a local context. We can enter this context with **context**.

```
context Foo
begin
```

Within the context we can explore it. As an example the locale assumption becomes a theorem within the context

```
thm N-le-M
```

Within the context the axiomatic part (fixes and assumes) are often accessed implicitly. For example the fixes are presented as free Variables with the declared type fixed.

```
ML <@{assert} (@{term N} = Free (N, @{typ nat}))>
```

The locale assumptions are wrapped up in the locale predicate *Foo*, and internally become part of the hypothesis of a (local theory) theorem. Theorems derived within the local theory will all carry around this implicit hypothesis.

```
declare [[show-hyps]]
thm N-le-M
ML <@{thm N-le-M}>
thm Foo-def
```

There is also an exported version of the theorem that holds on the theory level. Here the implicit hypothesis becomes an explicit precondition.

```
thm Foo.N-le-M
```

A definition in a local theory has multiple effects. A generalised definition, where the locale fixes become explicit parameters is issued in the background theory. Moreover, an abbreviation that hides these parameters is introduced.

```
definition G-def:  $G = M + N$ 
```

```
lemma  $G = \text{Foo.G } N \ M..$ 
```

ML $\langle @\{\text{assert}\} (@\{\text{term } G\} = @\{\text{term } \text{Foo}.G \ N \ M\}) \rangle$

When working within ML we also deal with another view, the *auxiliary context* as referred to in the Isabelle documentation. On the Isar top-level we are usually only exposed to the *background theory*, e.g. $\text{Foo}.G$ and the *target context*, e.g. G .

local-setup $\langle \text{fn } lthy0 \Rightarrow$
let

val $((\text{trm}, (\text{def-name}, \text{def-thm})), lthy1) = lthy0 \ |>$
Specification.definition (*SOME* (*Binding.name* F , *NONE*, *NoSyn*))
 $\square \square$ (*Binding.empty-atts*, $@\{\text{prop } F = G + M\}$)

— $lthy1$ is now the auxiliary context. Here F aka. trm is not an abbreviation but acutally a fixed variable in that auxiliary context.

val $- = @\{\text{print}\} \text{trm}$
val $- = @\{\text{assert}\} (\text{trm} = \text{Free } (F, @\{\text{typ } \text{nat}\}))$

— Theorems within this context will have an additional hypothesis reflecting the definition equation of the F , $F \equiv \text{Foo}.F \ N \ M$. This can be explored already in *def-thm*.

val $- = \text{writeln } (\text{Thm.string-of-thm } lthy1 \ \text{def-thm})$

— One can switch between the different views / versions of a theorem by *exporting* it, either by means of an explicit export-morphism or by an export function. Out of the auxiliary context into the target context (view of the locale):

val $[\text{target-def-thm}] = \text{Proof-Context.export } lthy1 \ lthy0 \ [\text{def-thm}]$
val $- = \text{writeln } (\text{target-def-thm}: \ \hat{\ } \ \text{Thm.string-of-thm } lthy1 \ \text{target-def-thm})$
val $\text{phi-target} = \text{Proof-Context.export-morphism } lthy1 \ lthy0$
val $\text{target-def-thm}' = \text{Morphism.thm } \text{phi-target} \ \text{def-thm}$
val $- = \text{writeln } (\text{target-def-thm}': \ \hat{\ } \ \text{Thm.string-of-thm } lthy1 \ \text{target-def-thm}')$

— Out on the theory level.

val $\text{thy-lthy} = \text{Proof-Context.init-global } (\text{Proof-Context.theory-of } lthy1)$
val $[\text{theory-def-thm}] = \text{Proof-Context.export } lthy1 \ \text{thy-lthy} \ [\text{def-thm}]$
val $- = \text{writeln } (\text{theory-def-thm}: \ \hat{\ } \ \text{Thm.string-of-thm } lthy1 \ \text{theory-def-thm})$
val $\text{phi-thy} = \text{Proof-Context.export-morphism } lthy1 \ \text{thy-lthy}$
val $\text{theory-def-thm}' = \text{Morphism.thm } \text{phi-thy} \ \text{def-thm}$
val $- = \text{writeln } (\text{theory-def-thm}': \ \hat{\ } \ \text{Thm.string-of-thm } lthy1 \ \text{theory-def-thm}')$

in

lthy1

end

\rangle

Here we have the target view of the freshly defined F .

ML $\langle @\{\text{assert}\} (@\{\text{term } F\} = @\{\text{term } \text{Foo}.F \ N \ M\}) \rangle$

One notable logical difference of the view on a definition within a auxiliary context and the target context is polymorphism. Within the auxiliary context the definition is always monomorphic, maybe referring to implicitly

fixed types 'variables', or better frees. Within the target context it might be polymorphic (depending on the definition), as it is just an abbreviation for the global constant applied to the locale parameters.

The fixed view of the auxiliary context can be quite handy when continuing work with the freshly defined function, do some proofs etc. in ML.

Working with nested contexts, like fixing and assuming new stuff opening and closing them and exporting results is supported within ML by `Local_Theory.begin_nested` and `Local_Theory.end_nested`. The term might be a little bit misleading as you don't actually open an close a target but open and close a context block within the same target.

The following example illustrates this first with toplevel commands and then with ML.

```
context
  fixes K
  assumes M-le-K:  $M < K$ 
begin
lemma N-le-K:  $N < K$ 
  by (rule less-trans [OF N-le-M M-le-K])
```

Here we have the local view on the theorem within the nested context.

```
thm N-le-K
end
```

Now we have the exported view of the theorem within the target context.

```
thm N-le-K

local-setup <fn lthy0 =>
  let
    val (-, lthy) = lthy0 |> Local_Theory.begin_nested
    val ([K], lthy) = Variable.add-fixes [K] lthy
    val - = writeln (is-fixed K: ^@{make-string} (Variable.is-fixed lthy K))
    val ([M-le-K], lthy-inner) = Assumption.add-assumes [(Thm.cterm-of lthy (Syntax.read-prop
lthy  $M < K$ ))] lthy
    val thm = @{thm less-trans} OF @[{thm N-le-M}, M-le-K]
    val (([th-name,thm]), lthy) = Local_Theory.notes [((Binding.name N-le-K',[]),
[[thm], []])] lthy-inner
    val - = writeln (thm: ^ Thm.string-of-thm lthy thm)
    val lthy = lthy |> Local_Theory.end_nested
    val - = writeln (is-fixed K: ^@{make-string} (Variable.is-fixed lthy K))
    val [target-thm] = Proof_Context.export lthy-inner lthy [thm]
    val - = writeln (target-thm: ^ Thm.string-of-thm lthy target-thm)
  in
    lthy
  end
>
```

Here we have the exported view of the theorem within the target context.

```
thm N-le-K'
```

```
end
```

After a definition in a local theory the context is extended with a fixed variable for the lhs and an equation that relates it to the global definition. So everything what happens thereafter happens within that extended context.

```
ML-val <  
  val lthy = Named-Target.theory-init @{theory}  
  val ((lhs,(name, thm)), lthy) = Local-Theory.define ((binding <foo>, Mixfix.NoSyn),  
  ((Binding.empty, []), @{term 42::nat})) lthy  
  val prems = Assumption.all-prems-of lthy  
>
```

By surrounding the definition with a `Local_Theory.begin_nested` and `Local_Theory.end_nested` you can step out of the extended context again. Don't forget to export the results as well.

```
ML-val <  
  val lthy = Named-Target.theory-init @{theory} |> Local-Theory.begin-nested |>  
  snd  
  val ((lhs,(name, thm)), lthy) = Local-Theory.define ((binding <foo>, Mixfix.NoSyn),  
  ((Binding.empty, []), @{term 42::nat})) lthy  
  val lthy' = Local-Theory.end-nested lthy  
  val prems = Assumption.all-prems-of lthy'  
  val [thm'] = Proof-Context.export lthy lthy' [thm]  
>
```

26.13 Morphisms and Declarations

As we have seen before, when working with locales and local theories, we deal with different views on the same results (like theorems). The locale infrastructure takes care to provide the results in the 'expected' way by a careful naming policy for facts and (local) constants.

When maintaining results in custom (generic) context data, Isabelle provides mechanisms around the notion of a *declaration*, which comes in various flavours. In particular attached to (local) theorems / facts in the form of attributes (rule and declaration attributes) or as local theory declarations. In the case of attributes the morphism is implicit, as the theorem / facts they are attached to are already in the expected localized version. So the function can refer to these theorem / facts. In case of local theory declarations the morphism is explicitly passed as an argument to the function.

In ML, attributes are functions from `thm * Proof.context -> thm * Proof.context`, in practice presented as either a declaration attribute (modifying the context) `thm -> Proof.context -> Proof.context` or as a rule attribute (modifying the theorem) `Proof.context -> thm -> thm`. Local

theory declarations are functions, taking a morphism and modifying the `local_theory`: `Morphism.morphism -> local_theory -> local_theory`.

All these declarations are part of the context-building-infrastructure of locales and local theories. In that sense a local theory declaration can be thought of as declaration attribute attached to a dummy fact. Whenever a context is entered and the fact is processed the declarations are reevaluated on the that context. As we have seen, at each point there are three contexts simultaneously, representing the background theory, the target context and the auxiliary context. So at each point there are also three views on the data, 'viewed' via the application of the explicit or implicit morphisms.

Note that when implementing context data and tools, morphisms come in two variants. Import and export morphisms. Exporting results or data out of a context into a more general context can often be treated explicit by `Proof_Context.export` or `Proof_Context.export_morphism` or variants of those functions. This is because going out of a context is unambiguous and therefor a single morphism is enough.

Whereas when entering into a context (like entering a locale) the morphisms are internally generated and provided by the locale infrastructure. As a locale might be imported several times in different variants into the same local theory there is no unambiguous morphism. Each import provides a different morphism.

When issuing a local theory declaration via `Local_Theory.declaration` there is also two flags to define, represented as a record `{syntax: bool, pervasive: bool}`.

- A pervasive declaration is also applied to the background-theory (besides the target and auxiliary context)
- A syntax declaration is activated already in the syntax phase of the locale roundup. This means, when your context data already has to be ready to enabling parsing of new assumptions declaring a new locale you have to set the syntax flag.

Note that theorem attributes are treated like `{pervasive = false, syntax = false}`. That means that they are not evaluated on the background-theory version of the facts. This makes perfect sense for typical attributes like *simp*. The theorem is only added to the simpset within the locale. The background-theorem is not added to the simpset of the theory. Keep in mind that the background-theorem is typically an implication with the locale predicate as precondition. However, when you interpret the locale on the theory level the interpreted version of the theorem is added to the simpset of the theory, here the locale-predicate precondition is discharged.

In the following we have some examples illustrating the described behaviour.

ML <

```
structure MyData = Generic-Data (  
  type T = int;  
  val empty = 0;  
  val merge = fst;  
)  
>
```

ML <

```
val n-theory = MyData.get (Context.Theory @ {theory})  
val - = @ {assert} (n-theory = 0)  
val n-aux = MyData.get (Context.Proof @ {context})  
val - = @ {assert} (n-aux = 0)  
>
```

setup <

```
Context.theory-map (MyData.put 1)  
>
```

ML <

```
val n-theory = MyData.get (Context.Theory @ {theory})  
val - = @ {assert} (n-theory = 1)  
val n-aux = MyData.get (Context.Proof @ {context})  
val - = @ {assert} (n-aux = 1)  
>
```

context Foo

begin

ML <

```
val n-theory = MyData.get (Context.Theory @ {theory})  
val - = @ {assert} (n-theory = 1)  
val n-aux = MyData.get (Context.Proof @ {context})  
val - = @ {assert} (n-aux = 1)  
val n-target = MyData.get (Context.Proof (Local-Theory.target-of @ {context}))  
val - = @ {assert} (n-target = 1)  
>
```

Manipulating data by functions like `Context.proof_map` will only have very limited and especially temporary effect. It is applied to the auxiliary context only. When exiting the auxiliary context it will be away. Also it has no effect on the target context.

local-setup

<fn lthy =>

let

```
  val lthy = lthy |> Context.proof-map (MyData.put 2)  
  val n-aux = MyData.get (Context.Proof lthy)  
  val - = @ {assert} (n-aux = 2)  
  val n-target = MyData.get (Context.Proof (Local-Theory.target-of lthy))
```

```

    val - = @{assert} (n-target = 1)
    val n-theory = MyData.get (Context.Theory (Proof-Context.theory-of lthy))
    val - = @{assert} (n-theory = 1)

  in
    lthy
  end
>

```

You might be surprised to see that the value is already reset here. The reason is that every toplevel command resets the context and is treated like a block. Think of a `Local_Theory.begin_nested / Local_Theory.end_nested`.

```

ML <
    val n-aux = MyData.get (Context.Proof @{context})
    val - = @{assert} (n-aux = 1)
  >

```

A non pervasive declaration takes effect on the auxiliary and the target context but not the theory context.

```

local-setup <fn lthy =>
  let
    val lthy = lthy |> Local_Theory.declaration {pervasive=false, syntax=false,
    pos=here} (fn phi => MyData.put 3)
    val n-aux = MyData.get (Context.Proof lthy)
    val - = @{assert} (n-aux = 3)
    val n-target = MyData.get (Context.Proof (Local_Theory.target-of lthy))
    val - = @{assert} (n-target = 3)
    val n-theory = MyData.get (Context.Theory (Proof-Context.theory-of lthy))
    val - = @{assert} (n-theory = 1)
  in lthy end
>

```

As expected the effect is still there at this point.

```

ML <
    val n-theory = MyData.get (Context.Theory (Proof-Context.theory-of @{context}))
    val - = @{assert} (n-theory = 1)
    val n-aux = MyData.get (Context.Proof @{context})
    val - = @{assert} (n-aux = 3)
    val n-target = MyData.get (Context.Proof (Local_Theory.target-of @{context}))
    val - = @{assert} (n-target = 3)
  >
end

```

Note that the declaration is persistent in the sense that when reentering the locale it will be evaluated again. So when designing custom data and implementing declarations or attributes one has to be clear about the order of things and that they might be applied in many different situations. Using `MyData.map` instead of `MyData.put` might be handy to design robust data management.

```

context Foo
begin
ML <
  val n-theory = MyData.get (Context.Theory (Proof-Context.theory-of @{context}))
  val - = @{assert} (n-theory = 1)
  val n-aux = MyData.get (Context.Proof @{context})
  val - = @{assert} (n-aux = 3)
  val n-target = MyData.get (Context.Proof (Local-Theory.target-of @{context}))
  val - = @{assert} (n-target = 3)
  >

```

A pervasive declaration also effects the theory.

```

local-setup <fn lthy =>
  let
    val lthy = lthy |> Local-Theory.declaration {pervasive=true, syntax=false,
pos=here} (fn phi => MyData.put 4)
    val n-aux = MyData.get (Context.Proof lthy)
    val - = @{assert} (n-aux = 4)
    val n-target = MyData.get (Context.Proof (Local-Theory.target-of lthy))
    val - = @{assert} (n-target = 4)
    val n-theory = MyData.get (Context.Theory (Proof-Context.theory-of lthy))
    val - = @{assert} (n-theory = 4)
  in lthy end
  >

```

```

ML <
  val n-theory = MyData.get (Context.Theory (Proof-Context.theory-of @{context}))
  val - = @{assert} (n-theory = 4)
  val n-aux = MyData.get (Context.Proof @{context})
  val - = @{assert} (n-aux = 4)
  val n-target = MyData.get (Context.Proof (Local-Theory.target-of @{context}))
  val - = @{assert} (n-target = 4)
  >

```

end

```

ML <
  val n-theory = MyData.get (Context.Theory @{theory})
  val - = @{assert} (n-theory = 4)
  val n-aux = MyData.get (Context.Proof @{context})
  val - = @{assert} (n-aux = 4)
  >

```

26.13.1 Excuse on Proof Context vs. Local Theory.

A local theory is represented as a proof context. So a local theory is a semantic subtype of a proof context. Every local theory is a proof context but not every proof context is a local theory. In particular a local theory is a proof context that is ready to accept (local theory) definitions

(`Local_Theory.define`) and notes (`Local_Theory.note`).

The Isabelle sources try to be consistent in the naming, i.e. 'ctxt' vs 'lthy' in parameters and variables and `Proof.context` vs. `local_theory` in signatures. But as `local_theory` is only a type synonym for `Proof.context`, there is no type check for this convention. The nesting level that you get with `Local_Theory.level` is an indicator. A Level of 0 is not a proper local theory.

With `Proof_Context.init_global` you get a proof context not a local theory.

```
ML-val <
val ctxt = Proof-Context.init-global @{theory}
val level = Local-Theory.level ctxt
val is-theory = Named-Target.is-theory ctxt
val locale = Named-Target.locale-of ctxt
val bottom-locale = Named-Target.bottom-locale-of ctxt
>
```

With `Named_Target.theory_init` you get a local theory.

```
ML-val <
val lthy = Named-Target.theory-init @{theory}
val level = Local-Theory.level lthy
val is-theory = Named-Target.is-theory lthy
val locale-of = Named-Target.locale-of lthy
val bottom-locale = Named-Target.bottom-locale-of lthy
>
```

With `Locale.init` you get a proof context not a local theory.

```
ML-val <
val ctxt = Locale.init (Locale.intern @{theory} Foo) @{theory}
val level = Local-Theory.level ctxt
val is-theory = Named-Target.is-theory ctxt
val locale-of = Named-Target.locale-of ctxt
val bottom-locale = Named-Target.bottom-locale-of ctxt
>
```

With `Named_Target.init` you get a local theory.

```
ML-val <
val lthy = Named-Target.init [] (Locale.intern @{theory} Foo) @{theory}
val level = Local-Theory.level lthy
val is-theory = Named-Target.is-theory lthy
val locale-of = Named-Target.locale-of lthy
val bottom-locale = Named-Target.bottom-locale-of lthy
>
```

26.13.2 Attributes vs. Local Theory Declarations

Attributes on local facts, are not applied on the underlying theory level foundation. Only when you interpret a locale on the theory level the attributes

get activated. This corresponds to general `Local_Theory.declaration` where you have the flag `pervasive=false`.

To demonstrate this we define an artificial tracing attribute.

```

attribute-setup trace-attr =
<
  Attrib.thms >> (fn ths =>
    Thm.declaration-attribute
      (fn thm => fn context =>
        let
          val (kind, level, is-theory, locale, bottom-locale) = case context of
            Context.Proof ctxt =>
              let
                val level = Local_Theory.level ctxt
              in
                (if level = 0 then context else local-theory,
                 string-of-int level,
                 @{make-string} (Named_Target.is-theory ctxt),
                 @{make-string} (Named_Target.locale-of ctxt),
                 @{make-string} (Named_Target.bottom-locale-of ctxt))
              end
            | - => (theory, 0, true, NONE, NONE)

          val - = tracing (trace-attr ( ^ kind ^, ^ level ^, ^
            is-theory ^, ^ locale ^, ^ bottom-locale ^): ^
            Thm.string-of-thm (Context.proof-of-context) thm)
            in context end))
>

```

First we try the attribute on a theory level theorem. When you put the cursor on or after the `simp` you see the tracing output of our attribute.

```

lemma foo[trace-attr]: x = x
  by simp

```

Interestingly we see four tracing outputs. Here the outputs and my interpretation:

- `trace-attr (local-theory, 1, true, NONE, NONE): ?x = ?x` This seems to be the immediate auxiliary context of the lemma.
- `trace-attr (theory, 0, true, NONE, NONE): ?x = ?x` This is the theory level.
- `trace-attr (context, 0, false, NONE, NONE): ?x = ?x` This seems to be a theory-context that might be generated with `Proof_Context.init_global`.
- `trace-attr (local-theory, 1, true, NONE, NONE): ?x = ?x` This seems to be from reentering the original lemma context.

Now let us look at a nested context.

```
context Foo
begin
context
begin
lemma silly[trace-attr]: M = M by simp
end
end
```

As you see there is no theory level trace:

- *trace-attr (local-theory, 2, false, NONE, SOME Scratch.Foo): M = M [Foo N M]* This is the original context of the lemma.
- *trace-attr (context, 0, false, NONE, SOME Scratch.Foo): M = M [Foo N M]* This seems to be the target context but not as a local theory.
- *trace-attr (local-theory, 1, false, SOME Scratch.Foo, SOME Scratch.Foo): M = M [Foo N M]* This seems to be the target context as a local theory.
- *trace-attr (local-theory, 2, false, NONE, SOME Scratch.Foo): M = M [Foo N M]* This seems to be from reentering the original lemma context.

Let us now try an interpretation of the locale on the theory level.

```
definition NN::nat where NN = 2
definition MM::nat where MM = 3
```

```
interpretation Foo NN MM
by (unfold-locales) (simp add: NN-def MM-def)
```

We can see the trace of the activation of the attribute: *trace-attr (theory, 0, true, NONE, NONE): MM = MM*

26.14 ML Antiquotations

The notion of quote / antiquote is related to lisps quotation and quasi-quotation mechanisms or the notion of 'interpolation' that is used in the context of macros in other programming languages. In lisp, "everything is a list", in particular the program itself and you can dynamically evaluate a list. With quotation you can express that a list is meant to be plain data and should not be immediately evaluated. With quasi-quotation you can

build data which inside uses a lisp-expression that evaluates / expands to some data that is inserted at that point.

This is a first intuition to understand ML antiquotations in Isabelle/ML. An ML antiquotation is something embedded into the Isabelle/ML program text, that is not itself plain ML, but something that evaluates to a piece of ML text that is inserted in that position. A prominent example is the term antiquotation, which transforms a logical term (presented in the inner-syntax) of the logic, to the ML representation of that term. `term`

```
ML-val <
term <x + y>
>
```

The expansion takes place during compile time of the piece of Isabelle/ML. To be more precise it is expanded already during lexing of Isabelle/ML. This is important to keep in mind when writing your own antiquotations. The context you can refer to during the expansion of the antiquotation is the context you have during the lexing phase. This design decision facilitates robustness and predictability of the resulting ML code. Note that the expanded ML-code itself might of course be dynamically used later on in various contexts, but it is always the same code.

The main interface to write your own ML antiquotations is `ML_Antiquotation`. The more low-level (and more flexible) interface it is based on is `ML_Context.add_antiquotation`. The interface in `ML_Antiquotation` distinguishes the antiquotations into the variants *inline*, *value* and *special-form*.

The *inline* variant is the closest to the intuition of macro expansion. The inline-antiquotation yields a piece of ML-text that is literally inserted at the position of the antiquotation. The term antiquotation is an example of this. `\<^term>\<open>x + y\<close>`. It is implemented with `ML_Antiquotation.inline_embedded`.

¹

For the *value* variant the intuition is that the expanded antiquotation is immediately evaluated in the compile time context and the resulting value is what is inserted at the position of the antiquotation. An example is the `cterm` antiquotation, e.g. `@{cterm "a = b"}`.

Following the static nature of antiquotations this abstract value is produced statically during compile time and is bound to some value as in *val x*

¹Note that there is also a variant without the 'embedded' suffix `ML_Antiquotation.inline`. What is the difference? According to a mail thread <https://lists.cam.ac.uk/mailman/htdig/cl-isabelle-users/2021-October/msg00023.html> the 'embedded' is a hint to Isabelle that antiquotation expects its argument as an 'embedded language' enclosed by a cartouche. This can nowadays be considered as the canonical way. Historically, Isabelle first distinguished between the outer-syntax (e.g. Isar or plain ML as Meta-Language) and the inner-syntax of the object logic like HOL or FOL. Meanwhile Isabelle embraces a lot of different languages that in some cases can even be nested, and the languages can be dynamically extended by declaring new commands or antiquotations.

= ...*compile-time-context*..., and then this value x is what appears in the expanded ML text. So to implement such an antiquotation means to provide two main ingredients: the code for the value binding (referred to as environment) and the code to reference that value (referred to as body). This is what the interface `ML_Antiquotation.declaration` offers. The argument `Proof.context -> string * string` is the central point. The pair of strings denotes the ML-text for the environment and the body respectively. See for example the theorem antiquotation: `@{thm refl}`. This can also be considered a *value* antiquotation, albeit being implemented by the more low-level interface.

Digging even deeper into `ML_Context.add_antiquotation` the `ML_Context.decl` also refers to a pair of ML-code, denoting an environment and the body, but here presented as `ML_Lex.token list` not as a string. This interface gives a lot of flexibility into the design of antiquotations. A notable example is the 'instantiate'-antiquotation. e.g:

```
ML-val <
fun foo t =
  instantiate <'a =typ <nat> and a =term <s::nat> and b = t in
    prop <a + b = b + a> for a b::<'a::plus>>
  >
```

Note the nesting of antiquotations in that example. The 'environment-part' here consists of all the right hand sides of the instantiations as well as the proposition. The 'body-part' is an ML-expression that instantiates the proposition with the terms. The body part cannot be immediately evaluated during compile time as the value of parameter t of the function is not yet known. So the result is an ML-expression that references t among the right and sides of the instantiations that are evaluated at compile time.

The special-form antiquotation packs the text in a function `think`, e.g. `\<^try>\<open>I\<close>`.

26.15 Markup and Reports

Isabelle provides a rich set of markup to display information in Isabelle/jEdit. Markup can either be directly attached to a string, or an existing source text can be decorated by sending *reports* to PIDE. An example for direct markup is the printing of terms, e.g.:

```
ML <
val str = Syntax.string-of-term @<context> @<term a + b>
  >
```

You can make the markup visible by sending it to an output function, e.g.

```
ML <
```

```
tracing str
>
```

The markup is itself encoded in the string by the format `YXML`. The plain information (without markup) can be extracted:

```
ML-val <
YXML.content-of str
>
```

The markup can be expanded to the `XML.tree` by `YXML.parse`

```
ML-val <
YXML.parse str
>
```

To build up markup you can use `Markup`. The type for markup is a pair of element name and a list of properties. A property `Properties.T` is a key value pair which are both strings. First you build up your markup by using the various functions supplied in `Markup`, then you attach it to a string via `Markup.markup`.

```
ML-val <
val m = Markup.url http:\\www.isabelle.in.tum.de
val decorated = Markup.markup m attach url to this string
val xml = YXML.parse decorated
>
```

With `Position.report` you can add markup to a given position, e.g. here we add the warning-twigglies and an url:

```
ML-val <
val pos = Position.range-position (Position.range (here, here))
val w = Markup.warning
val u = Markup.url http:\\www.isabelle.in.tum.de
val - = Position.report pos w
val - = Position.report pos u
>
```

The standard message channels `error`, `warning` and `tracing` scan the string for position information and automatically markup the position.

```
ML-val <
val this-spot = here (* wiggly line *)
val msg = Look here: ^ Position.here this-spot
val - = warning msg
>
```

There are some fine points regarding the command region where the message is issued. By default markup is only shown in the actual command region.

```
ML <
val this-spot-is-not-marked = here (* no wiggly line *)
```

```
val msg1 = Look here: ^ Position.here this-spot-is-not-marked
>
```

We do not get the wiggly lines in the **ML** above, instead the **ML-val**

```
ML-val <
val - = warning msg1
>
```

We can put markup to a previous command if we first save the command position of the first command, and temporary use this in the second command when issuing the message.

```
ML <
val command-thread-position = Position.thread-data ();
val this-spot-is-marked = here (* wiggly line *)
val msg2 = Look here: ^ Position.here this-spot-is-marked
>
```

```
ML-val <
val - = Position.setmp-thread-data command-thread-position warning msg2
>
```

To inspect the markup attached to the source you can use the panel Sidekick and select the language "isabelle-markup" When you hover over an entity in the upper half of the panel you see the attached markup in the lower half of the panel.

Reports are also used in autocorres **Feedback** to provide warning and error markup for C programs.

An illustrative example for custom markup and reports can be seen in the document antiquotation for rail diagrams: is `$ISABELLE_HOME/src/Pure/Tools/rail.ML`. It is used extensively in the Isabelle documentation, watch out for `\<^rail>\<open>\<close>` e.g. in `$ISABELLE_HOME/src/Doc/Datatypes/Datatypes.thy`

26.16 Term Synthesize via Intro Rules

A repeated task in the various AutoCorres phases is to synthesize a typically more abstract monadic function (output) from a given monadic function (input) together with a simulation theorem that connects both versions of the function. One strategy is to formulate introduction rules for the correspondence relation for the different language constructs and then recursively apply those rules guided by the constructs in the "input" function, and as a side effect of the rule application to synthesize the "output" function. Examples are, heap abstraction, word abstraction, exception rewriting (as described above) and type strengthening. Currently there is not (yet) an uniform implementation of this strategy, but the different instances are converging.

Currently the most recent incarnation is the setup for type strengthening to perform the *refines* proofs:

- A term net is used for efficient lookup of potentially matching rules. It is indexed by the input function as well as the desired result relation.
- Splitting of tuples in rules is automatically performed to match the current term.
- Priorities of rules are specified to guarantee that the most specific rules are tried first
- A cache is used both for positive and negative proof attempts. As there might be multiple applicable rules when decomposing the input function, partial results might already be proven or have failed. Using a cache helps to prune repeated subproof attempts.

The setup for this phase can be found in `../Refines_Spec.thy`. It employs **synthesize-rules**, as defined in `../lib/ml-helpers/Synthesize.thy`. This command can be seen as generalisation of **named-theorems**, with support for the kind of features just sketched.

For a set of **synthesize-rules** you can generate patterns to support indexing of rules via subterms: **add-synthesize-pattern**.

With **print-synthesize-rules** you can inspect the set:

print-synthesize-rules *pure*

You can also supply an term to select the matching rules:

print-synthesize-rules *pure* $\langle \text{Trueprop } (\text{refines-spec } (L2\text{-try } f) - (\text{return } ?f') - (\text{rel-prod } \text{rel-liftE } (=)))) \rangle$

Rules are added via attribute *synthesize-rule*, where you can also specify the rule sets the priority and whether to split some variables. In order to preserve the theorem name for debugging purposes you should apply that attribute in a separate **lemmas** and not directly in the original **lemma**

The main tactic to drive application of those rules is `CT.cache_deepen_tac`. This tactic has type `context_tactic` and supports implementing a cache within the `Proof.context`. The user provides a cache and a single-step tactic that is recursively tried on the emerging subgoals.

The interface of the cache is captured in record `CT.ctx_cache`:

- **#lookup**: `CT.ctx_cache -> Proof.context -> cterm -> int -> context_tactic` The lookup function gets the goal presented as a `cterm` and returns a subgoal tactic. A cache miss is implemented by `K CT.no_tac`.

- **#insert:** `CT.ctx_cache -> (Timing.timing * int * int) -> thm -> Proof.context -> Proof.context` The insert functions gets timing information tupled with the total number of alternatives and the current alternative as input together with the just proven subgoal. It can use that information to decide whether to really extend the cache or to ignore the result.
- **#propagate:** `CT.ctx_cache -> Proof.context -> Proof.context -> Proof.context` To propagate the cache throughout backtracking the propagate function is supplied. It gets the current context and the old context as argument and can propagate the cache from the current context to the old one (to which it backtracks).

The cache used for the type strengthening phase is implemented in `Monad_Convert.sim_nondet`. It is based on `Synthesize_Rules.gen_cond_cache`. It uses a term net to index the cached results and only adds rules for compound statements like *L2-seq*, not for atomic ones. The idea here is that proving simulation for an atomic statement is just a single rule application and is thus not worth to be cached.

Failed proof attempts are represented by $FALSE \implies PROP ?P$ instantiated with the failed subgoal. This is how they are presented to the cache and can be stored. When a cache lookup is performed and the cache results in such an instance of $FALSE \implies PROP ?P$ which matches the current subgoal the proof attempt will be immediately aborted at that depth and potential alternative proof steps 'higher up' (smaller depth) may get explored.

Next we illustrate the approach with a small example.

lemma *refines-liftE-nondet*: *refines (liftE f) f s s (rel-prod rel-liftE (=))*
by (*auto simp add: refines-def-old reaches-liftE*)

lemma *refines-L2-unknown*:

shows *refines (L2-unknown ns) (L2-VARS (select UNIV) ns) s s (rel-prod rel-liftE (=))*

by (*rule Refines-Spec.refines-L2-unknown-nondet*)

lemmas *refines-spec-monad-rules* =

refines-L2-seq-nondet [*simplified THIN-def*]

refines-L2-gets-nondet

refines-L2-unknown

lemmas *refines-option-monad-rules* =

refines-L2-seq-option [*simplified THIN-def*]

refines-L2-gets-option

schematic-goal $\langle \text{refines } (L2\text{-seq } (L2\text{-gets } (\lambda\cdot. n::\text{nat}) [])) (\lambda n. L2\text{-gets } (\lambda\cdot. n + m) []) \rangle$ $(?f'::(?'c, 'a) \text{ res-monad}) ?s ?s ?R$

apply (*rule refines-spec-monad-rules*)
prefer 2

— Note that `CT.cache_deepen_tac` follows the convention of tactics to solve subgoals from the back. So the second subgoal is solved first.

apply (*rule refines-spec-monad-rules*)
apply (*rule refines-spec-monad-rules*)
done

First we define the single-step tactics, which just try some rules.

ML \langle

```
fun get-nondet-tacs goal = @{thms refines-spec-monad-rules}
  |> map (fn thm => (Utils.guess-binding-of-thm @ {context} thm, thm))
  |> CT.binding-resolve-tac
```

```
fun get-option-tacs goal = @{thms refines-option-monad-rules}
  |> map (fn thm => (Utils.guess-binding-of-thm @ {context} thm, thm))
  |> CT.binding-resolve-tac
```

```
fun get-all-tacs goal = get-option-tacs goal @ get-nondet-tacs goal
  >
```

Now we define a simple cache which stores all emerging subgoals in a list and tries to resolve with those subgoals on a lookup.

ML \langle

```
structure List-Cache = Proof-Data (
  type T = thm list
  val init = K [])
```

```
val list-cache:CT.ctx-cache = {
  lookup = fn ctxt => fn goal =>
    let
      val - = tracing (lookup: ^ Syntax.string-of-term ctxt (Thm.term-of goal))
    in
      CT.resolve-tac (List-Cache.get ctxt)
    end,
  insert = fn timing => fn thm => fn ctxt =>
    let
      val thm' = thm |> Thm.forall-elim-vars ((Thm.maxidx-of thm) + 1) |>
zero-var-indexes
      val - = tracing (insert: ^ Thm.string-of-thm ctxt thm')
    in
      ctxt |> List-Cache.map (cons thm')
    end,
  propagate = fn current => fn old => current
}
```

>

We only try nondet-rules, so we end up in the nondet monad.

schematic-goal

```
⟨refines ( L2-seq (L2-gets (λ-. n::nat) []) (λn. L2-gets (λ-. n + m) []))
  (?!f'::(!'c, 'a) res-monad) ?s ?s ?R⟩
apply (tactic ⟨
  Context-Tactic.NO-CONTEXT-TACTIC @{context} (
  CT.cache-deepen-tac (K 1) list-cache get-nondet-tacs 1
  )⟩)
done
```

schematic-goal

```
⟨refines ( L2-seq (L2-gets (λ-. n::nat) []) (λn. L2-gets (λ-. n + m) [])) (?!f'::(!'c,
'a) res-monad) ?s ?s ?R⟩
thm refines-option-monad-rules
apply (rule refines-option-monad-rules)
apply (rule refines-option-monad-rules)
apply (rule refines-option-monad-rules)
done
```

We only try the option-rules, so we end up in the option monad.

schematic-goal

```
⟨refines ( L2-seq (L2-gets (λ-. n::nat) []) (λn. L2-gets (λ-. n + m) [])) (?!f'::(!'c,
'a) res-monad) ?s ?s ?R⟩
apply (tactic ⟨
  Context-Tactic.NO-CONTEXT-TACTIC @{context} (
  CT.cache-deepen-tac (K 1) list-cache get-option-tacs 1
  )⟩)
done
```

No we try both the option-rules and the nondet-rules. As the option rules come first we end up in the option monad

schematic-goal

```
⟨refines ( L2-seq (L2-gets (λ-. n::nat) []) (λn. L2-gets (λ-. n + m) [])) (?!f'::(!'c,
'a) res-monad) ?s ?s ?R⟩
apply (tactic ⟨
  Context-Tactic.NO-CONTEXT-TACTIC @{context} (
  CT.cache-deepen-tac (K 1) list-cache get-all-tacs 1
  )⟩)
done
```

By including a *L2-unknown* in the example we have to end up in the nondet monad, as it cannot be handled in the option monad. Note the difference in the outcome of the first and second proof attempt. In the first one we only provide the nondet-rules in the second one we also provide the option-rules. This results in a different "translation" of the first *L2-gets*.

schematic-goal

```

⟨refines ( L2-seq (L2-gets (λ-. n::nat) [])) (λn. L2-unknown []) ( ?f'::( ?'c, 'a)
res-monad) ?s ?s ?R⟩
  apply (tactic ⟨
    Context-Tactic.NO-CONTEXT-TACTIC @{context} (
      CT.cache-deepen-tac (K 1) list-cache get-all-tacs 1
    )⟩)
done

```

schematic-goal

```

⟨refines ( L2-seq (L2-gets (λ-. n::nat) [])) (λn. L2-unknown []) ( ?f'::( ?'c, 'a) res-monad)
?s ?s ?R⟩
  apply (tactic ⟨
    Context-Tactic.NO-CONTEXT-TACTIC @{context} (
      CT.cache-deepen-tac (K 1) list-cache get-nondet-tacs 1
    )⟩)
done

```

end

26.17 Pointers to Local Variables

```

theory pointers-to-locals imports AutoCorres
begin

```

```

install-C-file pointers-to-locals.c

```

```

init-autocorres [addressable-fields = pair.first buffer.buf 32 word[3]]pointers-to-locals.c

```

```

autocorres [no-heap-abs = inc-untyp]pointers-to-locals.c

```

This story began with the desire to support "pointers to local variables". The idea to support "pointers to local variables" might be quite fearful as it covers lots of use cases. When trying to literally support pointers to local variables (uniform with heap pointers) one has to answer questions about the memory layout of heap and stack, how to ensure that those regions are disjoint, what happens if we run out of stack space, how do we make sure that there is no dangling pointers to a stack-frame that is already popped from the stack etc.

So let us make a step back and ask ourselves for which C-idiomatic use cases are pointers to local variables actually used. C does only support call-by-value. So pointers are sometimes used to model call-by-reference semantics, for input as well as output parameters. Especially, the use case of an additional output parameter (besides the regular return value) is often implemented as a pointer parameter. An alternative might be to return a tuple encoded as a structure instead. But as there are no ad hoc tuples in C, this requires the boilerplate to define an auxiliary structure type. So typically a pointer parameter is used instead.

Here an example of a integer addition with overflow check. The return

value is used for the status, the actual result is returned as a pointer parameter.

```
int add_check(int* result, int a, int b)
{
    *result = a + b;
    if(a > 0 && b > 0 && *result < 0)
        return -1;
    if(a < 0 && b < 0 && *result > 0)
        return -1;
    return 0;
}
```

A call to the function could be the following with a local variable for the result.

```
int res, status;
status = add_check(&res, 2, 3)
```

Here `res` is an output parameter. If C would support tuples we might have code like:

```
int add_check(int a, int b)
{
    int result = a + b;
    if(a > 0 && b > 0 && *result < 0)
        return (-1, result);
    if(a < 0 && b < 0 && *result > 0)
        return (-1, result);
    return (0, result);
}
```

```
int res, status;

(status, res) = add_check(2, 3)
```

Another use case is to have in input / output parameter implemented by a pointer parameter.

```
void inc(int *a)
{
    (*a)++;
}

int x = 42;
inc(&x);
```

This could be also modelled as explicit input and output:

```
int inc(int a)
{
    return (a++);
}
```

```
int x = 42;
x = inc(x);
```

So one general idea is to model those kind of pointer-parameters as explicit input / output parameters. From a pattern like:

```
int f(int *p1, ..., int * p_n, int a1, ..., int a_m) {
    ... (*p1) ...
    ... (*p_i) ...
    return res;
}
```

```
r = f(q1,..., q_n, b1, ..., b_m)
```

We make

```
(int * int ... * int) f(int p1, ..., int p_n, int a1, ..., int
a_m) {
    ... p1 ...
    ... p_i ...
    return (res, p1, ... p_n);
}
```

```
(r, q1, ..., q_n) = f(q1,..., q_n, b1, ..., b_m)
```

In which cases is such a model faithful? Different behaviours might occur, when pointer-parameters are aliased. Moreover, the translation scheme only works out if the body of the function only uses the dereferenced pointer variables, i.e. `*p_i` to obtain the value or perform an update, and does not actually use the literal address value stored in the variable. E.g. it does not make things like `p1 = p2` or stores the pointer value in some global data structure. All examples we have seen so far are benign with this respect, as there was only one pointer-variable involved, and we only used it to dereference it.

What about this swap function:

```
void swap1(int* p, int* q) {
    int tmp = *p;
```

```

    *p = *q;
    *q = tmp;
}

int a = 1; int b = 2;
swap1(&a, &b);
/* a = 2, b = 1 */

```

```

(int * int) swap2(int p, int q) {
    int tmp = p;
    p = q;
    q = tmp;
    return (p, q);
}

```

```

int a = 1; int b = 2;
(a, b) = swap2(a, b);
/* a = 2, b = 1 */

```

These two swap functions behave the same, even if the parameters would alias. But in general aliasing is problematic.

```

void inc_both1(int* p, int* q) {
    *p = *p + 1;
    *q = *q + 1;
}

```

```

int x = 2;
void inc_both1(&x, &x);

/* x = 4 */

```

```

(int * int) inc_both2(int p, int q) {
    p = p + 1;
    q = q + 1;
    return (p, q);
}

```

```

int x = 2;
(x, x) = inc_both2(&x, &x);

/* x = 3 */

```

Whats the relevant difference between the increment and the swap example? In the swap example, we never read from a pointer-parameter after any pointer-parameter was assigned to. So an update within the function might not alter the values we read via any pointer-parameter.

26.17.1 Design choices

C does not distinguish pointers to local variables from heap pointers they are all the same. E.g. the increment functions above could be applied to a heap location as well as a stack location. Moreover, in the context of systems programming it is common that low-level code explicitly deals with heap-layout and pointer values and "tricks" like storing some flags as bits in the pointer values themselves are often used. That is why we aim for an uniform model for stack and heap pointers and don't want to impose assumptions on the layout.

Whenever a function creates a pointer to a local variable, we model it as part of the heap. Core ideas:

- As part of ordinary heap-typing, we track free stack locations by a distinguished C type *stack-byte*. All addresses with that type are considered free stack location.
- Additionally to the typing information with *stack-byte* (that denotes free stack space) We have a fixed set of addresses \mathcal{S} that describe all the stack space (free and allocated). Note that the set \mathcal{S} does not depend on the state. The dynamic aspects are modelled within the heap typing.
- As prelude in a function that needs pointers to local variables we allocate the variable in the heap, by non-deterministically retyping a region of free stack space that fits: We retype from *stack-byte* to the type we allocate.
- All accesses / updates to the variable within the function are modelled indirectly via the pointer
- As postlude of the function we again retype the stack space as free.

Observations:

- Ordinary C functions, that do not have an AUX-UPDATE on the heap-type also do not mess around with the stack as they do not change heap-typing. So stack-space is preserved when calling a function.
- Prelude / postlude ensures that the stack typing is the same after a function call to a function that has pointers to local variables.

Stack Allocation

The central definition for stack allocation is *stack-allocs* describing the set of possible pointers and modified heap type descriptions:

$$(p, d') \in \text{stack-allocs } n \ \mathcal{S} \ \text{TYPE}('a) \ d$$

thm *stack-allocs-def*

After such an allocation we know that:

- Pointer p is a root pointer with the expected type in the new heap type description d' .
- The allocated addresses are within set \mathcal{S} .
- The new heap typing d' is obtained from the original heap typing d by retyping addresses of *stack-byte* to the type of the pointer p . The address of pointer p denotes the start the retyped region. The parameter n can be used to allocate a consecutive range of pointers in one step.

Stack allocation might fail in the sense that the resulting set of *stack-allocs* can be empty. In *Simpl* we then fail in the terminal fault state reporting *StackOverflow*. In the autocorres monad we currently have decided to ignore possible stack overflows. We simply assume that stack allocation succeeds and does not return an empty set.

Stack Release

The counter part to the stack allocation above is *stack-releases* $n \ p \ d'$.

thm *stack-releases-def*

There is no non-determinism here. It is a plain function retyping back to *stack-byte*.

Stack discipline

We encapsulate the stacking discipline of allocation / release in some functions in the various layers.

Simpl In *Simpl* we have the command *With-Fresh-Stack-Ptr* $n \ \text{init } c$ where

- n denotes the number of consecutive pointers we want to allocate, typically $1::'a$. This number was introduced to allocate local arrays and obtain pointers to the elements. However, it turns out that allocating an array pointer $('a['b]) \ ptr$ instead is sufficient. Hence the parameter is actually always $1::'a$.

- *init s* returns a set of initial values from which we non-deterministically choose one. In case the addressed local variable is a function parameter this is a singleton set containing the value of the argument. In case of an ordinary local variable this is either the the universal set of all possible values (if uninitialized) or a singleton set of the initialisation value.
- *c* expects a fresh pointer *p* and then yields a Simpl command of the body.

The idea is simple, we first allocate a pointer *p* with *stack-allocs*, then initialise the fresh heap location according to *init* then execute the body *c p* and finally release the stack pointer again.

thm *globals.With-Fresh-Stack-Ptr-def*
term *stack-heap-state.With-Fresh-Stack-Ptr*

A central role of the definition is the *DynCom* statements, which is the means of Simpl to bind certain execution points and provide state dependent commands. As the body of a *DynCom* always depends on the complete state (and not such a thing like the pointer *p*) we use the function *allocated-ptrs* calculate the fresh pointer from the typing information.

thm *allocated-ptrs-def*
thm *stack-allocs-allocated-ptrs*

Note that *spec-monad.Spec* might fail if the resulting set is empty, which is exposed to the failure *StackOverflow* of the command.

L1 / L2 In both L1 and L2 we use *globals.with-fresh-stack-ptr n init f*, where *n* as well as *init* are just the same as in Simpl and *f* is a monadic computation depending on the fresh pointer *p*.

term *globals.with-fresh-stack-ptr*
thm *globals.with-fresh-stack-ptr-def*
term *stack-heap-state.with-fresh-stack-ptr*

In contrast to Simpl we assume that stack allocation succeeds by using *assume-result-and-state*.

HL / Split Heap

In the split heap model there is no singleton heap anymore. Hence we introduce a family of functions depending on the type that is allocated.

term *heap-w32.with-fresh-stack-ptr*
thm *heap-w32.with-fresh-stack-ptr-def*
term *heap-w32.guard-with-fresh-stack-ptr*
thm *heap-w32.guard-with-fresh-stack-ptr-def*

```

term heap-w32.assume-with-fresh-stack-ptr
thm heap-w32.assume-with-fresh-stack-ptr-def
term typ-heap-typing.with-fresh-stack-ptr
term typ-heap-typing.guard-with-fresh-stack-ptr
term typ-heap-typing.assume-with-fresh-stack-ptr

```

While stack allocation remains the same upon stack release some more work has to be done in order to make the simulation work.

In the monolithic heap the stack release of a pointer p immediately results in $\text{plift}(h, \text{stack-releases } n \ p \ d') \ p = \text{None}$. So the simulation demands $\text{the-default ZERO}'a) (\text{plift}(h, \text{stack-releases } n \ p \ d') \ p) = \text{zero}$.²

So to simulate this in the split heap we explicitly zero out the memory. Moreover, for the simulation to work we have to argue that the stack release did not mess up with any of the other split heaps. This argument can be given if know that pointer p is still a valid root pointer before we do the release. As we know that p is a valid root pointer immediately after the allocation we have to establish that this is still true at stack release. Most C function do not alter the heap typing at all. So it can easily be shown that $f \cdot s \ \text{?}\{\lambda r. \text{unchanged-typing-on } \mathcal{S} \ s \ \}\} holds and thus that p is still a valid root pointer after executing f . In case we cannot automatically proof the preservation of typing information, we enforce that p is still a valid root pointer by inserting a guard. So for code that changes the heap typing in more involved ways the argument is left for the user to prove.$

```

thm lifted-globals.unchanged-typing-on-def
term heap-typing-state.unchanged-typing-on

```

Some syntax

```

term PTR-VALID(32 word)
term IS-VALID(32 word) s
term IS-VALID(32 word) s p

```

```

context ts-definition-inc
begin
thm ts-def
end

```

Pointer to an uninitialized local variable.

```

context call-inc-local-uninitialized-impl
begin
thm call-inc-local-uninitialized-body-def
end

```

```

context ts-definition-call-inc-local-uninitialized
begin

```

²c.f. `open_struct.thy` for details on heap lifting simulation

thm *ts-def*
end

Pointer to an initialized local variable.

context *ts-definition-call-inc-local-initialized*
begin
thm *ts-def*
end

Pointer to parameter.

context *ts-definition-call-inc-parameter*
begin
thm *ts-def*
end

When we cannot prove that the heap typing is unchanged we fall back to *heap-w32.guard-with-fresh-stack-ptr* instead of *heap-w32.assume-with-fresh-stack-ptr*

context *ts-definition-call-inc-untyp*
begin
thm *ts-def*
end

Nested pointers among the phases.

context *call-inc-nested-impl*
begin
thm *call-inc-nested-body-def*
end

context *l1-definition-call-inc-nested*
begin
thm *l1-def*
end

context *l2-definition-call-inc-nested*
begin
thm *l2-def*
end

context *hl-definition-call-inc-nested*
begin
thm *hl-def*
end

context *wa-definition-call-inc-nested*
begin
thm *wa-def*
end

context *ts-definition-call-inc-nested*

```

begin
thm ts-def
end

```

Global variable

```

context ts-corres-call-inc-global
begin
thm ts-def
end

```

Local array variable.

```

context ts-definition-call-inc-array
begin
thm ts-def
end

```

Local structure variable.

```

context ts-definition-call-inc-first
begin
thm ts-def
end

```

Local array in structure variable.

```

context ts-definition-call-inc-buffer
begin
thm ts-def
end

```

Mutual recursion and pointer to local variables.

```

context l2-definition-odd-even
begin
thm unchanged-typing
end
context ts-corres-odd-even
begin
thm ts-def
end

```

Proof rules for *runs-to-vcg*

```

context pointers-to-locals-all-corres begin

```

```

thm stack-ptr-simps

```

```

lemma (call-inc-local-initialized') • s  $\{ \lambda r t. t = s \wedge r = \text{Result } 43 \}$ 
  apply (unfold call-inc-local-initialized'-def inc'-def)
  apply runs-to-vcg

```

```

apply (auto simp: stack-ptr-simps comp-def)
done

lemma (call-inc-local-uninitialized') • s { λ r t. t = s ∧ r = Result 42 }
apply (unfold call-inc-local-uninitialized'-def inc'-def)
apply runs-to-vcg
apply (auto simp: stack-ptr-simps comp-def)
done

lemma (call-inc-parameter' n) • s { λ r t. t = s ∧ r = Result (n + 1) }
apply (unfold call-inc-parameter'-def inc'-def)
apply runs-to-vcg
apply (auto simp: stack-ptr-simps comp-def)
done

end

```

26.17.2 Open Ends / TODOs

There are not yet any user level proof rules for *heap-w32.guard-with-fresh-stack-*ptr** and *heap-w32.assume-with-fresh-stack-*ptr** to include with *runs-to-vcg*. Note that there are a lot of theorems on the primitives *stack-allocs* and *stack-releases*.

```

thm stack-allocs-cases
thm stack-allocs-ptr-valid-cases
thm stack-releases-ptr-valid-cases
thm stack-releases-ptr-valid-cases1

```

Value parameters aka. In-Out parameters

Now that we have proper pointers to local variables there is room for further abstractions, motivated by the prominent use cases described in the beginning. Those use cases suggest that in a lot of cases one can completely get rid of stack allocation / release by introducing value parameters: Instead of just passing a pointer value into the function, one passes in the actual (dereferenced) value and tuples the return value of the function with the final (dereferenced) value of that parameter.

It turns out that the core transformation here is not so much about pointers to local variables but more on transforming functions from passing in a pointer parameter to another function value parameters. This part of the transformation is independent of the question heap pointer vs. stack pointer. After this transformation is done for the body of a function which allocates a stack pointer, the actual address of the stack pointer becomes meaningless as it is only used in "dereferenced form". So we can remove the stack allocation / stack release and replace it by an ordinary local variable.

See `In_Out_Parameters_Ex.thy` for more information.

```

end

```

26.18 In-Out Parameters, Abstracting Pointers to Values

```
theory In-Out-Parameters-Ex imports AutoCorres  
begin
```

```
consts abs-step:: word32  $\Rightarrow$  word32  $\Rightarrow$  bool
```

26.18.1 Overview

We introduce a new phase in AutoCorres to replace pointer parameters by *in/out parameters*: Instead of passing a pointer into a function we pass the initial value of the pointer (by dereferencing the pointer) into the function and return the value at the end of the function as additional output. E.g. a function `void inc(unsigned *n)` becomes `unsigned inc(unsigned n)`. The initial motivation for this phase was to get rid of explicit pointers to local variables and replace them by ordinary values instead. This conversion has two main aspects:

- Function signatures may change: pointer parameters become value parameters and the function may return the value of the pointer at the end as an additional return value (tupled with the ordinary return value). Functions with side effects on the heap may become pure functions on values by this transformation.
- As a result of the first transformation, pointers to local variables may be eliminated and be replaced by bound variables. This means that *stack-heap-state.with-fresh-stack-ptr* disappears.

The new phase is called *IO* and is placed between L2 and HL. This means local variables are already represented as bound variables in L2 and we still operate on a monolithic heap.

26.18.2 Building Blocks

What ingredients do we need for the abstraction relation in the refinement proof?

- Heaps: As we eliminate pointers, the original function manipulates the heap more often than the resulting function. So we need a notion to relate the heap states and keep track of the relevant pointers. As the stack of local variable pointers is also modelled in the heap the stack free locations are also of interest. The heaps of the original and the resulting program may differ in these locations. The heap value of a stack free location should be irrelevant for the program.

- Function signatures: The function signature changes, pointer parameters become ordinary values, the return value is tupled with output parameters.

In the following we refer to the original state (containing the heap) as s and the resulting / abstract state as t . As I first idea we want to relate states s and t such that the heaps are the same except for some pointers we want to eliminate and the stack free locations. It would be natural to describe this as a relation. However, it turns out that dealing with relations within the refinement proof are hard to make admissible in order to support recursive functions. Having a abstraction / lifting function (instead of a relation) from s to t makes admissibility straightforward.

As a prerequisite we encourage the reader to consider the documentation about pointers to local variables: `pointers_to_locals.thy`

context *heap-state*
begin

frame

We want to express that certain portions of the heap (typing and values) are 'irrelevant', in particular regarding (free) stack space. The notion of 'irrelevant' is a bit vague, it means that the behaviour of the resulting program does not depend on those locations and also that it does not modify those locations. Moreover, we prefer an abstraction function rather than an relation between states to avoid admissibility issues for refinement. The central property is $t = \text{frame } A \ t_0 \ s$, for A being a set of addresses and t, t_0 and s being states. Think of A as the set of *allocated* addresses containing those pointers we want to *abstract* aka. eliminate.

thm *frame-def*

The standard use case we have in mind is $A \cap \text{stack-free } (h\text{td } s) = \{\}$, hence $A - \text{stack-free } (h\text{td } s) = A$, but nevertheless the intuition is:

- stack free typing from s is preserved, the framed sate has at least as many stack free addresses as the original one. So we can simulate any stack allocation.
- heap values for stack free and A are taken from reference state t_0 , this captures that we do not depend on the original values in s for those addresses.
- typing for allocations in A is taken from reference state t_0 , this captures that we do not depend on the original typing in s for those addresses.

By taking the same reference state t_0 to frame two states s and s' , we can express that the 'irrelevant' parts of the heap did not change in the respective framed states $\text{frame } A \ t_0 \ s$ and $\text{frame } A \ t_0 \ s'$.

rel-alloc, rel-stack, rel-sum-stack

The core invariant between the states that is maintained by the refinement is *rel-alloc* $\mathcal{S} \ M \ A \ t_0 \ s \ t$:

- \mathcal{S} is a static set of addresses containing the stack. Stack allocation and stack release are contained within \mathcal{S} .
- M is the set of addresses that might be modified by the original program.
- A is the set of addresses that are eliminated (abstracted) in the resulting program. Another good intuition about the set A is that the original function might read from and depend on those addresses but the resulting function does not. Moreover, the resulting function does not change any of heap locations in A . The resulting function is agnostic to those locations. It neither reads nor modifies them. For any heap valuation of A the abstract function simulates the original one.
- t_0 is the reference state explained above.
- s is the state of the original program
- t is the state of the resulting program.

thm *rel-alloc-def* [of $\mathcal{S} \ M \ A \ t_0$]

The properties of *rel-alloc* are:

- $t = \text{frame } A \ t_0 \ s$, the resulting heap in t is the same as the original in s , except for the addresses in A and the stack free space.
- $\text{stack-free}(\text{htd } s) \subseteq \mathcal{S}$: The stack free space is contained in \mathcal{S} .
- $\text{stack-free}(\text{htd } s) \cap A = \{\}$: We only want to eliminate properly allocated pointers, which are not contained in the stack free space.
- $\text{stack-free}(\text{htd } s) \cap M = \{\}$: We only mention properly allocated pointers to be modified. Modifications within the stack free space are irrelevant, as they are abstracted by the framing anyways.

For an original function f and the abstract aka. lifted function g the refinement property has the following shape: $rel\text{-}alloc \ \mathcal{S} \ M \ A \ t_0 \ s \ t \implies refines \ f \ g \ s \ t \ (rel\text{-}stack \ \mathcal{S} \ M1 \ A \ s \ t_0 \ (rel\text{-}xval\text{-}stack \ L \ R))$

Note that the output relation of $refines$ relates both the resulting states and return values to each other. The return values of the error monad are related by $rel\text{-}sum\text{-}stack$, taking a relation L for error values (Inl) and the R for normal termination (Inr).

thm $rel\text{-}sum\text{-}stack\text{-}def$

thm $rel\text{-}stack\text{-}def$ [of $\mathcal{S} \ M1 \ A \ s \ t_0 \ rel\text{-}sum\text{-}stack \ L \ R$]

Consider the output states s' and t' and the output values v and w for the original and the abstracted function. Then we have:

- $rel\text{-}xval\text{-}stack \ L \ R \ (hmem \ s') \ v \ w$: The result values are related, (see below).
- $rel\text{-}alloc \ \mathcal{S} \ M1 \ A \ t_0 \ s' \ t'$: In particular $t' = frame \ A \ t_0 \ s'$. Note that we use the same A and the same reference state t_0 as for the initial states. So this means that the abstract program g does not change the heap values and typing in A and does not depend on the values.
- $equal\text{-}upto \ (M1 \cup \text{stack-free} \ (htd \ s')) \ (hmem \ s') \ (hmem \ s)$: The original program only modifies pointers contained in $M1$ or in the stack free portions of the heap (e.g. by nested function calls).
- $equal\text{-}upto \ M1 \ (htd \ s') \ (htd \ s)$: Changes to the heap typing of the original program are confined within set $M1$.
- $equal\text{-}on \ \mathcal{S} \ (htd \ s') \ (htd \ s)$: The stack typing of the original program remains unchanged. Note that it might change temporarily by nested function calls but the stacking discipline restores the state. By construction we have $M1 \subseteq M$.

Stacking pointers: $rel\text{-}singleton\text{-}stack$, $rel\text{-}push$ The relation on the output value captures the idea that portions of the heap are stacked / tupled into result values. On the boundaries of a function the relation on error values L is always the trivial $\lambda\text{-}$. ($=$). Errors are terminal as there is no exception handling in C. Within the boundaries of a function there can be more complex relations reflecting the nesting of `break` / `continue` / `return` and `goto`. For ordinary values the values of pointers are tupled by $rel\text{-}singleton\text{-}stack$, in case the original function returned no result (void), or nested by $rel\text{-}push$ in case the original function returned some result.

thm $rel\text{-}singleton\text{-}stack\text{-}def$

thm $rel\text{-}push\text{-}def$

A typical relation is $R = \text{rel-singleton-stack } p$ for some pointer p . This means that the abstract value can be obtained by looking into the heap at pointer p : $\text{rel-singleton-stack } p \ h \ () \ v$ means that $v = h\text{-val } h \ p$. The heap of the original program is propagated down the into the relations. Similar $\text{rel-push } p \ (\lambda\cdot. (=)) \ h \ x \ (v, x)$ relates the result value x to the tuple (v, x) where $v = h\text{-val } h \ p$. Note that rel-push can be nested represent multiple output parameters. Depending on the role of the pointers there are additional assumptions about the pointers, in particular things like $\text{ptr-span } p \subseteq A$ or $\text{ptr-span } p \subseteq M$ and disjointness properties like $\text{distinct-sets } [\text{ptr-span } p, \text{ptr-span } q]$.

IOcorres

There is an additional predicate *IOcorres* which represents the result of the refinement proof in the canonical autocorres ideom that is used within the other phases. This form is what is expected when constructing the final theorem combining all autocorres layers $\llbracket L1corres \ ?check\text{-termination} \ ?Gamma \ ?c\text{-L1.0} \ ?c; L2corres \ ?st\text{-L2.0} \ ?rx\text{-L2.0} \ ?ex\text{-L2.0} \ ?P\text{-L2.0} \ ?c\text{-L2.0} \ ?c\text{-L1.0}; IOcorres \ ?P\text{-IO} \ ?Q\text{-IO} \ ?st\text{-IO} \ ?rx\text{-IO} \ ?ex\text{-IO} \ ?c\text{-IO} \ ?c\text{-L2.0}; L2Tcorres \ ?st\text{-HL} \ ?c\text{-HL} \ ?c\text{-IO}; corresTA \ ?P\text{-WA} \ ?rx\text{-WA} \ ?ex\text{-WA} \ ?c\text{-WA} \ ?c\text{-HL}; \bigwedge s. \text{refines } ?c\text{-WA} \ ?A \ s \ s \ (\text{rel-prod } \text{rel-liftE} \ (=)) \rrbracket \implies ac\text{-corres}' \ (\lambda\cdot. \text{Exception default}) \ (?st\text{-HL} \circ \ ?st\text{-IO} \circ \ ?st\text{-L2.0}) \ ?check\text{-termination} \ \{ \text{AssumeError}, \ \text{StackOverflow} \} \ ?Gamma \ (\lambda s. \ (?rx\text{-WA} \circ \ (\lambda v. \ ?rx\text{-IO} \ v \ (?st\text{-L2.0} \ s)) \circ \ ?rx\text{-L2.0}) \ s) \ (\lambda s. \ (?ex\text{-WA} \circ \ (\lambda e. \ ?ex\text{-IO} \ e \ (?st\text{-L2.0} \ s)) \circ \ ?ex\text{-L2.0}) \ s) \ (?P\text{-L2.0} \ \text{and} \ ?P\text{-IO} \circ \ ?st\text{-L2.0} \ \text{and} \ ?P\text{-WA} \circ \ ?st\text{-HL} \circ \ ?st\text{-IO} \circ \ ?st\text{-L2.0}) \ ?A \ ?c$

$\llbracket L1corres \ ?check\text{-termination} \ ?Gamma \ ?c\text{-L1.0} \ ?c; L2corres \ ?st\text{-L2.0} \ ?rx\text{-L2.0} \ ?ex\text{-L2.0} \ ?P\text{-L2.0} \ ?c\text{-L2.0} \ ?c\text{-L1.0}; IOcorres \ ?P\text{-IO} \ ?Q\text{-IO} \ ?st\text{-IO} \ ?rx\text{-IO} \ ?ex\text{-IO} \ ?c\text{-IO} \ ?c\text{-L2.0}; L2Tcorres \ ?st\text{-HL} \ ?c\text{-HL} \ ?c\text{-IO}; corresTA \ ?P\text{-WA} \ ?rx\text{-WA} \ ?ex\text{-WA} \ ?c\text{-WA} \ ?c\text{-HL}; \bigwedge s. \text{refines } ?c\text{-WA} \ ?A \ s \ s \ (\text{rel-prod} \ (=) \ (=)) \rrbracket \implies ac\text{-corres}' \ \text{Exn} \ (?st\text{-HL} \circ \ ?st\text{-IO} \circ \ ?st\text{-L2.0}) \ ?check\text{-termination} \ \{ \text{AssumeError}, \ \text{StackOverflow} \} \ ?Gamma \ (\lambda s. \ (?rx\text{-WA} \circ \ (\lambda v. \ ?rx\text{-IO} \ v \ (?st\text{-L2.0} \ s)) \circ \ ?rx\text{-L2.0}) \ s) \ (\lambda s. \ (?ex\text{-WA} \circ \ (\lambda e. \ ?ex\text{-IO} \ e \ (?st\text{-L2.0} \ s)) \circ \ ?ex\text{-L2.0}) \ s) \ (?P\text{-L2.0} \ \text{and} \ ?P\text{-IO} \circ \ ?st\text{-L2.0} \ \text{and} \ ?P\text{-WA} \circ \ ?st\text{-HL} \circ \ ?st\text{-IO} \circ \ ?st\text{-L2.0}) \ ?A \ ?c$. The *refines* version and the *IOcorres* version can be converted into each other by $\llbracket L1corres \ ?check\text{-termination} \ ?Gamma \ ?c\text{-L1.0} \ ?c; L2corres \ ?st\text{-L2.0} \ ?rx\text{-L2.0} \ ?ex\text{-L2.0} \ ?P\text{-L2.0} \ ?c\text{-L2.0} \ ?c\text{-L1.0}; IOcorres \ ?P\text{-IO} \ ?Q\text{-IO} \ ?st\text{-IO} \ ?rx\text{-IO} \ ?ex\text{-IO} \ ?c\text{-IO} \ ?c\text{-L2.0}; L2Tcorres \ ?st\text{-HL} \ ?c\text{-HL} \ ?c\text{-IO}; corresTA \ ?P\text{-WA} \ ?rx\text{-WA} \ ?ex\text{-WA} \ ?c\text{-WA} \ ?c\text{-HL}; \bigwedge s. \text{refines } ?c\text{-WA} \ ?A \ s \ s \ (\text{rel-prod } \text{rel-liftE} \ (=)) \rrbracket \implies ac\text{-corres}' \ (\lambda\cdot. \text{Exception default}) \ (?st\text{-HL} \circ \ ?st\text{-IO} \circ \ ?st\text{-L2.0}) \ ?check\text{-termination} \ \{ \text{AssumeError}, \ \text{StackOverflow} \} \ ?Gamma \ (\lambda s. \ (?rx\text{-WA} \circ \ (\lambda v. \ ?rx\text{-IO} \ v \ (?st\text{-L2.0} \ s)) \circ \ ?rx\text{-L2.0}) \ s) \ (\lambda s. \ (?ex\text{-WA} \circ \ (\lambda e. \ ?ex\text{-IO} \ e \ (?st\text{-L2.0} \ s)) \circ \ ?ex\text{-L2.0}) \ s) \ (?P\text{-L2.0} \ \text{and} \ ?P\text{-IO} \circ \ ?st\text{-L2.0} \ \text{and} \ ?P\text{-WA} \circ \ ?st\text{-HL} \circ \ ?st\text{-IO} \circ \ ?st\text{-L2.0}) \ ?A \ ?c$

$$\llbracket L1corres \ ?check-termination \ ?Gamma \ ?c-L1.0 \ ?c; L2corres \ ?st-L2.0 \ ?rx-L2.0 \ ?ex-L2.0 \ ?P-L2.0 \ ?c-L2.0 \ ?c-L1.0; IOcorres \ ?P-IO \ ?Q-IO \ ?st-IO \ ?rx-IO \ ?ex-IO \ ?c-IO \ ?c-L2.0; L2Tcorres \ ?st-HL \ ?c-HL \ ?c-IO; corresTA \ ?P-WA \ ?rx-WA \ ?ex-WA \ ?c-WA \ ?c-HL; \wedge s. \ refines \ ?c-WA \ ?A \ s \ s \ (rel-prod \ (=) \ (=)) \rrbracket \implies ac-corres' \ Exn \ (?st-HL \circ \ ?st-IO \circ \ ?st-L2.0) \ ?check-termination \ \{AssumeError, StackOverflow\} \ ?Gamma \ (\lambda s. \ (?rx-WA \circ \ (\lambda v. \ ?rx-IO \ v \ (?st-L2.0 \ s)) \circ \ ?rx-L2.0) \ s) \ (\lambda s. \ (?ex-WA \circ \ (\lambda e. \ ?ex-IO \ e \ (?st-L2.0 \ s)) \circ \ ?ex-L2.0) \ s) \ (?P-L2.0 \ and \ ?P-IO \circ \ ?st-L2.0 \ and \ ?P-WA \circ \ ?st-HL \circ \ ?st-IO \circ \ ?st-L2.0) \ ?A \ ?c.$$

thm *IOcorres-def*
thm *IOcorres-refines-conv*
thm *ac-corres-chain-sims*
end

install-C-file *in-out-parameters.c*

26.18.3 Options

To control the in/out- parameter abstraction there are two options of `autocorres`:

- *skip-io-abs* a boolean which is *true* by default. This skips the complete phase for all functions.
- *in-out-parameters* which takes a list of in-out specifications for functions. As soon as there is at least one specification present the phase is enabled (taking precedence over *skip-io-abs*).

So to properly enable the IO phase you have two main choices to specify what should be done either in **init-autocorres** or subsequent calls to **autocorres**. You can only disable option *skip-io-abs* in **init-autocorres** and then provide the individual options for *in-out-parameters* in subsequent **autocorres** invocations, or you can already define all *in-out-parameters* already in **init-autocorres**.

More details on the options are provided in the following examples.

26.18.4 Examples

```

init-autocorres [ in-out-parameters =
  compare(cmp:out) and
  inc(y:in-out) and
  inc-int(y:in-out) and
  dec(y:in-out) and
  swap(x:in-out, y:in-out) and
  swap-pair(x:in-out, y:in-out) and
  call-inc-ptr(p: in-out) and

```

```

inc-twice(y:in-out) and
inc2(y:in-out, z:in-out) and
heap-inc2-part(z:in-out) and
heap-inc2-part-swap(y:in-out) and
safe-add(result: in-out) and
inc-pair(p:in-out) and
get-arr-idx(arr:in) and
get-pair-arr-idx-second(arr:in) and
inc-pair(p:in-out) and
xyz(x:in-out) and
resab(res:in-out) and
abcmp(cmpRst:in-out) and
out(out:out) and
out2(out:in-out) and
out-seq(out:out) and
inc-loop (x:in, y:in-out) and
call-inc-int-other(m: in-out) and
inc-int-no-exit(y:in-out) and
call-inc-int-other-mixed(m: in-out) and
inc-int2(y:in-out, z:in-out) and
call-inc-int2(n:in-out) and
call-inc-int-pair(p:in-out) and
call-inc-int-array(p:in-out) and
call-compare-ptr(m:out) and
mixed-global-local-inc(q:in-out) and
g1(out:in-out) and
set-void(p:data) and
set-void2(p:data) and
set-byte(p:data) and
read-char(p:in-out, len: in-out) and
goto-read-char1(p:in-out) and
goto-read-char5(elem: in),
in-out-globals =
    keep-inc-global-array
    inc-global-array
    keep-inc-global-array2
    inc-global-array2
    call-keep-inc-global-array
    call-inc-global-array
    mixed-global-local-inc shuffle global-array-update
    read-char call-read-char call-read-char-loop
goto-read-char1 goto-read-char2 goto-read-char3 goto-read-char4 goto-read-char5,
ignore-addressable-fields-error,
addressable-fields =
    pair.first
    pair.second
    array.elements
    int-pair.int-first
    int-pair.int-second

```

]in-out-parameters.c

In option *in-out-parameters* you provide a parameter specification for the functions you want to abstract. The parameter specs following the parameter name are

- *in*: pointer is only used as input to the function. The referenced value does not change during the function call.
- *out*: the pointer is only used to output a result from the function. The referenced value at the beginning is irrelevant, the abstracted function should return the referenced value at the end of the original function.
- *in-out* the pointer is used to input a value and to return a result.

When no specification is given for a function, the list is considered empty. This means that the signature of the function shall not be changed by the abstraction. Pointer parameters in the original function stay pointer parameters in the resulting function. Note that this has a quite different effect from just skipping the phase, as autocorres is still attempting to do a proper refinement proof. More details are provided by the examples.

```
locale keep-globals = in-out-parameters-global-addresses +
  assumes keep-global-array: {ptr-val global-array ..+128}  $\subseteq$   $\mathcal{G}$ 
begin
lemma global-array-contained1:
  assumes i-bound:  $i < 12$ 
  shows ptr-span (global-array +p (int i))  $\subseteq$   $\mathcal{G}$ 
proof -
  have ptr-span (global-array +p (int i))  $\subseteq$  {ptr-val global-array ..+128}
  using i-bound
  apply (clarsimp simp add: ptr-add-def intvl-def)
  subgoal for k
  apply (rule exI [where x= i * 4 + k])
  by force
  done
with keep-global-array show ?thesis by blast
qed

lemma gobal-array-contained2[polish]:
  assumes i-bound: numeral i < (12::nat)
  shows ptr-span (global-array +p (numeral i))  $\subseteq$   $\mathcal{G}$ 
  using global-array-contained1
  by (metis i-bound of-nat-numeral)

end
```

```

autocorres [
    ts-force nondet = shuffle] in-out-parameters.c

```

```

context in-out-parameters-all-corres
begin

```

```

thm io-corres
thm ts-def
term ptr-valid

```

```

inc'

```

The function `inc` increments the value which is passed as a pointer `y`.

```

void inc (unsigned* y) {
    *y = *y + 1;
}

```

When we abstract this function to have an in-out parameter instead, we obtain a function like:

```

void inc (unsigned y) {
    y = y + 1;
    return y;
}

```

Note that this is now a pure function, that does no longer depend on the heap. This is also the final output of `autocorres` in `inc' ?y ≡ ?y + 1`:

```

thm inc'-def

```

Let us zoom into the refinement step from L2 to IO.

```

thm l2-inc'-def
thm io-inc'-def

```

Note that the type of parameter `y` changes from a pointer type to a value type. Here is the generated refinement theorem.

```

thm io-inc'-corres
lemma c-guard y ⇒
    distinct-sets [ptr-span y] ⇒
    ptr-span y ⊆ A ⇒
    ptr-span y ⊆ M ⇒
    globals.rel-alloc S M A t0 s t ⇒
    refines
    (l2-inc' y)
    (io-inc' (h-val (hrs-mem (t-hrs-' s)) y)) s t
    (globals.rel-stack S (ptr-span y) A s t0 (rel-xval-stack (rel-exit (λ- - . False)))
    (rel-singleton-stack y)))

```

by (rule *io-inc'-corres*)

Parameter y was specified as *in-out*. So function *io-inc'* becomes the actual value of the pointer passed in: $h\text{-val } (hrs\text{-mem } (t\text{-hrs-}' s)) y$. To discharge the *c-guard* y guards within the body of *l2-inc'* we need the assumption that the guard holds at the beginning. As we attempt to abstract pointer y we have its pointer span included in set A . Moreover, the content of pointer y is modified by *l2-inc'* hence it appears in M as well as explicit in the final relation *globals.rel-stack*. Pointer parameters are assumed to be distinct. As we have only one parameter the assumption trivially *distinct-sets* [*ptr-span* y] holds. Note that set A is only constrained by *ptr-span* $y \subseteq A$. So we could instantiate it with the universal set *UNIV*. Recalling the meaning of A this means that the resulting function *io-inc'* is a pure function, in the sense that it does not depend on the heap anymore. The relation for the termination with **exit** is *rel-exit* ($\lambda\text{-} - \text{-} . False$), which expresses that this path is unreachable for the current function. The function will never terminate with **exit**.

call-inc-parameter'

Next we look at a function that calls our previously defined integer function on a pointer to its parameter.

```
unsigned call_inc_parameter(unsigned n) {
    inc(&n);
    return(n);
}
```

After autocorres the function is just a mere wrapper to the increment function

thm *call-inc-parameter'-def*

Here are the relevant definitions and the IO refinement theorem.

thm *l2-call-inc-parameter'-def*

thm *io-call-inc-parameter'-def*

thm *io-call-inc-parameter'-corres* [*no-vars*]

thm *io-call-inc-parameter'-corres*

lemma *distinct-sets* ($[\]::\text{addr set list}$) \implies

globals.rel-alloc $\mathcal{S} M A t_0 s t \implies$

refines

(*l2-call-inc-parameter' n*)

(*io-call-inc-parameter' n*) $s t$

(*globals.rel-stack* $\mathcal{S} \{ \} A s t_0$ (*rel-xval-stack* (*rel-exit* ($\lambda\text{-} - \text{-} . False$)) ($\lambda\text{-} . (=)$)))

by (rule *io-call-inc-parameter'-corres*)

Note that here we do not have any assumption on the set A and the modified variables in the final state are empty. This is because we eliminate

a pointer to a temporary stack variable that is out of scope after the function and in the beginning of the function it is part of the *stack-free* space.

const*keep-inc'*

Now let us consider a copy of the increment function were we do not convert to in out parameters but leave it as it is. Unsurprisingly, the resulting function still operates on pointers.

thm *keep-inc'-def*

Let us look into the IO corres phase.

thm *l2-keep-inc'-def*

thm *io-keep-inc'-def*

thm *io-keep-inc'-corres*

lemma *distinct-sets* [*ptr-span y*] \implies

ptr-span y \cap *A* = {} \implies

ptr-span y \cap *stack-free* (*hrs-htd* (*t-hrs-' s*)) = {} \implies — FIXME: redundant assumption?, follows from the next ones

globals.rel-alloc *S M A t₀ s t* \implies

ptr-span y \subseteq *M* \implies

refines

(*l2-keep-inc' y*)

(*io-keep-inc' y*) *s t*

(*globals.rel-stack* *S* (*ptr-span y*) *A s t₀* (*rel-xval-stack* (*rel-exit* (λ - - -. *False*)))

(λ -. (=)))

by (*rule io-keep-inc'-corres*)

As pointer *y* is an allocated pointer that we do not abstract it should not belong to *A* and must not collide with *stack-free* space. As we might modify the pointer it is contained in *M* as well as the final modifies set.

Note that the bodies of *io-keep-inc'* and *l2-keep-inc'* are equivalent but the refinement statement is not a trivial statement and in particular is not the same as when skipping the IO corres phase. For example from the refinement statement we can derive that any heap location different from *ptr-span y* is unchanged.

Next let's look what happens when we call this function on a local parameter.

keep-call-inc-parameter'

thm *keep-call-inc-parameter'-def*

thm *io-keep-call-inc-parameter'-def*

thm *l2-keep-call-inc-parameter'-def*

thm *io-keep-call-inc-parameter'-corres* [*no-vars*]

lemma *distinct-sets* ([*addr set list*]) \implies

```

globals.rel-alloc  $\mathcal{S} M A t_0 s t \implies$ 
refines
  (l2-keep-call-inc-parameter'  $n$ )
  (io-keep-call-inc-parameter'  $n$ )  $s t$ 
  (globals.rel-stack  $\mathcal{S} \{\}$   $A s t_0$  (rel-xval-stack (rel-exit ( $\lambda - - . False$ )) ( $\lambda - . (=)$ )))
by (rule io-keep-call-inc-parameter'-corres)

```

The local variable pointer is still present in the final function. Still the refinement statement indicates that we have a ‘pure’ functions, as we do not have constraints on A and also do not modify anything. The notion of pure we use here, is that the function does not depend on proper heap pointers. The *stack-free* space that we temporarily use is excluded by our definitions of *globals.frame*, *globals.rel-alloc* and *globals.rel-stack*.

safe-add'

This function is an example of an arithmetic function that checks for overflows. The status of the check is the return value and the result of the operation is passed via pointer. We make an in-out parameter for the result. So the final function returns a tuple of the result value and the status (*rel-push* in the refinement statement).

```

thm safe-add'-def
thm l2-safe-add'-def
thm io-safe-add'-def
thm io-safe-add'-corres

```

Recursion

```

thm l2-fac'.simps
thm io-fac'.simps
thm io-fac'-corres

```

```

thm l2-even'.simps l2-odd'.simps
thm io-even'.simps io-odd'.simps
thm io-even'-corres io-odd'-corres

```

swap'

We abstract a function that swaps the values referenced by two input pointers by a function that takes the values and returns the swapped values (as a pair). Note that in the refinement statement we assume disjointness of the input pointers. The swap function would still be correct if the pointers are equal. To generalise the refinement proofs to ‘disjoint or equal’ inputs is left as future work.

```

thm swap'-def
thm l2-swap'-def
thm io-swap'-def

```

thm *io-swap'-corres*

Out parameters

The function *compare* has an mere 'out' parameter. Until phase L2 the function has three parameters, after phase IO the function only has two parameters.

thm *l2-compare'-def*

thm *io-compare'-def*

thm *compare'-def*

thm *l2-call-compare'-def*

thm *io-call-compare'-def*

thm *call-compare'-def*

Pointers to compound types (structs / arrays)

Here are some examples of function that work on (parts) of a compound type.

thm *get-arr-idx'-def*

thm *l2-get-arr-idx'-def*

thm *io-get-arr-idx'-def*

thm *inc-pair'-def*

thm *l2-inc-pair'-def*

thm *io-inc-pair'-def*

Misc

thm *wa-def*

thm *ts-def*

thm *io-corres*

26.18.5 Handling exit in function calls

When we call a function that was refined by in-out-parameters there are various cases we have to consider, e.g. do we pass in a heap pointer or a local pointer, do we eliminate the local pointer or keep it? In general when we pass in a pointer that is not supposed to be eliminated as an in-out parameter we first pass in the dereferenced value and after the function returns we take the value from the embellished result and assign it to the pointer. So the pointer assignment that would take place in the middle of the called function in the original program, is moved outside of the call in the refined program. After this the heaps of the original program and the refined program are in sync again at the pointer. To establish the heap refinement it turns out that we also have to bring the heaps in sync again even if the function terminates with `exit`. We catch the `exit`, update the heap and

then re-throw the *exit*. So the called function also returns embellished exit values which hold the value of the relevant pointers. This extra work is a little annoying as the `exit` terminates the complete program and usually we do not catch an `exit`. So this step might introduce *L2-catch* to wrap up the procedure call. Note that this is currently the only remaining use case for *L2-catch*, the other occurrences of *L2-catch* were already transformed to *L2-try* as post processing in phase L2 when we move from ‘flat’ exception handling to ‘nested’ exception handling.

term *inc-int'*
thm *inc-int'-def*
term *call-inc-int-ptr'*
thm *call-inc-int-ptr'-def*

26.18.6 Global Heap Pointers

A core part of the IO phase is to keep track of pointers and in particular to do some bookkeeping of the parts of the heap that might get modified. This analysis and bookkeeping is focused around the pointers that are visible in the signature of a function. This fails short as soon as pointers to global heap are used within the body of a function, which are not mentioned in the function signature. To handle some common use cases with global heap we have a mechanism to over-approximate the impact of a function on the heap by referring to a static set \mathcal{G} of addresses that should not be abstracted by the IO phase. We can annotate a function to make use of this mechanism by mentioning them in **init-autocorres** with the option `in_out_globals`. For the refinement statement we assume that

- $\mathcal{G} \cap A = \{\}$: pointers in \mathcal{G} are not supposed to be abstracted, and
- $\mathcal{G} \subseteq M$: the function might modify any global variable.

Within the body of such a function (marked with option `in_out_globals`), whenever a pointer is used which does not occur in the signature as pointer parameter and is specified otherwise we assume that this pointer a global pointer and assert this formally by adding a guard *ptr-span* $x \subseteq \mathcal{G}$. If a function has multiple pointer parameters we also assert disjointness of pointers.

thm *io-keep-inc-global-array'-corres*
thm *io-keep-inc-global-array'-def*
thm *l2-keep-inc-global-array'-def*

thm *io-inc-global-array'-corres*
thm *io-inc-global-array'-def*
thm *l2-inc-global-array'-def*

thm *io-inc-global-array2'-corres*
thm *io-inc-global-array2'-def*
thm *l2-inc-global-array2'-def*
thm *ac-corres*

Note that a function can also have both in-out parameters and `in_out_globals`.

thm *io-read-char'-corres*
thm *io-read-char'-def*
thm *l2-read-char'-def*

thm *io-call-read-char'-corres*
thm *io-call-read-char'-def*
thm *l2-call-read-char'-def*

thm *io-call-read-char-loop'-corres*
thm *io-call-read-char-loop'-def*
thm *l2-call-read-char-loop'-def*

26.18.7 Disclaimer / Caution

As we have seen in the examples the refinement theorems can contain assumptions on the input pointers, in particular disjointness assumptions between different pointer parameters and also disjointness of pointer parameters to complete memory areas like *stack-free*. When a function is called these assumptions have to be discharged. When these pointer parameters are eliminated (replaced by in-out value parameters) the assumptions disappear. So when you work on the final output of `autocorres` you know nothing about pointers or disjointness.

In case a local pointer is eliminated these assumptions are be discharged locally as an intermediate step in the proof. So this use case is fine. However, if the function is called on a heap pointer these assumptions are propagated to the caller. So the assumptions move up the call stack and might end up as implicit hidden assumptions on your toplevel functions (API).

Rule of thumb: In case your toplevel API function has more then one pointer parameter don't specify any in-out parameters on that function to avoid implicit assumptions.

26.18.8 Implementation Aspects

The proof of the refinement theorem requires to keep track of various pointers and changes the signature of functions, including extending plain return values to tuples. We decided to make a rather explicit forward proof for this, to keep tight control of the various aspects. We make heavy use of ML antiquotations to match and build cterms. An obstacle here is that the state record *globals*, which holds the heap, is not yet defined but is only present as a locale. So our code is implemented within the locale

stack-heap-state. We have introduced the concept of *interpretation data* (c.f. `../lib/ml-helpers/interpretation_data.ML`) to get hold of the interpretation we are interested in. The data is initialised in `../AutoCorres.thy` as declaration within *stack-heap-state*. In particular this allows us to match against and build heap lookup and heap update expressions.

The interface is via `In_Out_Parameters.operations`. This takes the morphism of the interpretation as an argument to derive the instance we are interested in. Note that we have crafted the main functions to benefit from the eager evaluation of ML. In particular `In_Out_Parameters.operations` takes the morphism *phi* as its only parameter and thus the instances of the main functions are only evaluated once.

end

26.18.9 pointer-parameters as data

In a case where a pointer is not used to access the heap (e.g., *set-void*), the "modifies" part of the corres theorem should be empty. Otherwise functions like *set-byte*, that pass "arbitrarily" computed pointers fail to get IO-lifted. We enforce this by declaring pointer parameters as "data" in the *in-out-parameters* option of autocorres.

Future work: Automate this analysis to infer which pointer-parameters are treated as pure data.

context *io-corres-set-void* **begin**

lemma *distinct-sets* ($[\]::\text{addr set list}$)

— should be able to get rid of this assumption, too (for multiple parameters) \implies

globals.rel-alloc $\mathcal{S} M A t_0 s t \implies$

refines (*l2-set-void'* p) (*io-set-void'* p) $s t$

(*globals.rel-stack* $\mathcal{S} \{ \} A s t_0$

(*rel-xval-stack* (*rel-exit* ($\lambda - - . \text{False}$)) ($\lambda - . (=)$)))

by (*rule io-set-void'-corres*)

end

context *io-corres-set-void2* **begin**

thm *io-set-void2'-corres*

end

context *io-corres-set-byte* **begin**

thm *io-set-byte'-corres*

end

26.18.10 Future work / Open Ends

- Support function pointers
- support functions that might modify the heap typing (e.g. via *exec-concrete*) currently our *rel-alloc* / *rel-stack* setup enforces that heap typing does not change at all this is too restrictive. Before we had that heap typing

does not change in \mathcal{S} which was too weak to modify the A frame to handle invocations on heap pointers. I guess a good approximation would be to say that heap typing is unchanged on all relevant addresses: $\mathcal{S} \cup A \cup M$ Including M might be too restrictive, e.g. consider an alloc function that zeros some memory and adapts the heap-typing.

- Refined treatment of *in-out-globals*: allow the user to add a guard that is inserted in front of the abstract program. E.G. locale for procedure: $ptr\text{-}span\ (global\text{-}array + MAX\text{-}IDX) \subseteq \mathcal{G}$ + a Guard depending on a parameter idx , e.g. $idx < MAX\text{-}IDX$. Then we can automatically discharge local guards like $ptr\text{-}span\ (global\text{-}array + idx) \subseteq \mathcal{G}$. Not sure if this is useful though
- Cleanup unused *Generic-Data* that was replaced by interpretation data
- Remove struct-rewrite step from Heap Lifting. As we normalise pointers to root pointers in L2-Opt the pointers should already be in normal form and struct-rewrite does nothing.
- Add sanity checks to *in-out* specs, especially if parameter names actually occur in signature
- Remove redundant assumptions for keep parameters: $ptr\text{-}span\ y \cap stack\text{-}free\ (hrs\text{-}htd\ (t\text{-}hrs\text{-}'\ s)) = \{\} \implies globals.\text{rel}\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t \implies ptr\text{-}span\ y \subseteq M \implies \dots$
 - The first one can be derived from the following ones
- Peephole: remove *ptr-disjoint* of singletons
- Peephole simplification, especially Guard True. Not really necessary, will disappear in hl anyways.
- Eliminate pointers to global variables: When a pointer to a global variable is only used as a *in-out parameter* it would be nice to abstract it to a record field in *lifted-globals* (or to something similar, i.e. lookup / update function in the heap with disjointness and commutation of updates for other globals)
- Support more relaxed disjoint assumptions, especially *disjoint-or-eq*, e.g. swap also works if the input pointers are equal.
 - All ‘keep’ pointers are currently assumed disjoint and also are in the modifies set. We could be more relaxed here.
- HL phase has some explicit references to phase L2 instead of abstract *prev-phase* (probably related to function pointers)

- Would be nice if synthesize rules would also participate in thm, and add / del stuff like *named-rules*

end

26.19 Function Pointers

```
theory fnptr
imports AutoCorres
begin
```

```
install-C-file fnptr.c
```

```
autocorres [
  ts-force nondet = voidcaller,
  ts-force-known-functions = option ] fnptr.c
```

Dealing with function pointers in AutoCorres has the following main challenges:

- Parameter passing
- Correspondence proofs
- Mutual recursion

Parameter passing

In C a function pointer type does not have to specify the names of the function parameters. Hence when translating a call to a function pointer from C into SIMPL we need a uniform way to pass the function parameters. Recall that in SIMPL local variables are represented as part of the state, there is no lambda binding at that stage. For ordinary function calls we know the names of the parameters and can refer to them, for function pointers we do not know the names. Our approach is to generally switch to uniform names for the parameters, encoding the position and the type like: *in1-int*, *in2-unsigned*.

The mapping to the original names in the C file is implemented as syntactic sugar in the locale. As this mapping depends on the actual function we have to make sure to work in the correct context. The main interface to this mapping is defined in `../c-parser/HPInter.ML` e.g.:

- `HPInter.all_locvars`

- `HPInter.enter_scope`

Note that the current implementation is a little bit fragile as it also depends on the syntax translations for lookup and update in the underlying `statespace`.

Correspondence proofs

When we come along a function pointer call in a correspondence proof we need a correspondence proof for the function pointer. Where to we get it from? In full generality we do not know much about where the function pointer might be pointing to. One drastic way out could be to add a guard at the call point, which guarantees that there is such a proof. This makes the proof trivial, but the code gets polluted with correspondence guards which also show up at the user level. So whenever one wants to prove a property about the resulting function one has to deal with this guard.

To keep the user level proofs clean, we decided to take another approach, by restricting the programs we can handle to some common use cases:

- Constant function pointer as parameter to a function. Here constant means that the parameter is not modified within the function body.
- Function pointer via constant global variable. Think of the global variable as a kind of dispatcher or class table, e.g. Array of structs representing "object descriptions" with function pointers as fields.
- C-style object method calls: Instead of a pure function pointer, a pointer to a structure which contains function pointers is passed as a parameter.

In the first case we can add the correspondence assumption to the locale of the function. In the second case we can even resolve the correspondence proof as we can statically infer which functions might be called via the global variable.

Mutual recursion

The general approach to resolve a function pointer call is via a lookup in a program environment which maps the pointer to the definition of the function. In SIMPL the function is defined by a piece of syntax, and the program environment is an explicit context Γ which appears in the semantic rules for SIMPL. Moreover, for each function that might be called via a function pointer we introduce a pointer to that function. These pointers, and the assumption that all of them are distinct are put to the locale of global variables.

In each phase of Autocorres we introduce a similar program environment \mathcal{P} that maps the pointer to a monadic HOL function definition. As from phase L2 on the parameters of functions become lambda abstractions we introduce a distinct program environment for each function type. These program environments are introduced as locale parameters. Then we successively add the implementation equations of the form \mathcal{P} *some-function-pointer* = *some-function* as the functions are defined.

The correspondence theorems (or assumptions) for function pointer calls then relates the functions resolved via their respective environment, e.g. relate *the* (Γ p) with \mathcal{P} p .

We also support mutual recursion via global function pointers (case 2 above), e.g. a function might call itself indirectly via a global function pointer variable. The global program analysis takes this into account to determine the strongly connected components aka. (recursive) cliques.

This adds another twist to the approach with program environments just described. The assumption \mathcal{P} *some-function-pointer* = *some-recursive-function-via-P* can only be added after the function *some-recursive-function-via-P* is defined, but the definition already depends on the program environment. How do we cut the loop?

The core idea is to split the definition into two phases. First we define the **fixed-point** by explicitly extending the program environment at each call. Instead of calling \mathcal{P} p directly we call *map-of-default* \mathcal{P} [(*some-function-pointer*, *some-recursive-function-via-P*)] p where *some-recursive-function-via-P* participates in the fixed point construction. Once the function is defined, we create a new locale, extending the program environment with the assumption \mathcal{P} *some-function-pointer* = *some-recursive-function-via-P* and simplifying

$$\text{map-of-default } \mathcal{P} \text{ [(some-function-pointer, some-recursive-function-via-P)]}$$

$$p = \mathcal{P} p$$

in the function body. After this extra step we again uniformly represent recursive and non-recursive function pointer calls with \mathcal{P} p .

26.19.1 Global locales

The global locales fix the program environments (mapping function pointers to definitions) for a phase. These locales are created in the initialisation phase of **autocorres** before the individual functions are processed.

The fundamental locale is storing the addresses of global variables, including function pointers and some basic properties about them.

context *fnptr-global-addresses*
begin

For each function pointer a constant is declared e.g. *fnptr.odd-disp*, *fnptr.add*. Note that these constants have to be qualified by the program name. In case there would be a conflict with Isabelle internal names, e.g.

a function ending with `--` the name is suffixed with `Hoare.proc_deco`, cf. `NameGeneration.fun_ptr_name`.

ML \langle *Hoare.proc-deco* \rangle

Moreover the global variables and their defining equations, i.e. initialisation expressions are collected in $odd\text{-}even\text{-}dispatcher \equiv fupdate\ (Suc\ 0)\ (\lambda\cdot\ fnptr.\ even\text{-}disp)\ (fupdate\ 0\ (\lambda\cdot\ fnptr.\ odd\text{-}disp)\ (ARRAY\ \cdot\ NULL))$

$gi \equiv 0$

$dispatcher\text{-}u \equiv fupdate\ (Suc\ 0)\ (unop\text{-}u\text{-}C\text{-}update\ (\lambda\cdot\ fnptr.\ inc\text{-}u))\ (fupdate\ (Suc\ 0)\ (binop\text{-}u\text{-}C\text{-}update\ (\lambda\cdot\ fnptr.\ add\text{-}gu))\ (fupdate\ 0\ (unop\text{-}u\text{-}C\text{-}update\ (\lambda\cdot\ fnptr.\ inc\text{-}u))\ (fupdate\ 0\ (binop\text{-}u\text{-}C\text{-}update\ (\lambda\cdot\ fnptr.\ add\text{-}u))\ (ARRAY\ \cdot\ object\text{-}u\text{-}C\ NULL\ NULL))))$

$dispatcher \equiv fupdate\ 4\ (object\text{-}C.\ unop\text{-}C\text{-}update\ (\lambda\cdot\ NULL))\ (fupdate\ 4\ (binop\text{-}C\text{-}update\ (\lambda\cdot\ NULL))\ (fupdate\ 3\ (object\text{-}C.\ unop\text{-}C\text{-}update\ (\lambda\cdot\ fnptr.\ inc))\ (fupdate\ 3\ (binop\text{-}C\text{-}update\ (\lambda\cdot\ fnptr.\ minus))\ (fupdate\ 2\ (object\text{-}C.\ unop\text{-}C\text{-}update\ (\lambda\cdot\ fnptr.\ dec))\ (fupdate\ 2\ (binop\text{-}C\text{-}update\ (\lambda\cdot\ fnptr.\ mul))\ (fupdate\ (Suc\ 0)\ (object\text{-}C.\ unop\text{-}C\text{-}update\ (\lambda\cdot\ fnptr.\ dec))\ (fupdate\ (Suc\ 0)\ (binop\text{-}C\text{-}update\ (\lambda\cdot\ fnptr.\ minus))\ (fupdate\ 0\ (object\text{-}C.\ unop\text{-}C\text{-}update\ (\lambda\cdot\ fnptr.\ inc))\ (fupdate\ 0\ (binop\text{-}C\text{-}update\ (\lambda\cdot\ fnptr.\ add))\ (ARRAY\ \cdot\ object\text{-}C\ NULL\ NULL))))))))))$

$add\text{-}u\text{-}p \equiv fnptr.\ add\text{-}u$

$add\text{-}p \equiv fnptr.\ add$.

thm *global-const-defs*

thm *global-const-array-selectors* — more efficient access to array components

thm *global-const-non-array-selectors*

thm *global-const-selectors*

To express distinctness of function pointers we make use of the infrastructure in `$ISABELLE_HOME/src/HOL/Statespace/DistinctTreeProver.thy`, which is also the foundation for `statespace`. A tree data structure is used to efficiently derive distinctness and subset properties.

thm *fun-ptr-distinct*

thm *fun-ptr-subtree*

For every function pointer the locale assumes that they are not NULL, i.e. *c-fnptr-guard*. Based on the distinctness of the function pointers we derive some simplification rules for *map-of-default*.

thm *fun-ptr-simps*

end

Note that the program environment \mathcal{P} for l1 programs is only declared as constant (without a definition). This environment is 'populated' in derived locales as the definitions of function become available.

A program environment for each distinct function pointer type is introduced for each further phase. e.g. $\mathcal{P}\text{-}l2\text{-}unsigned\text{---}unsigned$, $\mathcal{P}\text{-}l2\text{-}int\text{---}int$

context *fnptr-global-addresses*

begin

For example if the function pointer would index into the dispatcher array *dispatcher-u* and select a *binop-u-C*, it would result into subgoals for each of the possible pointers, namely and *fnptr.add-u* *fnptr.add-gu*.

term *dispatcher-u*

In this bundle we define some introduction rules to support the correspondence proofs. In phase L1 and L2 the correspondence proof is quite explicitly done in ML and the rules for function pointer calls are explicitly instantiated in ML. Here in layer HL the proof is done by applying intro rules and synthesising the abstract version of the program. The following rules relate the abstract and concrete program environments that are used for function pointer calls.

context includes *fnptr-hl-impl-corres*

begin

thm *fun-ptr-intros*

end

Note for example, that in the current goal state the concrete program would be of the shape $\mathcal{P}\text{-}l2\text{---}int\ p$ whereas the abstract program would be a plain schematic variable. This variable is then instantiated with $\mathcal{P}\text{-}hl\text{---}int\ p$. Once the rule is applied the identity marker *DYN-CALL* triggers the sidecondition splitter / solver: `AutoCorresUtil.dyn_call_split_simp_sidecondition_tac`, which splits the generic correspondence of some generic function pointer to its possible values.

For example if the function pointer would index into the dispatcher array *dispatcher-u* and select a *binop-u-C*, it would result into subgoals for each of the possible pointers, namely *fnptr.add-u* and *fnptr.add-gu*.

context includes *fnptr-wa-impl-corres*

begin

Like in phase HL we also have some custom intro rules to support the instantiation of function pointer calls.

thm *fun-ptr-intros*

thm *global-const-defs*

thm *fun-ptr-simps*

end

The TS phase adds another dimension to the program environments, the monad type. E.g. $\mathcal{P}\text{-}pure\text{-unsigned}\text{---}unsigned$, $\mathcal{P}\text{-}gets\text{-unsigned}\text{---}unsigned$, $\mathcal{P}\text{-}option\text{-unsigned}\text{---}unsigned$, $\mathcal{P}\text{-}nondet\text{-unsigned}\text{---}unsigned$, $\mathcal{P}\text{-}exit\text{-unsigned}\text{---}unsigned$.

context includes *fnptr-ts-impl-corres*

begin

Again we have custom intro rules to support instantiation of the abstract program environment.

```

thm fun-ptr-intros
thm global-const-defs
thm fun-ptr-simps
end
end

```

26.19.2 Function / Recursive-clique specific locales

L1

```

context l1-definition-add — holds the definition of the function
begin
thm l1-def — all relevant l1 definitions
thm ac-def — definitions of all layers

```

Note that there are two variants of the definition of *l1-add'*. First the original definition and then an optimised version, already removing some exception handling. Here are the names of the theorems:

```

thm l1-add'-def
thm l1-opt-add'-def
end

```

```

context l1-impl-add
begin

```

As *l1-add'* might also be called indirectly via a function pointer, we populate the program environment with the definition

```

thm l1-add'-impl — Mapping of the function-pointer to the definition in the corresponding environment.

```

```

The equation is also added to gets-the (ogets ( $\lambda-. ?f$ )) = return ?f
gets-the (ogets ?f) = gets ?f
c-fnptr-guard fnptr.f
c-fnptr-guard fnptr.add
c-fnptr-guard fnptr.dec
c-fnptr-guard fnptr.inc
c-fnptr-guard fnptr.mul
c-fnptr-guard fnptr.add-u
c-fnptr-guard fnptr.inc-u
c-fnptr-guard fnptr.minus
c-fnptr-guard fnptr.add-gu
c-fnptr-guard fnptr.odd-disp
c-fnptr-guard fnptr.callable1
c-fnptr-guard fnptr.even-disp
c-fnptr-guard fnptr.intcallable2
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.odd-disp
= ?f1.0

```

```

    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.even-disp
= ?f2.0
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.intcallable2
= ?d fnptr.intcallable2
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.callable1
= ?d fnptr.callable1
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.add-gu
= ?d fnptr.add-gu
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.minus
= ?d fnptr.minus
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.add-u
= ?d fnptr.add-u
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.inc
= ?d fnptr.inc
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.mul
= ?d fnptr.mul
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.add
= ?d fnptr.add
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.f
= ?d fnptr.f
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.dec
= ?d fnptr.dec
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.inc-u
= ?d fnptr.inc-u
    P fnptr.add = l1-add'.

```

thm *fun-ptr-simps*

thm *l1-def*

end

context *l1-definition-call-binop*

begin

l1-call-binop' makes a function pointer call via the *dispatcher* array selecting field *binop-C*. So it might call *l1-add'*, *l1-minus'* or *l1-mul'*. The definitions of these functions are also imported into the locale (cf. *l1-add'* \equiv *L1-catch* (*L1-seq* (*L1-init* (λ upd. (*add.ret'*::32 signed word):= \mathcal{L} upd)) (*L1-seq* (*L1-modify* (*add.k'*::32 signed word):= \mathcal{L} (λ -. *gi*)) (*L1-seq* (*L1-guard* (λ s. $- 2147483648 \leq$ sint ($s \cdot \mathcal{L}$ (*add.n'*::32 signed word)) + sint ($s \cdot \mathcal{L}$ (*add.m'*::32 signed word)) \wedge sint ($s \cdot \mathcal{L}$ (*add.n'*::32 signed word)) + sint ($s \cdot \mathcal{L}$ (*add.m'*::32 signed word)) ≤ 2147483647)) (*L1-seq* (*L1-modify* (λ s. *s*(*add.ret'*::32 signed word) := \mathcal{L} λ -. ($s \cdot \mathcal{L}$ (*add.n'*::32 signed word)) + ($s \cdot \mathcal{L}$ (*add.m'*::32 signed word)))))) (*L1-seq* (*L1-modify* (*global-exn-var'-'-update* (λ -. *Return*))) (*L1-throw*)))))) (*L1-condition* (λ s. *is-local* (*global-exn-var'-'-s*)) *L1-skip* *L1-throw*)) *l1-add'* \equiv *L1-seq* (*L1-init* (λ upd. (*add.ret'*::32 signed word):= \mathcal{L} upd)) (*L1-seq* (*L1-modify* (*add.k'*::32 signed word):= \mathcal{L} (λ -. *gi*)) (*L1-seq* (*L1-guard*

$(\lambda s. - 2147483648 \leq \text{sint } (s \cdot_{\mathcal{L}} (\text{add.n}::32 \text{ signed word})) + \text{sint } (s \cdot_{\mathcal{L}} (\text{add.m}::32 \text{ signed word})) \wedge \text{sint } (s \cdot_{\mathcal{L}} (\text{add.n}::32 \text{ signed word})) + \text{sint } (s \cdot_{\mathcal{L}} (\text{add.m}::32 \text{ signed word})) \leq 2147483647)) (L1\text{-seq } (L1\text{-modify } (\lambda s. s(\text{add.ret}'::32 \text{ signed word } :=_{\mathcal{L}} \lambda-. (s \cdot_{\mathcal{L}} (\text{add.n}::32 \text{ signed word})) + (s \cdot_{\mathcal{L}} (\text{add.m}::32 \text{ signed word})))))) (L1\text{-modify } (\text{global-exn-var}'\text{'-update } (\lambda-. \text{Return}))))))$

$l1\text{-mul}' \equiv L1\text{-catch } (L1\text{-seq } (L1\text{-init } (\lambda \text{upd. } (\text{mul.ret}'::32 \text{ signed word}) :=_{\mathcal{L}} \text{upd})) (L1\text{-seq } (L1\text{-guard } (\lambda s. - 2147483648 \leq \text{sint } (s \cdot_{\mathcal{L}} (\text{mul.n}::32 \text{ signed word})) * \text{sint } (s \cdot_{\mathcal{L}} (\text{mul.m}::32 \text{ signed word})) \wedge \text{sint } (s \cdot_{\mathcal{L}} (\text{mul.n}::32 \text{ signed word})) * \text{sint } (s \cdot_{\mathcal{L}} (\text{mul.m}::32 \text{ signed word})) \leq 2147483647)) (L1\text{-seq } (L1\text{-modify } (\lambda s. s(\text{mul.ret}'::32 \text{ signed word } :=_{\mathcal{L}} \lambda-. (s \cdot_{\mathcal{L}} (\text{mul.n}::32 \text{ signed word})) * (s \cdot_{\mathcal{L}} (\text{mul.m}::32 \text{ signed word})))))) (L1\text{-seq } (L1\text{-modify } (\text{global-exn-var}'\text{'-update } (\lambda-. \text{Return}))) L1\text{-throw})))) (L1\text{-condition } (\lambda s. \text{is-local } (\text{global-exn-var}'\text{'-} s)) L1\text{-skip } L1\text{-throw}))$

$l1\text{-mul}' \equiv L1\text{-seq } (L1\text{-init } (\lambda \text{upd. } (\text{mul.ret}'::32 \text{ signed word}) :=_{\mathcal{L}} \text{upd})) (L1\text{-seq } (L1\text{-guard } (\lambda s. - 2147483648 \leq \text{sint } (s \cdot_{\mathcal{L}} (\text{mul.n}::32 \text{ signed word})) * \text{sint } (s \cdot_{\mathcal{L}} (\text{mul.m}::32 \text{ signed word})) \wedge \text{sint } (s \cdot_{\mathcal{L}} (\text{mul.n}::32 \text{ signed word})) * \text{sint } (s \cdot_{\mathcal{L}} (\text{mul.m}::32 \text{ signed word})) \leq 2147483647)) (L1\text{-seq } (L1\text{-modify } (\lambda s. s(\text{mul.ret}'::32 \text{ signed word } :=_{\mathcal{L}} \lambda-. (s \cdot_{\mathcal{L}} (\text{mul.n}::32 \text{ signed word})) * (s \cdot_{\mathcal{L}} (\text{mul.m}::32 \text{ signed word})))))) (L1\text{-modify } (\text{global-exn-var}'\text{'-update } (\lambda-. \text{Return}))))))$

$l1\text{-minus}' \equiv L1\text{-catch } (L1\text{-seq } (L1\text{-init } (\lambda \text{upd. } (\text{ret}'::32 \text{ signed word}) :=_{\mathcal{L}} \text{upd})) (L1\text{-seq } (L1\text{-guard } (\lambda s. - 2147483648 \leq \text{sint } (s \cdot_{\mathcal{L}} (\text{n}::32 \text{ signed word})) - \text{sint } (s \cdot_{\mathcal{L}} (\text{m}::32 \text{ signed word})) \wedge \text{sint } (s \cdot_{\mathcal{L}} (\text{n}::32 \text{ signed word})) - \text{sint } (s \cdot_{\mathcal{L}} (\text{m}::32 \text{ signed word})) \leq 2147483647)) (L1\text{-seq } (L1\text{-modify } (\lambda s. s(\text{ret}'::32 \text{ signed word } :=_{\mathcal{L}} \lambda-. (s \cdot_{\mathcal{L}} (\text{n}::32 \text{ signed word})) - (s \cdot_{\mathcal{L}} (\text{m}::32 \text{ signed word})))))) (L1\text{-seq } (L1\text{-modify } (\text{global-exn-var}'\text{'-update } (\lambda-. \text{Return}))) L1\text{-throw})))) (L1\text{-condition } (\lambda s. \text{is-local } (\text{global-exn-var}'\text{'-} s)) L1\text{-skip } L1\text{-throw}))$

$l1\text{-minus}' \equiv L1\text{-seq } (L1\text{-init } (\lambda \text{upd. } (\text{ret}'::32 \text{ signed word}) :=_{\mathcal{L}} \text{upd})) (L1\text{-seq } (L1\text{-guard } (\lambda s. - 2147483648 \leq \text{sint } (s \cdot_{\mathcal{L}} (\text{n}::32 \text{ signed word})) - \text{sint } (s \cdot_{\mathcal{L}} (\text{m}::32 \text{ signed word})) \wedge \text{sint } (s \cdot_{\mathcal{L}} (\text{n}::32 \text{ signed word})) - \text{sint } (s \cdot_{\mathcal{L}} (\text{m}::32 \text{ signed word})) \leq 2147483647)) (L1\text{-seq } (L1\text{-modify } (\lambda s. s(\text{ret}'::32 \text{ signed word } :=_{\mathcal{L}} \lambda-. (s \cdot_{\mathcal{L}} (\text{n}::32 \text{ signed word})) - (s \cdot_{\mathcal{L}} (\text{m}::32 \text{ signed word})))))) (L1\text{-modify } (\text{global-exn-var}'\text{'-update } (\lambda-. \text{Return}))))))$

$l1\text{-call-binop}' \equiv L1\text{-catch } (L1\text{-seq } (L1\text{-init } (\lambda \text{upd. } (\text{call-binop.ret}'::32 \text{ signed word}) :=_{\mathcal{L}} \text{upd})) (L1\text{-seq } (L1\text{-modify } (\text{call-binop.r}::32 \text{ signed word}) :=_{\mathcal{L}} (\lambda-. 0)) (L1\text{-seq } (L1\text{-guard } (\lambda s. \text{UCAST}(8 \rightarrow 32 \text{ signed}) (s \cdot_{\mathcal{L}} (\text{call-binop.i}::8 \text{ word})) < s 5)) (L1\text{-seq } (L1\text{-guard } (\lambda s. 0 \leq s \text{ UCAST}(8 \rightarrow 32 \text{ signed}) (s \cdot_{\mathcal{L}} (\text{call-binop.i}::8 \text{ word})))) (L1\text{-seq } (L1\text{-condition } (\lambda s. \text{PTR-COERCE}(\text{unit} \rightarrow \text{unit}) (\text{binop-C } (\text{dispatcher}.\text{[unat } (s \cdot_{\mathcal{L}} (\text{call-binop.i}::8 \text{ word})))))) \neq \text{NULL}) (L1\text{-guarded } (\lambda s. 0 \leq s \text{ UCAST}(8 \rightarrow 32 \text{ signed}) (s \cdot_{\mathcal{L}} (\text{call-binop.i}::8 \text{ word})) \wedge \text{UCAST}(8 \rightarrow 32 \text{ signed}) (s \cdot_{\mathcal{L}} (\text{call-binop.i}::8 \text{ word})) < s 5 \wedge \text{c-fnptr-guard } (\text{binop-C } (\text{dispatcher}.\text{[unat } (s \cdot_{\mathcal{L}} (\text{call-binop.i}::8 \text{ word})))))) (do { p ← gets } (\lambda s. \text{binop-C } (\text{dispatcher}.\text{[unat } (s \cdot_{\mathcal{L}} (\text{call-binop.i}::8 \text{ word}))))); L1\text{-call } (\lambda s. \text{locals-update } (\text{cupdate } (\text{Suc } 0) (\lambda-. s \cdot_{\mathcal{L}} (\text{call-binop.m}::32 \text{ signed word}))))$

(locals-update (cupdate 0 (λ-. s ·_ℒ (call-binop.n::32 signed word))) s) (P p)
(λs t. s(|globals := globals t)) (λs t. s(|global-exn-var'-':= Nonlocal (the-Nonlocal
(global-exn-var'-' t))) (λuu-. (call-binop.r::32 signed word):=_ℒ (λ-. clookup
2 (locals uu-))) }))) L1-skip) (L1-seq (L1-modify (λs. s(|call-binop.ret':=32
signed word :=_ℒ λ-. s ·_ℒ (call-binop.r::32 signed word)))) (L1-seq (L1-modify
(global-exn-var'-'-update (λ-. Return))) L1-throw)))))) (L1-condition (λs.
is-local (global-exn-var'-' s) L1-skip L1-throw)
l1-call-binop' ≡ L1-seq (L1-init (λupd. (call-binop.ret':=32 signed word):=_ℒ
upd)) (L1-seq (L1-modify (call-binop.r::32 signed word):=_ℒ (λ-. 0)) (L1-seq
(L1-guard (λs. UCAST(8 → 32 signed) (s ·_ℒ (call-binop.i::8 word)) <s 5))
(L1-seq (L1-guard (λs. 0 ≤s UCAST(8 → 32 signed) (s ·_ℒ (call-binop.i::8
word)))) (L1-catch (L1-seq (L1-condition (λs. PTR-COERCE(unit → unit)
(binop-C (dispatcher.[unat (s ·_ℒ (call-binop.i::8 word))])) ≠ NULL) (L1-guarded
(λs. 0 ≤s UCAST(8 → 32 signed) (s ·_ℒ (call-binop.i::8 word)) ∧ UCAST(8
→ 32 signed) (s ·_ℒ (call-binop.i::8 word)) <s 5 ∧ c-fnptr-guard (binop-C
(dispatcher.[unat (s ·_ℒ (call-binop.i::8 word))])) (do { p ← gets (λs.
binop-C (dispatcher.[unat (s ·_ℒ (call-binop.i::8 word))]); L1-call (λs. lo-
icals-update (cupdate (Suc 0) (λ-. s ·_ℒ (call-binop.m::32 signed word))) (locals-update
(cupdate 0 (λ-. s ·_ℒ (call-binop.n::32 signed word))) s) (P p) (λs t. s(|globals
:= globals t)) (λs t. s(|global-exn-var'-':= Nonlocal (the-Nonlocal (global-exn-var'-'
t))) (λuu-. (call-binop.r::32 signed word):=_ℒ (λ-. clookup 2 (locals uu-)))
})) L1-skip) (L1-seq (L1-modify (λs. s(|call-binop.ret':=32 signed word :=_ℒ
λ-. s ·_ℒ (call-binop.r::32 signed word)))) (L1-seq (L1-modify (global-exn-var'-'-update
(λ-. Return))) L1-throw)) (L1-condition (λs. is-local (global-exn-var'-' s)
L1-skip L1-throw)))))), as well as the corresponding entries in the program
environment (cf. *gets-the (ogets (λ-. ?f)) = return ?f*

gets-the (ogets ?f) = gets ?f
c-fnptr-guard fnptr.f
c-fnptr-guard fnptr.add
c-fnptr-guard fnptr.dec
c-fnptr-guard fnptr.inc
c-fnptr-guard fnptr.mul
c-fnptr-guard fnptr.add-u
c-fnptr-guard fnptr.inc-u
c-fnptr-guard fnptr.minus
c-fnptr-guard fnptr.add-gu
c-fnptr-guard fnptr.odd-disp
c-fnptr-guard fnptr.callable1
c-fnptr-guard fnptr.even-disp
c-fnptr-guard fnptr.intcallable2
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.odd-disp
= ?f1.0
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.even-disp
= ?f2.0


```

    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.intcallable2
= ?d fnptr.intcallable2
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.callable1
= ?d fnptr.callable1
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.add-gu
= ?d fnptr.add-gu
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.minus
= ?d fnptr.minus
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.add-u
= ?d fnptr.add-u
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.inc
= ?d fnptr.inc
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.mul
= ?d fnptr.mul
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.add
= ?d fnptr.add
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.f
= ?d fnptr.f
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.dec
= ?d fnptr.dec
    map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.inc-u
= ?d fnptr.inc-u

```

\mathcal{P} *fnptr.add* = *l1-add'*

\mathcal{P} *fnptr.mul* = *l1-mul'*

\mathcal{P} *fnptr.minus* = *l1-minus'*). Also note that the function pointer call is wrapped in a *L1-guarded*. In particular the guard ensures that the index into the array is in range. It is essential to have this information available in the correspondence proof to split the function pointer call to its potential targets.

thm *l1-call-binop'-def*

thm *global-const-defs*

thm *l1-def*

thm *fun-ptr-simps*

end

context *l1-corres-call-binop*

begin

The correspondence proofs are performed within the *corres* locales. The results are added to *L1corres True Γ l1-add' (Call fnptr.add)*

L1corres True Γ l1-mul' (Call fnptr.mul)

L1corres True Γ l1-minus' (Call fnptr.minus)

L1corres True Γ l1-call-binop' (Call fnptr.call-binop). Note that the correspondence proofs of the potential callees are present. **autocorres** resolves call dependencies and first performs the correspondence proofs of the

potential callees.

```
thm l1-call-binop'-corres  
thm l1-corres  
thm l1-def  
end
```

```
context l1-definition-parameter-call  
begin
```

This function performs a function pointer call on a function parameter. Recall that we only support the case where the function pointer parameter value stays the same within the function. The phase L1 is special, as we postpone the correspondence proof of the function pointer call to the L2 layer. The reason is that in the L2 layer we replace the parameters with lambda abstraction and thus the fact that the value does not change becomes trivial. In L1 we would have to propagate this information through every state update, from the invocation of the function to the function pointer call. This proof is anyway performed in the L2 phase. Postponing the correspondence proof is achieved by just assuming the correspondence in *L1-guarded*.

```
thm l1-parameter-call'-def  
thm l1-def  
end
```

```
context l1-corres-parameter-call  
begin  
thm l1-parameter-call'-corres  
thm l1-corres  
end
```

L2

```
context l2-definition-add  
begin  
thm fun-ptr-simps  
end
```

```
context l2-impl-add  
begin  
thm fun-ptr-simps  
end
```

```
context l2-definition-call-binop  
begin  
thm fun-ptr-simps  
thm l2-call-binop'-def  
thm l2-def  
end
```

```

context l2-corres-call-binop
begin
thm l2-call-binop'-corres
thm l2-corres
thm l2-def
end

```

```

context l2-definition-parameter-call
begin

```

Note that in contrast to L1 we do not have any correspondence assumption in *L2-guarded*. The function pointer parameter is just inserted in the function pointer call and thus is immediately the same value.

```

thm l2-parameter-call'-def
thm l2-def
end

```

```

context l2-corres-parameter-call
begin

```

In contrast to the function pointer call via a global variable, where we statically resolve which functions are potentially called, we add an explicit assumption on the parameter to the correspondence theorem. As in phase L2 we also have to proof the postponed L1 correspondence we actually have both assumptions here.

```

thm l2-parameter-call'-corres
thm l2-corres
thm l2-def
end

```

When defining (mutual) recursion indirectly via function pointers, there is a subtlety in the definition of the functions. In order to have an admissible *L2corres* property the program environment for the current clique is temporarily explicitly extended via *map-of-default* in the function bodies. After definition of the function and extending the hypothetical program environment with the new definitions this construction is hidden again.

Compare the variation of `in` in *l2-corres-odd-disp-even-disp* vs. *l2-impl-odd-disp-even-disp* and the extended program environment in *gets-the* (*ogets* $(\lambda-. ?f)$) = *return* *?f*

gets-the (*ogets* *?f*) = *gets* *?f*.

Also recall the general flow of how to arrive at a new level with definition and correspondence proof.

- First the induction step of the correspondence proof is performed, assuming that the recursive calls are in the correspondence relation.
- If that proof is successful the function body or bodies are used to do the actual definition(s) of the function(s).

- After the definition is done the actual correspondence proof is performed using the induction step as major ingredient, replacing the body / bodies with the new definition(s).
- The program environments are extended with the assumptions that the pointers are mapped to the new definitions and the right hand sides of the definitions are simplified.

```

context l2-corres-odd-disp-even-disp
begin
thm fun-ptr-simps
thm l2-corres
thm l2-def
thm l2-odd-disp'-def — Foundational definition of fixed-point. FIXME: should we conceale this?
thm l2-odd-disp'.simps
end

```

```

context l2-impl-odd-disp-even-disp
begin
thm fun-ptr-simps
thm l2-corres
thm l2-def
thm l2-odd-disp'.simps
thm l2-impl-odd-disp'-def — canonical variant. FIXME: should we rename this (simps?)?
end

```

HL

```

context hl-definition-call-binop
begin
thm fun-ptr-simps
thm hl-call-binop'-def
thm hl-def
end

```

```

context hl-corres-call-binop
begin
thm hl-call-binop'-corres
thm hl-corres
thm hl-def
end

```

```

context hl-definition-parameter-call
begin
thm hl-parameter-call'-def
thm hl-def
end

```

```

context hl-corres-parameter-call
begin
thm hl-parameter-call'-corres
thm hl-corres
thm hl-def
end

```

WA

```

context wa-definition-call-binop
begin
thm fun-ptr-simps
thm wa-call-binop'-def
thm wa-def
end

```

```

context wa-corres-call-binop
begin
thm wa-call-binop'-corres
thm wa-corres
thm wa-def
end

```

```

context wa-definition-parameter-call
begin
thm wa-parameter-call'-def
thm wa-def
end

```

```

context wa-corres-parameter-call
begin
thm wa-parameter-call'-corres
thm wa-corres
thm wa-def
end

```

TS

```

context ts-impl-add
begin

```

add' is defined within the option monad. Note that (lifted version) also end up in the more expressive program environments (cf. *gets-the* (*ogets* $(\lambda-. ?f) = \text{return } ?f$) *gets* $(\text{ogets } ?f) = \text{gets } ?f$ *c-fnptr-guard* *fnptr.f* *c-fnptr-guard* *fnptr.add* *c-fnptr-guard* *fnptr.dec*

```

c-fnptr-guard fnptr.inc
c-fnptr-guard fnptr.mul
c-fnptr-guard fnptr.add-u
c-fnptr-guard fnptr.inc-u
c-fnptr-guard fnptr.minus
c-fnptr-guard fnptr.add-gu
c-fnptr-guard fnptr.odd-disp
c-fnptr-guard fnptr.callable1
c-fnptr-guard fnptr.even-disp
c-fnptr-guard fnptr.intcallable2
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.odd-disp
= ?f1.0
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.even-disp
= ?f2.0
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.intcallable2
= ?d fnptr.intcallable2
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.callable1
= ?d fnptr.callable1
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.add-gu
= ?d fnptr.add-gu
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.minus
= ?d fnptr.minus
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.add-u
= ?d fnptr.add-u
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.inc
= ?d fnptr.inc
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.mul
= ?d fnptr.mul
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.add
= ?d fnptr.add
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.f
= ?d fnptr.f
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.dec
= ?d fnptr.dec
map-of-default ?d [(fnptr.odd-disp, ?f1.0), (fnptr.even-disp, ?f2.0)] fnptr.inc-u
= ?d fnptr.inc-u
P-option-int'int---int fnptr.add = add'
P-nondet-int'int---int fnptr.add = (λx1 x2. gets-the (add' x1 x2))
P-exit-int'int---int fnptr.add = (λx1 x2. liftE (gets-the (add' x1 x2))).

```

So function pointer calls to this function can be performed in any of those monads.

```

thm add'-def
thm fun-ptr-simps
thm ts-def

```

```

end

context ts-definition-call-binop
begin
thm fun-ptr-simps
thm call-binop'-def
thm ts-def
end

```

```

context ts-corres-call-binop
begin
thm call-binop'-corres
thm ts-corres
thm ts-def
end

```

```

context ts-definition-fac
begin
thm fun-ptr-simps
thm fac'.simps
thm ts-def
end

```

```

context ts-corres-fac
begin
thm fac'-corres
thm ts-corres
thm ts-def
end

```

Recall that in case of a function pointer call via a parameter we assume correspondence of the parameter. So in which monad should we end up? Which monad should we assume for the parameter? Currently we assume the same monad as for the function. We consecutively try the correspondence from the most restrictive to the most expressive monad and we end up in the monad with the first successful proof. If you want a different (more expressive) monad you can configure this via the autocorres option *ts-force*.

```

context ts-definition-parameter-call
begin
thm parameter-call'-def
thm wa-def
end

```

```

context ts-corres-parameter-call
begin
thm parameter-call'-corres
thm wa-corres
thm wa-def
end

```

```

context ts-impl-call-binop
begin
thm ts-def
end

```

Final AutoCorres

Once all phases are passed and the correspondence proofs between the consecutive layers are ready they are combined to the final correspondence layer, from SIMPL upto TS.

```

context corres-parameter-call
begin
thm parameter-call'-ac-corres
end

```

```

context corres-call-binop
begin
thm call-binop'-ac-corres
end

```

```

context corres-odd-disp-even-disp
begin
thm odd-disp'-ac-corres
thm even-disp'-ac-corres
end

```

When all functions of a C file are translated the following summary locales are introduced.

```

context fnptr-all-impl — All implementation locales of phase TS
begin
thm fun-ptr-simps
thm even-disp'.simps
thm impl-even-disp'-def
thm ac-def
end

```

```

context fnptr-all-corres — All corres locales of all phases
begin
thm l1-corres
thm l2-corres
thm hl-corres
thm wa-corres
thm ts-corres
thm ac-corres

```

```

thm l1-def
thm l2-def
thm l2-even-disp'-def

```



```

thm l2-even-disp'.simps
thm hl-def
thm wa-def
thm ts-def
thm ac-def

thm global-const-defs
thm fun-ptr-simps
thm fun-ptr-intros
thm fun-ptr-distinct
thm fun-ptr-subtree
end

```

Method calls of C-style objects

The goal is to also support function pointer invocations via C-style object methods: Instead of a plain function pointer a function gets a pointer to a structure (object) that contains function pointers as structure fields. This adds an important twist to the refinement proofs of autocorres. When we invoke a method via `p->method(...)` we actually calculate the function pointer from the heap object referenced by pointer `p`. As with the cases described before at that point we have to argue that the referenced function fulfills the 'corres' predicate. However, now we have the indirection via the heap. So somehow we have to keep track of the function pointers stored in the heap. As other pointer objects might also be stored and manipulated in the heap this typically requires user-level reasoning about the heap layout and disjointness of certain pointers. As autocorres is mainly used as a preprocessing tool for arbitrary C programs we want to avoid making too restrictive assumptions here. Instead we postpone the argument to the user. The core idea is that we want to argue that the actually referenced function is a 'known function' for which we know or can assume that the 'corres' predicate holds. We introduce a predicate *known-function* into our locale hierarchy, which takes a function pointer *p* and tells us if this is a known function for which we can assume 'corres'. On every invocation of a object method we introduce a guard *known-function (method p)* into the code. Autocorres can then utilise this guard together with the assumption that the correspondence holds for all known functions. It is then left as an exercise for the user to discharge the guards. This boils down to an invariant argument that every time you assign something to *method p* it has to be a *known-function*. To support this, everytime a function is defined which might be such a function pointer, we utilise the implementation locales to store the information that this is a *known-function*.

Here are the locales about the *known-function* which state correspondence. These are imported as parents in all individual corres locales.

```

context fnptr-l1-corres

```

```
begin  
thm known-function-corres  
end
```

```
context fnptr-l2-corres  
begin  
thm known-function-corres  
end
```

```
context fnptr-hl-corres  
begin  
thm known-function-corres  
end
```

The locales for phases WA and TS are special as they might change the signature of the resulting function depending on the called functions. We make assumptions about called methods to derive the definition of the calling function. E.g. for phase WA we have to know whether a called method will do signed or unsigned word abstractions. For phase TS we have to know in which monad the called methods are defined in order to select the right monad for the caller. We can use the autocorres options `ts_force_known_functions`, `unsigned_word_abs_known_functions` and `no_signed_word_abs_known_functions` analogous to the corresponding options for individual functions.

In our example program we have chosen the option monad as a result monad. Note that autocorres does some sanity checks to ensure that e.g. the chosen monad is expressible enough, but ultimately it is the responsibility of the user to ensure that the monad and the word abstraction options match all potential instances of the methods.

```
context fnptr-wa-corres  
begin  
thm known-function-corres  
end
```

```
context fnptr-ts-corres  
begin  
thm known-function-corres  
end
```

```
context ts-definition-call-object-binop-return  
begin  
thm ts-def  
end
```

```
context ts-definition-call-object-binop-assign  
begin  
thm ts-def  
end
```

```

context ts-definition-call-object-binop-emb
begin
thm ts-def
end

```

Here you can see all known functions that autocorres has detected. Note that for this analysis autocorres assumes that it has total knowledge of the complete program. It collects all functions for which addresses are calculated in the program and considers them as *known-function*.

```

context fnptr-all-impl
begin
thm known-function
end

```

```

end

```

26.20 Unions

```

theory union-ac
imports
  AutoCorres

```

```

begin

```

26.20.1 Union support in C-Parser and Autocorres

Fully fledged C unions can have a lot of semantic subtleties in particular with respect to undefined behavior when it comes to different padding in the variants, effective types and accessing the union via different variants, e.g. writing a pointer through one variant and subsequently reading or writing the same union via another variant. See for example <https://robbertkrebbers.nl/research/thesis.pdf> in subsections 2.5.5 ff. for a discussion of some fine points.

Our current solution is very pragmatic by only supporting a subset of unions that does not exhibit the more "weired" behaviours:

- Single variant unions
- Multi-variant unions where each variant is packed (no implicit padding) and has the same size.

For these limited unions the semantics is deterministic and basically boils down to casting / coercing different C-types. Note that every *c-type* is equipped with conversions *from-bytes* and *to-bytes* which allows us to take a value, convert it to a byte list and then make another

value (potentially with a different type) from that byte list again. So the basic machinery is already there.

The C parser analyses which variants of a union are actually actively used in the code you want to verify. Only those variants (internally referred to as active variants) are taken into account. If there is only a single variant we are fine and a single record type is constructed in HOL as the C-type representing that type. It is the same construction as for a structure type of that variant.

In case there are multiple active variants the C parser checks that all those variants have the same size and are *packed* which means that there is no padding. The original C code might have to be rewritten with explicit padding fields to fulfill this requirement.

We create a record type for each variant of the union, like a 'struct' for each variant. One variant is considered as the *canonical* variant. Currently this always is the first variant of the union. The idea of the canonical variant is that every access or update to a union is always done via this canonical variant and then 'casted' to the variant you need. This casting is implemented via *PTR-COERCE*('a → 'b) for pointers and *COERCE*('a → 'b) and *coerce-map* for values.

Example:

```
typedef struct {
    unsigned fst;
    unsigned snd;
} unsigned_pair;
```

```
typedef struct {
    signed fst;
    signed snd;
} signed_pair;
```

```
typedef union {
    unsigned_pair u;
    signed_pair s;
} open_union;
```

The union *open-union* has two variants. The first one *u* is considered to be the canonical one. So when you want to update a variant *s* in the heap via a pointer to *open-union* we first read the canonical variant *u* from the heap then coerce the value to a variant *s*, then the update on the value is performed then the resulting value is coerced back to variant *u* and finally written back to the heap.

A good mental model for this implementation is that accesses to unions are normalised towards the canonical variants of the root pointers. This also carries over to the split heap model you obtain when you perform the HL phase of autocorres. There is only one relevant heap for the union, which is the heap for the canonical variant. This means that you can also define *addressable-fields* for the canonical variant. Currently you cannot define *addressable-fields* for the other variants.

```
include-C-file union.h for union.c
install-C-file union.c
autocorres [addressable-fields = open-union.u] union.c
```

```
context union-all-corres
begin
thm ts-def
thm l2-def
```

Names

The union *open-union* has two variants *u* and *s*. The first one, *u*, is considered to be the canonical one. The corresponding c-type becomes "THE" type of union and is named *open-union-C*. The other variant *s* gets the qualified name *open-union-C's-C*.

```
term open-union-C
term u-C
thm u-C-def
term u-C-update
thm u-C-update-def
```

```
term open-union-C's-C
term s-C
thm s-C-def
term s-C-update
thm s-C-update-def
```

```
term heap-open-union-C
thm heap-open-union-C-def
```

Anonymous Structures and Union

For anonymous structures and types we create names by considering the surrounding typedef and structure / union. For example see the typedef of *my-union*. The canonical variant for the union gets *my-union-C*. The variant *f* is an anonymous structure. The type of the anonymous structure is type *my-union-C'f-C'struct*, whereas the type of the variant *f* is *my-union-C'f-C*. Analogously there is also a suffix *union* in case of an anonymous union. Note the naming convention: For nested names note that suffix *union* or *struct* is for the nested union or structure itself, in contrast to the name without suffix which denotes the variant of the enclosing union.

```
typ my-union-C  
term my-union-C
```

```
typ my-union-C'f-C'struct  
term f1-C  
thm f1-C-def
```

```
typ my-union-C'f-C  
term f-C  
thm f-C-def
```

```
typ my-union-C'g-C'union  
term x1-C  
thm x1-C-def
```

```
typ my-union-C'g-C  
term g-C  
thm g-C-def
```

Lemmas to coerce values / heap accesses

```
thm coerce-id  
thm coerce-cancel-packed  
thm coerce-map-id  
thm coerce-coerce-map-cancel-packed
```

```
thm h-val-coerce-ptr-coerce-packed  
thm h-val-field-ptr-coerce-from-bytes-packed  
thm heap-update-field-ptr-coerce-from-bytes-packed  
thm heap-state.L2-modify-heap-update-ptr-coerce-packed-conv  
thm heap-state.L2-modify-heap-update-ptr-coerce-packed-field-root-conv  
thm L2-modify-heap-update-field-root-conv
```

```
end
```

end

26.21 Pointers into Structures in Split Heap

```
theory open-struct
  imports AutoCorres
begin
```

26.21.1 Overview

Heap lifting in AutoCorres transforms the monolithic byte-level heap *heap-mem* with explicit term level heap type description *heap-typ-desc* and typing constraints, like *c-guard*, *h-t-valid* and $d \models_t p$ to a more abstract split heap model with implicit HOL-typing. The major advantage of the split heap model is that, updates to one particular heap do not affect all other distinct heaps, removing the obligation to explicitly take care about aliasing. Aliasing between different heaps is ruled out by the model already.

The question that arises is on which granularity do we split the byte-level heap? The original AutoCorres splitted at the granularity of types. Each distinct c-type is mapped into a separate heap. In that model you can still have intra-type aliasing (i.e. pointers of the same type can alias), but no inter-type aliasing (i.e. pointers of different type are distinct, and cannot alias). As an example a pointer to a list structure refers to a different (split) heap as a pointer to a tree structure. So when a function modifies lists, no pointer to a tree is affected.

Of course not every C-program can be represented in this split-heap model. In those cases the abstraction fails. In particular think of a low-level memory allocator, that allocates some raw-bytes and delivers a void-pointer, that is then retyped to point to the actual structure. To be able to combine those byte-level functions with high-level typed functions (in the split heap) one can switch between both layers of abstraction by *exec-concrete* and *exec-abstract*. The good thing is that this gives us a very expressive tool to switch between byte-level and typed view. We can do the specification and proofs of different parts of the program on the adequate level and still combine the results. But switching between the layers can be tedious and thus should be limited to the low-level functions that really inherently need byte-level reasoning.

Unfortunately the type granularity of the split heaps also rules out programs dealing with pointers into a structure. For example a structure containing a *int* field *count* cannot pass a *int* pointer to that field *count* to a another function, that might update it. The complete structure has a separate (split) heap that is distinct from the heap containing *ints*. This would mean that we have to resort to the byte-level heap to reason about those functions.

We extended the split heap model to support this common use-case. The core idea is that in addition to split heaps on the granularity of types you can also specify split heaps on the granularity of structure fields. Conceptually you 'open' the structure into its components (aka. fields) and each field gets a separate heap. Additionally you can specify which fields should be *addressable* and thus shared. In the example you can specify that *count* is addressable, so it is represented within the same heap as all other *int*. This means that an *int* pointer can now point to a plain *int* or to a *count* field within a structure. You explicitly allow this limited inter-type aliasing on *int* pointers. All other fields that are not explicitly marked as *addressable* still have their own heap and thus cannot alias with another pointer of the same field type. Note that mainly for performance reasons we actually avoid to have a separate heap for each single field but try to minimise the number of heaps by clustering the fields that are treated the same. Some more details on this construction are below.

26.21.2 Example Program and some Intuition

We illustrate the approach with the following example specifying some structures, with addressable and non-addressable fields. Note that for an addressable field that is of an array type, each element of the array is considered to be addressable.

install-C-file *open-struct.c*

```
declare [[show-types=false, show-sorts=false,
verbose = 0, verbose-timing = 0, ML-print-depth = 1000, goals-limit = 20]]
```

```
init-autocorres [addressable-fields =
inner.fld1
inner.fld2
outer.inner
outer-array.inner-array
array.elements
other.fy
two-dimensional.matrix
data-struct1.d1.y1
data-struct1.d1.y2
data-struct1.d1.y3
data-struct1.d1.y4
data-struct1.d2
data-struct2.d
data-array.array,
single-threaded ] open-struct.c
```

autocorres *open-struct.c*

Which split heaps are generated? Each split heap is a component of the record *lifted-globals*.

print-record *lifted-globals*

Structure *closed* does not have any addressable field. So there is a separate heap for that component *closed-C*.

Structure *inner* has three *unsigned* fields *fld1-C*, *fld2-C* and *fld3-C*, The first two fields are addressable whereas the third one is not. This means that the first two fields are put into the same common heap for unsigned integers *heap-w32* whereas the third field is stored in a dedicated heap for all remaining fields *dedicated-heap-inner-C*.

We use the following nomenclature to distinguish the various kinds of split heaps and types:

- closed type or structure: There is no addressable field within the type / structure, e.g. *closed-C*.
- open type or structure: There is at least one addressable field within the type / structure, e.g. *inner-C*.
- atomic heap: Heap for a closed type, or a common heap that might be nested in an open structure. e.g. *heap-closed-C*.
- dedicated heap: Internal heap for all non-addressable fields of an open type, e.g. *dedicated-heap-inner-C*. These dedicated heaps are used for the construction of a derived heap.
- derived heap: Heap for an open type / structure. The name 'derived' indicates that it is not a fundamental heap but a composition of several common heaps and a dedicated heap, e.g. *outer-C.inner-C*
- fundamental heap: those heaps that are directly present in the new global state *lifted-globals*. These are atomic heaps, and dedicated heaps.
- virtual heap: heaps that represent the user level view. They are directly visible in the program after heap-lifting. This excludes dedicated heaps, which non-the-less may be used in the model as representation of that type.

Dedicated heaps play a special role, as they are subsumed within derived heaps. In a sense they are not part of the 'API' for split heaps: After heap lifting the resulting program only directly mentions atomic, common or derived heaps. Nevertheless dedicated heaps are important for the construction of derived heaps and might show up after unfolding the definitions for derived heaps.

Also dedicated heaps may be used by the user as the actual heap for a specific type. Only the dedicated heaps and the atomic heaps commute, the derived heaps do not necessary commute with each other as they might have common components that alias each other.

26.21.3 Background

The starting point for the split heap abstraction is still the unified memory model (UMM) by Harvey Tuch (<https://trustworthy.systems/publications/papers/Tuch%3Aphd.pdf>). The UMM provides the general concept of *c-type* to convert between a byte-list representations of C values to abstract HOL-types. It also provides an explicit notion of typing and what a well-typed byte level heap is (with respect to a heap type description). With these notions in place one can describe the set of valid pointers for a C program. The model is quite elaborate and respects the potential nesting of structures. A well-typed pointer to a structure in the heap gives rise to a bunch of other valid sub-pointers, pointing to sub-fields, which again could point to a nested structure. In the UMM, array types are modelled as a special case of a structure, where each index corresponds to an artificial field, where the field name is a unary encoding of the index. Hence, for most of the explanation in this file it is sufficient to elaborate on structures. The main limitation of the UMM is that it does not handle unions.

For the original split heap version of AutoCorres https://trustworthy.systems/publications/nicta_full_text/8758.pdf, David Greenaway put a simplified layer on top of the UMM focusing only on the root-pointers of a structure, ignoring any nested pointers to a field. This provides a split heap abstraction on the granularity of types.

The new split heap model takes an intermediate view. For closed structures it coincides with the original split heap model. Only root-pointers to a closed structure are valid. In addition for open structures also the nested pointers to addressable fields are welcome and valid. This flexibility allows the user to carefully open up the structures only as far as needed. Note the trade-off between expressibility and "proof-burden" of open structures. Aliasing is by construction ruled out for distinct split heaps, but has to be dealt with explicitly on the user level for common heaps.

To allow this flexibility some existing notions were slightly refined and some notions were introduced.

Footprints and valid pointers

The central typing judgement of UMM is *h-t-valid*, with infix syntax $d, g \models_t p$, meaning pointer p is valid with respect to heap type description d and a guard g . The default guard is *c-guard*, ensuring that the pointer is aligned, is

not NULL and that the referenced value fits into the address space, meaning that the addresses of the individual bytes do not overflow the address space. This instantiation with *c-guard* is abbreviated with $d \models_t p$.

thm *c-guard-def*
thm *TypHeap.h-t-valid-def*
thm *h-t-valid-valid-footprint*

The judgement $d, g \models_t p$ also ensures that the footprint is valid: *valid-footprint* d (*ptr-val* p) (*typ-uinfo-t* $TYPE('a)$). Note that p is a typed pointer, carrying the (phantom) type of the value it points to p . This type annotation is no longer present in *valid-footprint* which only talks about the address of the pointer *ptr-val* and relates it to a type tag via *typ-uinfo-t*.

thm *valid-footprint-def*

Without going into details of the definition of *valid-footprint* we get guarantees for all nested pointers of the structure which also have *valid-footprint*. Moreover, what we do not know is whether the pointer p is itself a nested pointer of some enclosing structure.

To get the additional knowledge that term p is a root pointer, meaning it is not enclosed in a bigger structure, we have the notion *valid-root-footprint*. This notion was refined from the original version of AutoCorres *valid-simple-footprint* to get the important property that every *valid-root-footprint* is indeed also a *valid-footprint*.

thm *valid-root-footprint-valid-footprint*

Also for all types that fit into the address space we can go from *valid-root-footprint* to *valid-simple-footprint*. The restriction on the address space size follows from the implicit property of the legacy *valid-simple-footprint*: The first byte and all the rest have a different tag, so there must not be a complete wrap around in the address space.

thm *valid-simple-footprint-size-td*
thm *valid-root-footprint-valid-simple-footprint*

The corresponding judgment to $d \models_t p$ for root pointers is *root-ptr-valid* d p . It implies *c-guard* p as well as *valid-root-footprint* d (*ptr-val* p) (*typ-uinfo-t* $TYPE('a)$).

thm *root-ptr-valid-def*

From byte heap to typed heaps

In UMM the core wrapper functions to provide a typed view on the raw byte level heap are:

- *h-val* to lookup typed values by typed pointers and

- *heap-update* to update the heap on the location designated by a typed pointer with a typed value.

Provided a byte level heap and a heap type description there is a function to lift the heap into a typed heap:

- *lift-t* takes a pointer guard and a pair of byte level heap and heap type description and provides a partial function from typed pointers to typed values. The most prominent instance is abbreviation *clift* = *clift*.

The resulting function is partial as it only lifts those pointers that actually point to valid addresses according to *h-t-valid*. This fact and the connection to *h-val* is reflected in $\text{lift-t } ?g \text{ (?h, ?d)} = (\lambda p. \text{if } ?d, ?g \models_t p \text{ then Some (h-val ?h p) else None})$.

thm *lift-t-if*

To lift root pointers there is

- *simple-lift*. It yields only defined values for root pointers, according to *root-ptr-valid*.

thm *simple-lift-def*

For the derived heaps we introduce the family of functions *plift*, which are defined in *wf-ptr-valid* and instantiated for each collection of open types. Like *clift* the definition is based on *lift-t*, but instead of the plain *c-guard* a custom validity-guard for the type is supplied *PTR-VALID('a)*. The relevant definitions are collected as named theorems:

thm *plift-defs*

Note that we used to introduced a separate distinct instance of *plift* and *PTR-VALID('a)* for each type. Now we have encapsulated this construction in *open-types*. From a specification of which fields of a type should be addressable we first derive an variant without phantom typing of all the concepts especially *open-types.ptr-valid-u* and then define the phantom typed version based on that. This separation works around the limitation of the Isabelle type system that are also reflected in locales. It does not allow explicit quantification on type variables. So we can only have fixed type "variables" in the locale assumptions. However, the lemmas derived within a locale can still be polymorphic. So we express the needed assumptions in the locale *open-types* in the "untyped" world and derive the "typed" polymorphic versions as lemmas within that locale.

thm *open-struct.ptr-valid-u.simps*

thm *open-types.ptr-valid-u.simps*

thm *open-struct.ptr-valid-def*

thm *open-types.ptr-valid-def*

Let us look into the definitions for the open structure *inner-C* which itself is part of the open structure *outer-C*.

lemma *plift h = lift-t (ptr-valid (hrs-htd h)) h*
by (*simp add: open-struct.plift-def*)

thm *inner-C.ptr-valid.unfold*

lemma

fixes *p::inner-C ptr*
shows *ptr-valid d p =*
(root-ptr-valid d p ∨
(∃ q::outer-C ptr. ptr-valid d q ∧ ptr-val p = &(q→["inner-C"])) ∨
(∃ i<5. ∃ q::(inner-C[5]) ptr. ptr-valid d q ∧ ptr-val p = &(q→[replicate i CHR
"1"])))
by (*rule inner-C.ptr-valid.unfold*)

lemma

fixes *p::outer-C ptr*
shows *ptr-valid d p = root-ptr-valid d p*
by (*simp add: ptr-valid*)

A pointer to an *inner-C* is valid according to *PTR-VALID('a) d p*, if it is either

- a root pointer to an *inner-C*, or
- there is a valid pointer to an enclosing pointer *q* to *outer-C*, where *p* is obtained by dereferencing field *"inner-C"*, (Note that valid *outer-C* are already root pointers). or
- there is a valid pointer to an array *q*, where *p* is obtained by indexing into the array. Indexing into arrays is modelled by unary encoded field names *replicate i CHR "1"*. (Note that valid array pointers themselves might have various possible extensions to a root pointer of an enclosing structure)

These pointer-valid predicates are derived from the specification of addressable fields in the corresponding **autocorres** command. They enumerate the possibilities to arrive at a valid pointer starting from a valid root-pointer. The specification is converted to the parameter \mathcal{T} of *open-types*. In our example program this is:

thm *T-def*

The \mathcal{T} is an association list mapping a type tag *typ-uinfo-t* of a *mem-type* to the list of addressable fields within that structure. From this specification we derive a lot of useful notions within the locale. Notably:

thm *ptr-valid* — These go up the chain to the root pointer.

From typed heaps to split heaps

Note that the functions in the previous subsection still are defined on the level of the monolithic UMM memory model. They are central building blocks to finally arrive at the split heap model. Essentially there is a split heap component in the lifted globals corresponding to *plift* for all the fundamental types.

However, instead of partial functions as provided by *wf-ptr-valid.plift* the split heap separates definedness from the actual value, by providing a pair of a total function from pointer to value and a validity predicate which indicates definedness. This design choice was introduced in the original AutoCorres.

As a prerequisite to lift the Functions into the split heap model, the new global state record *lifted-globals* is defined. For each fundamental heap this record contains a field for the split heap e.g. *heap-w32*.

These are the lookup functions provided by the record package, there are of course also the corresponding update functions, e.g.: *heap-w32-update*. Note that the update function behaves like a "map", which maps an update function on the selected heap to an update function on *lifted-globals*.

The typing information (heap type description) is preserved from the monolithic heap in component *heap-typing*.

print-record *lifted-globals*

For the derived heaps we provide a similar abstraction level by defining a derived heap lookup and a derived heap update function. These functions are a composition of the underlying functions for the fields of the structure.

thm *derived-heap-defs*

For example for *outer-C* we have.

- Lookup: *heap-outer-C*
- Pointwise update: *heap-outer-C-map*
- Validity is generically based on the heap typing for all heaps: $\lambda s. IS-VALID('a) s$

Note the difference in the signature of the derived update *heap-outer-C-map* and the fundamental *heap-w32-update*. Besides the difference in naming between 'map' and 'update', the derived update is defined pointwise instead of heap-wide. The reason is that the pointwise update is what we actually need for autocorres and it is natural to break down a pointwise update defined via a function on *f* to the update functions on the fields of *outer-C*.

thm *heap-outer-C-def*

thm *heap-outer-C-map-def*

The connection between *globals* (containing the byte level heap) and *lifted-globals* is defined via the function *lift-global-heap*. The fundamental heaps are lifted via the corresponding *wf-ptr-valid.plift* functions. Validity of pointers is maintained by literally preserving the heap typing from the monolithic heap.

thm *lift-global-heap-def*

thm *lift-t-def*

thm *plift-defs*

26.21.4 User Level

Due to the abstraction layer of derived heap types, the structure of the programs after heap lifting is analogous for closed and opened structures.

context *ts-definition-set-c1*

begin

thm *ts-def*

lemma *set-c1' p c* \equiv *do* {*guard* (λs . *IS-VALID*(*closed-C*) *s p*);

modify (*heap-closed-C-update* (λh . *h*(*p* := *c1-C-update* (λ -. *c*) (*h*

p))))

}

by (*rule ts-def*)

end

context *ts-definition-outer-fld1-upd*

begin

thm *ts-def*

end

context *ts-definition-set-element*

begin

thm *ts-def*

end

context *ts-definition-set-matrix-element*

begin

thm *ts-def*

end

Of course, when it comes to specifications and proving properties about the programs, the user has to take care about aliasing of addressable fields. *AutoCorres* generates a bunch of theorems and sets up the simpsets with some obvious rules for derived heaps, which hopefully helps in doing so. However, as a last resort one might still have to unfold the definitions and

work with the parts. Here is some examples of some theorems or collections of theorems.

thm *lifted-globals-ext-simps*

thm *ptr-valid*

thm *heap-inner-C.write-comp*

thm *heap-inner-C.write-id*

thm *heap-inner-C.write-other-commute*

thm *heap-inner-C.write-same*

thm *update-commute*

thm *read-write-same*

thm *read-write-other*

thm *write-cong*

thm *update-compose*

thm *valid-implies-c-guard*

thm *read-commutes*

thm *write-commutes*

thm *fg-cons-simps*

thm *typ-info-simps*

thm *td-names-simps*

thm *typ-name-simps*

thm *upd-lift-simps*

thm *upd-other-simps*

thm *size-align-simps*

thm *fl-Some-simps*

thm *fl-ti-simps*

thm *typ-tag-defs*

thm *size-simps*

thm *typ-name-itself*

addressable-fields, merge-addressable-fields, read-dedicated-heap, write-dedicated-heap

We distinguish three aspects when thinking and reasoning about split heaps: the monolithic heap stored in *globals* the split heap components in *lifted-globals* and the function *lift-global-heap* which transforms between both types. Autocorres does a refinement proof that a program operating on *lifted-globals* refines a program on *globals* where the heaps are related by function *lift-global-heap*.

For open structures, the fields which are addressable have two representations in *lifted-globals*:

1. the relevant value in the common heap, i.e. *heap-w32* for the *inner.fld1* pointers
2. an unused value in the field of the dedicate heap, i.e. the *fld1-C* in the *dedicated-heap-inner-C*. Having the dedicated heap allows us to keep all non-addressable fields of a structure in a single split heap component of *lifted-globals*. The values of the addressable fields are irrelevant

for the virtual heap. When reading from a virtual heap we start with the value in the dedicated heap and overwrite the addressable fields by considering the common heaps.

thm *heap-inner-C-def*

When writing to a virtual heap we have a series of updates on *lifted-globals*. We start by updating the non-addressable fields in the dedicated heap and then updating the common heaps with the new values for the addressable fields:

thm *heap-inner-C-map-def*

To express "update the non-addressable fields of the dedicated heap" in an uniform way we make use of the concept of *scenes* from the world of *lenses*. In contrast to lenses, scenes are formalised by properties of a single function $'a \Rightarrow 'a \Rightarrow 'a$ which is referred to as *merge* or 'overrider' operation. The intuition of *merge* $x y$ is that we take certain portions of a compound value x and merge (or override) them into another compound value y . Scenes are formalised in locale *is-scene*. The notion *merge* emphasises the symmetric nature of the merge operation. Like a lense, a scene 'focuses' on a portion of a compound value $'a$. The merge operation takes that portion from the first argument and the complement of that portion from the second argument. Think for example of a pair and the 'portion' being the first component. Then $merge\ x\ y = (fst\ x,\ snd\ y)$. The notion *overrider* puts more emphasis on the merge as an update function, in particular this view becomes obvious if we think of a partial application *merge* x : this is an update function which overrides the 'portion' within an value y to that portion of x . You can also think of fixing the 'portion' to those values in x . Analogously $\lambda x. merge\ x\ y$ fixes the complement of the 'portion' to those values in y . Scenes are closely related to lenses in the sense that it is straightforward to derive a scene from a lense $lense\ ?r\ ?w \implies is-scene\ (\lambda a. ?w\ (\lambda-. ?r\ a))$.

thm *is-scene-of-lense*

For HOL the decisive advantage of scenes compared to lenses is that they only have one type parameter, which fixes the type of the compound value. But still you can formally 'talk' about different 'portions' of that value in an uniform way. For example you can have a list or a set of scenes to describe their composition. This property can be used to uniformly describe "all addressable fields" of a value or "all non-addressable fields" of a value.

The relevant merging function for us is *merge-addressable-fields* which is an abbreviation $\lambda \mathcal{T}. merge-ti-list\ (map\ snd\ (open-types.addressable-fields\ \mathcal{T}\ TYPE('a)))$.

term $\langle open-types.merge-addressable-fields \rangle$

The term *merge-addressable-fields* $old\ new$ has to effects:

- The addressable fields are taken from *old*
- The non-addressable fields are taken from *new*.

To express the same semantics that we have just described for *lifted-globals* on the UMM heap in *globals* we make use of the functions *read-dedicated-heap* and *write-dedicated-heap*. These functions internally again make use of *merge-addressable-fields*. We zero out all parts of the compound value we are not actually interested in. This gives us canonical values which we can easily relate to the *lifted-globals*. For example for *read-dedicated-heap* we zero out all addressable fields and also all values which are not even *PTR-VALID('a)* (according to *plift*).

lemma (in *open-types*)

read-dedicated-heap *h p* =
merge-addressable-fields *ZERO('a::mem-type)* (*the-default ZERO('a)* (*plift h p*))
by (*simp add: read-dedicated-heap-def*)

thm *open-types.read-dedicated-heap-def*

thm *open-types.write-dedicated-heap-def*

The lifting function *lift-global-heap* directly uses *read-dedicated-heap* to construct a dedicated heap from the UMM heap:

thm *lift-global-heap-def*

To have a canonical representation we overwrite the fields in the dedicated heap with zeros, using the generic constants *addressable-fields* and *merge-addressable-fields* defined in *open-types*.

read-dedicated-heap, write-dedicated-heap combine these with *h-val*, to get the canonical heap access.

Why all the variants? *h-val* / *read-dedicated-heap*

thm *lifted-globals-ext-simps*

thm *lift-global-heap-def*

thm *open-types.read-dedicated-heap-def*

When taking a close look in the definition of *lift-global-heap* one sees that the abstraction for a split heap component from the monolithic heap is:

read-dedicated-heap (*t-hrs-' g*)

So this might seem a bit surprising at first sight. What is all the indication useful for:

1. the *read-dedicated-heap* adds an option layer to the plain *h-val* and *plift*

2. with *t-hrs-'* we get the heap representation (including bytes and types)

When we know that a pointer is valid there is a tight connection between *read-dedicated-heap* and *h-val*

thm *open-struct.ptr-valid.ptr-valid-plift-Some-hval*

thm *open-struct.read-dedicated-heap-def*

This connection directly carries over to lifting.

thm *read-commutes*

The difference between *read-dedicated-heap* and *h-val* becomes apparent when looking at partial heaps and invalid pointers. While *read-dedicated-heap* always yields *ZERO('a)* the plain *h-val* will still construct some value out of the bytes in the heap. So in a sense the indirection to *read-dedicated-heap* makes heaps equal if they agree on the valid pointers only. So we encode "heap equality only on valid pointers" in an ordinary equality on the complete lifted heap. This choice ensures compositionality for the polymorphic *ZERO('a)*, in the sense that the *ZERO('a)* of a structure means that all subcomponents are *ZERO('a)*.

Moreover *read-dedicated-heap* only reads the fields which are not addressable, and overwrites the addressable fields with *ZERO('a)*. This gives us a canonical view on the partial heap.

thm *zero-simps*

thm *open-struct.ptr-valid.plift-None*

I miss the typing in the split heap!

A peculiar property of the original split heap model of autocorres was that you lose parts of the type information that were directly available in the byte-level heap. There was no heap-type description in *lifted-globals* but only the more abstract *is-valid-<type>* fields for each type. For closed types this isn't so much of an issue as essentially all relevant type information is captured in the fact that all pointers to that type have a distinct split heap, which is separate from all other split heaps. But for shared types this is no longer true. Thus we maintain the typing information also in the split heap in the component *heap-typing*. The relation to the typing information of the original byte-level heap is encapsulated in *lift-global-heap*. For low-level operations that are embedded via *exec-concrete* you can directly connect the typing of the monolithic and the split heap. In particular you can derive *c-guard p* from that information. Or you can derive that if you have two valid pointers of the same type, where you only know that the address is different, you can still conclude that the complete pointer span of the pointers do not overlap.

But keep in mind that typing is only a "discipline". As in the split heap you can manipulate the values on the heap or split heap indepen-

dent of the typing information. When well-typedness or $PTR-VALID('a)$ is available you might be able to derive properties like distinctness of pointer spans. The distinctness of pointers spans is the actual reason that certain heap updates commute, e.g. $disjnt (ptr-span ?p) (ptr-span ?q) \implies heap-inner-C-map ?q ?g (heap-inner-C-map ?p ?f ?s) = heap-inner-C-map ?p ?f (heap-inner-C-map ?q ?g ?s)$

thm *heap-inner-C.write-other-commute*

26.21.5 Simulation Proof

The simulation of a concrete program C operating on the byte-level heap by an abstract program A operating on the split heaps is captured in predicate $L2Tcorres\ lift-global-heap\ A\ C$. The proof for an instance of C follows the syntactic structure of C by applying introduction rules and synthesizing the abstract program A .

Intuitively the core properties on which the simulation proof builds are:

- Abstract programs only operate on abstract heap values not on byte-lists. Byte-level concepts like padding fields are irrelevant for the abstract program. The abstract heaps and programs don't distinguish between two byte-level heaps if they agree on all values of non-padding fields of properly allocated and typed portions of the heap.
- Lookup and update via a pointer into a structure can be simulated by an lookup and update via the root pointer of the structure.
- Each non-addressable field of a structure is mapped into exactly one split heap. For a 'closed' structure without any addressable fields this is the heap for the type.
- For an 'open' structure an addressable field is mapped into a shared heap or multiple shared heaps in case the field is again an open structure.
- The pointer spans of two valid root pointers of different types do not overlap.
- The pointer spans of two valid root pointers to the same type might overlap only if the pointer value is the same.

The actual proof of a simulation is divided into three main steps.

- First some general theorems relating byte-level and split heaps are derived. Most prominently *typ-heap-simulation* and related predicates.
- These theorems are used to instantiate the syntax driven introduction rules collected in named theorems $\llbracket abs-expr\ ?st\ ?X\ ?f1.0\ ?f1'\ ;\ abs-expr$

$\llbracket ?st \ ?Y \ ?f2.0 \ ?f2 \rrbracket \Longrightarrow \text{abs-expr } ?st \ (\lambda s. \ ?X \ s \wedge \ ?Y \ s) \ (\lambda s. \ (?f1.0 \ s, \ ?f2.0 \ s)) \ (\lambda s. \ (?f1' \ s, \ ?f2' \ s))$
 $\text{abs-expr } ?st \ (\lambda-. \ \text{True}) \ (\lambda s. \ ?a) \ (\lambda s. \ ?a)$
 $\text{abs-expr } ?st \ ?P \ ?a' \ ?a \Longrightarrow \text{abs-guard } ?st \ (\lambda s. \ ?P \ s \wedge \ ?a' \ s) \ ?a$
 $\text{abs-guard } ?st \ (\lambda-. \ ?P) \ (\lambda-. \ ?P)$
 $\llbracket \text{abs-guard } ?st \ ?G \ ?G'; \text{abs-guard } ?st \ ?H \ ?H \rrbracket \Longrightarrow \text{abs-guard } ?st \ (\lambda s. \ ?G \ s \wedge \ ?H \ s) \ (\lambda s. \ ?G' \ s \wedge \ ?H' \ s)$
 $\llbracket \text{struct-rewrite-modifies } ?P \ ?b \ ?c; \text{abs-guard } ?st \ ?P' \ ?P; \text{abs-modifies } ?st \ ?Q \ ?a \ ?b \rrbracket \Longrightarrow \text{L2Tcorres } ?st \ (\text{L2-seq} \ (\text{L2-guard} \ (\lambda s. \ ?P' \ s \wedge \ ?Q \ s)) \ (\lambda-. \ \text{L2-modify } ?a)) \ (\text{L2-modify } ?c)$
 $\llbracket \text{struct-rewrite-expr } ?P \ ?b \ ?c; \text{abs-guard } ?st \ ?P' \ ?P; \text{abs-expr } ?st \ ?Q \ ?a \ ?b \rrbracket \Longrightarrow \text{L2Tcorres } ?st \ (\text{L2-seq} \ (\text{L2-guard} \ (\lambda s. \ ?P' \ s \wedge \ ?Q \ s)) \ (\lambda-. \ \text{L2-gets } ?a \ ?n)) \ (\text{L2-gets } ?c \ ?n)$
 $\text{L2Tcorres } ?st \ (\text{L2-gets} \ (\lambda-. \ ?a) \ ?n) \ (\text{L2-gets} \ (\lambda-. \ ?a) \ ?n)$
 $\llbracket \text{struct-rewrite-guard } ?b \ ?c; \text{abs-guard } ?st \ ?a \ ?b \rrbracket \Longrightarrow \text{L2Tcorres } ?st \ (\text{L2-guard } ?a) \ (\text{L2-guard } ?c)$
 $\llbracket \text{THIN} \ (\bigwedge x. \ \text{L2Tcorres } ?st \ (?B' \ x) \ (?B \ x)); \text{THIN} \ (\bigwedge r. \ \text{struct-rewrite-expr} \ (?G \ r) \ (?C' \ r) \ (?C \ r)); \text{THIN} \ (\bigwedge r. \ \text{abs-guard } ?st \ (?G' \ r) \ (?G \ r)); \text{THIN} \ (\bigwedge r. \ \text{abs-expr } ?st \ (?H \ r) \ (?C'' \ r) \ (?C' \ r)) \rrbracket \Longrightarrow \text{L2Tcorres } ?st \ (\text{L2-seq} \ (\text{L2-guard} \ (\lambda s. \ ?G' \ ?i \ s \wedge \ ?H \ ?i \ s)) \ (\lambda-. \ \text{L2-while } ?C'' \ (\lambda i. \ \text{L2-seq} \ (?B' \ i) \ (\lambda r. \ \text{L2-seq} \ (\text{L2-guard} \ (\lambda s. \ ?G' \ r \ s \wedge \ ?H \ r \ s)) \ (\lambda-. \ \text{L2-gets} \ (\lambda-. \ r) \ ?n)))) \ ?i \ ?n)) \ (\text{L2-while } ?C \ ?B \ ?i \ ?n)$
 $\text{abs-spec } ?st \ ?P \ ?A \ ?C \Longrightarrow \text{L2Tcorres } ?st \ (\text{L2-seq} \ (\text{L2-guard } ?P) \ (\lambda-. \ \text{L2-spec } ?A)) \ (\text{L2-spec } ?C)$
 $\text{abs-assume } ?st \ ?P \ ?A \ ?C \Longrightarrow \text{L2Tcorres } ?st \ (\text{L2-seq} \ (\text{L2-guard } ?P) \ (\lambda-. \ \text{L2-assume } ?A)) \ (\text{L2-assume } ?C)$
 $\text{abs-spec } ?st \ (\lambda-. \ \text{True}) \ \{(a, b). \ ?C\} \ \{(a, b). \ ?C\}$
 $\llbracket \text{THIN} \ (\text{L2Tcorres } ?st \ ?L \ ?L'); \text{THIN} \ (\text{L2Tcorres } ?st \ ?R \ ?R'); \text{THIN} \ (\text{struct-rewrite-expr } ?P \ ?C' \ ?C); \text{THIN} \ (\text{abs-guard } ?st \ ?P' \ ?P); \text{THIN} \ (\text{abs-expr } ?st \ ?Q \ ?C'' \ ?C') \rrbracket \Longrightarrow \text{L2Tcorres } ?st \ (\text{L2-seq} \ (\text{L2-guard} \ (\lambda s. \ ?P' \ s \wedge \ ?Q \ s)) \ (\lambda-. \ \text{L2-condition } ?C'' \ ?L \ ?R)) \ (\text{L2-condition } ?C \ ?L' \ ?R')$
 $\llbracket \text{THIN} \ (\text{L2Tcorres } ?st \ ?L' \ ?L); \text{THIN} \ (\bigwedge r. \ \text{L2Tcorres } ?st \ (?R' \ r) \ (?R \ r)) \rrbracket \Longrightarrow \text{L2Tcorres } ?st \ (\text{L2-seq} \ ?L' \ ?R') \ (\text{L2-seq} \ ?L \ ?R)$
 $\llbracket \text{struct-rewrite-guard } ?b \ ?c; \text{abs-guard } ?st \ ?a \ ?b; \bigwedge s \ s'. \ \llbracket ?c \ s; \ s' = ?st \ s \rrbracket \rrbracket \Longrightarrow \text{L2Tcorres } ?st \ ?f \ ?g \rrbracket \Longrightarrow \text{L2Tcorres } ?st \ (\text{L2-guarded } ?a \ ?f) \ (\text{L2-guarded } ?c \ ?g)$
 $\llbracket \text{THIN} \ (\text{L2Tcorres } ?st \ ?L \ ?L'); \text{THIN} \ (\bigwedge r. \ \text{L2Tcorres } ?st \ (?R \ r) \ (?R' \ r)) \rrbracket \Longrightarrow \text{L2Tcorres } ?st \ (\text{L2-catch } ?L \ ?R) \ (\text{L2-catch } ?L' \ ?R')$

$L2Tcorres \text{ ?st } ?L \text{ ?L}' \Longrightarrow L2Tcorres \text{ ?st } (L2\text{-try } ?L) (L2\text{-try } ?L')$
 $L2Tcorres \text{ ?st } (L2\text{-unknown } ?ns) (L2\text{-unknown } ?ns)$
 $L2Tcorres \text{ ?st } (L2\text{-throw } ?x \text{ ?n}) (L2\text{-throw } ?x \text{ ?n})$
 $(\bigwedge x y. L2Tcorres \text{ ?st } (?P \ x \ y) (?P' \ x \ y)) \Longrightarrow L2Tcorres \text{ ?st } (case \ ?a \ of \ (x, y) \Rightarrow ?P \ x \ y) (case \ ?a \ of \ (x, y) \Rightarrow ?P' \ x \ y)$
 $\llbracket THIN (L2Tcorres \text{ ?st } ?L \text{ ?L}'); THIN (L2Tcorres \text{ ?st } ?R \text{ ?R}') \rrbracket \Longrightarrow L2Tcorres \text{ ?st } (L2\text{-seq } ?L (\lambda-. ?R)) (L2\text{-seq } ?L' (\lambda-. ?R'))$
 $(\bigwedge a b. abs\text{-expr } \text{ ?st } (?P \ a \ b) (?A \ a \ b) (?C \ a \ b)) \Longrightarrow abs\text{-expr } \text{ ?st } (case \ ?r \ of \ (a, b) \Rightarrow ?P \ a \ b) (case \ ?r \ of \ (a, b) \Rightarrow ?A \ a \ b) (case \ ?r \ of \ (a, b) \Rightarrow ?C \ a \ b)$
 $(\bigwedge a b. abs\text{-guard } \text{ ?st } (?A \ a \ b) (?C \ a \ b)) \Longrightarrow abs\text{-guard } \text{ ?st } (case \ ?r \ of \ (a, b) \Rightarrow ?A \ a \ b) (case \ ?r \ of \ (a, b) \Rightarrow ?C \ a \ b)$
 $L2Tcorres \text{ ?st } L2\text{-fail } L2\text{-fail}$
 $abs\text{-expr } id (\lambda-. True) \ ?A \ ?A$
 $abs\text{-expr } \text{ ?st } ?P \ ?A \ ?C \Longrightarrow abs\text{-expr } \text{ ?st } ?P (\lambda s r. ?A \ s) (\lambda s r. ?C \ s)$
 $abs\text{-modifies } id (\lambda-. True) \ ?A \ ?A$
 $L2Tcorres \text{ id } ?A \ ?C \Longrightarrow L2Tcorres \text{ ?st } (exec\text{-concrete } \text{ ?st } (L2\text{-call } ?A \ ?emb \ ?ns)) (L2\text{-call } ?C \ ?emb \ ?ns)$
 $L2Tcorres \text{ id } ?A \ ?C \Longrightarrow L2Tcorres \text{ ?st } (exec\text{-concrete } \text{ ?st } (L2\text{-call-L1 } ?arg\text{-xf } ?gs \ ?ret\text{-xf } ?A)) (L2\text{-call-L1 } ?arg\text{-xf } ?gs \ ?ret\text{-xf } ?C)$
 $L2Tcorres \text{ ?st } ?A \ ?C \Longrightarrow L2Tcorres \text{ id } (exec\text{-abstract } \text{ ?st } (L2\text{-call } ?A \ ?emb \ ?ns)) (L2\text{-call } ?C \ ?emb \ ?ns)$
 $L2Tcorres \text{ ?st } ?A \ ?C \Longrightarrow L2Tcorres \text{ ?st } (L2\text{-call } ?A \ ?emb \ ?ns) (L2\text{-call } ?C \ ?emb \ ?ns)$
 $\llbracket typ\text{-heap-simulation } \text{ ?st } ?getter \ ?setter \ ?vgetter \ ?t\text{-hrs } ?t\text{-hrs-update}; abs\text{-expr } \text{ ?st } ?P \ ?x' \ ?x \rrbracket \Longrightarrow abs\text{-guard } \text{ ?st } (\lambda s. ?P \ s \wedge ?vgetter \ s \ (?x' \ s)) (\lambda s. c\text{-guard } (?x \ s))$
 $\llbracket typ\text{-heap-simulation } \text{ ?st } ?r \ ?w \ ?v \ ?t\text{-hrs } ?t\text{-hrs-update}; abs\text{-expr } \text{ ?st } ?P \ ?x' \ ?x \rrbracket \Longrightarrow abs\text{-expr } \text{ ?st } (\lambda s. ?P \ s \wedge ?v \ s \ (?x' \ s)) (\lambda s. ?r \ s \ (?x' \ s)) (\lambda s. h\text{-val } (hrs\text{-mem } (?t\text{-hrs } s)) (?x \ s))$
 $\llbracket typ\text{-heap-simulation } \text{ ?st } ?r \ ?w \ ?v \ ?t\text{-hrs } ?t\text{-hrs-update}; abs\text{-expr } \text{ ?st } ?Pb \ ?b' \ ?b; \bigwedge v. abs\text{-expr } \text{ ?st } ?Pc \ (?c' \ v) (?c \ v) \rrbracket \Longrightarrow abs\text{-modifies } \text{ ?st } (\lambda s. ?Pb \ s \wedge ?Pc \ s \wedge ?v \ s \ (?b' \ s)) (\lambda s. ?w \ (?b' \ s) (\lambda-. ?c' \ (?r \ s \ (?b' \ s)) \ s) \ s) (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (?b \ s) (?c \ (heap\text{-lift-h-val } (hrs\text{-mem } (?t\text{-hrs } s)) (?b \ s)) \ s))) \ s)$
 $struct\text{-rewrite-expr } ?P \ ?a' \ ?a \Longrightarrow struct\text{-rewrite-guard } (\lambda s. ?P \ s \wedge ?a' \ s) \ ?a$
 $struct\text{-rewrite-guard } (\lambda-. ?P) (\lambda-. ?P)$

$\llbracket \text{struct-rewrite-guard } ?b' ?b; \text{struct-rewrite-guard } ?a' ?a \rrbracket \Longrightarrow \text{struct-rewrite-guard}$
 $(\lambda s. ?a' s \wedge ?b' s) (\lambda s. ?a s \wedge ?b s)$

$(\bigwedge a b. \text{struct-rewrite-guard } (?A a b) (?C a b)) \Longrightarrow \text{struct-rewrite-guard}$
 $(\text{case } ?r \text{ of } (a, b) \Rightarrow ?A a b) (\text{case } ?r \text{ of } (a, b) \Rightarrow ?C a b)$

$\llbracket \text{valid-struct-field } ?\text{field-name } ?\text{field-getter } ?\text{field-setter } ?t\text{-hrs } ?t\text{-hrs-update};$
 $\text{struct-rewrite-expr } ?P ?p' ?p; \text{struct-rewrite-guard } ?Q (\lambda s. \text{c-guard}$
 $(?p' s)) \rrbracket \Longrightarrow \text{struct-rewrite-guard } (\lambda s. ?P s \wedge ?Q s) (\lambda s. \text{c-guard}$
 $(\text{PTR}(?'f) \ \& (?p \ s \rightarrow ?\text{field-name})))$

$\llbracket \text{valid-struct-field } ?\text{field-name } ?\text{field-getter } ?\text{field-setter } ?t\text{-hrs } ?t\text{-hrs-update};$
 $\text{struct-rewrite-expr } ?P ?p' ?p; \text{struct-rewrite-guard } ?Q (\lambda s. \text{c-guard}$
 $(?p' s)) \rrbracket \Longrightarrow \text{struct-rewrite-guard } (\lambda s. ?P s \wedge ?Q s \wedge 0 \leq ?k \wedge$
 $\text{nat } ?k < \text{CARD}(?'n)) (\lambda s. \text{c-guard } (\text{PTR-COERCE}(?'f[?'n] \rightarrow ?'f)$
 $(\text{PTR}(?'f[?'n]) \ \& (?p \ s \rightarrow ?\text{field-name})) +_p ?k))$

$\text{struct-rewrite-expr } (\lambda -. \text{True}) (\lambda -. ?a) (\lambda -. ?a)$

$\text{struct-rewrite-expr } ?P ?A ?C \Longrightarrow \text{struct-rewrite-expr } ?P (\lambda s -. ?A s)$
 $(\lambda s -. ?C s)$

$(\bigwedge a b. \text{struct-rewrite-expr } (?P a b) (?A a b) (?C a b)) \Longrightarrow \text{struct-rewrite-expr}$
 $(\text{case } ?r \text{ of } (a, b) \Rightarrow ?P a b) (\text{case } ?r \text{ of } (a, b) \Rightarrow ?A a b) (\text{case } ?r \text{ of}$
 $(a, b) \Rightarrow ?C a b)$

$\text{struct-rewrite-expr } (\lambda -. \text{True}) (\lambda s. \text{h-val } (?h s) (?p s)) (\lambda s. \text{h-val } (?h$
 $s) (?p s))$

$\text{struct-rewrite-expr } ?P ?a ?c \Longrightarrow \text{struct-rewrite-expr } ?P (\lambda s. \text{h-val } (?h$
 $s) (?a s)) (\lambda s. \text{h-val } (?h s) (?c s))$

$\llbracket \text{valid-struct-field } ?\text{field-name } ?\text{field-getter } ?\text{field-setter } ?t\text{-hrs } ?t\text{-hrs-update};$
 $\text{struct-rewrite-expr } ?P ?p' ?p; \text{struct-rewrite-expr } ?Q ?a (\lambda s. \text{h-val}$
 $(\text{hrs-mem } (?t\text{-hrs } s)) (?p' s)) \rrbracket \Longrightarrow \text{struct-rewrite-expr } (\lambda s. ?P s \wedge ?Q$
 $s) (\lambda s. ?\text{field-getter } (?a s)) (\lambda s. \text{h-val } (\text{hrs-mem } (?t\text{-hrs } s)) (\text{PTR}(?'f)$
 $\ \& (?p \ s \rightarrow ?\text{field-name})))$

$\llbracket \text{valid-struct-field } ?\text{field-name } ?\text{field-getter } ?\text{field-setter } ?t\text{-hrs } ?t\text{-hrs-update};$
 $\text{abs-expr } ?st ?P ?a ?c \rrbracket \Longrightarrow \text{abs-expr } ?st ?P (\lambda s. ?\text{field-getter } (?a s))$
 $(\lambda s. ?\text{field-getter } (?c s))$

$\llbracket \text{valid-struct-field } ?\text{field-name } ?\text{field-getter } ?\text{field-setter } ?t\text{-hrs } ?t\text{-hrs-update};$
 $\text{struct-rewrite-expr } ?P ?p' ?p; \text{struct-rewrite-expr } ?Q ?a (\lambda s. \text{h-val}$
 $(\text{hrs-mem } (?t\text{-hrs } s)) (?p' s)) \rrbracket \Longrightarrow \text{struct-rewrite-expr } (\lambda s. ?P s \wedge$
 $?Q s \wedge 0 \leq ?k \wedge \text{nat } ?k < \text{CARD}(?'n)) (\lambda s. ?\text{field-getter } (?a s).[\text{nat}$
 $?k]) (\lambda s. \text{h-val } (\text{hrs-mem } (?t\text{-hrs } s)) (\text{PTR-COERCE}(?'f[?'n] \rightarrow ?'f)$
 $(\text{PTR}(?'f[?'n]) \ \& (?p \ s \rightarrow ?\text{field-name})) +_p ?k))$

$\llbracket \text{valid-struct-field } ?\text{field-name } ?\text{field-getter } ?\text{field-setter } ?t\text{-hrs } ?t\text{-hrs-update};$
 $\text{struct-rewrite-expr } ?P ?p' ?p; \text{struct-rewrite-expr } ?Q ?a (\lambda s. \text{h-val}$
 $(\text{hrs-mem } (?t\text{-hrs } s)) (?p' s)) \rrbracket \Longrightarrow \text{struct-rewrite-expr } (\lambda s. ?P s \wedge$

$?Q\ s \wedge 0 \leq ?k \wedge \text{nat } ?k < \text{CARD}(?'n)$ ($\lambda s. ?\text{field-getter } (?a\ s).[nat\ ?k]$) ($\lambda s. h\text{-val } (hrs\text{-mem } (?t\text{-hrs } s)) (PTR(?'f) \ \& (?p\ s \rightarrow ?\text{field-name}) +_p\ ?k)$)

struct-rewrite-modifies ($\lambda\text{-}$. *True*) $?A\ ?A$

$\llbracket \text{typ-heap-simulation } ?st\ ?getter\ ?setter\ ?vgetter\ ?t\text{-hrs}\ ?t\text{-hrs-update}; \text{struct-rewrite-expr } ?P\ ?p'\ ?p; \text{struct-rewrite-expr } ?Q\ ?v'\ ?v \rrbracket \implies \text{struct-rewrite-modifies}$
 $(\lambda s. ?P\ s \wedge ?Q\ s) (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (?p'\ s) (?v'\ s))) s) (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (?p\ s) (?v\ s))) s)$

heap-lift--wrap-h-val (*heap-lift--h-val* $?s\ ?p$) (*h-val* $?s\ ?p$)

heap-lift--wrap-h-val (*h-val* $?s (PTR(?'a) \ \& (?p \rightarrow ?f))$) (*h-val* $?s (PTR(?'a) \ \& (?p \rightarrow ?f))$)

heap-lift--wrap-h-val (*h-val* $?s (PTR\text{-COERCE}(?'b \rightarrow ?'a)\ ?p +_p\ ?k)$)
(h-val $?s (PTR\text{-COERCE}(?'b \rightarrow ?'a)\ ?p +_p\ ?k)$)

$\llbracket \text{valid-struct-field } ?\text{field-name } ?\text{field-getter } ?\text{field-setter } ?t\text{-hrs } ?t\text{-hrs-update}; \text{struct-rewrite-expr } ?P\ ?p'\ ?p; \text{struct-rewrite-expr } ?Q\ ?f'\ ?f; \bigwedge s. \text{heap-lift--wrap-h-val } (?h\text{-val-p}'\ s) (h\text{-val } (hrs\text{-mem } (?t\text{-hrs } s)) (?p'\ s)); \text{struct-rewrite-modifies } ?R (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (?p''\ s) (?u\ s) (?field\text{-setter } (?f'\ s)))) s) (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (?p'\ s) (?field\text{-setter } (?f'\ s) (?h\text{-val-p}'\ s)))) s); \text{struct-rewrite-guard } ?S (\lambda s. c\text{-guard } (?p'\ s)) \rrbracket \implies \text{struct-rewrite-modifies } (\lambda s. ?P\ s \wedge ?Q\ s \wedge ?R\ s \wedge ?S\ s) (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (?p''\ s) (?u\ s) (?field\text{-setter } (?f'\ s)))) s) (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (PTR(?'f) \ \& (?p\ s \rightarrow ?\text{field-name})) (?f\ s) (h\text{-val } (hrs\text{-mem } (?t\text{-hrs } s)) (PTR(?'f) \ \& (?p\ s \rightarrow ?\text{field-name})))) s)$

$\llbracket \text{valid-struct-field } ?\text{field-name } ?\text{field-getter } ?\text{field-setter } ?t\text{-hrs } ?t\text{-hrs-update}; \text{struct-rewrite-expr } ?P\ ?p'\ ?p; \text{struct-rewrite-expr } ?Q\ ?f'\ ?f; \bigwedge s. \text{heap-lift--wrap-h-val } (?h\text{-val-p}'\ s) (h\text{-val } (hrs\text{-mem } (?t\text{-hrs } s)) (?p'\ s)); \text{struct-rewrite-modifies } ?R (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (?p''\ s) (?u\ s) (?field\text{-setter } (\lambda a. \text{Arrays.update } a (nat\ ?k) (?f'\ s) (a.[nat\ ?k])))))) s) (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (?p'\ s) (?field\text{-setter } (\lambda a. \text{Arrays.update } a (nat\ ?k) (?f'\ s) (a.[nat\ ?k]))) (?h\text{-val-p}'\ s)))) s); \text{struct-rewrite-guard } ?S (\lambda s. c\text{-guard } (?p'\ s)) \rrbracket \implies \text{struct-rewrite-modifies}$
 $(\lambda s. ?P\ s \wedge ?Q\ s \wedge ?R\ s \wedge ?S\ s \wedge 0 \leq ?k \wedge \text{nat } ?k < \text{CARD}(?'n)) (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (?p''\ s) (?u\ s) (?field\text{-setter } (\lambda a. \text{Arrays.update } a (nat\ ?k) (?f'\ s) (a.[nat\ ?k])))))) s) (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (PTR\text{-COERCE}(?'f[?'n] \rightarrow ?'f) (PTR(?'f[?'n]) \ \& (?p\ s \rightarrow ?\text{field-name}) +_p\ ?k) (?f\ s) (h\text{-val } (hrs\text{-mem } (?t\text{-hrs } s)) (PTR\text{-COERCE}(?'f[?'n] \rightarrow ?'f) (PTR(?'f[?'n]) \ \& (?p\ s \rightarrow ?\text{field-name}) +_p\ ?k)))) s)$

valid-globals-field $?st\ ?old\text{-getter}\ ?old\text{-setter}\ ?new\text{-getter}\ ?new\text{-setter}$
 $\implies \text{abs-expr } ?st (\lambda\text{-}$. *True*) $?new\text{-getter}\ ?old\text{-getter}$

$\llbracket \text{valid-globals-field } ?st \text{ } ?old\text{-getter } ?old\text{-setter } ?new\text{-getter } ?new\text{-setter};$
 $\bigwedge \text{old. abs-expr } ?st \text{ } (?P \text{ old}) \text{ } (?v \text{ old}) \text{ } (?v' \text{ old}) \rrbracket \implies \text{abs-modifies } ?st$
 $(\lambda s. \forall \text{old. } ?P \text{ old } s) (\lambda s. ?new\text{-setter } (\lambda \text{old. } ?v \text{ old } s) s) (\lambda s. ?old\text{-setter}$
 $(\lambda \text{old. } ?v' \text{ old } s) s)$

$\text{valid-globals-field } ?st \text{ } ?old\text{-getter } ?old\text{-setter } ?new\text{-getter } ?new\text{-setter}$
 $\implies \text{struct-rewrite-expr } (\lambda-. \text{True}) \text{ } ?old\text{-getter } ?old\text{-getter}$

$\llbracket \text{valid-globals-field } ?st \text{ } ?old\text{-getter } ?old\text{-setter } ?new\text{-getter } ?new\text{-setter};$
 $\bigwedge \text{old. struct-rewrite-expr } (?P \text{ old}) \text{ } (?v' \text{ old}) \text{ } (?v \text{ old}) \rrbracket \implies \text{struct-rewrite-modifies}$
 $(\lambda s. \forall \text{old. } ?P \text{ old } s) (\lambda s. ?old\text{-setter } (\lambda \text{old. } ?v' \text{ old } s) s) (\lambda s. ?old\text{-setter}$
 $(\lambda \text{old. } ?v \text{ old } s) s)$

$\text{abs-spec } ?st \text{ } (\lambda-. \text{True}) \{ (a, b). b \text{ may-not-modify-globals } a \} \{ (a, b). b$
 $\text{may-not-modify-globals } a \}$

$\llbracket \text{valid-globals-field } ?st \text{ } ?old\text{-getter } ?old\text{-setter } ?new\text{-getter } ?new\text{-setter};$
 $\text{abs-spec } ?st \text{ } (\lambda-. \text{True}) \{ (a, b). ?C \ a \ b \} \{ (a, b). ?C' \ a \ b \} \rrbracket \implies \text{abs-spec}$
 $?st \text{ } (\lambda-. \text{True}) \{ (a, b). \text{mex } (\lambda x. ?C \text{ } (?new\text{-setter } (\lambda-. x) \ a) \ b) \} \{ (a,$
 $b). \text{mex } (\lambda x. ?C' \text{ } (?old\text{-setter } (\lambda-. x) \ a) \ b) \}$

$\llbracket \text{typ-heap-simulation } ?st \text{ } ?r \text{ } ?w \text{ } ?v \text{ } ?t\text{-hrs } ?t\text{-hrs-update}; \text{abs-expr } ?st$
 $?P \text{ } ?x' \text{ } (\lambda s. \text{PTR-COERCE}(?'a[?'b] \rightarrow ?'a) \text{ } (?x \ s)) \rrbracket \implies \text{abs-guard } ?st$
 $(\lambda s. ?P \ s \wedge (\forall a \in \text{set } (\text{array-addr} \text{ } (?x' \ s) \ \text{CARD}(?'b)). ?v \ s \ a)) (\lambda s.$
 $\text{c-guard } (?x \ s))$

$\llbracket \text{typ-heap-simulation } ?st \text{ } ?r \text{ } ?w \text{ } ?v \text{ } ?t\text{-hrs } ?t\text{-hrs-update}; \text{abs-expr } ?st$
 $?Pb \text{ } ?b' \text{ } ?b; \text{abs-expr } ?st \text{ } ?Pn \text{ } ?n' \text{ } ?n; \text{abs-expr } ?st \text{ } ?Pv \text{ } ?y' \text{ } ?y \rrbracket \implies$
 $\text{abs-modifies } ?st \text{ } (\lambda s. ?Pb \ s \wedge ?Pn \ s \wedge ?Pv \ s \wedge ?n' \ s < \text{CARD}(?'b)$
 $\wedge (\forall \text{ptr} \in \text{set } (\text{array-addr} \text{ } (\text{PTR-COERCE}(?'a[?'b] \rightarrow ?'a) \text{ } (?b' \ s))$
 $\text{CARD}(?'b)). ?v \ s \ \text{ptr})) (\lambda s. ?w \text{ } (\text{PTR-COERCE}(?'a[?'b] \rightarrow ?'a) \text{ } (?b'$
 $s) +_p \text{int } (?n' \ s)) (\lambda v. ?y' \ s) \ s) (\lambda s. ?t\text{-hrs-update } (\text{hrs-mem-update}$
 $(\text{heap-update } (?b \ s) \text{ } (\text{Arrays.update } (\text{h-val } (\text{hrs-mem } (?t\text{-hrs } \ s)) \text{ } (?b \ s))$
 $(?n \ s) \text{ } (?y \ s)))) \ s)$

$\llbracket \text{typ-heap-simulation } ?st \text{ } ?r \text{ } ?w \text{ } ?v \text{ } ?t\text{-hrs } ?t\text{-hrs-update}; \text{abs-expr } ?st$
 $?Pb \text{ } ?b' \text{ } ?b; \text{abs-expr } ?st \text{ } ?Pn \text{ } ?n' \text{ } ?n \rrbracket \implies \text{abs-expr } ?st \text{ } (\lambda s. ?Pb \ s \wedge$
 $?Pn \ s \wedge ?n' \ s < \text{CARD}(?'b) \wedge ?v \ s \text{ } (\text{PTR-COERCE}(?'a[?'b] \rightarrow ?'a)$
 $(?b' \ s) +_p \text{int } (?n' \ s))) (\lambda s. ?r \ s \text{ } (\text{PTR-COERCE}(?'a[?'b] \rightarrow ?'a) \text{ } (?b'$
 $s) +_p \text{int } (?n' \ s))) (\lambda s. \text{h-val } (\text{hrs-mem } (?t\text{-hrs } \ s)) \text{ } (?b \ s).[?n \ s])$

$\text{abs-expr lift-global-heap } ?P \text{ } ?a \text{ } ?c \implies \text{L2Tcorres lift-global-heap } (\text{L2-seq}$
 $(\text{L2-guard } (\lambda t. \text{IS-VALID}(32 \ \text{word}) \ t \text{ } ?p \wedge ?P \ t)) (\lambda-. \text{L2-modify } (\lambda s.$
 $\text{heap-w32-update } (\lambda h. \text{h}(?p := ?a \ s) \ s))) (\text{globals.IO-modify-heap-paddingE}$
 $?p \text{ } ?c)$

$\text{abs-expr lift-global-heap } ?P \text{ } ?a \text{ } ?c \implies \text{L2Tcorres lift-global-heap } (\text{L2-seq}$
 $(\text{L2-guard } (\lambda t. \text{IS-VALID}(32 \ \text{word}) \ t \text{ } (\text{PTR-COERCE}(32 \ \text{signed word}$
 $\rightarrow 32 \ \text{word}) \ ?p) \wedge ?P \ t)) (\lambda-. \text{L2-modify } (\lambda s. \text{heap-w32-update } (\lambda h.$

$h(\text{PTR-COERCE}(32 \text{ signed word} \rightarrow 32 \text{ word}) ?p := \text{UCAST}(32 \text{ signed} \rightarrow 32) (?a \ s))) \text{ (globals.IO-modify-heap-paddingE } ?p \ ?c)$

$\text{abs-expr lift-global-heap } ?P \ ?a \ ?c \Longrightarrow \text{L2Tcorres lift-global-heap (L2-seq (L2-guard } (\lambda t. \text{IS-VALID}(8 \text{ word}) \ t \ ?p \wedge \ ?P \ t)) (\lambda-. \text{L2-modify } (\lambda s. \text{heap-w8-update } (\lambda h. \text{h}(?p := ?a \ s)) \ s))) \text{ (globals.IO-modify-heap-paddingE } ?p \ ?c)$

$\text{abs-expr lift-global-heap } ?P \ ?a \ ?c \Longrightarrow \text{L2Tcorres lift-global-heap (L2-seq (L2-guard } (\lambda t. \text{IS-VALID}(8 \text{ word}) \ t \ (\text{PTR-COERCE}(8 \text{ signed word} \rightarrow 8 \text{ word}) \ ?p) \wedge \ ?P \ t)) (\lambda-. \text{L2-modify } (\lambda s. \text{heap-w8-update } (\lambda h. \text{h}(\text{PTR-COERCE}(8 \text{ signed word} \rightarrow 8 \text{ word}) \ ?p := \text{UCAST}(8 \text{ signed} \rightarrow 8) (?a \ s))) \ s))) \text{ (globals.IO-modify-heap-paddingE } ?p \ ?c)$

$\text{abs-expr lift-global-heap } ?P \ ?a \ ?c \Longrightarrow \text{L2Tcorres lift-global-heap (L2-seq (L2-guard } (\lambda t. \text{IS-VALID}(32 \text{ word ptr}) \ t \ ?p \wedge \ ?P \ t)) (\lambda-. \text{L2-modify } (\lambda s. \text{heap-w32'ptr-update } (\lambda h. \text{h}(?p := ?a \ s)) \ s))) \text{ (globals.IO-modify-heap-paddingE } ?p \ ?c)$

$\text{abs-expr lift-global-heap } ?P \ ?a \ ?c \Longrightarrow \text{L2Tcorres lift-global-heap (L2-seq (L2-guard } (\lambda t. \text{IS-VALID}(\text{unit ptr}) \ t \ ?p \wedge \ ?P \ t)) (\lambda-. \text{L2-modify } (\lambda s. \text{heap-unit'ptr-update } (\lambda h. \text{h}(?p := ?a \ s)) \ s))) \text{ (globals.IO-modify-heap-paddingE } ?p \ ?c)$

$\text{abs-expr lift-global-heap } ?P \ ?a \ ?c \Longrightarrow \text{L2Tcorres lift-global-heap (L2-seq (L2-guard } (\lambda t. \text{IS-VALID}(\text{data-C}) \ t \ ?p \wedge \ ?P \ t)) (\lambda-. \text{L2-modify } (\lambda s. \text{heap-data-C-update } (\lambda h. \text{h}(?p := ?a \ s)) \ s))) \text{ (globals.IO-modify-heap-paddingE } ?p \ ?c)$

$\text{abs-expr lift-global-heap } ?P \ ?a \ ?c \Longrightarrow \text{L2Tcorres lift-global-heap (L2-seq (L2-guard } (\lambda t. \text{IS-VALID}(\text{closed-C}) \ t \ ?p \wedge \ ?P \ t)) (\lambda-. \text{L2-modify } (\lambda s. \text{heap-closed-C-update } (\lambda h. \text{h}(?p := ?a \ s)) \ s))) \text{ (globals.IO-modify-heap-paddingE } ?p \ ?c)$

$\text{abs-expr lift-global-heap } ?P \ ?a \ ?c \Longrightarrow \text{L2Tcorres lift-global-heap (L2-seq (L2-guard } (\lambda t. \text{IS-VALID}(\text{unpacked-C}) \ t \ ?p \wedge \ ?P \ t)) (\lambda-. \text{L2-modify } (\lambda s. \text{heap-unpacked-C-update } (\lambda h. \text{h}(?p := ?a \ s)) \ s))) \text{ (globals.IO-modify-heap-paddingE } ?p \ ?c)$

$\text{abs-expr lift-global-heap } ?P \ ?a \ ?c \Longrightarrow \text{L2Tcorres lift-global-heap (L2-seq (L2-guard } (\lambda t. \text{valid-array-base.valid-array } (\lambda h. \text{IS-VALID}(\text{unpacked-C}) \ h) \ t \ ?p \wedge \ ?P \ t)) (\lambda-. \text{L2-modify } (\lambda s. \text{unpacked-C.heap-array-map } ?p \ (\lambda-. \ ?a \ s) \ s))) \text{ (globals.IO-modify-heap-paddingE } ?p \ ?c)$

$\text{abs-expr lift-global-heap } ?P \ ?a \ ?c \Longrightarrow \text{L2Tcorres lift-global-heap (L2-seq (L2-guard } (\lambda t. \text{valid-array-base.valid-array } (\text{valid-array-base.valid-array } (\lambda h. \text{IS-VALID}(\text{unpacked-C}) \ h) \ h) \ t \ ?p \wedge \ ?P \ t)) (\lambda-. \text{L2-modify } (\lambda s. \text{unpacked-C.outer.heap-array-map } ?p \ (\lambda-. \ ?a \ s) \ s))) \text{ (globals.IO-modify-heap-paddingE } ?p \ ?c)$

abs-expr lift-global-heap ?P ?a ?c \implies *L2Tcorres lift-global-heap (L2-seq (L2-guard ($\lambda t.$ IS-VALID(*two-dimensional-C*) t ?p \wedge ?P t)) ($\lambda-$. L2-modify ($\lambda s.$ *heap-two-dimensional-C-map* ?p ($\lambda-$. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)*

abs-expr lift-global-heap ?P ?a ?c \implies *L2Tcorres lift-global-heap (L2-seq (L2-guard ($\lambda t.$ IS-VALID(*data-struct1-C*) t ?p \wedge ?P t)) ($\lambda-$. L2-modify ($\lambda s.$ *heap-data-struct1-C-map* ?p ($\lambda-$. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)*

abs-expr lift-global-heap ?P ?a ?c \implies *L2Tcorres lift-global-heap (L2-seq (L2-guard ($\lambda t.$ IS-VALID(*data-struct2-C*) t ?p \wedge ?P t)) ($\lambda-$. L2-modify ($\lambda s.$ *heap-data-struct2-C-map* ?p ($\lambda-$. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)*

abs-expr lift-global-heap ?P ?a ?c \implies *L2Tcorres lift-global-heap (L2-seq (L2-guard ($\lambda t.$ IS-VALID(*inner-C*) t ?p \wedge ?P t)) ($\lambda-$. L2-modify ($\lambda s.$ *heap-inner-C-map* ?p ($\lambda-$. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)*

abs-expr lift-global-heap ?P ?a ?c \implies *L2Tcorres lift-global-heap (L2-seq (L2-guard ($\lambda t.$ *valid-array-base.valid-array* ($\lambda h.$ IS-VALID(*inner-C*) h) t ?p \wedge ?P t)) ($\lambda-$. L2-modify ($\lambda s.$ *inner-C.heap-array-map* ?p ($\lambda-$. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)*

abs-expr lift-global-heap ?P ?a ?c \implies *L2Tcorres lift-global-heap (L2-seq (L2-guard ($\lambda t.$ IS-VALID(*outer-array-C*) t ?p \wedge ?P t)) ($\lambda-$. L2-modify ($\lambda s.$ *heap-outer-array-C-map* ?p ($\lambda-$. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)*

abs-expr lift-global-heap ?P ?a ?c \implies *L2Tcorres lift-global-heap (L2-seq (L2-guard ($\lambda t.$ *valid-array-base.valid-array* ($\lambda h.$ IS-VALID(*data-C*) h) t ?p \wedge ?P t)) ($\lambda-$. L2-modify ($\lambda s.$ *data-C.heap-array-map* ?p ($\lambda-$. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)*

abs-expr lift-global-heap ?P ?a ?c \implies *L2Tcorres lift-global-heap (L2-seq (L2-guard ($\lambda t.$ IS-VALID(*data-array-C*) t ?p \wedge ?P t)) ($\lambda-$. L2-modify ($\lambda s.$ *heap-data-array-C-map* ?p ($\lambda-$. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)*

abs-expr lift-global-heap ?P ?a ?c \implies *L2Tcorres lift-global-heap (L2-seq (L2-guard ($\lambda t.$ IS-VALID(*outer-C*) t ?p \wedge ?P t)) ($\lambda-$. L2-modify ($\lambda s.$ *heap-outer-C-map* ?p ($\lambda-$. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)*

abs-expr lift-global-heap ?P ?a ?c \implies *L2Tcorres lift-global-heap (L2-seq (L2-guard ($\lambda t.$ IS-VALID(*other-C*) t ?p \wedge ?P t)) ($\lambda-$. L2-modify ($\lambda s.$ *heap-other-C-map* ?p ($\lambda-$. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)*

$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda t. valid\text{-}array\text{-}base.valid\text{-}array\ (\lambda h. IS\text{-}VALID(unpacked\text{-}C)\ h)\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}. L2\text{-}modify\ (\lambda s. unpacked\text{-}C.heap\text{-}array\text{-}map\ ?p\ (\lambda\text{-}. ?a\ s)\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$

$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda t. IS\text{-}VALID(array\text{-}C)\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}. L2\text{-}modify\ (\lambda s. heap\text{-}array\text{-}C\text{-}map\ ?p\ (\lambda\text{-}. ?a\ s)\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$

$\llbracket struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \rrbracket \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}. heap\text{-}w32.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$\llbracket \bigwedge s\ p. ?f_c\ p \cdot s\ ? \rrbracket \lambda r. globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ \llbracket ; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \rrbracket \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}. heap\text{-}w32.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$\llbracket struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \rrbracket \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}. heap\text{-}w8.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$\llbracket \bigwedge s\ p. ?f_c\ p \cdot s\ ? \rrbracket \lambda r. globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ \llbracket ; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \rrbracket \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}. heap\text{-}w8.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$\llbracket struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \rrbracket \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}. heap\text{-}w32'ptr.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$\llbracket \bigwedge s\ p. ?f_c\ p \cdot s\ ? \rrbracket \lambda r. globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ \llbracket ; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P;$

$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s\ \wedge\ ?Q\ s))\ (\lambda-. heap\text{-}w32'\ ptr.\ assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.\ with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$\llbracket struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s\ \wedge\ ?Q\ s))\ (\lambda-. heap\text{-}unit'\ ptr.\ guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.\ with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm)) \rrbracket$

$\llbracket \bigwedge s\ p. ?f_c\ p \cdot s\ ? \rrbracket \lambda r. globals.\ typing.\ unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ \llbracket ; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s\ \wedge\ ?Q\ s))\ (\lambda-. heap\text{-}unit'\ ptr.\ assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.\ with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm)) \rrbracket$

$\llbracket struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s\ \wedge\ ?Q\ s))\ (\lambda-. heap\text{-}data\text{-}C.\ guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.\ with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm)) \rrbracket$

$\llbracket \bigwedge s\ p. ?f_c\ p \cdot s\ ? \rrbracket \lambda r. globals.\ typing.\ unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ \llbracket ; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s\ \wedge\ ?Q\ s))\ (\lambda-. heap\text{-}data\text{-}C.\ assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.\ with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm)) \rrbracket$

$\llbracket struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s\ \wedge\ ?Q\ s))\ (\lambda-. heap\text{-}closed\text{-}C.\ guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.\ with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm)) \rrbracket$

$\llbracket \bigwedge s\ p. ?f_c\ p \cdot s\ ? \rrbracket \lambda r. globals.\ typing.\ unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ \llbracket ; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s\ \wedge\ ?Q\ s))\ (\lambda-. heap\text{-}closed\text{-}C.\ assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr$

$?n ?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket struct-rewrite-expr ?P ?init_c' ?init_c; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres lift-global-heap (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres lift-global-heap (L2-seq (L2-guard (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. heap-unpacked-C.guard-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \bigwedge s p. ?f_c p \cdot s ? \rrbracket \lambda r. globals.typing.unchanged-typing-on \mathcal{S} s \llbracket ; struct-rewrite-expr ?P ?init_c' ?init_c; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres lift-global-heap (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres lift-global-heap (L2-seq (L2-guard (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. heap-unpacked-C.assume-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$

$abs-expr lift-global-heap ?P ?a ?c \implies L2Tcorres lift-global-heap (L2-seq (L2-guard (\lambda t. IS-VALID(unpacked-C[3][2]) t ?p \wedge ?P t)) (\lambda-. L2-modify (\lambda s. unpacked-C.outer.heap-array-map ?p (\lambda-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)$

$\llbracket struct-rewrite-expr ?P ?init_c' ?init_c; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres lift-global-heap (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres lift-global-heap (L2-seq (L2-guard (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. heap-unpacked-C'array-3'array-2.guard-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \bigwedge s p. ?f_c p \cdot s ? \rrbracket \lambda r. globals.typing.unchanged-typing-on \mathcal{S} s \llbracket ; struct-rewrite-expr ?P ?init_c' ?init_c; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres lift-global-heap (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres lift-global-heap (L2-seq (L2-guard (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. heap-unpacked-C'array-3'array-2.assume-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket struct-rewrite-expr ?P ?init_c' ?init_c; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres lift-global-heap (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres lift-global-heap (L2-seq (L2-guard (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. heap-two-dimensional-C.guard-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \bigwedge s p. ?f_c p \cdot s ? \rrbracket \lambda r. globals.typing.unchanged-typing-on \mathcal{S} s \llbracket ; struct-rewrite-expr ?P ?init_c' ?init_c; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres lift-global-heap (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres lift-global-heap (L2-seq (L2-guard (\lambda s.$

$?P' s \wedge ?Q s)) (\lambda-. \text{heap-two-dimensional-C.assume-with-fresh-stack-ptr } ?n ?init_a (L2-VARS ?f_a ?nm))) (\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \text{struct-rewrite-expr } ?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P; \text{abs-expr lift-global-heap } ?Q ?init_a ?init_c'; \text{THIN } (\bigwedge p. L2Tcorres \text{ lift-global-heap } (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres \text{ lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. \text{heap-data-struct1-C.guard-with-fresh-stack-ptr } ?n ?init_a (L2-VARS ?f_a ?nm))) (\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \bigwedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. \text{globals.typing.unchanged-typing-on } \mathcal{S} s \rrbracket; \text{struct-rewrite-expr } ?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P; \text{abs-expr lift-global-heap } ?Q ?init_a ?init_c'; \text{THIN } (\bigwedge p. L2Tcorres \text{ lift-global-heap } (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres \text{ lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. \text{heap-data-struct1-C.assume-with-fresh-stack-ptr } ?n ?init_a (L2-VARS ?f_a ?nm))) (\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \text{struct-rewrite-expr } ?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P; \text{abs-expr lift-global-heap } ?Q ?init_a ?init_c'; \text{THIN } (\bigwedge p. L2Tcorres \text{ lift-global-heap } (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres \text{ lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. \text{heap-data-struct2-C.guard-with-fresh-stack-ptr } ?n ?init_a (L2-VARS ?f_a ?nm))) (\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \bigwedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. \text{globals.typing.unchanged-typing-on } \mathcal{S} s \rrbracket; \text{struct-rewrite-expr } ?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P; \text{abs-expr lift-global-heap } ?Q ?init_a ?init_c'; \text{THIN } (\bigwedge p. L2Tcorres \text{ lift-global-heap } (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres \text{ lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. \text{heap-data-struct2-C.assume-with-fresh-stack-ptr } ?n ?init_a (L2-VARS ?f_a ?nm))) (\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \text{struct-rewrite-expr } ?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P; \text{abs-expr lift-global-heap } ?Q ?init_a ?init_c'; \text{THIN } (\bigwedge p. L2Tcorres \text{ lift-global-heap } (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres \text{ lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. \text{heap-inner-C.guard-with-fresh-stack-ptr } ?n ?init_a (L2-VARS ?f_a ?nm))) (\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \bigwedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. \text{globals.typing.unchanged-typing-on } \mathcal{S} s \rrbracket; \text{struct-rewrite-expr } ?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P; \text{abs-expr lift-global-heap } ?Q ?init_a ?init_c'; \text{THIN } (\bigwedge p. L2Tcorres \text{ lift-global-heap } (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres \text{ lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. \text{heap-inner-C.assume-with-fresh-stack-ptr } ?n ?init_a (L2-VARS ?f_a ?nm))) (\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c ?nm))$

$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \implies L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda t. IS\text{-}VALID(inner\text{-}C[5])\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s. inner\text{-}C.heap\text{-}array\text{-}map\ ?p\ (\lambda\text{-}.\ ?a\ s)\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$

$\llbracket struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \rrbracket \implies L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}inner\text{-}C'\text{-}array\text{-}5.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$\llbracket \bigwedge s\ p. ?f_c\ p \cdot s\ ?\ \lambda r. globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ \rrbracket; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \rrbracket \implies L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}inner\text{-}C'\text{-}array\text{-}5.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$\llbracket struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \rrbracket \implies L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}outer\text{-}array\text{-}C.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$\llbracket \bigwedge s\ p. ?f_c\ p \cdot s\ ?\ \lambda r. globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ \rrbracket; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \rrbracket \implies L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}outer\text{-}array\text{-}C.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \implies L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda t. IS\text{-}VALID(data\text{-}C[10])\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s. data\text{-}C.heap\text{-}array\text{-}map\ ?p\ (\lambda\text{-}.\ ?a\ s)\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$

$\llbracket struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p. L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p)) \rrbracket \implies L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s. ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}data\text{-}C'\text{-}array\text{-}10.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$\llbracket \bigwedge s\ p. ?f_c\ p \cdot s\ ?\ \lambda r. globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ \rrbracket; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P;$

$?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$
 $abs-expr \text{ lift-global-heap } ?P ?a ?c \implies L2Tcorres \text{ lift-global-heap } (L2-seq (L2-guard (\lambda t. IS-VALID(unpacked-C[2]) t ?p \wedge ?P t)) (\lambda-. L2-modify (\lambda s. unpacked-C.heap-array-map ?p (\lambda-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)$

$\llbracket struct-rewrite-expr ?P ?init_c' ?init_c; abs-guard \text{ lift-global-heap } ?P' ?P; abs-expr \text{ lift-global-heap } ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres \text{ lift-global-heap } (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres \text{ lift-global-heap } (L2-seq (L2-guard (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. heap-unpacked-C'array-2.guard-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \bigwedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. globals.typing.unchanged-typing-on \mathcal{S} s \rrbracket;$
 $struct-rewrite-expr ?P ?init_c' ?init_c; abs-guard \text{ lift-global-heap } ?P' ?P;$
 $abs-expr \text{ lift-global-heap } ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres \text{ lift-global-heap } (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres \text{ lift-global-heap } (L2-seq (L2-guard (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. heap-unpacked-C'array-2.assume-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket struct-rewrite-expr ?P ?init_c' ?init_c; abs-guard \text{ lift-global-heap } ?P' ?P; abs-expr \text{ lift-global-heap } ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres \text{ lift-global-heap } (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres \text{ lift-global-heap } (L2-seq (L2-guard (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. heap-array-C.guard-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \bigwedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. globals.typing.unchanged-typing-on \mathcal{S} s \rrbracket;$
 $struct-rewrite-expr ?P ?init_c' ?init_c; abs-guard \text{ lift-global-heap } ?P' ?P;$
 $abs-expr \text{ lift-global-heap } ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres \text{ lift-global-heap } (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres \text{ lift-global-heap } (L2-seq (L2-guard (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. heap-array-C.assume-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm)).$

- The derived introduction rules are recursively applied to the program. The first step is the one that was significantly extended to support addressable fields in open structures. The second and third step remained almost unchanged.

thm *heap-abs*

When thoroughly inspecting the rules $\llbracket abs-expr ?st ?X ?f1.0 ?f1'; abs-expr ?st ?Y ?f2.0 ?f2 \rrbracket \implies abs-expr ?st (\lambda s. ?X s \wedge ?Y s) (\lambda s. (?f1.0 s, ?f2.0 s)) (\lambda s. (?f1' s, ?f2' s))$

$abs\text{-expr } ?st (\lambda\text{-. True}) (\lambda s. ?a) (\lambda s. ?a)$
 $abs\text{-expr } ?st ?P ?a' ?a \implies abs\text{-guard } ?st (\lambda s. ?P s \wedge ?a' s) ?a$
 $abs\text{-guard } ?st (\lambda\text{-. ?P}) (\lambda\text{-. ?P})$
 $\llbracket abs\text{-guard } ?st ?G ?G'; abs\text{-guard } ?st ?H ?H' \rrbracket \implies abs\text{-guard } ?st (\lambda s. ?G s \wedge ?H s) (\lambda s. ?G' s \wedge ?H' s)$
 $\llbracket struct\text{-rewrite-modifies } ?P ?b ?c; abs\text{-guard } ?st ?P' ?P; abs\text{-modifies } ?st ?Q ?a ?b \rrbracket \implies L2Tcorres ?st (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-. } L2\text{-modify } ?a)) (L2\text{-modify } ?c)$
 $\llbracket struct\text{-rewrite-expr } ?P ?b ?c; abs\text{-guard } ?st ?P' ?P; abs\text{-expr } ?st ?Q ?a ?b \rrbracket \implies L2Tcorres ?st (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-. } L2\text{-gets } ?a ?n)) (L2\text{-gets } ?c ?n)$
 $L2Tcorres ?st (L2\text{-gets } (\lambda\text{-. } ?a) ?n) (L2\text{-gets } (\lambda\text{-. } ?a) ?n)$
 $\llbracket struct\text{-rewrite-guard } ?b ?c; abs\text{-guard } ?st ?a ?b \rrbracket \implies L2Tcorres ?st (L2\text{-guard } ?a) (L2\text{-guard } ?c)$
 $\llbracket THIN (\bigwedge x. L2Tcorres ?st (?B' x) (?B x)); THIN (\bigwedge r. struct\text{-rewrite-expr } (?G r) (?C' r) (?C r)); THIN (\bigwedge r. abs\text{-guard } ?st (?G' r) (?G r)); THIN (\bigwedge r. abs\text{-expr } ?st (?H r) (?C'' r) (?C' r)) \rrbracket \implies L2Tcorres ?st (L2\text{-seq } (L2\text{-guard } (\lambda s. ?G' ?i s \wedge ?H ?i s)) (\lambda\text{-. } L2\text{-while } ?C'' (\lambda i. L2\text{-seq } (?B' i) (\lambda r. L2\text{-seq } (L2\text{-guard } (\lambda s. ?G' r s \wedge ?H r s)) (\lambda\text{-. } L2\text{-gets } (\lambda\text{-. } r) ?n)))) ?i ?n)) (L2\text{-while } ?C ?B ?i ?n)$
 $abs\text{-spec } ?st ?P ?A ?C \implies L2Tcorres ?st (L2\text{-seq } (L2\text{-guard } ?P) (\lambda\text{-. } L2\text{-spec } ?A)) (L2\text{-spec } ?C)$
 $abs\text{-assume } ?st ?P ?A ?C \implies L2Tcorres ?st (L2\text{-seq } (L2\text{-guard } ?P) (\lambda\text{-. } L2\text{-assume } ?A)) (L2\text{-assume } ?C)$
 $abs\text{-spec } ?st (\lambda\text{-. True}) \{(a, b). ?C\} \{(a, b). ?C\}$
 $\llbracket THIN (L2Tcorres ?st ?L ?L'); THIN (L2Tcorres ?st ?R ?R'); THIN (struct\text{-rewrite-expr } ?P ?C' ?C); THIN (abs\text{-guard } ?st ?P' ?P); THIN (abs\text{-expr } ?st ?Q ?C'' ?C') \rrbracket \implies L2Tcorres ?st (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-. } L2\text{-condition } ?C'' ?L ?R)) (L2\text{-condition } ?C ?L' ?R')$
 $\llbracket THIN (L2Tcorres ?st ?L' ?L); THIN (\bigwedge r. L2Tcorres ?st (?R' r) (?R r)) \rrbracket \implies L2Tcorres ?st (L2\text{-seq } ?L' ?R') (L2\text{-seq } ?L ?R)$
 $\llbracket struct\text{-rewrite-guard } ?b ?c; abs\text{-guard } ?st ?a ?b; \bigwedge s s'. \llbracket ?c s; s' = ?st s \rrbracket \implies L2Tcorres ?st ?f ?g \rrbracket \implies L2Tcorres ?st (L2\text{-guarded } ?a ?f) (L2\text{-guarded } ?c ?g)$
 $\llbracket THIN (L2Tcorres ?st ?L ?L'); THIN (\bigwedge r. L2Tcorres ?st (?R r) (?R' r)) \rrbracket \implies L2Tcorres ?st (L2\text{-catch } ?L ?R) (L2\text{-catch } ?L' ?R')$
 $L2Tcorres ?st ?L ?L' \implies L2Tcorres ?st (L2\text{-try } ?L) (L2\text{-try } ?L')$
 $L2Tcorres ?st (L2\text{-unknown } ?ns) (L2\text{-unknown } ?ns)$
 $L2Tcorres ?st (L2\text{-throw } ?x ?n) (L2\text{-throw } ?x ?n)$
 $(\bigwedge x y. L2Tcorres ?st (?P x y) (?P' x y)) \implies L2Tcorres ?st (case ?a of (x, y) \implies ?P x y) (case ?a of (x, y) \implies ?P' x y)$
 $\llbracket THIN (L2Tcorres ?st ?L ?L'); THIN (L2Tcorres ?st ?R ?R') \rrbracket \implies L2Tcorres ?st (L2\text{-seq } ?L (\lambda\text{-. } ?R)) (L2\text{-seq } ?L' (\lambda\text{-. } ?R'))$
 $(\bigwedge a b. abs\text{-expr } ?st (?P a b) (?A a b) (?C a b)) \implies abs\text{-expr } ?st (case$

$?r \text{ of } (a, b) \Rightarrow ?P a b$ (case $?r \text{ of } (a, b) \Rightarrow ?A a b$) (case $?r \text{ of } (a, b) \Rightarrow ?C a b$)
 $(\bigwedge a b. \text{abs-guard } ?st (?A a b) (?C a b)) \Longrightarrow \text{abs-guard } ?st (\text{case } ?r \text{ of } (a, b) \Rightarrow ?A a b) (\text{case } ?r \text{ of } (a, b) \Rightarrow ?C a b)$
 $L2Tcorres ?st L2-fail L2-fail$
 $\text{abs-expr id } (\lambda-. \text{True}) ?A ?A$
 $\text{abs-expr } ?st ?P ?A ?C \Longrightarrow \text{abs-expr } ?st ?P (\lambda s r. ?A s) (\lambda s r. ?C s)$
 $\text{abs-modifies id } (\lambda-. \text{True}) ?A ?A$
 $L2Tcorres id ?A ?C \Longrightarrow L2Tcorres ?st (\text{exec-concrete } ?st (L2-call ?A ?emb ?ns)) (L2-call ?C ?emb ?ns)$
 $L2Tcorres id ?A ?C \Longrightarrow L2Tcorres ?st (\text{exec-concrete } ?st (L2-call-L1 ?arg-xf ?gs ?ret-xf ?A)) (L2-call-L1 ?arg-xf ?gs ?ret-xf ?C)$
 $L2Tcorres ?st ?A ?C \Longrightarrow L2Tcorres id (\text{exec-abstract } ?st (L2-call ?A ?emb ?ns)) (L2-call ?C ?emb ?ns)$
 $L2Tcorres ?st ?A ?C \Longrightarrow L2Tcorres ?st (L2-call ?A ?emb ?ns) (L2-call ?C ?emb ?ns)$
 $\llbracket \text{typ-heap-simulation } ?st ?getter ?setter ?vgetter ?t-hrs ?t-hrs-update; \text{abs-expr } ?st ?P ?x' ?x \rrbracket \Longrightarrow \text{abs-guard } ?st (\lambda s. ?P s \wedge ?vgetter s (?x' s)) (\lambda s. \text{c-guard } (?x s))$
 $\llbracket \text{typ-heap-simulation } ?st ?r ?w ?v ?t-hrs ?t-hrs-update; \text{abs-expr } ?st ?P ?x' ?x \rrbracket \Longrightarrow \text{abs-expr } ?st (\lambda s. ?P s \wedge ?v s (?x' s)) (\lambda s. ?r s (?x' s)) (\lambda s. \text{h-val } (\text{hrs-mem } (?t-hrs s)) (?x s))$
 $\llbracket \text{typ-heap-simulation } ?st ?r ?w ?v ?t-hrs ?t-hrs-update; \text{abs-expr } ?st ?Pb ?b' ?b; \bigwedge v. \text{abs-expr } ?st ?Pc (?c' v) (?c v) \rrbracket \Longrightarrow \text{abs-modifies } ?st (\lambda s. ?Pb s \wedge ?Pc s \wedge ?v s (?b' s)) (\lambda s. ?w (?b' s) (\lambda-. ?c' (?r s (?b' s)) s) s) (\lambda s. ?t-hrs-update (\text{hrs-mem-update } (\text{heap-update } (?b s) (?c (\text{heap-lift--h-val } (\text{hrs-mem } (?t-hrs s)) (?b s)) s))) s)$
 $\text{struct-rewrite-expr } ?P ?a' ?a \Longrightarrow \text{struct-rewrite-guard } (\lambda s. ?P s \wedge ?a' s) ?a$
 $\text{struct-rewrite-guard } (\lambda-. ?P) (\lambda-. ?P)$
 $\llbracket \text{struct-rewrite-guard } ?b' ?b; \text{struct-rewrite-guard } ?a' ?a \rrbracket \Longrightarrow \text{struct-rewrite-guard } (\lambda s. ?a' s \wedge ?b' s) (\lambda s. ?a s \wedge ?b s)$
 $(\bigwedge a b. \text{struct-rewrite-guard } (?A a b) (?C a b)) \Longrightarrow \text{struct-rewrite-guard } (\text{case } ?r \text{ of } (a, b) \Rightarrow ?A a b) (\text{case } ?r \text{ of } (a, b) \Rightarrow ?C a b)$
 $\llbracket \text{valid-struct-field } ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update; \text{struct-rewrite-expr } ?P ?p' ?p; \text{struct-rewrite-guard } ?Q (\lambda s. \text{c-guard } (?p' s)) \rrbracket \Longrightarrow \text{struct-rewrite-guard } (\lambda s. ?P s \wedge ?Q s) (\lambda s. \text{c-guard } (\text{PTR} (?f) \& (?p s \rightarrow ?field-name)))$
 $\llbracket \text{valid-struct-field } ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update; \text{struct-rewrite-expr } ?P ?p' ?p; \text{struct-rewrite-guard } ?Q (\lambda s. \text{c-guard } (?p' s)) \rrbracket \Longrightarrow \text{struct-rewrite-guard } (\lambda s. ?P s \wedge ?Q s \wedge 0 \leq ?k \wedge \text{nat } ?k < \text{CARD} (?n)) (\lambda s. \text{c-guard } (\text{PTR-COERCE} (?f[?n] \rightarrow ?f) (\text{PTR} (?f[?n]) \& (?p s \rightarrow ?field-name)) +_p ?k))$
 $\text{struct-rewrite-expr } (\lambda-. \text{True}) (\lambda-. ?a) (\lambda-. ?a)$

$struct\text{-rewrite}\text{-expr } ?P \text{ ?}A \text{ ?}C \implies struct\text{-rewrite}\text{-expr } ?P (\lambda s \text{ . } ?A s)$
 $(\lambda s \text{ . } ?C s)$
 $(\bigwedge a \text{ b. } struct\text{-rewrite}\text{-expr } (?P a \text{ b}) (?A a \text{ b}) (?C a \text{ b})) \implies struct\text{-rewrite}\text{-expr}$
 $(case \text{ ?}r \text{ of } (a, b) \implies ?P a \text{ b}) (case \text{ ?}r \text{ of } (a, b) \implies ?A a \text{ b}) (case \text{ ?}r \text{ of } (a,$
 $b) \implies ?C a \text{ b})$
 $struct\text{-rewrite}\text{-expr } (\lambda \text{ . } True) (\lambda s. h\text{-val } (?h s) (?p s)) (\lambda s. h\text{-val } (?h s)$
 $(?p s))$
 $struct\text{-rewrite}\text{-expr } ?P \text{ ?}a \text{ ?}c \implies struct\text{-rewrite}\text{-expr } ?P (\lambda s. h\text{-val } (?h$
 $s) (?a s)) (\lambda s. h\text{-val } (?h s) (?c s))$
 $\llbracket valid\text{-struct}\text{-field } ?field\text{-name } ?field\text{-getter } ?field\text{-setter } ?t\text{-hrs } ?t\text{-hrs}\text{-update};$
 $struct\text{-rewrite}\text{-expr } ?P \text{ ?}p' \text{ ?}p; struct\text{-rewrite}\text{-expr } ?Q \text{ ?}a (\lambda s. h\text{-val } (hrs\text{-mem}$
 $(?t\text{-hrs } s)) (?p' s)) \rrbracket \implies struct\text{-rewrite}\text{-expr } (\lambda s. ?P s \wedge ?Q s) (\lambda s. ?field\text{-getter}$
 $(?a s)) (\lambda s. h\text{-val } (hrs\text{-mem } (?t\text{-hrs } s)) (PTR(?f) \& (?p s \rightarrow ?field\text{-name})))$
 $\llbracket valid\text{-struct}\text{-field } ?field\text{-name } ?field\text{-getter } ?field\text{-setter } ?t\text{-hrs } ?t\text{-hrs}\text{-update};$
 $abs\text{-expr } ?st \text{ ?}P \text{ ?}a \text{ ?}c \rrbracket \implies abs\text{-expr } ?st \text{ ?}P (\lambda s. ?field\text{-getter } (?a s)) (\lambda s.$
 $?field\text{-getter } (?c s))$
 $\llbracket valid\text{-struct}\text{-field } ?field\text{-name } ?field\text{-getter } ?field\text{-setter } ?t\text{-hrs } ?t\text{-hrs}\text{-update};$
 $struct\text{-rewrite}\text{-expr } ?P \text{ ?}p' \text{ ?}p; struct\text{-rewrite}\text{-expr } ?Q \text{ ?}a (\lambda s. h\text{-val } (hrs\text{-mem}$
 $(?t\text{-hrs } s)) (?p' s)) \rrbracket \implies struct\text{-rewrite}\text{-expr } (\lambda s. ?P s \wedge ?Q s \wedge 0 \leq ?k \wedge$
 $nat \text{ ?}k < CARD(?n)) (\lambda s. ?field\text{-getter } (?a s).[nat \text{ ?}k]) (\lambda s. h\text{-val } (hrs\text{-mem}$
 $(?t\text{-hrs } s)) (PTR\text{-COERCE}(?f[?n] \rightarrow ?f) (PTR(?f[?n]) \& (?p s \rightarrow ?field\text{-name}))$
 $+_p \text{ ?}k))$
 $\llbracket valid\text{-struct}\text{-field } ?field\text{-name } ?field\text{-getter } ?field\text{-setter } ?t\text{-hrs } ?t\text{-hrs}\text{-update};$
 $struct\text{-rewrite}\text{-expr } ?P \text{ ?}p' \text{ ?}p; struct\text{-rewrite}\text{-expr } ?Q \text{ ?}a (\lambda s. h\text{-val } (hrs\text{-mem}$
 $(?t\text{-hrs } s)) (?p' s)) \rrbracket \implies struct\text{-rewrite}\text{-expr } (\lambda s. ?P s \wedge ?Q s \wedge 0 \leq ?k \wedge$
 $nat \text{ ?}k < CARD(?n)) (\lambda s. ?field\text{-getter } (?a s).[nat \text{ ?}k]) (\lambda s. h\text{-val } (hrs\text{-mem}$
 $(?t\text{-hrs } s)) (PTR(?f) \& (?p s \rightarrow ?field\text{-name}) +_p \text{ ?}k))$
 $struct\text{-rewrite}\text{-modifies } (\lambda \text{ . } True) \text{ ?}A \text{ ?}A$
 $\llbracket typ\text{-heap}\text{-simulation } ?st \text{ ?}getter \text{ ?}setter \text{ ?}vgetter \text{ ?}t\text{-hrs } ?t\text{-hrs}\text{-update};$
 $struct\text{-rewrite}\text{-expr } ?P \text{ ?}p' \text{ ?}p; struct\text{-rewrite}\text{-expr } ?Q \text{ ?}v' \text{ ?}v \rrbracket \implies struct\text{-rewrite}\text{-modifies}$
 $(\lambda s. ?P s \wedge ?Q s) (\lambda s. ?t\text{-hrs}\text{-update } (hrs\text{-mem}\text{-update } (heap\text{-update } (?p' s)$
 $(?v' s))) s) (\lambda s. ?t\text{-hrs}\text{-update } (hrs\text{-mem}\text{-update } (heap\text{-update } (?p s) (?v s)))$
 $s)$
 $heap\text{-lift}\text{-wrap}\text{-h}\text{-val } (heap\text{-lift}\text{-h}\text{-val } ?s \text{ ?}p) (h\text{-val } ?s \text{ ?}p)$
 $heap\text{-lift}\text{-wrap}\text{-h}\text{-val } (h\text{-val } ?s (PTR(?a) \& (?p \rightarrow ?f))) (h\text{-val } ?s (PTR(?a)$
 $\& (?p \rightarrow ?f)))$
 $heap\text{-lift}\text{-wrap}\text{-h}\text{-val } (h\text{-val } ?s (PTR\text{-COERCE}(?b \rightarrow ?a) ?p +_p \text{ ?}k))$
 $(h\text{-val } ?s (PTR\text{-COERCE}(?b \rightarrow ?a) ?p +_p \text{ ?}k))$
 $\llbracket valid\text{-struct}\text{-field } ?field\text{-name } ?field\text{-getter } ?field\text{-setter } ?t\text{-hrs } ?t\text{-hrs}\text{-update};$
 $struct\text{-rewrite}\text{-expr } ?P \text{ ?}p' \text{ ?}p; struct\text{-rewrite}\text{-expr } ?Q \text{ ?}f' \text{ ?}f; \bigwedge s. heap\text{-lift}\text{-wrap}\text{-h}\text{-val}$
 $(?h\text{-val}\text{-}p' s) (h\text{-val } (hrs\text{-mem } (?t\text{-hrs } s)) (?p' s)); struct\text{-rewrite}\text{-modifies } ?R$
 $(\lambda s. ?t\text{-hrs}\text{-update } (hrs\text{-mem}\text{-update } (heap\text{-update } (?p'' s) (?u s) (?field\text{-setter}$
 $(?f' s)))) s) (\lambda s. ?t\text{-hrs}\text{-update } (hrs\text{-mem}\text{-update } (heap\text{-update } (?p' s) (?field\text{-setter}$
 $(?f' s) (?h\text{-val}\text{-}p' s)))) s); struct\text{-rewrite}\text{-guard } ?S (\lambda s. c\text{-guard } (?p' s)) \rrbracket \implies$

$struct\text{-rewrite-modifies } (\lambda s. ?P s \wedge ?Q s \wedge ?R s \wedge ?S s) (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (?p'' s) (?u s (?field\text{-setter } (?f' s)))))) s) (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (PTR(?f) \& (?p s \rightarrow ?field\text{-name})))) s) (?f s (h\text{-val } (hrs\text{-mem } (?t\text{-hrs } s)) (PTR(?f) \& (?p s \rightarrow ?field\text{-name})))) s)$
 $\llbracket valid\text{-struct-field } ?field\text{-name } ?field\text{-getter } ?field\text{-setter } ?t\text{-hrs } ?t\text{-hrs-update}; struct\text{-rewrite-expr } ?P ?p' ?p; struct\text{-rewrite-expr } ?Q ?f' ?f; \bigwedge s. heap\text{-lift--wrap-h-val } (?h\text{-val-p' } s) (h\text{-val } (hrs\text{-mem } (?t\text{-hrs } s)) (?p' s)); struct\text{-rewrite-modifies } ?R (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (?p'' s) (?u s (?field\text{-setter } (\lambda a. Arrays.update a (nat ?k) (?f' s (a.[nat ?k])))))))) s) (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (?p' s) (?field\text{-setter } (\lambda a. Arrays.update a (nat ?k) (?f' s (a.[nat ?k])))) (?h\text{-val-p' } s)))) s); struct\text{-rewrite-guard } ?S (\lambda s. c\text{-guard } (?p' s)) \rrbracket \implies struct\text{-rewrite-modifies } (\lambda s. ?P s \wedge ?Q s \wedge ?R s \wedge ?S s \wedge 0 \leq ?k \wedge nat ?k < CARD(?n)) (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (?p'' s) (?u s (?field\text{-setter } (\lambda a. Arrays.update a (nat ?k) (?f' s (a.[nat ?k])))))))) s) (\lambda s. ?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (PTR\text{-COERCE} (?f[?n] \rightarrow ?f) (PTR(?f[?n]) \& (?p s \rightarrow ?field\text{-name})) +_p ?k) (?f s (h\text{-val } (hrs\text{-mem } (?t\text{-hrs } s)) (PTR\text{-COERCE} (?f[?n] \rightarrow ?f) (PTR(?f[?n]) \& (?p s \rightarrow ?field\text{-name})) +_p ?k)))))) s)$
 $valid\text{-globals-field } ?st ?old\text{-getter } ?old\text{-setter } ?new\text{-getter } ?new\text{-setter} \implies abs\text{-expr } ?st (\lambda -. True) ?new\text{-getter } ?old\text{-getter}$
 $\llbracket valid\text{-globals-field } ?st ?old\text{-getter } ?old\text{-setter } ?new\text{-getter } ?new\text{-setter}; \bigwedge old. abs\text{-expr } ?st (?P old) (?v old) (?v' old) \rrbracket \implies abs\text{-modifies } ?st (\lambda s. \forall old. ?P old s) (\lambda s. ?new\text{-setter } (\lambda old. ?v old s) s) (\lambda s. ?old\text{-setter } (\lambda old. ?v' old s) s)$
 $valid\text{-globals-field } ?st ?old\text{-getter } ?old\text{-setter } ?new\text{-getter } ?new\text{-setter} \implies struct\text{-rewrite-expr } (\lambda -. True) ?old\text{-getter } ?old\text{-getter}$
 $\llbracket valid\text{-globals-field } ?st ?old\text{-getter } ?old\text{-setter } ?new\text{-getter } ?new\text{-setter}; \bigwedge old. struct\text{-rewrite-expr } (?P old) (?v' old) (?v old) \rrbracket \implies struct\text{-rewrite-modifies } (\lambda s. \forall old. ?P old s) (\lambda s. ?old\text{-setter } (\lambda old. ?v' old s) s) (\lambda s. ?old\text{-setter } (\lambda old. ?v old s) s)$
 $abs\text{-spec } ?st (\lambda -. True) \{(a, b). b \text{ may-not-modify-globals } a\} \{(a, b). b \text{ may-not-modify-globals } a\}$
 $\llbracket valid\text{-globals-field } ?st ?old\text{-getter } ?old\text{-setter } ?new\text{-getter } ?new\text{-setter}; abs\text{-spec } ?st (\lambda -. True) \{(a, b). ?C a b\} \{(a, b). ?C' a b\} \rrbracket \implies abs\text{-spec } ?st (\lambda -. True) \{(a, b). mex (\lambda x. ?C (?new\text{-setter } (\lambda -. x) a) b)\} \{(a, b). mex (\lambda x. ?C' (?old\text{-setter } (\lambda -. x) a) b)\}$
 $\llbracket typ\text{-heap-simulation } ?st ?r ?w ?v ?t\text{-hrs } ?t\text{-hrs-update}; abs\text{-expr } ?st ?P ?x' (\lambda s. PTR\text{-COERCE} (?a[?b] \rightarrow ?a) (?x s)) \rrbracket \implies abs\text{-guard } ?st (\lambda s. ?P s \wedge (\forall a \in set (array\text{-addrs } (?x' s) CARD(?b)). ?v s a)) (\lambda s. c\text{-guard } (?x s))$
 $\llbracket typ\text{-heap-simulation } ?st ?r ?w ?v ?t\text{-hrs } ?t\text{-hrs-update}; abs\text{-expr } ?st ?Pb ?b' ?b; abs\text{-expr } ?st ?Pn ?n' ?n; abs\text{-expr } ?st ?Pv ?y' ?y \rrbracket \implies abs\text{-modifies } ?st (\lambda s. ?Pb s \wedge ?Pn s \wedge ?Pv s \wedge ?n' s < CARD(?b) \wedge (\forall ptr \in set (array\text{-addrs } (PTR\text{-COERCE} (?a[?b] \rightarrow ?a) (?b' s)) CARD(?b)). ?v s ptr)) (\lambda s. ?w (PTR\text{-COERCE} (?a[?b] \rightarrow ?a) (?b' s) +_p int (?n' s)) (\lambda v. ?y' s) s) (\lambda s.$

$?t\text{-hrs-update } (hrs\text{-mem-update } (heap\text{-update } (?b \ s) \ (Arrays.update \ (h\text{-val} \ (hrs\text{-mem} \ (?t\text{-hrs} \ s) \ (?b \ s) \ (?n \ s) \ (?y \ s)))) \ s)$
 $\llbracket typ\text{-heap-simulation } ?st \ ?r \ ?w \ ?v \ ?t\text{-hrs} \ ?t\text{-hrs-update}; \ abs\text{-expr } ?st \ ?Pb \ ?b' \ ?b; \ abs\text{-expr } ?st \ ?Pn \ ?n' \ ?n \rrbracket \implies \ abs\text{-expr } ?st \ (\lambda s. \ ?Pb \ s \wedge \ ?Pn \ s \wedge \ ?n' \ s < \ CARD(?b) \wedge \ ?v \ s \ (PTR\text{-COERCE}(?'a[?'b] \rightarrow \ ?'a) \ (?b' \ s) \ +_p \ int \ (?n' \ s))) \ (\lambda s. \ ?r \ s \ (PTR\text{-COERCE}(?'a[?'b] \rightarrow \ ?'a) \ (?b' \ s) \ +_p \ int \ (?n' \ s))) \ (\lambda s. \ h\text{-val} \ (hrs\text{-mem} \ (?t\text{-hrs} \ s) \ (?b \ s).[?'n \ s])$
 $\ abs\text{-expr lift-global-heap } ?P \ ?a \ ?c \implies \ L2Tcorres \ lift\text{-global-heap} \ (L2\text{-seq} \ (L2\text{-guard} \ (\lambda t. \ IS\text{-VALID}(32 \ word) \ t \ ?p \wedge \ ?P \ t)) \ (\lambda\text{-}. \ L2\text{-modify} \ (\lambda s. \ heap\text{-w32-update} \ (\lambda h. \ h(?p := \ ?a \ s) \ s))) \ (globals.IO\text{-modify-heap-paddingE} \ ?p \ ?c)$
 $\ abs\text{-expr lift-global-heap } ?P \ ?a \ ?c \implies \ L2Tcorres \ lift\text{-global-heap} \ (L2\text{-seq} \ (L2\text{-guard} \ (\lambda t. \ IS\text{-VALID}(32 \ word) \ t \ (PTR\text{-COERCE}(32 \ signed \ word \rightarrow \ 32 \ word) \ ?p) \wedge \ ?P \ t)) \ (\lambda\text{-}. \ L2\text{-modify} \ (\lambda s. \ heap\text{-w32-update} \ (\lambda h. \ h(PTR\text{-COERCE}(32 \ signed \ word \rightarrow \ 32 \ word) \ ?p := \ UCAST(32 \ signed \rightarrow \ 32) \ (?a \ s))) \ s))) \ (globals.IO\text{-modify-heap-paddingE} \ ?p \ ?c)$
 $\ abs\text{-expr lift-global-heap } ?P \ ?a \ ?c \implies \ L2Tcorres \ lift\text{-global-heap} \ (L2\text{-seq} \ (L2\text{-guard} \ (\lambda t. \ IS\text{-VALID}(8 \ word) \ t \ ?p \wedge \ ?P \ t)) \ (\lambda\text{-}. \ L2\text{-modify} \ (\lambda s. \ heap\text{-w8-update} \ (\lambda h. \ h(?p := \ ?a \ s) \ s))) \ (globals.IO\text{-modify-heap-paddingE} \ ?p \ ?c)$
 $\ abs\text{-expr lift-global-heap } ?P \ ?a \ ?c \implies \ L2Tcorres \ lift\text{-global-heap} \ (L2\text{-seq} \ (L2\text{-guard} \ (\lambda t. \ IS\text{-VALID}(8 \ word) \ t \ (PTR\text{-COERCE}(8 \ signed \ word \rightarrow \ 8 \ word) \ ?p) \wedge \ ?P \ t)) \ (\lambda\text{-}. \ L2\text{-modify} \ (\lambda s. \ heap\text{-w8-update} \ (\lambda h. \ h(PTR\text{-COERCE}(8 \ signed \ word \rightarrow \ 8 \ word) \ ?p := \ UCAST(8 \ signed \rightarrow \ 8) \ (?a \ s))) \ s))) \ (globals.IO\text{-modify-heap-paddingE} \ ?p \ ?c)$
 $\ abs\text{-expr lift-global-heap } ?P \ ?a \ ?c \implies \ L2Tcorres \ lift\text{-global-heap} \ (L2\text{-seq} \ (L2\text{-guard} \ (\lambda t. \ IS\text{-VALID}(32 \ word \ ptr) \ t \ ?p \wedge \ ?P \ t)) \ (\lambda\text{-}. \ L2\text{-modify} \ (\lambda s. \ heap\text{-w32}'ptr\text{-update} \ (\lambda h. \ h(?p := \ ?a \ s) \ s))) \ (globals.IO\text{-modify-heap-paddingE} \ ?p \ ?c)$
 $\ abs\text{-expr lift-global-heap } ?P \ ?a \ ?c \implies \ L2Tcorres \ lift\text{-global-heap} \ (L2\text{-seq} \ (L2\text{-guard} \ (\lambda t. \ IS\text{-VALID}(unit \ ptr) \ t \ ?p \wedge \ ?P \ t)) \ (\lambda\text{-}. \ L2\text{-modify} \ (\lambda s. \ heap\text{-unit}'ptr\text{-update} \ (\lambda h. \ h(?p := \ ?a \ s) \ s))) \ (globals.IO\text{-modify-heap-paddingE} \ ?p \ ?c)$
 $\ abs\text{-expr lift-global-heap } ?P \ ?a \ ?c \implies \ L2Tcorres \ lift\text{-global-heap} \ (L2\text{-seq} \ (L2\text{-guard} \ (\lambda t. \ IS\text{-VALID}(data\text{-C}) \ t \ ?p \wedge \ ?P \ t)) \ (\lambda\text{-}. \ L2\text{-modify} \ (\lambda s. \ heap\text{-data-C-update} \ (\lambda h. \ h(?p := \ ?a \ s) \ s))) \ (globals.IO\text{-modify-heap-paddingE} \ ?p \ ?c)$
 $\ abs\text{-expr lift-global-heap } ?P \ ?a \ ?c \implies \ L2Tcorres \ lift\text{-global-heap} \ (L2\text{-seq} \ (L2\text{-guard} \ (\lambda t. \ IS\text{-VALID}(closed\text{-C}) \ t \ ?p \wedge \ ?P \ t)) \ (\lambda\text{-}. \ L2\text{-modify} \ (\lambda s. \ heap\text{-closed-C-update} \ (\lambda h. \ h(?p := \ ?a \ s) \ s))) \ (globals.IO\text{-modify-heap-paddingE} \ ?p \ ?c)$
 $\ abs\text{-expr lift-global-heap } ?P \ ?a \ ?c \implies \ L2Tcorres \ lift\text{-global-heap} \ (L2\text{-seq} \ (L2\text{-guard} \ (\lambda t. \ IS\text{-VALID}(unpacked\text{-C}) \ t \ ?p \wedge \ ?P \ t)) \ (\lambda\text{-}. \ L2\text{-modify} \ (\lambda s. \ heap\text{-unpacked-C-update} \ (\lambda h. \ h(?p := \ ?a \ s) \ s))) \ (globals.IO\text{-modify-heap-paddingE} \ ?p \ ?c)$
 $\ abs\text{-expr lift-global-heap } ?P \ ?a \ ?c \implies \ L2Tcorres \ lift\text{-global-heap} \ (L2\text{-seq} \ (L2\text{-guard} \ (\lambda t. \ valid\text{-array-base.valid-array} \ (\lambda h. \ IS\text{-VALID}(unpacked\text{-C}) \ h) \ t \ ?p \wedge \ ?P \ t)) \ (\lambda\text{-}. \ L2\text{-modify} \ (\lambda s. \ unpacked\text{-C.heap-array-map} \ ?p \ (\lambda\text{-}. \ ?a \ s) \ s))) \ (globals.IO\text{-modify-heap-paddingE} \ ?p \ ?c)$

$s))$ ($globals.IO-modify-heap-paddingE$ $?p$ $?c$)
 $abs-expr$ $lift-global-heap$ $?P$ $?a$ $?c \implies L2Tcorres$ $lift-global-heap$ ($L2-seq$
 $(L2-guard$ ($\lambda t. IS-VALID(array-C)$ t $?p \wedge ?P t$)) ($\lambda s. L2-modify$ ($\lambda s. heap-array-C-map$
 $?p$ ($\lambda s. ?a s$ s))) ($globals.IO-modify-heap-paddingE$ $?p$ $?c$)
 $\llbracket struct-rewrite-expr$ $?P$ $?init_c'$ $?init_c$; $abs-guard$ $lift-global-heap$ $?P'$ $?P$;
 $abs-expr$ $lift-global-heap$ $?Q$ $?init_a$ $?init_c'$; $THIN$ ($\wedge p. L2Tcorres$ $lift-global-heap$
 $(?f_a p)$ ($?f_c p$)) $\rrbracket \implies L2Tcorres$ $lift-global-heap$ ($L2-seq$ ($L2-guard$ ($\lambda s. ?P' s$
 $\wedge ?Q s$)) ($\lambda s. heap-w32.guard-with-fresh-stack-ptr$ $?n$ $?init_a$ ($L2-VARS$ $?f_a$
 $?nm$))) ($globals.with-fresh-stack-ptr$ $?n$ $?init_c$ ($L2-VARS$ $?f_c$ $?nm$))
 $\llbracket \wedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. globals.typing.unchanged-typing-on$ \mathcal{S} s \rrbracket ; $struct-rewrite-expr$
 $?P$ $?init_c'$ $?init_c$; $abs-guard$ $lift-global-heap$ $?P'$ $?P$; $abs-expr$ $lift-global-heap$
 $?Q$ $?init_a$ $?init_c'$; $THIN$ ($\wedge p. L2Tcorres$ $lift-global-heap$ ($?f_a p)$ ($?f_c p$)) \rrbracket
 $\implies L2Tcorres$ $lift-global-heap$ ($L2-seq$ ($L2-guard$ ($\lambda s. ?P' s \wedge ?Q s$)) ($\lambda s.$
 $heap-w32.assume-with-fresh-stack-ptr$ $?n$ $?init_a$ ($L2-VARS$ $?f_a$ $?nm$))) ($globals.with-fresh-stack-ptr$
 $?n$ $?init_c$ ($L2-VARS$ $?f_c$ $?nm$))
 $\llbracket struct-rewrite-expr$ $?P$ $?init_c'$ $?init_c$; $abs-guard$ $lift-global-heap$ $?P'$ $?P$;
 $abs-expr$ $lift-global-heap$ $?Q$ $?init_a$ $?init_c'$; $THIN$ ($\wedge p. L2Tcorres$ $lift-global-heap$
 $(?f_a p)$ ($?f_c p$)) $\rrbracket \implies L2Tcorres$ $lift-global-heap$ ($L2-seq$ ($L2-guard$ ($\lambda s. ?P'$
 $s \wedge ?Q s$)) ($\lambda s. heap-w8.guard-with-fresh-stack-ptr$ $?n$ $?init_a$ ($L2-VARS$ $?f_a$
 $?nm$))) ($globals.with-fresh-stack-ptr$ $?n$ $?init_c$ ($L2-VARS$ $?f_c$ $?nm$))
 $\llbracket \wedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. globals.typing.unchanged-typing-on$ \mathcal{S} s \rrbracket ; $struct-rewrite-expr$
 $?P$ $?init_c'$ $?init_c$; $abs-guard$ $lift-global-heap$ $?P'$ $?P$; $abs-expr$ $lift-global-heap$
 $?Q$ $?init_a$ $?init_c'$; $THIN$ ($\wedge p. L2Tcorres$ $lift-global-heap$ ($?f_a p)$ ($?f_c p$)) \rrbracket
 $\implies L2Tcorres$ $lift-global-heap$ ($L2-seq$ ($L2-guard$ ($\lambda s. ?P' s \wedge ?Q s$)) ($\lambda s.$
 $heap-w8.assume-with-fresh-stack-ptr$ $?n$ $?init_a$ ($L2-VARS$ $?f_a$ $?nm$))) ($globals.with-fresh-stack-ptr$
 $?n$ $?init_c$ ($L2-VARS$ $?f_c$ $?nm$))
 $\llbracket struct-rewrite-expr$ $?P$ $?init_c'$ $?init_c$; $abs-guard$ $lift-global-heap$ $?P'$ $?P$;
 $abs-expr$ $lift-global-heap$ $?Q$ $?init_a$ $?init_c'$; $THIN$ ($\wedge p. L2Tcorres$ $lift-global-heap$
 $(?f_a p)$ ($?f_c p$)) $\rrbracket \implies L2Tcorres$ $lift-global-heap$ ($L2-seq$ ($L2-guard$ ($\lambda s. ?P'$
 $s \wedge ?Q s$)) ($\lambda s. heap-w32'ptr.guard-with-fresh-stack-ptr$ $?n$ $?init_a$ ($L2-VARS$
 $?f_a$ $?nm$))) ($globals.with-fresh-stack-ptr$ $?n$ $?init_c$ ($L2-VARS$ $?f_c$ $?nm$))
 $\llbracket \wedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. globals.typing.unchanged-typing-on$ \mathcal{S} s \rrbracket ; $struct-rewrite-expr$
 $?P$ $?init_c'$ $?init_c$; $abs-guard$ $lift-global-heap$ $?P'$ $?P$; $abs-expr$ $lift-global-heap$
 $?Q$ $?init_a$ $?init_c'$; $THIN$ ($\wedge p. L2Tcorres$ $lift-global-heap$ ($?f_a p)$ ($?f_c p$)) \rrbracket
 $\implies L2Tcorres$ $lift-global-heap$ ($L2-seq$ ($L2-guard$ ($\lambda s. ?P' s \wedge ?Q s$)) ($\lambda s.$
 $heap-w32'ptr.assume-with-fresh-stack-ptr$ $?n$ $?init_a$ ($L2-VARS$ $?f_a$ $?nm$))) ($globals.with-fresh-stack-ptr$
 $?n$ $?init_c$ ($L2-VARS$ $?f_c$ $?nm$))
 $\llbracket struct-rewrite-expr$ $?P$ $?init_c'$ $?init_c$; $abs-guard$ $lift-global-heap$ $?P'$ $?P$;
 $abs-expr$ $lift-global-heap$ $?Q$ $?init_a$ $?init_c'$; $THIN$ ($\wedge p. L2Tcorres$ $lift-global-heap$
 $(?f_a p)$ ($?f_c p$)) $\rrbracket \implies L2Tcorres$ $lift-global-heap$ ($L2-seq$ ($L2-guard$ ($\lambda s. ?P'$
 $s \wedge ?Q s$)) ($\lambda s. heap-unit'ptr.guard-with-fresh-stack-ptr$ $?n$ $?init_a$ ($L2-VARS$
 $?f_a$ $?nm$))) ($globals.with-fresh-stack-ptr$ $?n$ $?init_c$ ($L2-VARS$ $?f_c$ $?nm$))
 $\llbracket \wedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. globals.typing.unchanged-typing-on$ \mathcal{S} s \rrbracket ; $struct-rewrite-expr$
 $?P$ $?init_c'$ $?init_c$; $abs-guard$ $lift-global-heap$ $?P'$ $?P$; $abs-expr$ $lift-global-heap$

$?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres \text{lift-global-heap } (?f_a p) (?f_c p))\|$
 $\implies L2Tcorres \text{lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-heap-unit}'ptr.\text{assume-with-fresh-stack-ptr } ?n ?init_a (L2\text{-VARS } ?f_a ?nm)))$
 $(\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2\text{-VARS } ?f_c ?nm))$
 $\|struct\text{-rewrite-expr } ?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P;$
 $\text{abs-expr lift-global-heap } ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres \text{lift-global-heap } (?f_a p) (?f_c p))\|$
 $\implies L2Tcorres \text{lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-heap-data-C.guard-with-fresh-stack-ptr } ?n ?init_a (L2\text{-VARS } ?f_a ?nm)))$
 $(\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2\text{-VARS } ?f_c ?nm))$
 $\|\bigwedge s p. ?f_c p \cdot s ?\| \lambda r. \text{globals.typing.unchanged-typing-on } \mathcal{S} s \|;$ *struct-rewrite-expr*
 $?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P; \text{abs-expr lift-global-heap } ?Q$
 $?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres \text{lift-global-heap } (?f_a p) (?f_c p))\|$
 $\implies L2Tcorres \text{lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-heap-data-C.assume-with-fresh-stack-ptr } ?n ?init_a (L2\text{-VARS } ?f_a ?nm)))$
 $(\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2\text{-VARS } ?f_c ?nm))$
 $\|struct\text{-rewrite-expr } ?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P;$
 $\text{abs-expr lift-global-heap } ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres \text{lift-global-heap } (?f_a p) (?f_c p))\|$
 $\implies L2Tcorres \text{lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-heap-closed-C.guard-with-fresh-stack-ptr } ?n ?init_a (L2\text{-VARS } ?f_a ?nm)))$
 $(\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2\text{-VARS } ?f_c ?nm))$
 $\|\bigwedge s p. ?f_c p \cdot s ?\| \lambda r. \text{globals.typing.unchanged-typing-on } \mathcal{S} s \|;$ *struct-rewrite-expr*
 $?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P; \text{abs-expr lift-global-heap } ?Q$
 $?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres \text{lift-global-heap } (?f_a p) (?f_c p))\|$
 $\implies L2Tcorres \text{lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-heap-closed-C.assume-with-fresh-stack-ptr } ?n ?init_a (L2\text{-VARS } ?f_a ?nm)))$
 $(\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2\text{-VARS } ?f_c ?nm))$
 $\|struct\text{-rewrite-expr } ?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P;$
 $\text{abs-expr lift-global-heap } ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres \text{lift-global-heap } (?f_a p) (?f_c p))\|$
 $\implies L2Tcorres \text{lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-heap-unpacked-C.guard-with-fresh-stack-ptr } ?n ?init_a (L2\text{-VARS } ?f_a ?nm)))$
 $(\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2\text{-VARS } ?f_c ?nm))$
 $\|\bigwedge s p. ?f_c p \cdot s ?\| \lambda r. \text{globals.typing.unchanged-typing-on } \mathcal{S} s \|;$ *struct-rewrite-expr*
 $?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P; \text{abs-expr lift-global-heap } ?Q$
 $?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres \text{lift-global-heap } (?f_a p) (?f_c p))\|$
 $\implies L2Tcorres \text{lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-heap-unpacked-C.assume-with-fresh-stack-ptr } ?n ?init_a (L2\text{-VARS } ?f_a ?nm)))$
 $(\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2\text{-VARS } ?f_c ?nm))$
 $\text{abs-expr lift-global-heap } ?P ?a ?c \implies L2Tcorres \text{lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda t. \text{IS-VALID}(\text{unpacked-C}[3][2]) t ?p \wedge ?P t)) (\lambda\text{-L2-modify } (\lambda s. \text{unpacked-C.outer.heap-array-map } ?p (\lambda\text{-?a } s) s)))$
 $(\text{globals.IO-modify-heap-paddingE } ?p ?c)$
 $\|struct\text{-rewrite-expr } ?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P;$
 $\text{abs-expr lift-global-heap } ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres \text{lift-global-heap } (?f_a p) (?f_c p))\|$
 $\implies L2Tcorres \text{lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-L2-modify } (\lambda s. \text{unpacked-C.outer.heap-array-map } ?p (\lambda\text{-?a } s) s)))$
 $(\text{globals.IO-modify-heap-paddingE } ?p ?c)$

$s \wedge ?Q s)$ ($\lambda\text{-heap-unpacked-C'array-3'array-2.guard-with-fresh-stack-ptr}$
 $?n ?init_a (L2-VARS ?f_a ?nm))$) ($globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS$
 $?f_c ?nm)$)
 $\llbracket \bigwedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. globals.typing.unchanged-typing-on \mathcal{S} s \rrbracket$; $struct\text{-rewrite}\text{-expr}$
 $?P ?init_c' ?init_c$; $abs\text{-guard lift-global-heap } ?P' ?P$; $abs\text{-expr lift-global-heap}$
 $?Q ?init_a ?init_c'$; $THIN (\bigwedge p. L2Tcorres lift-global-heap (?f_a p) (?f_c p))$
 $\implies L2Tcorres lift-global-heap (L2\text{-seq} (L2\text{-guard} (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-}$
 $heap-unpacked-C'array-3'array-2.assume-with-fresh-stack-ptr ?n ?init_a (L2-VARS$
 $?f_a ?nm))$) ($globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm)$)
 $\llbracket struct\text{-rewrite}\text{-expr } ?P ?init_c' ?init_c$; $abs\text{-guard lift-global-heap } ?P' ?P$;
 $abs\text{-expr lift-global-heap } ?Q ?init_a ?init_c'$; $THIN (\bigwedge p. L2Tcorres lift-global-heap$
 $(?f_a p) (?f_c p))$
 $\implies L2Tcorres lift-global-heap (L2\text{-seq} (L2\text{-guard} (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-}$
 $heap-two-dimensional-C.guard-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a$
 $?nm))$) ($globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm)$)
 $\llbracket \bigwedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. globals.typing.unchanged-typing-on \mathcal{S} s \rrbracket$; $struct\text{-rewrite}\text{-expr}$
 $?P ?init_c' ?init_c$; $abs\text{-guard lift-global-heap } ?P' ?P$; $abs\text{-expr lift-global-heap}$
 $?Q ?init_a ?init_c'$; $THIN (\bigwedge p. L2Tcorres lift-global-heap (?f_a p) (?f_c p))$
 $\implies L2Tcorres lift-global-heap (L2\text{-seq} (L2\text{-guard} (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-}$
 $heap-two-dimensional-C.assume-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a$
 $?nm))$) ($globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm)$)
 $\llbracket struct\text{-rewrite}\text{-expr } ?P ?init_c' ?init_c$; $abs\text{-guard lift-global-heap } ?P' ?P$;
 $abs\text{-expr lift-global-heap } ?Q ?init_a ?init_c'$; $THIN (\bigwedge p. L2Tcorres lift-global-heap$
 $(?f_a p) (?f_c p))$
 $\implies L2Tcorres lift-global-heap (L2\text{-seq} (L2\text{-guard} (\lambda s. ?P' s$
 $\wedge ?Q s)) (\lambda\text{-heap-data-struct1-C.guard-with-fresh-stack-ptr ?n ?init_a (L2-VARS$
 $?f_a ?nm))$) ($globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm)$)
 $\llbracket \bigwedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. globals.typing.unchanged-typing-on \mathcal{S} s \rrbracket$; $struct\text{-rewrite}\text{-expr}$
 $?P ?init_c' ?init_c$; $abs\text{-guard lift-global-heap } ?P' ?P$; $abs\text{-expr lift-global-heap}$
 $?Q ?init_a ?init_c'$; $THIN (\bigwedge p. L2Tcorres lift-global-heap (?f_a p) (?f_c p))$
 $\implies L2Tcorres lift-global-heap (L2\text{-seq} (L2\text{-guard} (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-}$
 $heap-data-struct1-C.assume-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))$)
 $(globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$
 $\llbracket struct\text{-rewrite}\text{-expr } ?P ?init_c' ?init_c$; $abs\text{-guard lift-global-heap } ?P' ?P$;
 $abs\text{-expr lift-global-heap } ?Q ?init_a ?init_c'$; $THIN (\bigwedge p. L2Tcorres lift-global-heap$
 $(?f_a p) (?f_c p))$
 $\implies L2Tcorres lift-global-heap (L2\text{-seq} (L2\text{-guard} (\lambda s. ?P' s$
 $\wedge ?Q s)) (\lambda\text{-heap-data-struct2-C.guard-with-fresh-stack-ptr ?n ?init_a (L2-VARS$
 $?f_a ?nm))$) ($globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm)$)
 $\llbracket \bigwedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. globals.typing.unchanged-typing-on \mathcal{S} s \rrbracket$; $struct\text{-rewrite}\text{-expr}$
 $?P ?init_c' ?init_c$; $abs\text{-guard lift-global-heap } ?P' ?P$; $abs\text{-expr lift-global-heap}$
 $?Q ?init_a ?init_c'$; $THIN (\bigwedge p. L2Tcorres lift-global-heap (?f_a p) (?f_c p))$
 $\implies L2Tcorres lift-global-heap (L2\text{-seq} (L2\text{-guard} (\lambda s. ?P' s \wedge ?Q s)) (\lambda\text{-}$
 $heap-data-struct2-C.assume-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))$)
 $(globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$
 $\llbracket struct\text{-rewrite}\text{-expr } ?P ?init_c' ?init_c$; $abs\text{-guard lift-global-heap } ?P' ?P$;

$s \wedge ?Q s)) (\lambda-. \text{heap-data-}C'\text{array-10.guard-with-fresh-stack-ptr } ?n ?init_a$
 $(L2-VARS ?f_a ?nm))) (\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c$
 $?nm))$

$\llbracket \wedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. \text{globals.typing.unchanged-typing-on } \mathcal{S} s \rrbracket; \text{struct-rewrite-expr}$
 $?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P; \text{abs-expr lift-global-heap}$
 $?Q ?init_a ?init_c'; \text{THIN } (\wedge p. L2Tcorres \text{ lift-global-heap } (?f_a p) (?f_c p)) \rrbracket$
 $\implies L2Tcorres \text{ lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s))) (\lambda-$
 $\text{heap-data-}C'\text{array-10.assume-with-fresh-stack-ptr } ?n ?init_a (L2-VARS ?f_a$
 $?nm))) (\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \text{struct-rewrite-expr } ?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P;$
 $\text{abs-expr lift-global-heap } ?Q ?init_a ?init_c'; \text{THIN } (\wedge p. L2Tcorres \text{ lift-global-heap}$
 $(?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres \text{ lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s$
 $\wedge ?Q s))) (\lambda-. \text{heap-data-array-}C.\text{guard-with-fresh-stack-ptr } ?n ?init_a (L2-VARS$
 $?f_a ?nm))) (\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \wedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. \text{globals.typing.unchanged-typing-on } \mathcal{S} s \rrbracket; \text{struct-rewrite-expr}$
 $?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P; \text{abs-expr lift-global-heap}$
 $?Q ?init_a ?init_c'; \text{THIN } (\wedge p. L2Tcorres \text{ lift-global-heap } (?f_a p) (?f_c p)) \rrbracket$
 $\implies L2Tcorres \text{ lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s))) (\lambda-$
 $\text{heap-data-array-}C.\text{assume-with-fresh-stack-ptr } ?n ?init_a (L2-VARS ?f_a ?nm)))$
 $(\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \text{struct-rewrite-expr } ?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P;$
 $\text{abs-expr lift-global-heap } ?Q ?init_a ?init_c'; \text{THIN } (\wedge p. L2Tcorres \text{ lift-global-heap}$
 $(?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres \text{ lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s$
 $\wedge ?Q s))) (\lambda-. \text{heap-outer-}C.\text{guard-with-fresh-stack-ptr } ?n ?init_a (L2-VARS$
 $?f_a ?nm))) (\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \wedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. \text{globals.typing.unchanged-typing-on } \mathcal{S} s \rrbracket; \text{struct-rewrite-expr}$
 $?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P; \text{abs-expr lift-global-heap}$
 $?Q ?init_a ?init_c'; \text{THIN } (\wedge p. L2Tcorres \text{ lift-global-heap } (?f_a p) (?f_c p)) \rrbracket$
 $\implies L2Tcorres \text{ lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s))) (\lambda-$
 $\text{heap-outer-}C.\text{assume-with-fresh-stack-ptr } ?n ?init_a (L2-VARS ?f_a ?nm)))$
 $(\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \text{struct-rewrite-expr } ?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P;$
 $\text{abs-expr lift-global-heap } ?Q ?init_a ?init_c'; \text{THIN } (\wedge p. L2Tcorres \text{ lift-global-heap}$
 $(?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres \text{ lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P'$
 $s \wedge ?Q s))) (\lambda-. \text{heap-other-}C.\text{guard-with-fresh-stack-ptr } ?n ?init_a (L2-VARS$
 $?f_a ?nm))) (\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \wedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. \text{globals.typing.unchanged-typing-on } \mathcal{S} s \rrbracket; \text{struct-rewrite-expr}$
 $?P ?init_c' ?init_c; \text{abs-guard lift-global-heap } ?P' ?P; \text{abs-expr lift-global-heap}$
 $?Q ?init_a ?init_c'; \text{THIN } (\wedge p. L2Tcorres \text{ lift-global-heap } (?f_a p) (?f_c p)) \rrbracket$
 $\implies L2Tcorres \text{ lift-global-heap } (L2\text{-seq } (L2\text{-guard } (\lambda s. ?P' s \wedge ?Q s))) (\lambda-$
 $\text{heap-other-}C.\text{assume-with-fresh-stack-ptr } ?n ?init_a (L2-VARS ?f_a ?nm)))$
 $(\text{globals.with-fresh-stack-ptr } ?n ?init_c (L2-VARS ?f_c ?nm))$

$\text{abs-expr lift-global-heap } ?P ?a ?c \implies L2Tcorres \text{ lift-global-heap } (L2\text{-seq}$
 $(L2\text{-guard } (\lambda t. \text{IS-VALID}(\text{unpacked-}C[2]) t ?p \wedge ?P t)) (\lambda-. L2\text{-modify } (\lambda s.$

$unpacked-C.heap-array-map ?p (\lambda-. ?a s s))) (globals.IO-modify-heap-paddingE ?p ?c)$

$\llbracket struct-rewrite-expr ?P ?init_c' ?init_c; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres lift-global-heap (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres lift-global-heap (L2-seq (L2-guard (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. heap-unpacked-C'array-2.guard-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \bigwedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. globals.typing.unchanged-typing-on \mathcal{S} s \rrbracket; struct-rewrite-expr ?P ?init_c' ?init_c; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres lift-global-heap (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres lift-global-heap (L2-seq (L2-guard (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. heap-unpacked-C'array-2.assume-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket struct-rewrite-expr ?P ?init_c' ?init_c; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres lift-global-heap (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres lift-global-heap (L2-seq (L2-guard (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. heap-array-C.guard-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$

$\llbracket \bigwedge s p. ?f_c p \cdot s ?\rrbracket \lambda r. globals.typing.unchanged-typing-on \mathcal{S} s \rrbracket; struct-rewrite-expr ?P ?init_c' ?init_c; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init_a ?init_c'; THIN (\bigwedge p. L2Tcorres lift-global-heap (?f_a p) (?f_c p)) \rrbracket \implies L2Tcorres lift-global-heap (L2-seq (L2-guard (\lambda s. ?P' s \wedge ?Q s)) (\lambda-. heap-array-C.assume-with-fresh-stack-ptr ?n ?init_a (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?init_c (L2-VARS ?f_c ?nm))$ and keeping in mind that they are used by recursively applying them as introduction rules to a concrete program C and a schematic variable for the abstract program A one can see the following strategy:

- First some normalisation of expressions involving pointers is performed on the concrete program C . In particular an update to a dereferenced pointer $\&(p \rightarrow f)$ by a value v is transformed to an update on p , where first the value of p is fetched from memory and then field f of this value is updated by v and the resulting compound value is put back to memory.
- Only after the normalisation on C , the actual translation to an abstract A on the split heap is performed.

The normalisation step is guided by predicates *struct-rewrite-guard*, *struct-rewrite-expr* and *struct-rewrite-modifies*. After that normalisation the atomic building block for the simulation is provided by an instance of *typ-heap-simulation*, which tells how a lookup and update of a pointer of a given type in the byte heap is simulated in the split heap.

The proofs of the necessary instances of *typ-heap-simulation* are provided once and for all in the initialisation phase of **autocorres** or **init-autocorres**. They are provided for atomic and shared heaps as well as derived heaps.

Proving *typ-heap-simulation*

As stated before this step was extended quite excessively and quite some infrastructure of both theorems as well as ML code were built to support it.

Typed vs. Untyped Dialects/ Records vs. Byte-Lists First some general remarks on the general concepts and setup. The UMM was designed in the spirit to take advantage of HOL type and class inference to support reasoning about C-types and pointers. As a consequence some of the definitions and theorems are rather subtle and somehow live on the edge of what can be typed and expressed in the HOL type system. Sometimes the essence or the limitations of a theorem only become 'visible' when also looking at the types of expressions, especially the pointer types. Unfortunately writing about these concepts is rather ambiguous with respect to the notion type. 'Type' might refer to the original C-structure in the C program, this type is related to an abstract HOL-type, which is a record. This record is an instance of various type-classes: *c-type*, *mem-type*, *xmem-type*. For types of *c-type* we define overloaded *typ-info-t*, which associates a HOL-term describing the type (referred to as type-description or type-information). Moreover, pointers represented in HOL carry the HOL-type they point to as phantom type.

An illustration for the subtle typing is the two terms $\&(p \rightarrow f) \neq \&(q \rightarrow g)$ vs. $p \neq q \implies \&(p \rightarrow f) \neq \&(q \rightarrow g)$. Whereas in first term the pointers p and q have a different type, in the second term they both have the same type as the inequation in the precondition also makes the types equal. This might not always be the intended meaning. When the type of the pointer is irrelevant one might resort to plain addresses instead of pointers $ptr\text{-}val\ p = ptr\text{-}val\ q \implies \&(p \rightarrow f) = \&(q \rightarrow g)$. This general theme occurs in various places. There often is a typed variant of a concept and also an untyped variant, based on addresses or byte lists, like p vs. $ptr\text{-}val\ p$. Both are closely related. The typed variant somehow reflects the 'API' of the concept and is more abstract, whereas during a proof one actually resort to the untyped variant to gain flexibility. These two views on a concepts leads to a duplication of lemmas. Moreover, a lemma might not immediately be available in the form one expects, but can be 'easily' derived from some related lemmas.

The central UMM HOL-datatype to reflect C-types into HOL terms is $(\prime a, \prime b)\ typ\text{-}desc$ and is defined mutually recursive with $(\prime a, \prime b)\ typ\text{-}struct$. It

is basically a tree describing the nested structure of an aggregate type. Field names, alignment and size information is encoded, and lookup and update of sub-fields can be derived from this information.

The most prominent instances of this type are the typed variant *'a xtyp-info* vs. the untyped variant *typ-uinfo*. The former can be transformed to the latter by *export-uinfo*. As with pointer types the type variable in *'a xtyp-info* denotes the abstract HOL-C-type that is described by the type information. So to derive a field-lookup or field-update on HOL-types from the type information one needs to resort to the typed variant. Exporting the type information via *export-uinfo* maintains the shape of the tree, but removes all the HOL-C-type dependent components. The resulting *typ-uinfo* has two main use cases:

- It can be seen as a type-tag identifying a C-type. It can be used to relate two C-types on the HOL-term level, e.g. ask if they are equal or if one is contained in the other.
- It can be used to normalise a byte-list representing a value of that type. Normalisation means that all padding bytes in the byte-list are set to zero.

For each C-type the C-Parser also generates the type information which can be accessed via the overloaded function: *typ-info-t TYPE('a)*.

The main use cases for *typ-info-t TYPE('a)* are

- Provide lenses for fields (access / update) on the abstract value (record), via *access-ti*, *update-ti* and *field-lookup* The main use cases for *typ-uinfo-t TYPE('a)* are
- comparison of type descriptors: equality and order $s \leq_{\tau} t$
- normalisation of byte-lists *norm-tu*, in particular setting all padding bytes to zero.

thm *sub-typ-def*

thm *typ-tag-le-def*

So *typ-info-t* is more related to the abstract view on a value (as a HOL record), whereas *typ-uinfo-t* is more related to the byte-list encoding of the value. This duality somehow also reflects the dual nature of types in C. On the one hand C is a statically typed language and on the other hand it allows to break the abstraction and switch to a low-level byte oriented view.

context

fixes *p::'a::c-type ptr*

fixes *f::qualified-field-name*

fixes *t::'a xtyp-info*

fixes $n:: \text{nat}$
begin

A central function on both typed and untyped typ-information is the function *field-lookup* which retrieves the type-information for a field of a type. A common application of this function is to state some property on a dereferenced pointer: $\text{PTR}('b) \ \&(p \rightarrow f)$. Note that p is an pointer to an ' a ': p . A typical precondition is to retrieve the type-information for field f from the type information of ' a ': $\text{field-lookup}(\text{typ-info-t TYPE}('a)) \ f \ 0 = \text{Some}(t, n)$

Note that the retrieved type information t is still tagged with ' a '. The number n is the offset of the selected field. Typically we want to relate t to the type of the selected field ' b ' (which only happens to be the same ' a ' in case the field is the empty list). The relation can be established via *export-uinfo*, namely $\text{export-uinfo} \ t = \text{export-uinfo}(\text{typ-info-t TYPE}('b))$. Note that directly equating t to $\text{typ-info-t TYPE}('b)$ is not even a well-typed expression in HOL, as ' a ' is not equal to ' b '.

The right hand side abbreviates to constant $\text{typ-uinfo-t TYPE}('b)$.

end

A concrete example might give some further insight to the relation of *typ-info-t* and *typ-uinfo-t* and how the type-information is constructed. Let us consider the field "*inner-C*" of *outer-C*, which selects a field of *inner-C*. We can retrieve the information for the field:

lemma $\text{field-lookup}(\text{typ-info-t TYPE}(\text{outer-C})) \ [\"inner-C\"] \ 0 =$
 $\text{Some}(\text{adjust-ti}(\text{typ-info-t TYPE}(\text{inner-C})) \ \text{outer-C.inner-C} \ (\text{inner-C-update} \circ$
 $(\lambda x \ . \ x)), \ 0)$
by *(simp)*

Note that the retrieved type information is constructed from the nested type information for *inner-C*, by adjusting it. Adjusting means that we say how we can lookup and update the sub-field of the record *outer-C*. This adjustment only affects the typed-view of the type information. Exporting the type information collapses to the adjusted inner type:

lemma $\text{export-uinfo}(\text{adjust-ti}(\text{typ-info-t TYPE}(\text{inner-C})) \ \text{outer-C.inner-C} \ (\text{inner-C-update}$
 $\circ (\lambda x \ . \ x)))$
 $= \text{export-uinfo}(\text{typ-info-t TYPE}(\text{inner-C}))$
by *simp*

In the realm of 'lenses', *adjust-ti* can be viewed as a form of composition of lenses. A lense for an inner type is transformed to a lense on the outer type. The lense associated to type information is captured in *access-ti*, for the 'lense-lookup' aka. 'lense-get' part, and *update-ti* for the 'lense-update' aka 'lense-put'.

Function *field-lookup* is the essential building block to relate field names e.g. as part of dereferencing a pointer to their abstracts operations on the

associated HOL-type. One can convert between the typed and untyped version:

thm *field-lookup-export-uinfo-Some-rev*
thm *field-lookup-export-uinfo-Some*

Here are some closely related, mostly derived concepts around *field-lookup*:

- *field-lookup*, *field-ti*, *field-offset*, *field-of*
- *td-set*, *sub-typ* ($s \leq_{\tau} t$)
- Family of field name functions: *all-field-names*, *TypHeap.field-names*, *field-names-u*, *field-names-no-padding*, *all-field-names-no-padding*

Often lemmas might not be available for all variants, but via some simple indirections, via definitions or conversion lemmas. In an Isar-proof, **sledgehammer** can often help to find the connections.

thm *td-set-field-lookupD*
thm *td-set-field-lookup*
thm *all-field-names-union-field-names-export-uinfo-conv*
thm *set-field-names-all-field-names-conv*
thm *field-names-u-field-names-export-uinfo-conv(1)*
thm *set-field-names-no-padding-all-field-names-no-padding-conv*
thm *all-field-names-no-padding-typ-uinfo-t-conv*

Doing the Proof The fundamental goal is to derive interpretations *typ-heap-simulation* for every relevant type. It states that the virtual heap for a type as obtained from *lifted-globals* simulates the original UMM heap in *globals*. Both states are connected by the abstraction function *lift-global-heap* which is the abstract function *st* in *typ-heap-simulation*. To facilitate the construction of the proofs we introduce intermediate helper locale *typ-heap-simulation-open-types* and make use of the infrastructure of lenses and scenes that we mentioned before.

To optimize the construction (in particular to minimise the number of heaps we have to put into into *lifted-globals*) we distinguish three main cases:

- A structure is closed and has no addressable fields: $\llbracket \text{open-types } ?\mathcal{T}; \text{heap-typing-simulation } ?\mathcal{T} \text{ ?hrs ?hrs-upd ?heap-typing ?heap-typing-upd } ?l; \text{lense } ?R \text{ ?u}; \text{map-of } ?\mathcal{T} (\text{typ-uinfo-t TYPE}(?'a)) = \text{None}; \bigwedge p \ x \ s. \text{open-types.ptr-valid } ?\mathcal{T} (\text{hrs-htd } (?hrs \ s)) \ p \implies ?l \ (?hrs-upd (\text{open-types.write-dedicated-heap } ?\mathcal{T} \ p \ x) \ s) = ?u \ (\text{upd-fun } p \ (\lambda \text{old. merge-ti-list } (\text{map snd } (\text{open-types.addressable-fields } ?\mathcal{T} \ \text{TYPE}(?'a))) \ \text{old } x)) \ (?l \ s); \bigwedge x \ d \ h. \ ?\text{heap-typing-upd } d \ (?u \ x \ h) = ?u \ x \ (?heap-typing-upd \ d \ h); \bigwedge p \ s. \forall a \in \text{ptr-span } p. \text{root-ptr-valid } (\text{hrs-htd } (?hrs \ s)) \ (\text{PTR}(\text{stack-byte } a)) \implies ?R \ (?l \ s) \ p = \text{ZERO}(?'a); \bigwedge h \ p. \ ?V \ h \ p = \text{open-types.ptr-valid}$

$?T$ ($?heap\text{-}typing$ h) p ; $\wedge p f h. ?W p f h = ?u (\lambda h'. h'(p := f (h' p)))$
 $h\]] \implies typ\text{-}heap\text{-}simulation\text{-}open\text{-}types ?T ?l ?R ?W ?V ?hrs ?hrs\text{-}upd$
 $?heap\text{-}typing ?heap\text{-}typing\text{-}upd$ In this case there is only one relevant
heap in *lifted-globals*. No overlay of a dedicated heap with some com-
mon heaps is necessary.

- A structure is completely open, meaning that all fields are addressable:
 $\llbracket open\text{-}types ?T; heap\text{-}typing\text{-}simulation ?T ?hrs ?hrs\text{-}upd ?heap\text{-}typing$
 $?heap\text{-}typing\text{-}upd ?l; map\text{-}of ?T (typ\text{-}uinfo\text{-}t TYPE(?a)) = Some ?fs;$
 $length ?rs = length ?fs; length ?ws = length ?fs; list\text{-}all (\lambda(f, r, w).$
 $open\text{-}types.typ\text{-}heap\text{-}simulation\text{-}of\text{-}field ?T ?l ?hrs ?hrs\text{-}upd ?heap\text{-}typing\text{-}upd$
 $f r w) (zip ?fs (zip ?rs ?ws)); distinct\text{-}prop (\lambda(f1, w1) (f2, w2). disj\text{-}fn$
 $f1 f2 \longrightarrow pointer\text{-}writer\text{-}disjnt\text{-}eq w1 w2) (zip ?fs ?ws); \wedge a b. fold$
 $(\lambda x. merge\text{-}ti (the (field\text{-}ti TYPE(?a) x)) a) ?fs b = a; \wedge h p. ?V h$
 $p = open\text{-}types.ptr\text{-}valid ?T (?heap\text{-}typing h) p; \wedge h p x. ?R h p =$
 $fold (\lambda r. r h p) ?rs x; \wedge p f h. ?W p f h = fold (\lambda w. w p (f (?R$
 $h p))) ?ws h\]] \implies typ\text{-}heap\text{-}simulation\text{-}open\text{-}types ?T ?l ?R ?W ?V$
 $?hrs ?hrs\text{-}upd ?heap\text{-}typing ?heap\text{-}typing\text{-}upd \wedge (\forall w. (\forall x. list\text{-}all (\lambda w'.$
 $pointer\text{-}writer\text{-}disjnt (\lambda p. w' p x) w) ?ws) \longrightarrow (\forall f. pointer\text{-}writer\text{-}disjnt$
 $(\lambda p. ?W p f) w)) \wedge (\forall w p. (\forall x. list\text{-}all (\lambda w'. w' p x \circ w = w \circ w'$
 $p x) ?ws) \longrightarrow (\forall f. ?W p f \circ w = w \circ ?W p f))$. In this case we do
not need a dedicated heap. The structure is described by a overlay of
common heaps.
- A structure is partially open, some fields are addressable and some not:
 $\llbracket open\text{-}types ?T; heap\text{-}typing\text{-}simulation ?T ?hrs ?hrs\text{-}upd ?heap\text{-}typing$
 $?heap\text{-}typing\text{-}upd ?l; map\text{-}of ?T (typ\text{-}uinfo\text{-}t TYPE(?a)) = Some ?fs;$
 $lense ?g ?u; length ?rs = length ?fs; length ?ws = length ?fs; list\text{-}all$
 $(\lambda(f, r, w). open\text{-}types.typ\text{-}heap\text{-}simulation\text{-}of\text{-}field ?T ?l ?hrs ?hrs\text{-}upd$
 $?heap\text{-}typing\text{-}upd f r w) (zip ?fs (zip ?rs ?ws)); distinct\text{-}prop (\lambda(f1,$
 $w1) (f2, w2). disj\text{-}fn f1 f2 \longrightarrow pointer\text{-}writer\text{-}disjnt\text{-}eq w1 w2) (zip$
 $?fs ?ws); list\text{-}all (\lambda w. \forall p a f. w p a \circ ?u f = ?u f \circ w p a) ?ws;$
 $\wedge p x s. open\text{-}types.ptr\text{-}valid ?T (hrs\text{-}htd (?hrs s)) p \implies ?l (?hrs\text{-}upd$
 $(open\text{-}types.write\text{-}dedicated\text{-}heap ?T p x) s) = ?u (upd\text{-}fun p (\lambda old.$
 $merge\text{-}ti\text{-}list (map snd (open\text{-}types.addressable\text{-}fields ?T TYPE(?a)))$
 $old x)) (?l s); \wedge p s. \forall a \in ptr\text{-}span p. root\text{-}ptr\text{-}valid (hrs\text{-}htd (?hrs s))$
 $(PTR(stack\text{-}byte) a) \implies ?g (?l s) p = ZERO(?a); \wedge x d h. ?heap\text{-}typing\text{-}upd$
 $d (?u x h) = ?u x (?heap\text{-}typing\text{-}upd d h); \wedge h p. ?V h p = open\text{-}types.ptr\text{-}valid$
 $?T (?heap\text{-}typing h) p; \wedge h p. ?R h p = fold (\lambda r. r h p) ?rs (?g h p); \wedge p$
 $f h. ?W p f h = fold (\lambda w. w p (f (?R h p))) ?ws (?u (upd\text{-}fun p (\lambda old.$
 $merge\text{-}ti\text{-}list (map snd (open\text{-}types.addressable\text{-}fields ?T TYPE(?a)))$
 $old (f (?R h p)))) h\]] \implies typ\text{-}heap\text{-}simulation\text{-}open\text{-}types ?T ?l ?R$
 $?W ?V ?hrs ?hrs\text{-}upd ?heap\text{-}typing ?heap\text{-}typing\text{-}upd \wedge pointer\text{-}lense$
 $?g (\lambda p f. ?u (upd\text{-}fun p f)) \wedge (\forall w. (\forall x. list\text{-}all (\lambda w'. pointer\text{-}writer\text{-}disjnt$

$(\lambda p. w' p x) w) ?ws) \longrightarrow (\forall x. \text{pointer-writer-disjnt } (\lambda p. ?u (\text{upd-fun } p (\lambda-. x))) w) \longrightarrow (\forall f. \text{pointer-writer-disjnt } (\lambda p. ?W p f) w) \wedge (\forall w p. (\forall x. \text{list-all } (\lambda w'. w' p x \circ w = w \circ w' p x) ?ws) \longrightarrow (\forall x. ?u (\text{upd-fun } p (\lambda-. x)) \circ w = w \circ ?u (\text{upd-fun } p (\lambda-. x))) \longrightarrow (\forall f. ?W p f \circ w = w \circ ?W p f))$. In that case we need an overlay of a dedicated heap and some common heaps.

On an intuitive abstract level the lemmas and proof argue about composing field-level lookup and updates within some heap(s) to lookup and updates of the complete compound structure. We want to argue that lookup and update in the UMM heap is simulated by the overlaid updates of a dedicated heap and some common heaps in the split heap. To argue about different fields of a structure we build on the idea of scenes and also extend it to reads. Note that the general problem is that a structure is represented as a HOL record and each field has a distinct type. Because of the limitations of the HOL type system we cannot directly combine e.g. functions like readers for different fields like $'a \Rightarrow 'b1$ and $'a \Rightarrow 'b2$ in a HOL list. Here the scene idea to express everything via a merge or update on $'a$ is a helpful 'trick'. E.g. 'reading' the value of a field can in a sense be as well expressed as a function $'a \Rightarrow 'a \Rightarrow 'a$ that reads the field of the first argument and puts it into any structure you give it as second argument:

thm *lift-global-heap-def*
thm *open-types.typ-heap-simulationI-all-addressable*
thm *open-types.typ-heap-simulationI-part-addressable*
thm *open-types.typ-heap-simulationI-no-addressable*
thm *pointer-lense-def*
thm *partial-pointer-lense-def*
thm *typ-heap-simulation-of-field-def*

Other important building blocks are:

- *pointer-lense* which describes a virtual heap.
- *partial-pointer-lense* which describes the effect of field lookup / updates on a virtual heap by combining a scene identifying the field and the pointer lense for the virtual heap.
- *typ-heap-simulation-of-field* $?st ?t\text{-hrs} ?t\text{-hrs}\text{-update} ?\text{heap}\text{-typing}\text{-upd} ?f' ?r' ?w' = ((\forall d p f. ?\text{heap}\text{-typing}\text{-upd } d \circ ?w' p f = ?w' p f \circ ?\text{heap}\text{-typing}\text{-upd } d) \wedge (\forall t u n. \text{field-ti } \text{TYPE}('a) ?f' = \text{Some } t \longrightarrow \text{field-lookup } (\text{typ-uinfo-t } \text{TYPE}('a)) ?f' 0 = \text{Some } (u, n) \longrightarrow \text{partial-pointer-lense } (\text{merge-ti } t) ?r' ?w' \wedge (\forall p s. (\forall a \in \text{ptr-span } p. \text{root-ptr-valid } (\text{hrs-htd } (?t\text{-hrs } s)) (\text{PTR}(\text{stack-byte } a)) \longrightarrow ?r' (?st s) p \text{ZERO}('a) = \text{ZERO}('a)) \wedge (\forall p x h. \text{ptr-valid-u } u (\text{hrs-htd } (?t\text{-hrs } h)) \&(p \rightarrow ?f') \longrightarrow ?st (?t\text{-hrs}\text{-update } (\text{hrs-mem}\text{-update } (\text{heap-upd-list$

$(\text{size-td } u) \ \&(p \rightarrow ?f') \ (\text{access-ti } t \ x)) \ h) = ?w' \ p \ x \ (?st \ h))$ which is used to break down *typ-heap-simulation* to the level of fields in the structure.

The automation is implemented in `HeapLiftBase.gen_new_heap`.

The following paragraphs describe some abstract arguments and lemma collections. In a previous version the proof *typ-heap-simulation* was more directly based on those arguments implemented in ML. Meanwhile we were able to replace most parts of the ML code by lemmas described before based on scenes and pointer lenses. Nevertheless the arguments are still valid and the lemma collections might be useful in other places.

Fundamental heaps We start with the fundamental heaps. The simulation property for heap-updates is captured in *write-simulation*. It is a commutation property. Provided we have a valid pointer in the split heap (obtained from lifting from the byte-level heap), we can either first perform the heap update in the byte-level heap and then lift the state via *lift-global-heap*, or first lift the byte-level heap into the split heap and apply the corresponding update there. Both ways yield the same final state. So we have to prove equality of two states of *lifted-globals*. The first question is which parts of the state did change? Intuitively only the affected split heap did change, all other heaps stayed the same. But all split heaps are derived from the same byte level heap. So the proof decomposes into two main arguments, one is for the affected heap where we have to show that both updates, the one on the byte-level heap and and the one on the affected split heap, are the same. The other argument is that all other split heaps remain unchanged. Actually the first case is the more straight forward one and could be handled with theorems like $PTR\text{-}VALID(?a) \ (\text{hrs-htd } ?h) \ ?p \Longrightarrow \text{plift} \ (\text{hrs-mem-update} \ (\text{heap-update } ?p \ ?v) \ ?h) = (\text{plift } ?h)(?p \mapsto ?v)$.

thm *plift-heap-update*

The second case, to prove what is not changed is more involved. As a split heap might not only contain root pointers but also valid pointers that are nested in other types we have to distinguish several cases. We developed a general theory in *open-types* and the essential theorems for the commutation proof are collected in $\llbracket PTR\text{-}VALID(?a) \ ?d \ ?p; PTR\text{-}VALID(?a) \ ?d \ ?q \rrbracket \Longrightarrow h\text{-val} \ (\text{heap-update } ?p \ ?v \ ?h) \ ?q = ((h\text{-val } ?h)(?p := ?v)) \ ?q$

$\llbracket PTR\text{-}VALID(?a) \ ?d \ ?p; PTR\text{-}VALID(?a) \ ?d \ ?q; \text{length } ?bs = \text{size-of } TYPE(?a) \rrbracket \Longrightarrow h\text{-val} \ (\text{heap-update-padding } ?p \ ?v \ ?bs \ ?h) \ ?q = ((h\text{-val } ?h)(?p := ?v)) \ ?q$

$PTR\text{-}VALID(?a) \ (\text{hrs-htd } ?h) \ ?p \Longrightarrow \text{plift } ?h \ ?p = \text{Some} \ (h\text{-val} \ (\text{hrs-mem } ?h) \ ?p)$. When we update the heap at pointer p and lookup the value of pointer q we can distinguish various cases by analysing the type relation

(e.g. if one type is nested in the other). Note that by introducing guards into the code and the design of *lift-global-heap* we only have to care about the case where both pointers are valid according to $PTR-VALID('a)$. To be more specific. That the pointer that is updated is $PTR-VALID('a)$ is ensured by the corresponding guard on the update. That the pointer we read from is actually $PTR-VALID('a)$ is more subtle. Here we rely on the construction of *lift-global-heap* where a split heap is constructed by $\lambda p. the-default\ ZERO('a)\ (plift\ (t-hrs-' g)\ p)$. In case the pointer is invalid we always obtain $ZERO('a)$ and hence only the valid pointers may be affected by an heap update. This reduction to valid pointers and $h-val$ is encapsulated in theorems $root-ptr-valid\ (hrs-htd\ ?h)\ ?p \implies the-default\ ZERO(?'a)\ (simple-lift\ (hrs-mem-update\ (heap-update\ ?p\ ?v)\ ?h)\ ?q) = ((\lambda p. the-default\ ZERO(?'a)\ (simple-lift\ ?h\ p)))(?p := ?v)\ ?q$

$\llbracket root-ptr-valid\ (hrs-htd\ ?h)\ ?p; length\ ?bs = size-of\ TYPE(?'a) \rrbracket \implies the-default\ ZERO(?'a)\ (simple-lift\ (hrs-mem-update\ (heap-update-padding\ ?p\ ?v\ ?bs)\ ?h)\ ?q) = ((\lambda p. the-default\ ZERO(?'a)\ (simple-lift\ ?h\ p)))(?p := ?v)\ ?q$

$root-ptr-valid\ (hrs-htd\ ?h)\ ?p \wedge ?f\ ?x = ?v \implies ?f\ (the-default\ ZERO(?'a)\ (simple-lift\ (hrs-mem-update\ (heap-update\ ?p\ ?x)\ ?h)\ ?q)) = ((\lambda p. ?f\ (the-default\ ZERO(?'a)\ (simple-lift\ ?h\ p)))(?p := ?v)\ ?q$

$\llbracket root-ptr-valid\ (hrs-htd\ ?h)\ ?p \wedge ?f\ ?x = ?v; length\ ?bs = size-of\ TYPE(?'a) \rrbracket \implies ?f\ (the-default\ ZERO(?'a)\ (simple-lift\ (hrs-mem-update\ (heap-update-padding\ ?p\ ?x\ ?bs)\ ?h)\ ?q)) = ((\lambda p. ?f\ (the-default\ ZERO(?'a)\ (simple-lift\ ?h\ p)))(?p := ?v)\ ?q$

$(\llbracket root-ptr-valid\ (hrs-htd\ ?h)\ ?q; hrs-htd\ ?h \models_t ?q \rrbracket \implies h-val\ (heap-update\ ?p\ ?v\ (hrs-mem\ ?h))\ ?q = h-val\ (hrs-mem\ ?h)\ ?q \implies the-default\ ZERO(?'a)\ (simple-lift\ (hrs-mem-update\ (heap-update\ ?p\ ?v)\ ?h)\ ?q) = the-default\ ZERO(?'a)\ (simple-lift\ ?h\ ?q)$

$\llbracket length\ ?bs = size-of\ TYPE(?'b); \llbracket root-ptr-valid\ (hrs-htd\ ?h)\ ?q; hrs-htd\ ?h \models_t ?q \rrbracket \implies h-val\ (heap-update-padding\ ?p\ ?v\ ?bs\ (hrs-mem\ ?h))\ ?q = h-val\ (hrs-mem\ ?h)\ ?q \rrbracket \implies the-default\ ZERO(?'a)\ (simple-lift\ (hrs-mem-update\ (heap-update-padding\ ?p\ ?v\ ?bs)\ ?h)\ ?q) = the-default\ ZERO(?'a)\ (simple-lift\ ?h\ ?q)$

$(\llbracket root-ptr-valid\ (hrs-htd\ ?h)\ ?q; hrs-htd\ ?h \models_t ?q \rrbracket \implies ?f\ (h-val\ (heap-update\ ?p\ ?v\ (hrs-mem\ ?h))\ ?q) = ?f\ (h-val\ (hrs-mem\ ?h)\ ?q) \implies ?f\ (the-default\ ZERO(?'a)\ (simple-lift\ (hrs-mem-update\ (heap-update\ ?p\ ?v)\ ?h)\ ?q)) = ?f\ (the-default\ ZERO(?'a)\ (simple-lift\ ?h\ ?q))$

$\llbracket length\ ?bs = size-of\ TYPE(?'b); \llbracket root-ptr-valid\ (hrs-htd\ ?h)\ ?q; hrs-htd\ ?h \models_t ?q \rrbracket \implies ?f\ (h-val\ (heap-update-padding\ ?p\ ?v\ ?bs\ (hrs-mem\ ?h))\ ?q) = ?f\ (h-val\ (hrs-mem\ ?h)\ ?q) \rrbracket \implies ?f\ (the-default\ ZERO(?'a)\ (simple-lift\ (hrs-mem-update\ (heap-update-padding\ ?p\ ?v\ ?bs)\ ?h)\ ?q)) = ?f\ (the-default\ ZERO(?'a)\ (simple-lift\ ?h\ ?q))$

$PTR-VALID(?'a)\ (hrs-htd\ ?h)\ ?p \implies the-default\ ZERO(?'a)\ (plift\ (hrs-mem-update\ (heap-update\ ?p\ ?v)\ ?h)\ ?q) = ((\lambda p. the-default\ ZERO(?'a)\ (plift\ ?h\ p)))(?p$

$:= ?v)) ?q$
 $\llbracket \text{PTR-VALID}(?a) (\text{hrs-htd } ?h) ?p; \text{length } ?bs = \text{size-of TYPE}(?a) \rrbracket$
 $\implies \text{the-default ZERO}(?a) (\text{plift } (\text{hrs-mem-update } (\text{heap-update-padding } ?p$
 $?v ?bs) ?h) ?q) = ((\lambda p. \text{the-default ZERO}(?a) (\text{plift } ?h p)) (?p := ?v)) ?q$
 $\text{PTR-VALID}(?a) (\text{hrs-htd } ?h) ?p \wedge ?f ?x = ?v \implies ?f (\text{the-default}$
 $\text{ZERO}(?a) (\text{plift } (\text{hrs-mem-update } (\text{heap-update } ?p ?x) ?h) ?q)) = ((\lambda p.$
 $?f (\text{the-default ZERO}(?a) (\text{plift } ?h p))) (?p := ?v)) ?q$
 $\llbracket \text{PTR-VALID}(?a) (\text{hrs-htd } ?h) ?p \wedge ?f ?x = ?v; \text{length } ?bs = \text{size-of}$
 $\text{TYPE}(?a) \rrbracket \implies ?f (\text{the-default ZERO}(?a) (\text{plift } (\text{hrs-mem-update } (\text{heap-update-padding}$
 $?p ?x ?bs) ?h) ?q)) = ((\lambda p. ?f (\text{the-default ZERO}(?a) (\text{plift } ?h p))) (?p :=$
 $?v)) ?q$
 $(\llbracket \text{PTR-VALID}(?a) (\text{hrs-htd } ?h) ?q; \text{hrs-htd } ?h \models_t ?q \rrbracket \implies h\text{-val } (\text{heap-update}$
 $?p ?v (\text{hrs-mem } ?h)) ?q = h\text{-val } (\text{hrs-mem } ?h) ?q \implies \text{the-default ZERO}(?a)$
 $(\text{plift } (\text{hrs-mem-update } (\text{heap-update } ?p ?v) ?h) ?q) = \text{the-default ZERO}(?a)$
 $(\text{plift } ?h ?q)$
 $\llbracket \text{length } ?bs = \text{size-of TYPE}(?b); \llbracket \text{PTR-VALID}(?a) (\text{hrs-htd } ?h) ?q;$
 $\text{hrs-htd } ?h \models_t ?q \rrbracket \implies h\text{-val } (\text{heap-update-padding } ?p ?v ?bs (\text{hrs-mem } ?h))$
 $?q = h\text{-val } (\text{hrs-mem } ?h) ?q \rrbracket \implies \text{the-default ZERO}(?a) (\text{plift } (\text{hrs-mem-update}$
 $(\text{heap-update-padding } ?p ?v ?bs) ?h) ?q) = \text{the-default ZERO}(?a) (\text{plift } ?h$
 $?q)$
 $(\llbracket \text{PTR-VALID}(?a) (\text{hrs-htd } ?h) ?q; \text{hrs-htd } ?h \models_t ?q \rrbracket \implies ?f (h\text{-val}$
 $(\text{heap-update } ?p ?v (\text{hrs-mem } ?h)) ?q) = ?f (h\text{-val } (\text{hrs-mem } ?h) ?q) \implies$
 $?f (\text{the-default ZERO}(?a) (\text{plift } (\text{hrs-mem-update } (\text{heap-update } ?p ?v) ?h)$
 $?q)) = ?f (\text{the-default ZERO}(?a) (\text{plift } ?h ?q))$
 $\llbracket \text{length } ?bs = \text{size-of TYPE}(?b); \llbracket \text{PTR-VALID}(?a) (\text{hrs-htd } ?h) ?q;$
 $\text{hrs-htd } ?h \models_t ?q \rrbracket \implies ?f (h\text{-val } (\text{heap-update-padding } ?p ?v ?bs (\text{hrs-mem}$
 $?h)) ?q) = ?f (h\text{-val } (\text{hrs-mem } ?h) ?q) \rrbracket \implies ?f (\text{the-default ZERO}(?a) (\text{plift}$
 $(\text{hrs-mem-update } (\text{heap-update-padding } ?p ?v ?bs) ?h) ?q)) = ?f (\text{the-default}$
 $\text{ZERO}(?a) (\text{plift } ?h ?q)).$

thm *lift-global-heap-def*
thm *the-plift-hval-eqI*
thm *plift-eqI*
thm *plift-simps*

Treatment of Array Types At the core there is no special type-information for array types. It is treated analogously to structures. Each index gets an individual field name which is the unary encoding of the index. The good thing is that one does not need any new fundamental lemmas to deal with array types. But it is not a good idea to unfold the type-information for arrays and to work on that expanded version. On the one hand things like simplification might become slow, on the other hand we can make use of the regular structure of array types and once and for all derive general lemmas.

The central theorems to work with field-lookup in arrays are $?n < \text{CARD}(?b) \implies \text{field-lookup } (\text{typ-info-t } \text{TYPE}(?a[?b])) [\text{replicate } ?n \text{ CHR}$

"1'" $?i = \text{Some } (\text{adjust-ti } (\text{typ-info-t } \text{TYPE}(\text{'a})) (\lambda x. x.[?n]) (\lambda x f. \text{Arrays.update } f \text{'n } x), ?i + ?n * \text{size-of } \text{TYPE}(\text{'a}))$ and $?i < \text{CARD}(\text{'b}) \implies \text{array-ptr-index } ?p \text{ False } ?i = \text{PTR}(\text{'a}) \ \&(\text{'p} \rightarrow [\text{replicate } ?i \text{ CHR "1"}])$.
The latter one allows to convert between an array-index-arithmetic based view of array pointers $\text{array-ptr-index } p \text{ False } i$ and the field-name view $\text{PTR}(\text{'a}) \ \&(p \rightarrow [\text{replicate } i \text{ CHR "1"}])$.

thm *field-lookup-array*

thm *array-ptr-index-field-lvalue-conv*

These theorems are used to normalise array accesses towards symbolic field name accesses of the format term $\text{Ptr } \&(p \rightarrow [\text{replicate } i \text{ CHR "1"}])$. Note that we don't unfold the definition of *replicate* here, and index i stays abstract, constrained by precondition that limits its range.

thm *unpacked-C.ptr-valid.unfold*

thm *unpacked-C'array-2.ptr-valid.unfold*

The instances of *typ-heap-simulation* can be established from *typ-heap-simulation* of the element types. We introduce the locales *array-heap-simulation* and *two-dimensional-array-heap-simulation* to automatically provide the sublocale relations when the element type falls into *array-outer-max-size* or *array-inner-max-size* respectively.

Derived Heaps The commutation proofs for derived heaps follow another path. For derived heaps we can assume that we have all the instances of *typ-heap-simulation* of the component types already available. Unaddressable fields are never shared. Pointers only appear within a dedicated enclosing parent structure. Validity of the pointer coincides with validity of the parent pointer. Pointers that are mapped by the field heap coincide with the parent pointers, there is no need to calculate the offset of the field. So deriving the commutation proof from the available theorems boils down to their composition. The central lemma connects an update of a derived heap to an fold over the updates of the toplevel fields: $c\text{-guard } ?p \implies \text{heap-update } ?p \ ?x = \text{heap-update } (\text{PTR}(32 \text{ word}) \ \&(\text{'p} \rightarrow [\text{"y4-C"}]))$
 $(y4\text{-C } ?x) \circ (\text{heap-update } (\text{PTR}(32 \text{ word}) \ \&(\text{'p} \rightarrow [\text{"y3-C"}])) (y3\text{-C } ?x) \circ$
 $(\text{heap-update } (\text{PTR}(32 \text{ word}) \ \&(\text{'p} \rightarrow [\text{"y2-C"}])) (y2\text{-C } ?x) \circ (\text{heap-update } (\text{PTR}(32 \text{ word}) \ \&(\text{'p} \rightarrow [\text{"y1-C"}])) (y1\text{-C } ?x) \circ \text{heap-update } (\text{PTR}(32 \text{ signed}$
 $\text{word}) \ \&(\text{'p} \rightarrow [\text{"x-C"}])) (x\text{-C } ?x))$

$c\text{-guard } ?p \implies \text{heap-update } ?p \ ?x = \text{heap-update } (\text{PTR}(32 \text{ word}) \ \&(\text{'p} \rightarrow [\text{"c2-C"}]))$
 $(c2\text{-C } ?x) \circ \text{heap-update } (\text{PTR}(32 \text{ word}) \ \&(\text{'p} \rightarrow [\text{"c1-C"}])) (c1\text{-C } ?x)$

$c\text{-guard } ?p \implies \text{heap-update } ?p \ ?x = \text{heap-update } (\text{PTR}(\text{closed-C}) \ \&(\text{'p} \rightarrow [\text{"fld4-C"}]))$
 $(fld4\text{-C } ?x) \circ (\text{heap-update } (\text{PTR}(32 \text{ word}) \ \&(\text{'p} \rightarrow [\text{"fld3-C"}])) (fld3\text{-C } ?x) \circ$
 $(\text{heap-update } (\text{PTR}(32 \text{ word}) \ \&(\text{'p} \rightarrow [\text{"fld2-C"}])) (fld2\text{-C } ?x) \circ \text{heap-update } (\text{PTR}(32 \text{ word}) \ \&(\text{'p} \rightarrow [\text{"fld1-C"}])) (fld1\text{-C } ?x))$

$c\text{-guard } ?p \implies \text{heap-update } ?p \ ?x = \text{heap-update } (\text{PTR}(32 \text{ word}) \ \&(\text{'p} \rightarrow [\text{"lng-C"}]))$
 $(lng\text{-C } ?x) \circ \text{heap-update } (\text{PTR}(8 \text{ word}) \ \&(\text{'p} \rightarrow [\text{"chr-C"}])) (chr\text{-C } ?x)$

$c\text{-guard } ?p \implies \text{heap-update } ?p \ ?x = \text{heap-update } (\text{PTR}(32 \text{ word}) \ \&(\ ?p \rightarrow ["\text{fld-C}"]))$
 $(\text{outer-C.fld-C } ?x) \circ \text{heap-update } (\text{PTR}(\text{inner-C}) \ \&(\ ?p \rightarrow ["\text{inner-C}"]))$ $(\text{outer-C.inner-C } ?x)$

$c\text{-guard } ?p \implies \text{heap-update } ?p \ ?x = \text{heap-update } (\text{PTR}(32 \text{ word}) \ \&(\ ?p \rightarrow ["\text{count-C}"]))$
 $(\text{count-C } ?x) \circ \text{heap-update } (\text{PTR}(\text{unpacked-C}[2]) \ \&(\ ?p \rightarrow ["\text{elements-C}"]))$
 $(\text{elements-C } ?x)$

$c\text{-guard } ?p \implies \text{heap-update } ?p \ ?x = \text{heap-update } (\text{PTR}(\text{unpacked-C})$
 $\ \&(\ ?p \rightarrow ["\text{fy-C}"]))$ $(\text{fy-C } ?x) \circ (\text{heap-update } (\text{PTR}(\text{closed-C}) \ \&(\ ?p \rightarrow ["\text{fz-C}"])))$
 $(\text{fz-C } ?x) \circ \text{heap-update } (\text{PTR}(32 \text{ word}) \ \&(\ ?p \rightarrow ["\text{fx-C}"]))$ $(\text{fx-C } ?x)$

$c\text{-guard } ?p \implies \text{heap-update } ?p \ ?x = \text{heap-update } (\text{PTR}(\text{unpacked-C}[3][2])$
 $\ \&(\ ?p \rightarrow ["\text{matrix-C}"]))$ $(\text{matrix-C } ?x)$

$c\text{-guard } ?p \implies \text{heap-update } ?p \ ?x = \text{heap-update } (\text{PTR}(32 \text{ word}) \ \&(\ ?p \rightarrow ["\text{fld-C}"]))$
 $(\text{outer-array-C.fld-C } ?x) \circ \text{heap-update } (\text{PTR}(\text{inner-C}[5]) \ \&(\ ?p \rightarrow ["\text{inner-array-C}"]))$
 $(\text{inner-array-C } ?x)$

$c\text{-guard } ?p \implies \text{heap-update } ?p \ ?x = \text{heap-update } (\text{PTR}(\text{data-C}[10])$
 $\ \&(\ ?p \rightarrow ["\text{array-C}"]))$ $(\text{data-array-C.array-C } ?x)$

$c\text{-guard } ?p \implies \text{heap-update } ?p \ ?x = \text{heap-update } (\text{PTR}(\text{data-C}) \ \&(\ ?p \rightarrow ["\text{d2-C}"]))$
 $(\text{d2-C } ?x) \circ \text{heap-update } (\text{PTR}(\text{data-C}) \ \&(\ ?p \rightarrow ["\text{d1-C}"]))$ $(\text{d1-C } ?x)$

$c\text{-guard } ?p \implies \text{heap-update } ?p \ ?x = \text{heap-update } (\text{PTR}(\text{data-struct1-C})$
 $\ \&(\ ?p \rightarrow ["\text{d-C}"]))$ $(\text{d-C } ?x)$. Moreover, we establish that *heap-update-padding*
 is equivalent to *heap-update* under the state lifting function. Padding bytes
 become irrelevant in the split heap. For each toplevel field we have the
 commutation proof connecting the monolithic byte-level heap update to the
 split-heap update collected in $\llbracket \text{IS-VALID}(32 \text{ word}) (\text{lift-global-heap } ?s) \ ?p;$
 $\text{length } ?bs = \text{size-of TYPE}(32 \text{ word}) \rrbracket \implies \text{lift-global-heap } (\text{t-hrs-'-update}$
 $(\text{hrs-mem-update } (\text{heap-update-padding } ?p \ ?x \ ?bs)) \ ?s) = \text{heap-w32-update}$
 $(\lambda h. h(\ ?p := ?x)) (\text{lift-global-heap } ?s)$

$\text{IS-VALID}(32 \text{ word}) (\text{lift-global-heap } ?s) \ ?p \implies \text{lift-global-heap } (\text{t-hrs-'-update}$
 $(\text{hrs-mem-update } (\text{heap-update } ?p \ ?x)) \ ?s) = \text{heap-w32-update } (\lambda h. h(\ ?p :=$
 $?x)) (\text{lift-global-heap } ?s)$

$\llbracket \text{IS-VALID}(32 \text{ word}) (\text{lift-global-heap } ?s) (\text{PTR-COERCE}(32 \text{ signed word}$
 $\rightarrow 32 \text{ word}) \ ?p); \text{length } ?bs = \text{size-of TYPE}(32 \text{ signed word}) \rrbracket \implies \text{lift-global-heap}$
 $(\text{t-hrs-'-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p \ ?x \ ?bs)) \ ?s) =$
 $\text{heap-w32-update } (\lambda h. h(\text{PTR-COERCE}(32 \text{ signed word } \rightarrow 32 \text{ word}) \ ?p :=$
 $\text{UCAST}(32 \text{ signed } \rightarrow 32) \ ?x)) (\text{lift-global-heap } ?s)$

$\text{IS-VALID}(32 \text{ word}) (\text{lift-global-heap } ?s) (\text{PTR-COERCE}(32 \text{ signed word}$
 $\rightarrow 32 \text{ word}) \ ?p) \implies \text{lift-global-heap } (\text{t-hrs-'-update } (\text{hrs-mem-update } (\text{heap-update}$
 $?p \ ?x)) \ ?s) = \text{heap-w32-update } (\lambda h. h(\text{PTR-COERCE}(32 \text{ signed word } \rightarrow$
 $32 \text{ word}) \ ?p := \text{UCAST}(32 \text{ signed } \rightarrow 32) \ ?x)) (\text{lift-global-heap } ?s)$

$\llbracket \text{IS-VALID}(8 \text{ word}) (\text{lift-global-heap } ?s) \ ?p; \text{length } ?bs = \text{size-of TYPE}(8$
 $\text{word}) \rrbracket \implies \text{lift-global-heap } (\text{t-hrs-'-update } (\text{hrs-mem-update } (\text{heap-update-padding}$
 $?p \ ?x \ ?bs)) \ ?s) = \text{heap-w8-update } (\lambda h. h(\ ?p := ?x)) (\text{lift-global-heap } ?s)$

$\text{IS-VALID}(8 \text{ word}) (\text{lift-global-heap } ?s) \ ?p \implies \text{lift-global-heap } (\text{t-hrs-'-update}$
 $(\text{hrs-mem-update } (\text{heap-update } ?p \ ?x)) \ ?s) = \text{heap-w8-update } (\lambda h. h(\ ?p :=$

$?x)$) (*lift-global-heap* $?s$)

$\llbracket IS-VALID(8 \text{ word}) (lift-global-heap ?s) (PTR-COERCE(8 \text{ signed word} \rightarrow 8 \text{ word}) ?p); \text{length } ?bs = \text{size-of } TYPE(8 \text{ signed word}) \rrbracket \Longrightarrow lift-global-heap (t-hrs-'-update (hrs-mem-update (heap-update-padding ?p ?x ?bs)) ?s) = heap-w8-update (\lambda h. h(PTR-COERCE(8 \text{ signed word} \rightarrow 8 \text{ word}) ?p := UCAST(8 \text{ signed} \rightarrow 8) ?x)) (lift-global-heap ?s)$

$IS-VALID(8 \text{ word}) (lift-global-heap ?s) (PTR-COERCE(8 \text{ signed word} \rightarrow 8 \text{ word}) ?p) \Longrightarrow lift-global-heap (t-hrs-'-update (hrs-mem-update (heap-update ?p ?x)) ?s) = heap-w8-update (\lambda h. h(PTR-COERCE(8 \text{ signed word} \rightarrow 8 \text{ word}) ?p := UCAST(8 \text{ signed} \rightarrow 8) ?x)) (lift-global-heap ?s)$

$\llbracket IS-VALID(32 \text{ word ptr}) (lift-global-heap ?s) ?p; \text{length } ?bs = \text{size-of } TYPE(32 \text{ word ptr}) \rrbracket \Longrightarrow lift-global-heap (t-hrs-'-update (hrs-mem-update (heap-update-padding ?p ?x ?bs)) ?s) = heap-w32'ptr-update (\lambda h. h(?p := ?x)) (lift-global-heap ?s)$

$IS-VALID(32 \text{ word ptr}) (lift-global-heap ?s) ?p \Longrightarrow lift-global-heap (t-hrs-'-update (hrs-mem-update (heap-update ?p ?x)) ?s) = heap-w32'ptr-update (\lambda h. h(?p := ?x)) (lift-global-heap ?s)$

$\llbracket IS-VALID(unit \text{ ptr}) (lift-global-heap ?s) ?p; \text{length } ?bs = \text{size-of } TYPE(unit \text{ ptr}) \rrbracket \Longrightarrow lift-global-heap (t-hrs-'-update (hrs-mem-update (heap-update-padding ?p ?x ?bs)) ?s) = heap-unit'ptr-update (\lambda h. h(?p := ?x)) (lift-global-heap ?s)$

$IS-VALID(unit \text{ ptr}) (lift-global-heap ?s) ?p \Longrightarrow lift-global-heap (t-hrs-'-update (hrs-mem-update (heap-update ?p ?x)) ?s) = heap-unit'ptr-update (\lambda h. h(?p := ?x)) (lift-global-heap ?s)$

$\llbracket IS-VALID(data-C) (lift-global-heap ?s) ?p; \text{length } ?bs = \text{size-of } TYPE(data-C) \rrbracket \Longrightarrow lift-global-heap (t-hrs-'-update (hrs-mem-update (heap-update-padding ?p ?x ?bs)) ?s) = heap-data-C-update (\lambda h. h(?p := ?x)) (lift-global-heap ?s)$

$IS-VALID(data-C) (lift-global-heap ?s) ?p \Longrightarrow lift-global-heap (t-hrs-'-update (hrs-mem-update (heap-update ?p ?x)) ?s) = heap-data-C-update (\lambda h. h(?p := ?x)) (lift-global-heap ?s)$

$\llbracket IS-VALID(closed-C) (lift-global-heap ?s) ?p; \text{length } ?bs = \text{size-of } TYPE(closed-C) \rrbracket \Longrightarrow lift-global-heap (t-hrs-'-update (hrs-mem-update (heap-update-padding ?p ?x ?bs)) ?s) = heap-closed-C-update (\lambda h. h(?p := ?x)) (lift-global-heap ?s)$

$IS-VALID(closed-C) (lift-global-heap ?s) ?p \Longrightarrow lift-global-heap (t-hrs-'-update (hrs-mem-update (heap-update ?p ?x)) ?s) = heap-closed-C-update (\lambda h. h(?p := ?x)) (lift-global-heap ?s)$

$\llbracket IS-VALID(unpacked-C) (lift-global-heap ?s) ?p; \text{length } ?bs = \text{size-of } TYPE(unpacked-C) \rrbracket \Longrightarrow lift-global-heap (t-hrs-'-update (hrs-mem-update (heap-update-padding ?p ?x ?bs)) ?s) = heap-unpacked-C-update (\lambda h. h(?p := ?x)) (lift-global-heap ?s)$

$IS-VALID(unpacked-C) (lift-global-heap ?s) ?p \Longrightarrow lift-global-heap (t-hrs-'-update (hrs-mem-update (heap-update ?p ?x)) ?s) = heap-unpacked-C-update (\lambda h. h(?p := ?x)) (lift-global-heap ?s)$

$\llbracket \text{valid-array-base.valid-array } (\lambda h. \text{IS-VALID}(\text{unpacked-C}) h) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{unpacked-C}[3]) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p ?x ?bs)) ?s) = \text{unpacked-C.heap-array-map } ?p (\lambda-. ?x) (\text{lift-global-heap } ?s)$

$\text{valid-array-base.valid-array } (\lambda h. \text{IS-VALID}(\text{unpacked-C}) h) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update } ?p ?x)) ?s) = \text{unpacked-C.heap-array-map } ?p (\lambda-. ?x) (\text{lift-global-heap } ?s)$

$\llbracket \text{valid-array-base.valid-array } (\text{valid-array-base.valid-array } (\lambda h. \text{IS-VALID}(\text{unpacked-C}) h)) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{unpacked-C}[3][2]) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p ?x ?bs)) ?s) = \text{unpacked-C.outer.heap-array-map } ?p (\lambda-. ?x) (\text{lift-global-heap } ?s)$

$\text{valid-array-base.valid-array } (\text{valid-array-base.valid-array } (\lambda h. \text{IS-VALID}(\text{unpacked-C}) h)) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update } ?p ?x)) ?s) = \text{unpacked-C.outer.heap-array-map } ?p (\lambda-. ?x) (\text{lift-global-heap } ?s)$

$\llbracket \text{IS-VALID}(\text{two-dimensional-C}) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{two-dimensional-C}) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p ?x ?bs)) ?s) = \text{heap-two-dimensional-C-map } ?p (\lambda-. ?x) (\text{lift-global-heap } ?s)$

$\text{IS-VALID}(\text{two-dimensional-C}) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update } ?p ?x)) ?s) = \text{heap-two-dimensional-C-map } ?p (\lambda-. ?x) (\text{lift-global-heap } ?s)$

$\llbracket \text{IS-VALID}(\text{data-struct1-C}) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{data-struct1-C}) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p ?x ?bs)) ?s) = \text{heap-data-struct1-C-map } ?p (\lambda-. ?x) (\text{lift-global-heap } ?s)$

$\text{IS-VALID}(\text{data-struct1-C}) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update } ?p ?x)) ?s) = \text{heap-data-struct1-C-map } ?p (\lambda-. ?x) (\text{lift-global-heap } ?s)$

$\llbracket \text{IS-VALID}(\text{data-struct2-C}) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{data-struct2-C}) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p ?x ?bs)) ?s) = \text{heap-data-struct2-C-map } ?p (\lambda-. ?x) (\text{lift-global-heap } ?s)$

$\text{IS-VALID}(\text{data-struct2-C}) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update } ?p ?x)) ?s) = \text{heap-data-struct2-C-map } ?p (\lambda-. ?x) (\text{lift-global-heap } ?s)$

$\llbracket \text{IS-VALID}(\text{inner-C}) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{inner-C}) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p ?x ?bs)) ?s) = \text{heap-inner-C-map } ?p (\lambda-. ?x) (\text{lift-global-heap } ?s)$

$\text{IS-VALID}(\text{inner-C}) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update } ?p ?x)) ?s) = \text{heap-inner-C-map } ?p (\lambda-. ?x) (\text{lift-global-heap } ?s)$

$\llbracket \text{valid-array-base.valid-array } (\lambda h. \text{IS-VALID}(\text{inner-C}) h) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{inner-C}) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p ?x ?bs)) ?s) = \text{heap-inner-C-map } ?p (\lambda-. ?x) (\text{lift-global-heap } ?s)$

$?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{inner-C}[5]) \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p ?x ?bs)) ?s) = \text{inner-C.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\text{valid-array-base.valid-array } (\lambda h. \text{IS-VALID}(\text{inner-C}) h) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update } ?p ?x)) ?s) = \text{inner-C.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\llbracket \text{IS-VALID}(\text{outer-array-C}) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{outer-array-C}) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p ?x ?bs)) ?s) = \text{heap-outer-array-C-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\text{IS-VALID}(\text{outer-array-C}) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update } ?p ?x)) ?s) = \text{heap-outer-array-C-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\llbracket \text{valid-array-base.valid-array } (\lambda h. \text{IS-VALID}(\text{data-C}) h) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{data-C}[10]) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p ?x ?bs)) ?s) = \text{data-C.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\text{valid-array-base.valid-array } (\lambda h. \text{IS-VALID}(\text{data-C}) h) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update } ?p ?x)) ?s) = \text{data-C.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\llbracket \text{IS-VALID}(\text{data-array-C}) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{data-array-C}) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p ?x ?bs)) ?s) = \text{heap-data-array-C-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\text{IS-VALID}(\text{data-array-C}) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update } ?p ?x)) ?s) = \text{heap-data-array-C-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\llbracket \text{IS-VALID}(\text{outer-C}) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{outer-C}) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p ?x ?bs)) ?s) = \text{heap-outer-C-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\text{IS-VALID}(\text{outer-C}) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update } ?p ?x)) ?s) = \text{heap-outer-C-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\llbracket \text{IS-VALID}(\text{other-C}) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{other-C}) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p ?x ?bs)) ?s) = \text{heap-other-C-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\text{IS-VALID}(\text{other-C}) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update } ?p ?x)) ?s) = \text{heap-other-C-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\llbracket \text{valid-array-base.valid-array } (\lambda h. \text{IS-VALID}(\text{unpacked-C}) h) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{unpacked-C}[2]) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update-padding } ?p ?x ?bs)) ?s) = \text{unpacked-C.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\text{valid-array-base.valid-array } (\lambda h. \text{IS-VALID}(\text{unpacked-C}) h) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (\text{hrs-mem-update } (\text{heap-update } ?p ?x)) ?s) = \text{unpacked-C.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$

$?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (hrs\text{-mem-update } (heap\text{-update } ?p ?x)) ?s) = \text{unpacked-C.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\llbracket IS\text{-VALID}(\text{array-C}) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{array-C}) \rrbracket$
 $\implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (hrs\text{-mem-update } (heap\text{-update-padding } ?p ?x ?bs)) ?s) = \text{heap-array-C-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $IS\text{-VALID}(\text{array-C}) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (hrs\text{-mem-update } (heap\text{-update } ?p ?x)) ?s) = \text{heap-array-C-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\llbracket IS\text{-VALID}(\text{unpacked-C}[3][2]) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{unpacked-C}[3][2]) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (hrs\text{-mem-update } (heap\text{-update-padding } ?p ?x ?bs)) ?s) = \text{unpacked-C.outer.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $IS\text{-VALID}(\text{unpacked-C}[3][2]) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (hrs\text{-mem-update } (heap\text{-update } ?p ?x)) ?s) = \text{unpacked-C.outer.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\llbracket IS\text{-VALID}(\text{inner-C}[5]) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{inner-C}[5]) \rrbracket$
 $\implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (hrs\text{-mem-update } (heap\text{-update-padding } ?p ?x ?bs)) ?s) = \text{inner-C.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $IS\text{-VALID}(\text{inner-C}[5]) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (hrs\text{-mem-update } (heap\text{-update } ?p ?x)) ?s) = \text{inner-C.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\llbracket IS\text{-VALID}(\text{data-C}[10]) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{data-C}[10]) \rrbracket$
 $\implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (hrs\text{-mem-update } (heap\text{-update-padding } ?p ?x ?bs)) ?s) = \text{data-C.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $IS\text{-VALID}(\text{data-C}[10]) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (hrs\text{-mem-update } (heap\text{-update } ?p ?x)) ?s) = \text{data-C.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $\llbracket IS\text{-VALID}(\text{unpacked-C}[2]) (\text{lift-global-heap } ?s) ?p; \text{length } ?bs = \text{size-of TYPE}(\text{unpacked-C}[2]) \rrbracket \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (hrs\text{-mem-update } (heap\text{-update-padding } ?p ?x ?bs)) ?s) = \text{unpacked-C.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s)$
 $IS\text{-VALID}(\text{unpacked-C}[2]) (\text{lift-global-heap } ?s) ?p \implies \text{lift-global-heap } (t\text{-hrs}'\text{-update } (hrs\text{-mem-update } (heap\text{-update } ?p ?x)) ?s) = \text{unpacked-C.heap-array-map } ?p (\lambda\text{-. } ?x) (\text{lift-global-heap } ?s).$

thm *heap-update-fold-toplevel-fields-pointless*

thm *plift-heap-update-padding-heap-update-pointless-conv*

thm *lift-heap-update-padding-heap-update-conv*

thm *write-commutes*

No restriction to *packed-type*!

In the original version of AutoCorres there was a restriction of heap lifting to types that are also in *packed-type*. Those are types that internally have no padding fields at all. The reason was that certain lemmas especially

regarding heap update become more involved in the presence of padding fields. A *heap-update* preserves the value of all padding bytes, whereas the abstract value obtained from a corresponding *h-val* is independent of the value of the padding bytes. So in principle these notions fit together well but it seems that some formal language was missing to reason about padding bytes. Meanwhile we added a theory to reason about those padding bytes in *AutoCorres2.Padding-Equivalence*. This made it possible to liberate heap lifting from the restriction to *packed-type*. For example $\text{heap-update } ?p \text{ ?arr} = (\lambda s. \text{foldl } (\lambda s n. \text{heap-update } (\text{array-ptr-index } ?p \text{ False } n) \text{ (?arr.[n]) } s) s [0..<CARD(?'b)])$ for *packed-type* holds in general $c\text{-guard } ?p \implies \text{heap-update } ?p \text{ ?arr } ?h = \text{fold } (\lambda i. \text{heap-update } (\text{array-ptr-index } ?p \text{ False } i) \text{ (?arr.[i])}) [0..<CARD(?'b)] \text{ ?h}$.

thm *heap-update-array*
thm *heap-update-Array*

Padding Equivalence and *xmem-type*

Padding bytes and padding fields are introduced via the construction of new C-Types via the combinators *ti-pad-combine* and *ti-typ-pad-combine* to satisfy alignment properties. However, the concept of a padding byte is not a first-class citizen of a $(\text{'a'}, \text{'b'})$ *typ-desc*, but just happens to be a field with some special properties. In practice these fields are generated by *ti-pad-combine*, so we know that the field name starts with *"!pad"*, and the associated 'lense' for lookup and update have the 'passthrough' properties of a padding field.

We made these properties explicit by supplying a theory to identify padding-bytes in a list of bytes associated with an C-Type, and having notions to compare such byte-lists, e.g. telling if two byte lists are equal up-to the padding bytes, or if they agree on the padding bytes. This information is also backed into the properties of *xmem-type*, which is a subclass of *mem-type*. All primitive word and pointer types as well as all types that are constructed by the UMM module of the C-Parser are proved to belong to that class. The construction of array types propagates the class as expected.

Again the notions come in pairs of a 'typed' and a 'untyped' version. The 'typed' version associated with *typ-info-t* is a lense based formalisation, the 'untyped' version associated with *typ-uinfo-t* is based on byte lists.

The lense based version is introduced by *padding-lense*. Access and update follow the approach of *'a field-desc*. Lookup / access has type $\text{'a} \Rightarrow \text{byte list} \Rightarrow \text{byte list}$. It takes an abstract value and a supply list of padding bytes and retrieves the bytes encoding the field. The intuition of the padding byte list is the following. It is supposed to bridge the gap between the abstract value and the byte representation. The abstract value is independent of the padding bytes. Hence in general there is no one-to-one correspondence between a byte list encoding of an abstract value and the abstract value.

When converting from a byte list to an abstract value this is fine, the padding bytes are just ignored. But when we go the other way we have to invent the padding bytes. One solution could be to just fill up with zeros. Another solution could be to yield a non-deterministic result for the padding bytes. The UMM model chose another way. The possible valuation for padding bytes is supplied as an additional argument. So whenever a padding byte is needed we just take it from the position in that list. The update has type *byte list* \Rightarrow 'a \Rightarrow 'a. It takes the value of a field, encoded as a byte list, and transforms it to an update on the abstract value.

context *padding-lense*
begin

The concept of a *padding-lense* follows the signature of 'a *field-desc* and describes the notions of padding bytes as semantic properties of the access function *acc* and the update function *upd*. Based on them it introduces the notions of

- *local.is-padding-byte*, is a byte a padding with respect to the lense?
- *local.is-value-byte*, is a byte a value byte with respect to the lense, i.e. does the abstract value associated with the byte list depend on that byte.
- *local.eq-padding*, access cannot distinguish the byte lists.
- *local.eq-upto-padding*, update cannot distinguish the byte list.

thm *is-padding-byte-def*
thm *is-value-byte-def*
thm *eq-padding-def*
thm *eq-upto-padding-def*

As the definitions are semantically defined the effect on *acc* and *upd* are rather immediate. For example, if two byte lists are equal upto padding, then an update with *upd* yields the same result. If the padding bytes within two supply byte lists are the same, then *acc v* yields the same result for both byte lists.

end

In the untyped *typ-uinfo-t* we define the corresponding notions, as properties of the byte list instead of the access and update functions.

- *is-padding-byte*, a byte is a padding byte in a byte list, if normalisation of the byte list is independent of its value. Normalisation *norm-tu* is defined on *typ-uinfo-t* and sets all padding bytes to zero.
- *is-value-byte*, normalisation depends on the value of the byte.

- *eq-padding*, all padding bytes are the same.
- *eq-upto-padding*, all value bytes are the same.

```

thm is-padding-byte-def
thm is-value-byte-def
thm eq-padding-def
thm eq-upto-padding-def

```

For instances of *xmem-type* we can go back and forth between both characterisations.

```

thm is-padding-byte-lense-conv
thm field-lookup-is-padding-byte-lense-conv
thm is-value-byte-lense-conv
thm field-lookup-is-value-byte-lense-conv
thm eq-padding-lense-conv
thm field-lookup-eq-padding-lense-conv
thm eq-upto-padding-lense-conv
thm field-lookup-eq-upto-padding-lense-conv

```

With those theorems in place it is easy to show that padding fields only consist of padding bytes and thus do not account to the abstract value.

```

thm field-lookup-access-ti-to-bytes-field-conv
thm access-ti-update-ti-eq-upto-padding
thm field-lookup-qualified-padding-field-name(1)
thm is-padding-tag-def padding-tag-def padding-desc-def

```

UMM-Simprocs Cache

To solve sideconditions on UMM-Types we have implemented some simprocs within `UMM_Proofs`. Currently their scope is limited to the usecases we have in the proofs described above. It should be straightforward to extend them to more usecases. Their purpose is not to support abstract reasoning on types but to provide properties of concrete instances of C-types, for example deciding whether $TYPE(32\ word) \leq_{\tau} TYPE(outer-C)$ holds.

The benefit of the simprocs is that we do not have to guess and prove lots of lemmas already during the definition of a new UMM type, but can postpone it until we actually need them. Once they are proven they are added to the cache, so they are only proven once. In our use-case the lemmas we need depend on the addressable fields the user specifies on an `autocorres` invocation.

```

ML-val <
  val - = simproc <field-lookup>
  val - = simproc <type-calculations>
  val - = simproc <typuinfo-calculations>
  >

```



```

ML-val <
  Cached-Theory-Simproc.trace-cache @{context}
>

```

The simprocs make use of a common cache. The cache itself is implemented as a simpset. Cache lookup means we try to rewrite with the cache:

- cache miss: term is unchanged.
- cache hit: term is changed.

In case of a cache hit we are finished and return the resulting equation. In case of a cache miss we invoke the simplifier with a tailored simpset to derive a new equation. This equation is then added to the cache as well as returned from the simproc.

Some fine points of this setup are related to context management. Conceptually we only prove theory level theorems about an UMM type. So even if we prove them after the definition of the UMM type they should all be applicable as if we have immediately proven them at the point where the UMM type was generated. However, the simproc is invoked in some context later on where the theory has already advanced. In order to produce a result after a cache miss we have to somehow 'travel back in time' to the original theory context after the definition of the UMM type, such that the simplifier properly works on it and the result is reusable in other contexts. We maintain that original theory state and recertify terms before simplifying them.

Another point is, that the terms we attempt to simplify and cache might depend on each other. In order not to miss to cache intermediate results one has to carefully craft the simpsets.

In general the simplifier tries simprocs only after unconditional and conditional rules. So when a rewrite rule has already transformed the redex the simproc will never see that redex. This has two main implications:

- In order to have the simproc-based cache applied, there must not be a rule in the simpset that removes the redex before the cache actually has a chance to see it. This is why we maintain several simpsets to control that behaviour.
- When a cache miss was encountered and we apply the simplifier recursively to rewrite the redex we have to take care that the simproc is not invoked again on the same redex. When there is an unconditional rule in the simpset that rewrites that redex we are on the safe side. But beware of *conditional* rules. When the solver fails to solve the conditions the rule can fail and then the simproc is invoked again on

the same redex. To prevent the setup from looping in these cases we maintain the redexes the simproc has already seen.

Here are some principles in the design of the simprocs and their simpsets:

- The type descriptions obtained by *typ-info-t* are not fully expanded. Instead we use a simplifier setup that works on the combinators, like *empty-typ-info*, *final-pad*, *ti-typ-pad-combine*. This maintains the more compact representation of a type-descriptor as a combinator expression, which is anyways the way it is originally defined.
- Fieldnames for array indexes stay symbolic: *replicate i "1"*. We employ derived rules for array indexes and do not have to expand the type-descriptor of arrays. For these rules to apply we typically need the information that the index is in the bounds of the array type. In those cases the simproc generates a conditional rewrite rule.
- To test equality on *export-uinfo t = export-uinfo s* we normalise towards *export-uinfo (typ-info-t TYPE('a)) = export-uinfo (typ-info-t TYPE('b))*, where both 'a and 'b are concrete instances. Then when both expressions happen to be the same we have proved equality. Note that this approach is incomplete and in particular not sufficient to disprove the equality and prove the inequality. However, as these equality often appears as sidecondition in a conditional rewrite rule, not being able to prove the equality is somehow "equivalent" to disproving it: If we cannot prove the sidecondition the rule cannot be applied. The most prominent pattern for this is a *field-lookup* yielding the type-description of the selected field which is then compared to some type-description that is derived from a pointer type. To also disprove equality we make use of rule $(\text{export-uinfo } (typ\text{-info-t } TYPE(?'a)) = \text{export-uinfo } (typ\text{-info-t } TYPE(?'b))) = (TYPE(?'a) \leq_{\tau} TYPE(?'b) \wedge TYPE(?'b) \leq_{\tau} TYPE(?'a))$ and use the simproc on *sub-typ*.
- To decide whether e.g. $TYPE(32\ word) \leq_{\tau} TYPE(outer-C)$ holds or not, we first try to disprove it by using type name information $\llbracket typ\text{-name } (typ\text{-info-t } TYPE(?'a)) \neq pad\text{-typ-name}; typ\text{-name } (typ\text{-info-t } TYPE(?'a)) \notin td\text{-names } (typ\text{-info-t } TYPE(?'b)) \rrbracket \implies \neg TYPE(?'a) \leq_{\tau} TYPE(?'b)$. both *typ-name* and *td-names* are supplied by the UMM package: *typ-name (typ-info-t TYPE(data-struct2-C))* \equiv "data-struct2-C"
typ-name (typ-info-t TYPE(data-struct1-C)) \equiv "data-struct1-C"
typ-name (typ-info-t TYPE(data-array-C)) \equiv "data-array-C"
typ-name (typ-info-t TYPE(outer-array-C)) \equiv "outer-array-C"
typ-name (typ-info-t TYPE(two-dimensional-C)) \equiv "two-dimensional-C"
typ-name (typ-info-t TYPE(other-C)) \equiv "other-C"

$typ\text{-}name (typ\text{-}info\text{-}t \text{TYPE}(array\text{-}C)) \equiv "array\text{-}C"$
 $typ\text{-}name (typ\text{-}info\text{-}t \text{TYPE}(outer\text{-}C)) \equiv "outer\text{-}C"$
 $typ\text{-}name (typ\text{-}info\text{-}t \text{TYPE}(unpacked\text{-}C)) \equiv "unpacked\text{-}C"$
 $typ\text{-}name (typ\text{-}info\text{-}t \text{TYPE}(inner\text{-}C)) \equiv "inner\text{-}C"$
 $typ\text{-}name (typ\text{-}info\text{-}t \text{TYPE}(closed\text{-}C)) \equiv "closed\text{-}C"$
 $typ\text{-}name (typ\text{-}info\text{-}t \text{TYPE}(data\text{-}C)) \equiv "data\text{-}C", td\text{-}names (typ\text{-}info\text{-}t ?x) \equiv \{"data\text{-}C", "word0000010", "data\text{-}struct2\text{-}C", "data\text{-}struct1\text{-}C"\}$
 $td\text{-}names (typ\text{-}info\text{-}t ?x) \equiv \{"data\text{-}C", "word0000010", "data\text{-}struct1\text{-}C"\}$
 $td\text{-}names (typ\text{-}info\text{-}t ?x) \equiv \{"data\text{-}C", "word0000010", "data\text{-}array\text{-}C", "data\text{-}C\text{-}array\text{-}01010"\}$
 $td\text{-}names (typ\text{-}info\text{-}t ?x) \equiv \{"inner\text{-}C", "closed\text{-}C", "word0000010", "outer\text{-}array\text{-}C", "inner\text{-}C\text{-}array\text{-}1010"\}$
 $td\text{-}names (typ\text{-}info\text{-}t ?x) \equiv \{"word00010", "unpacked\text{-}C", "word0000010", "two\text{-}dimensional\text{-}C", "unpacked\text{-}C\text{-}array\text{-}110", "unpacked\text{-}C\text{-}array\text{-}110\text{-}array\text{-}010"\}$
 $td\text{-}names (typ\text{-}info\text{-}t ?x) \equiv \{"other\text{-}C", "closed\text{-}C", "word00010", "unpacked\text{-}C", "word0000010"\}$
 $td\text{-}names (typ\text{-}info\text{-}t ?x) \equiv \{"array\text{-}C", "word00010", "unpacked\text{-}C", "word0000010", "unpacked\text{-}C\text{-}array\text{-}010"\}$
 $td\text{-}names (typ\text{-}info\text{-}t ?x) \equiv \{"inner\text{-}C", "outer\text{-}C", "closed\text{-}C", "word0000010"\}$
 $td\text{-}names (typ\text{-}info\text{-}t ?x) \equiv \{"word00010", "unpacked\text{-}C", "word0000010"\}$
 $td\text{-}names (typ\text{-}info\text{-}t ?x) \equiv \{"inner\text{-}C", "closed\text{-}C", "word0000010"\}$
 $td\text{-}names (typ\text{-}info\text{-}t ?x) \equiv \{"closed\text{-}C", "word0000010"\}$
 $td\text{-}names (typ\text{-}info\text{-}t ?x) \equiv \{"data\text{-}C", "word0000010"\}$. By construction every distinct type gets an individual name. If that does not work we try proving it instead, by using a transitivity prover on the single step sub-type relations provided by $\text{TYPE}(32 \text{ signed word}) \leq_{\tau} \text{TYPE}(data\text{-}C)$

$\text{TYPE}(32 \text{ word}) \leq_{\tau} \text{TYPE}(data\text{-}C)$
 $\text{TYPE}(32 \text{ word}) \leq_{\tau} \text{TYPE}(closed\text{-}C)$
 $\text{TYPE}(32 \text{ word}) \leq_{\tau} \text{TYPE}(inner\text{-}C)$
 $\text{TYPE}(closed\text{-}C) \leq_{\tau} \text{TYPE}(inner\text{-}C)$
 $\text{TYPE}(8 \text{ word}) \leq_{\tau} \text{TYPE}(unpacked\text{-}C)$
 $\text{TYPE}(32 \text{ word}) \leq_{\tau} \text{TYPE}(unpacked\text{-}C)$
 $\text{TYPE}(inner\text{-}C) \leq_{\tau} \text{TYPE}(outer\text{-}C)$
 $\text{TYPE}(32 \text{ word}) \leq_{\tau} \text{TYPE}(outer\text{-}C)$
 $\text{TYPE}(unpacked\text{-}C[2]) \leq_{\tau} \text{TYPE}(array\text{-}C)$

$TYPE(32\ word) \leq_{\tau} TYPE(array-C)$
 $TYPE(32\ word) \leq_{\tau} TYPE(other-C)$
 $TYPE(closed-C) \leq_{\tau} TYPE(other-C)$
 $TYPE(unpacked-C) \leq_{\tau} TYPE(other-C)$
 $TYPE(unpacked-C[3][2]) \leq_{\tau} TYPE(two-dimensional-C)$
 $TYPE(inner-C[5]) \leq_{\tau} TYPE(outer-array-C)$
 $TYPE(32\ word) \leq_{\tau} TYPE(outer-array-C)$
 $TYPE(data-C[10]) \leq_{\tau} TYPE(data-array-C)$
 $TYPE(data-C) \leq_{\tau} TYPE(data-struct1-C)$
 $TYPE(data-struct1-C) \leq_{\tau} TYPE(data-struct2-C)$ and the rule for arrays
 $TYPE(?a) \leq_{\tau} TYPE(?a[?b])$.

thm *not-sub-typ-via-td-name*
thm *typ-name-simps*
thm *td-names-simps*
thm *sub-typ-simps*
thm *element-typ-subtyp-array-typ*

Here some examples

lemma *export-uinfo* (*typ-info-t* $TYPE(array-C)$) = *export-uinfo* (*typ-info-t* $TYPE(other-C)$)
apply *simp*
oops

schematic-goal *field-lookup* (*typ-info-t* $TYPE(outer-C)$) ["inner-C", "fld1-C"] 0
= ?X
apply *simp*
done

lemma $TYPE(8\ word) \leq_{\tau} TYPE(outer-C) = False$
by *simp*

schematic-goal $i < 2 \implies j < 3 \implies$
field-ti $TYPE(two-dimensional-C)$ ["matrix-C", replicate *i* CHR "1", replicate *j*
CHR "1"] = ?X
apply *simp*
done

lemma $TYPE(8\ word) \leq_{\tau} TYPE(8\ word[42])$
apply *simp*
done

ML-val ‹
Cached-Theory-Simproc.trace-cache @{context}
›

26.21.6 Examples for normalisation of array indexes

lemma

```

fixes p::array-C ptr
assumes bnd[simp]: i < 2
shows (PTR(unpacked-C) &(p→["elements-C", replicate i CHR "1"])) = PTR(unpacked-C)
&(p→["elements-C"]) +p int i
supply [[simp-trace=false, verbose=5]]
apply (array-index-to-ptr-arith-simp)
done

```

lemma

```

fixes p::array-C ptr
assumes bnd: (i::32 word) < 2
shows (PTR(unpacked-C) &(p→["elements-C", replicate (unat i) CHR "1"]))
= PTR(unpacked-C) &(p→["elements-C"]) +p uint i
supply [[simp-trace=false, verbose=5]]
apply (array-index-to-ptr-arith-simp simp: bnd)
done

```

lemma

```

fixes p::array-C ptr
shows do {- <- guard (λ-. unat (i::32 word) < 2);
return ((PTR(unpacked-C) &(p→["elements-C", replicate (unat i) CHR "1"])))}
=
do {
- <- guard (λs. unat i < 2);
return (PTR(unpacked-C) &(p→["elements-C"]) +p uint i)
}
supply [[simp-trace=false, verbose=6]]
apply array-index-to-ptr-arith-simp
done

```

lemma

```

fixes p::array-C ptr
shows do {- <- guard (λ-. (i::32 word) < 2);
return ((PTR(unpacked-C) &(p→["elements-C", replicate (unat i) CHR "1"])))}
=
do {
- <- guard (λs. i < 2);
return (PTR(unpacked-C) &(p→["elements-C"]) +p uint i)
}
supply [[simp-trace=false, verbose=6]]
apply array-index-to-ptr-arith-simp
done

```

lemma

```

fixes p::array-C ptr

```

```

assumes bnd[simp]: i < 2
shows (PTR(unpacked-C) &(p→["elements-C", replicate i CHR "1", "chr-C"]))
=
  PTR(unpacked-C)
  &(PTR(unpacked-C) &(p→["elements-C"]) +p int i→["chr-C"])
supply [simp-trace=false, verbose=5]
apply (array-index-to-ptr-arith-simp)
done

lemma (i::32 word) < 4  $\implies$  unat i < 4
supply [array-bound-mksimps, verbose=5]
apply (simp)
done

lemma (i::32 signed word) < s 4  $\wedge$   $0 \leq s\ i \implies$  nat (sint i) < 4
supply [array-bound-mksimps, verbose=5]
apply (simp)
done

```

26.21.7 Essence of Heap Lifting

Some birds eye view on what heap lifting is about. Consider the following C program.

```

typedef struct foo {
    int myint;
    bool mybool;
} foo;

int * p;
foo * q;

(*p) = 42;
i = q->myint;
b = q->mybool;

```

Is value *i* and *b* affected by update to **p*
 From the C-Parser we only get *c-guard p* and *c-guard q*:
 {*c_guard p* \<and> *c_guard q*}
 (*p) = 42;
 i = q->myint;
 b = q->mybool;

This only tells us something about alignment of pointers and that the pointer span does not overflow, but nothing about disjointness of the pointer spans.

What about stronger guarantees, if we assume well-typedness of the pointers:

```
{d \<Turnstile>\<^sub>t p \<and> d \<Turnstile>\<^sub>t p q}
(*p) = 42;
i = q->myint;
b = q->mybool;
```

Now we know that p might alias with $q->myint$ as they have the same type, but that pointer span p is actually disjoint from pointer span $q->mybool$. Hence for the read of b we can skip the heap update via p .

Now consider that the structure *foo* is a closed structure in the split heap model. This means that we have even stronger assumptions (which are guaranteed by guards in the lifted code):

```
{root_ptr_valid d p \<and> root_ptr_valid p q}
(*p) = 42;
i = q->myint;
b = q->mybool;
```

From this we can infer that the pointer spans p and the entire pointer span q don't overlap. hence reading i as well as b can skip the heap update via p .

In general the heap lifting phase introduces the potentially weaker guards for open types:

```
{ptr_valid d p \<and> ptr_valid p q}
(*p) = 42;
i = q->myint;
b = q->mybool;
```

So whether i might be affected by p depends on whether field *myint* is addressable or not.

The essential semantic part of heap lifting is to introduce these guards at every pointer access. In a second step these guards then allow a change in a representation from the monolithic heap to the split heap, with a separate heap for each atomic type. But in a sense this second step could be viewed as optional or cosmetic in the semantic sense. There is all the information available to perform the "skipping" steps as sketched above in the monolithic heap. This is what is performed in the simulation proof. So the question is if we could omit introduction of "lifted globals" completely and provide the necessary automation to the user to simplify accesses on the monolithic heap that mimic that behaviour in the split heap. In a sense instead of doing the simulation proof upfront it is done adhoc at every heap access / update. This avoids to provide quadratically many commutation lemmas (in the number of fields of open types) that are introduces for the lifted

globals.

From the perspective of the simulation proof alone such a "virtual" split heap, which does not introduce a new lifted-globals type, seems to be equivalent to the one which introduces a new type with concrete distinct record fields for each fundamental heap. The simulation proof only works when every pointer access is guarded and under that assumption the two representations behave the same.

However, besides the simulation there are other aspects of the split heap that might pay off for verification or further abstractions. The two models behave differently for invalid pointers: In the split-heap (with different record fields), updates to one heap still do not affect other heaps but we cannot properly model such updates in the monolithic heap at all.

Another aspect was introduced later on for stack allocation: non-deterministic padding bytes. The *typ-heap-simulation* was generalised to state that lifting to the split heap is invariant for arbitrary padding bytes, i.e. *write-simulation*. The lifted heap only cares about the abstract values were the values of the padding bytes become irrelevant.

thm *h-val-heap-update-padding*

thm *plift-heap-update-padding-heap-update-pointless-conv*

thm *lift-heap-update-padding-heap-update-conv*

context *open-struct-all-corres*

begin

thm *ts-def*

thm *ac-corres*

end

end

theory *AutoCorres-Documentation*

imports

AutoCorresInfrastructure

pointers-to-locals

In-Out-Parameters-Ex

fnptr

union-ac

open-struct

begin

end

Chapter 27

C-Translation Infrastructure

```
theory CTranslationInfrastructure
  imports
    CTranslation
begin
```

```
term guarded-spec-body
```

27.1 Local Variables

Local variables in the SIMPL outcome of the C-Parser are represented in *locals*. The natural number in the domain encodes the canonical position in the scope of the C-function: input arguments, then return variable, then local variables. The byte list in the range of the function is the encoding of the value according to the UMM (unified memory model) *c-type*. The typing of variables is maintained as data in `CLocals` and a typed view is achieved via *lookup* and *cupdate*.

In previous attempts local variables were represented first as a **record** and more recently as **statespace**. The representation as *locals* tries to combine the best of all previous approaches:

- Uniform (program independent) representation of local variables.
- Positional 'naming' enables a canonical parameter passing for function calls, even if the names of parameters are unknown, as in the case of function-pointers.
- Lightweight "typing" via ML infrastructure instead of the 'fixes' of **statespace**. Unfortunately, the more flexible **statespace** representation for local variables turned out to reveal some performance bottleneck in the locale infrastructure, which itself boils down to the resources consumed by the omnipresent instantiation of types and terms.

```
declare [[ML-print-depth = 1000]]
```

27.1.1 Basic ML primitives

To define new variables the basic primitive is `CLocals.define_locals` which takes a scope (currently a two element list of qualifiers: program-name followed by function-name) and the list of local variable declarations (name, type and kind). Note that the order is relevant as it represents the canonical positional ordering.

```
setup <CLocals.define-locals [prog, myfun] [  
  (x, @{typ <32 word>}, NameGeneration.Loc),  
  (p, @{typ <32 word ptr>}, NameGeneration.Loc),  
  (z, @{typ <64 word>}, NameGeneration.Loc)] >
```

```
ML <CLocals.Data.get (Context.Proof @{context})>
```

For each local variable a constant is defined with the position of the variable. The constant name encodes the original name of the local variable. Whenever available we use this symbolic name in favour of the plain literal number. However, at some places we use the plain numbers, in particular for parameter passing. There is some simplifier setup which we explain later that allows to use either representation.

To enable syntax translations from short names to the symbolic constants together with their typing one has to enter the correct scope, e.g.:

```
local-setup <  
Local-Theory.declaration {pervasive=false, syntax=true, pos = here} (fn - =>  
  CLocals.scope-map (K [prog, myfun]))  
>
```

```
term prog.myfun.p-'  
thm prog.myfun.p'-def
```

Some remarks on the naming scheme for those symbolic constants. The qualifiers for both program-name and function-name are mandatory. The name mangling with the suffix serves two purposes:

- Avoid name clashes with user input via variable names in the C sources (in particular for the artificial return variable)
- Avoid name clashes for bound or free variables (or new constants) in HOL. Note that even a constant with mandatory qualifier e.g. `foo.c` would force the pretty printer to avoid a bound variable being named just as `c`. The pretty printer only takes the `Long_Name.base_name` into account.

The general suffix `-'` avoids clashes with C variable names as the prime is not a legal character for C identifiers. Syntax translations will truncate this suffix. The artificial return variable `ret` that the C-Parser introduces is internally `ret'-'` to avoid a potential conflict with a local variable named `ret`.

ML $\langle CLocals.Data.get (Context.Proof @\{context\}) \rangle$

27.1.2 Syntax

Plain lookup and update in locals.

term $\langle l \cdot p \rangle$

ML-val $\langle term \langle l \cdot p \rangle \rangle$

term $\langle s \langle x := y \rangle \rangle$

term $\langle s \langle x := y, p := k \rangle \rangle$

ML-val $\langle term \langle s \langle x := y \rangle \rangle \rangle$

ML-val $\langle term \langle s \langle x := y, p := k \rangle \rangle \rangle$

Lookup and update in a complete Simpl state, selecting the *locals*

term $\langle s \cdot_{\mathcal{L}} p \rangle$

ML-val $\langle @\{term \langle s \cdot_{\mathcal{L}} p \rangle \} \rangle$

term $\langle s \langle x :=_{\mathcal{L}} y \rangle \rangle$

term $\langle s \langle x :=_{\mathcal{L}} y, p :=_{\mathcal{L}} k \rangle \rangle$

27.1.3 Simplifier setup

The simplifier is setup to be able to deal with symbolic as well as literal numerals. Symbolic ones are expanded on the fly employing the infrastructure from the code generator. The trigger for this expansion is via a simplification procedure that. Conditional rules can be configured via the attribute:

ML $\langle @\{attributes [code-simproc]\} \rangle$

The simplification procedure first checks wheter the precondition evaluates to true, via the code-generator. In case of success the simplifier is invoked to replay that proof and trigger the rule. `Code_Simproc.code_simp_prem`s, `Code_Simproc.code_prove`.

Note that the attribute also consumes an optional name to index the simprocs. This name is taken from **named-theorems**.

The relevant setup of the rules can be found in `CLocals.thy`.

schematic-goal $\langle (lookup\ 2\ (cupdate\ prog.myfun.z^{-1}\ (\lambda-. 2::64\ word)\ l)) = (?X::64\ word) \rangle$

apply *simp*

done

schematic-goal $\langle (lookup\ prog.myfun.z^{-1}\ (cupdate\ prog.myfun.z^{-1}\ (\lambda-. 2::64\ word)\ l)) = (?X::64\ word) \rangle$

apply *simp*

done

schematic-goal $s \langle x := \lambda-. 2 \rangle \cdot x = ?X$

```
apply simp
done
```

```
schematic-goal  $s\langle x := \lambda-. \varrho \rangle \cdot p = ?X$ 
apply simp
done
```

```
install-C-file ex.c
```

```
thm size-td-simps
```

The root locale storing the global content *ex-global-addresses*. This is also the locale where the bodies of the imported functions are defined.

```
context ex-global-addresses
begin
```

The usage of global variables is analysed and three cases are distinguished:

- static variables do not change their value. So they are defined as HOL constant with their initial value. In case no initialiser is given they are zero initialised.
- 'ordinary' global variables that are also updated somewhere in the code are stored in the record of global variables *globals*. They are accessed by the lookup and update function of the **record** package.
- In case the address of a global variable is taken somewhere in the code, we declare a global constant for a pointer to that variable. Lookup and update is then indirectly via the heap. Note that the pointer is only declared. There is no defining equation. Currently there are also no assumptions maintained about distinctness of global variable pointers. This is up to the user.

```
thm global-const-defs
term g-static
term g-ordinary-' term g-ordinary-'-update
term g-addressed-'
thm main-body-def
thm globals.typing.get-upd
thm state.typing.get-upd
```

Functions also result in a declaration of a constant representing the global function pointer. *ex.add*. They have the type *unit ptr*.

```
term ex.add
```

When the code actually uses function pointers to perform indirect calls, some more infrastructure is provided, cf `../doc/fnptr.thy`

When looking into the function bodies the symbolic names for the positional local variables are displayed, fully qualified with program name as well as function name.

```
thm add-body-def
```

To have short names printed, one can either enter the scope of the function by activating the corresponding *variables* bundle, or use the option to always display short names.

```
context includes add-variables
begin
thm add-body-def
end
```

```
declare [[locals-short-names]]
thm add-body-def
end
```

The canonical locale to do verification of a procedure is the *impl* locale. This activates the scope of the function and also stores the equation that the lookup of the function pointer in the environment Γ retrieves the expected body.

```
context add-impl
begin
thm add-body-def
thm add-impl
```

The symbolic names can be folded and unfolded by an attribute.

```
thm add-body-def [unfold-locals]
thm add-body-def [unfold-locals, fold-locals]
```

These attributes can also take qualifier in case an alternative scope should be added

```
thm call-add-body-def
thm call-add-body-def [unfold-locals] — nothing happens as we are in the wrong scope
thm call-add-body-def [unfold-locals call-add] — now they are expanded as we qualify the scope

end
```

```
context call-add-impl
begin
thm call-add-body-def
declare [[hoare-use-call-tr' = false]]
thm call-add-body-def
end
```

All the implementations of the program are gathered in the *simpl* locale

```

context ex-simpl
begin
thm add-impl
thm call-add-impl
end

```

27.2 Infrastructure for states

```

type-synonym state = (globals, locals, 32 signed word) CProof.state

```

```

context add-impl
begin

```

As a final part of the verification generator generator terms containing local and global variables are generalised to a 'splitted' view, where each component becomes has a bound variable.

```

lemma
   $\exists s::state. s \cdot_{\mathcal{L}} n = \mathfrak{B}$ 
  apply (tactic  $\langle \text{HPInter.GENERALISE } @\{\text{context}\} 1 \rangle$ )
  by simp

```

```

lemma
   $\exists s::state. s \cdot_{\mathcal{L}} n = \mathfrak{B}$ 
  apply (tactic  $\langle \text{HPInter.GENERALISE } @\{\text{context}\} 1 \rangle$ )
  by simp

```

```

lemma
   $\exists s::state. s \cdot_{\mathcal{L}} n = \mathfrak{B} \wedge s \cdot_{\mathcal{L}} m = \mathfrak{B}$ 
  apply (tactic  $\langle \text{HPInter.GENERALISE } @\{\text{context}\} 1 \rangle$ )
  by simp

```

```

lemma
   $\exists s::state. s \cdot_{\mathcal{L}} n = \mathfrak{B} \wedge s \cdot_{\mathcal{L}} m = \mathfrak{B} \wedge g'(\text{globals } s) = 4 \wedge \text{global-exn-var}'\text{'-}' s =$ 
  Break
  apply (tactic  $\langle \text{HPInter.GENERALISE } @\{\text{context}\} 1 \rangle$ )
  by simp

```

Simpl syntax

```

ML  $\langle$ 
   $@\{\text{term } \langle \{ 'n = x \} \rangle \}$ 
 $\rangle$ 
term  $\langle \{ 'n = x \} \rangle$ 
ML  $\langle$ 
   $@\{\text{term } \langle 'n ::= x \rangle \}$ 
 $\rangle$ 
term  $\langle 'n ::= x \rangle$ 

```

```

term ‹'n := CALL add(x, y)›
term ‹'n := CALL ex.add(x, y)›
ML ‹@{term ‹'n := CALL add(x, y)››

```

```

ML ‹@{term ‹'ret' := PROC add(2, 3)››

```

```

term ‹'ret' := PROC add(2, 3)›

```

```

end

```

27.3 Cached simproc examples

Some more explanation on this is in `../doc/AutoCorresInfrastructure.thy`

```

schematic-goal TYPE(addr-bitsize word) ≤τ TYPE(unpacked-C[12])
  apply simp
  done

```

```

lemma TYPE(8 word) ≤τ TYPE(array-C)
  supply [[verbose=3]]
  apply simp
  done

```

```

lemma TYPE(8 word) <τ TYPE(array-C)
  supply [[verbose=3]]
  apply simp
  done

```

```

lemma TYPE(addr-bitsize word) ≤τ TYPE(matrix-C)
  supply [[verbose=3]]
  apply simp
  done

```

```

lemma TYPE(8 word) ≤τ TYPE(matrix-C)
  apply simp
  done

```

```

ML ‹
  Cached-Theory-Simproc.trace-cache @{context}
›

```

```

schematic-goal ‹
  field-names-no-padding (typ-info-t TYPE(outer-C)) (export-uinfo (typ-info-t TYPE(inner-C)))
  = ?XX›
  supply [[simp-trace=true, verbose=5]]
  apply simp

```

```

done
ML <
  Cached-Theory-Simproc.trace-cache @{context}
>

```

```

schematic-goal <
  set (field-names-no-padding (typ-info-t TYPE(outer-C)) (export-uinfo (typ-info-t
TYPE(inner-C)))) = ?XX>
  supply [[simp-trace=true, verbose=5]]
  apply simp
  done
ML <
  Cached-Theory-Simproc.trace-cache @{context}
>

```

```

lemma export-uinfo (typ-info-t TYPE(unpacked-C)) = export-uinfo (typ-info-t
(TYPE(32 word)))
  supply [[simp-trace=true, verbose=3]]
  apply simp
  oops

```

```

lemma typ-uinfo-t TYPE(unpacked-C) = export-uinfo (typ-info-t (TYPE(32 word)))
  apply simp
  oops

```

```

lemma typ-uinfo-t TYPE(32 signed word) = export-uinfo (typ-info-t (TYPE(32
word)))
  supply [[simp-trace=true, verbose=3]]
  apply simp
  done

```

```

lemma typ-uinfo-t TYPE(32 signed word[3]) = export-uinfo (typ-info-t (TYPE(32
word[3])))
  supply [[simp-trace=true, verbose=3]]
  apply simp
  done

```

```

schematic-goal <
  set (field-names-no-padding (typ-info-t TYPE(unpacked-C)) (export-uinfo (typ-info-t
TYPE(8 word)))) = ?XX>
  supply [[simp-trace=true, verbose=3]]
  apply simp
  done

```

```

lemma export-uinfo (typ-info-t TYPE(unpacked-C)) = export-uinfo (typ-info-t
(TYPE(unpacked-C)))
  supply [[simp-trace=true, verbose=3]]

```



```

apply simp
done

ML <
  Cached-Theory-Simproc.trace-cache @{context}
>
lemma TYPE (32 word)  $\leq_\tau$  TYPE(32 word[256])
  supply [[simp-trace=true,verbose=3]]
  apply simp
  done

lemma TYPE (32 word)  $\leq_\tau$  TYPE(32 word[256])
  supply [[simp-trace=true,verbose=3]]
  apply simp
  done

ML <
  Cached-Theory-Simproc.trace-cache @{context}
>

schematic-goal  $n < 12 \implies \text{field-lookup}(\text{typ-info-t } \text{TYPE}(\text{array-C})) ["\text{elements-C}"]$ ,
  replicate n CHR "1", "chr-C" 0 = ?X
  supply [[simp-trace=true, simp-trace-depth-limit=1, verbose=5]]
  apply simp
  done

schematic-goal  $n < 12 \implies \text{field-lookup}(\text{typ-uinfo-t } \text{TYPE}(\text{array-C})) ["\text{elements-C}"]$ ,
  replicate n CHR "1", "chr-C" 0 = ?X
  supply [[simp-trace=true, simp-trace-depth-limit=1, verbose=5]]
  apply simp
  done

schematic-goal  $n < 12 \implies \text{field-ti } \text{TYPE}(\text{array-C}) ["\text{elements-C}"]$ , replicate n
  CHR "1", "chr-C" = ?X
  supply [[simp-trace=false, simp-trace-depth-limit=1, verbose=3]]
  apply simp
  done

schematic-goal  $n < 12 \implies \text{field-ti } \text{TYPE}(\text{array-C}) ["\text{elements-C}"]$ , replicate n
  CHR "1", "chr-C" = ?X
  supply [[simp-trace=false, simp-trace-depth-limit=1, verbose=3]]
  apply simp
  done

lemma TYPE(unpacked-C[12])  $\leq_\tau$  TYPE(array-C[2])
  supply [[simp-trace=false, simp-trace-depth-limit=1, verbose=3]]
  apply simp
  done

lemma  $\neg \text{TYPE}(\text{unpacked-C}[12]) \leq_\tau \text{TYPE}(\text{unpacked-C}[24])$ 

```

```

supply [[simp-trace=false, simp-trace-depth-limit=2, verbose=3]]
apply simp
done

```

```

lemma TYPE(unpacked-C[12]) ≤τ TYPE(array-C)
apply simp
done

```

```

lemma typ-uinfo-t TYPE(unit ptr) ≠ typ-uinfo-t TYPE(32 word ptr)
apply simp
done

```

Matching of the interpreted locale *stack-heap-raw-state*

```

ML-val ⟨
  val cterm-match = With-Fresh-Stack-Ptr.cterm-match @{context}
  val {n=n1, init=init1, c=c1, instantiate=inst1, ...} = cterm-match @{cterm
state.With-Fresh-Stack-Ptr n i1 c1}
  val {n=n2, init=init2, c=c2, instantiate=inst2, ...} = cterm-match @{cterm
state.With-Fresh-Stack-Ptr n i2 c2}
  val t = inst1 {n=n2, init = init2, c= c2}
  ⟨

```

```

ML-val ⟨
  val match = With-Fresh-Stack-Ptr.match @{context}
  val {n = n1, init=init1, c=c1, instantiate=inst1, ...} = match @{term globals.With-Fresh-Stack-Ptr
n i1 c1}
  val {n = n2, init=init2, c=c2, instantiate=inst2, ...} = match @{term globals.With-Fresh-Stack-Ptr
n i2 c2}
  val t = inst1 {n=n2, init = init2, c= c2}
  ⟨

```

```

thm zero-simps
thm make-zero
thm size-simps
thm size-align-simps

```

```

thm h-val-fields
thm heap-update-fields
thm fg-cons-simps
thm fl-ti-simps
thm fl-Some-simps

```

```

end

```

Chapter 28

Translation of the **StrictC** Dialect (Outdated)

MICHAEL NORRISH

28.1 Introduction

This report describes the translation program that imports C source into a running Isabelle/HOL process, making a series of Isabelle definitions in the process, as well as discharging a number of (relatively minor) proof obligations that arise along the way.

The source code for the translator is found in the directory `c-parser`. Source files, *e.g.*, `file.ML`, are found in this directory unless otherwise noted.

The translation expects its input to be a well-formed C source file. Such a source file must additionally satisfy a number of other constraints, giving rise to a subset of C that is here called “**StrictC**”. Files in **StrictC** may also include special annotations intended only for consumption by Isabelle (and the human code-verifier).

In fact, there are two important **StrictC** programs: the translation program, and the analysis program. The analysis program is entirely independent of Isabelle, and can be used to check that a source file conforms to the **StrictC** subset. It also implements a number of analyses that can be performed on source code. For example, it can output the input file’s call-graph, and can list the globals that are read or modified in each function. The additional source-files supporting the analysis program are found in

`c-parser/standalone-parser`

The rest of this report describes both the functionality provided by these programs (focussing on the translator), how this functionality is implemented, and *where* it is implemented. The aim is to give a picture of the systems’ design in a way that should make future modification of the code possible.

28.1.1 StrictC Subset Summary

This is a brief list summarising the simplest restrictions imposed by the StrictC subset.

- No `goto` statements.
- No fall-through cases in `switch` statements. Cases can be terminated with `continue` and `return`, as well as `break`.
- Labels for `switch` statement cases must all appear at the syntactic level immediately below the block statement that must appear below the `switch` statement.
- No unions. (These are handled by a separate tool; see Cock [1].)
- No `struct`, `enum` or `typedef` declarations anywhere except at the top, global, level.

28.2 Abstract Syntax

The core data types in the translator represent the input program. These abstract syntax values are the product of parsing the concrete syntax (see Appendix 28.6 for the grammar used), and is the subsequent input to all further analyses and translation. The abstract syntax declarations are given in `Absyn.ML`. For example, the definition of the syntax type corresponding to C statements is given in Figure 28.1. There are also definitions for C types, expressions, and declarations in `Absyn.ML`.

The type `statement` is actually a `statement_node` wrapped inside a “region” (see `Region.ML` and Section 28.2.1 below), which provides information about where the original concrete syntax originated. This is used for providing error messages.

All strings in the statement declaration correspond to Isabelle terms (*e.g.*, the loop invariant in the `While` case). These will be parsed as such later in the translation process, but are just uninterpreted strings when the C parser finishes. Function calls can return their results into l-values, have the return value ignored, or have the return value itself `return`-ed. The first option corresponds to having the `expr_option` argument be `SOME e` in the `AssignFnCall` constructor, the second would have that parameter be `NONE`, and the last is handled by the `ReturnFnCall` constructor.

Syntactically, these options correspond to writing

```
var = f(x,y);
```

or

```
f(x,y);
```

```

datatype statement_node =
  Assign of expr * expr
  | AssignFnCall of expr option * expr * expr list
  | EmbFnCall of expr * expr * expr list
  | Block of block_item list
  | While of expr * string wrap option * statement
  | Trap of trappable * statement
  | Return of expr option
  | ReturnFnCall of expr * expr list
  | Break
  | Continue
  | IfStmt of expr * statement * statement
  | Switch of expr * (expr option list * block_item list) list
  | EmptyStmt
  | Auxupd of string
  | Spec of ((string * string) * statement list * string)
  | AsmStmt of {volatile : bool, asmblock : asmblock}
and statement = Stmt of statement_node Region.Wrap.t
and block_item =
  BI_Stmt of statement
  | BI_Decl of declaration wrap

```

Figure 28.1: The Abstract Syntax Data Type for C Statements

or

```
return f(x,y);
```

C99 Block Items Conforming to the C99 grammar, the input language allows declarations at any point inside a block, not just in a sequence at the head of the block. In other words, the following

```
{
  x = f(z);
  int y = x + 1;
  while (x < y) { ... }
}
```

is legal in C99. This means that a block has to take a list of `block_item` values as an argument, where a `block_item` is either a statement or a declaration.

Syntactic Sugar for Loops The abstract syntax has just one form of loop, the `While` constructor. The optional string argument to `While` is used to represent any user-supplied invariant. Together with the `Trap` constructor, this is used to implement all three forms of C loop. The translation follows the model from Norrish's PhD [6, p60]. It also supports `for`-loops with declarations in the first position. This latter, for example,

```
for (int i = 3; i < 10; i++) ...
```

is a feature of C99.

Breaking from C, the grammar has the third component of the `for` loop form be a restricted form of statement, rather than an expression. The parser syntax only allows comma-separated increments (*i.e.*, `++`), decrements and assignments.

The Isabelle translation eventually compiles all loops to one underlying the loop primitive in the VCG environment called `While`. This form does not handle exceptional control-flow forms like `break` and `continue`. These are instead handled by the `Trap` constructor, mapping to the VCG language's `TRY-CATCH` form.

28.2.1 Regions

The `Region` module implements a method for annotating arbitrary data types with location information. This module has been taken from the `MLton` compiler project (which has a BSD-style open source licence). It is used a great deal in the system, and its use could probably be extended still further. Region information is used to produce good error messages.

The basic type is that of the `region` which is essentially a pair of “source positions” (which are in turn implemented in `SourcePos.ML`). One useful source position is `SourcePos.bogus`, corresponding to “nowhere” (perhaps because some syntax has been conjured out of nowhere and doesn’t really exist in a file).

Regions are then used to implement the concept of a “wrap” (SML type `Region.Wrap.t`), a polymorphic data type. The file `Absyn.ML` declares the following abbreviation:

```
type 'a wrap = 'a Region.Wrap.t
```

An `'a wrap` (read “alpha wrap”) is an `'a` value coupled with a region. The important functions for manipulating wraps are

```
val wrap      : 'a * SourcePos.t * SourcePos.t -> 'a wrap
val bogwrap  : 'a -> 'a wrap
val left     : 'a wrap -> SourcePos.t
val right    : 'a wrap -> SourcePos.t
val node     : 'a wrap -> 'a
val apnode   : ('a -> 'b) -> 'a wrap -> 'b wrap
```

For example, the grammar code in `StrictC.grm` manipulates a number of values of type `string wrap`. If an error relating to this value arises, both the string and its position can be reported to the user.

Things become more complicated when the type to be wrapped is recursive. The standard idiom in the project is illustrated with the definition of the `statement` data type (shown in Figure 28.1). The constructors for values of the type are actually given in an auxiliary type (`statement_node`), but the recursive constructors take arguments of type `statement`. The type `statement` is then a type that is mutually recursive with `statement_node`, and which has just one constructor, `Stmt`.¹

Because a `statement` is not a wrap, the project cannot directly call functions like `node` on `statement` values. Instead, helper functions are declared:

```
val sleft   : statement -> SourcePos.t
val sright  : statement -> SourcePos.t
val snode   : statement -> statement_node
val swrap   : SourcePos.t * SourcePos.t * statement_node ->
              statement
```

When code wishes to pattern-match against the multiple possible forms a `statement s` may have, the idiom is to write

¹It would be nice if one could just declare `statement` to be an abbreviation of `statement_node wrap`, but SML doesn’t permit this. It must be a `datatype` itself, and thus must have at least one constructor.

```

case snode s of
  EmptyStmt => ...
| While(g,inv,body) => ...
| IfStmt(g,ts,es) => ...
| ...

```

The strength of this idiom is that one *always* manipulates values of type `statement`. In particular, if the case analysis above is to make recursive calls of its analysis on sub-statements such as `body`, `ts` and `es`, these values are of the correct type for this to be done immediately.

The `expr` (expression) type (home to constructors such as `Deref`, `Var` and `TypeCast`) is set up in the same style, giving rise to functions `elleft`, `eright`, `enode`, `ewrap` and `ebogwrap`.

The type of C types The type of C types is `'a ctype`, with constructors such as `Void` and `Ptr`. Though recursive, it doesn't use the wrap idiom. On the other hand, this type is polymorphic. The `'a` parameter is instantiated with an SML type that corresponds to the forms that give the size of arrays when they are declared. This type parameter is instantiated with `expr` when the input file is first parsed. In this way, a declaration like

```

unsigned char array[EnumConst1 * sizeof(int*)];

```

can be handled. Thus, the `VarDecl` constructor of the declaration type

```

val VarDecl :
  expr ctype * string wrap * bool * initializer option ->
  declaration

```

takes an `expr ctype` as its first parameter. Within subsequent phases of the analysis and translation, it is much more convenient to work with values of type `int ctype`, where the (constant) expression has been evaluated. For example, the `get_reftype` function from `program_analysis.ML`, takes a `csenv` value (see Section 28.3 below) and the name of a function, and returns the return type of the function. The value returned is an `int ctype`. The conversion of an `expr ctype` into an `int ctype` is done by the function `Absyn.constify_abtype`.

28.3 The Symbol Table

All of the work done in analysis and translation of `StrictC` revolves around the information stored in two important data structures implemented in the module `program-analysis.ML`. The first of these is the `var_info` type. This stores information about individual variables. The second type, `csenv` ("C state environment"), stores information about the program as a whole,

including its collection of variables, but also recording details such as where variables are read and modified, and the program's call-graph structure.

The `var_info` type stores information about declared identifiers living in the name-space that encompasses normal objects, functions and enumeration constants. In addition to the type of the variable (*e.g.*, `int`, `char *` *etc.*), the `var_info` also includes information about where the variable was declared in terms of program locations, and also in terms of scope (it might be global, or declared local to a particular function).

The `csenv` type accumulates its information about the program by performing traversals of the abstract syntax tree. **The StrictC translator makes no effort to be a one-pass compiler**, but the number of traversals is no greater than three, and will probably be reduced in future versions of the implementation. These traversals are performed after the parser has constructed all of the tree. There are also places in this analysis where **the translator assumes that it has seen the whole program**. In particular, the translator cannot be used to translate *translation units* that are to be separately compiled. It must be presented with a concatenation of the complete sources.

The bulk of the API for manipulating values of type `csenv` is concerned with pulling information out of the symbol table. For example, it is possible to calculate the type of a C expression with the function

```
val cse_typing : csenv -> expr -> int ctype
```

In addition, `program-analysis.ML` contains the one entry-point for taking a sequence of external declarations (once parsed) and creating a `csenv` value:

```
val process_decls :  
  Absyn.ext_decl list ->  
  ((Absyn.ext_decl list * Absyn.statement list) * csenv)
```

The return type includes a modified version of the syntax that was provided as input, a list of the initialising assignments for the global variables, and the `csenv` value.

28.3.1 Functional Record Updates in SML

The code in `program-analysis.ML` uses a powerful, but cryptic SML idiom that makes it easy to define SML records along with functions for updating their fields. Done naïvely, writing code to do this represents a quadratic amount of work for the programmer. Put another way, the naïve approach requires $O(n)$ much typing whenever a field is added to or removed from a record definition of n fields.

The cryptic technique is fully described at

<http://mlton.org/FunctionalRecordUpdate>

and allows the addition or deletion of a field to be done with $O(1)$ much typing. Supporting code is in `FunctionalRecordUpdate.ML`.

The cryptic code is isolated within `program-analysis.ML`, where it is used to define update functions that are subsequently used exclusively. In general, when one of the two types has a field `fld` of type τ , then there will typically be a function `{cse|vi}_fupd_fld` defined, with type

$$(\tau \rightarrow \tau) \rightarrow \text{rcd} \rightarrow \text{rcd}$$

where `rcd` is either `var_info` or `csenv`. Such functions can be used to update the fields of a record: the user provides a function that is given the old value of the field, and which returns the new value.

28.4 Creation of the Hoare Environment State

The major oddity about Norbert Schirmer’s Hoare environment, into which we are translating our programs, is that all local variables, including function parameters, have to be part of the “global state”. This state must be declared before any functions can be translated, because a function becomes an Isabelle definition (conceptually at least) that operates over that state space.

Slightly simplifying, the state space is an Isabelle type that is a record with fields for every local and global variable. Each field has a type (Isabelle/HOL is a typed logic after all), which means that all local variables of the same name in the same `StrictC` translation unit must have the same type. This is rather an arbitrary requirement, but easy both to enforce and to comply with. Thankfully, signed and unsigned variants of the same underlying type (such as `signed short` and `unsigned short`) are given the same Isabelle/HOL type, so there is a little leeway. Nonetheless, if `i` is of type `int` in one function, it can not be of type `char` in another.²

The arrangement of global and local variables is actually slightly complicated. In essence, the state-space is set up to look like:

```
statespace = record
  globals :: global_var_type
  local_var1 :: lvar1_type
  local_var2 :: lvar2_type
  ...
  local_varn :: lvarn_type
```

²If this is attempted, the system will “munge” one of the variables so that it has a different name when translated into Isabelle. The “munged” name is stored in the variable’s `var_info` record.

where the field `globals` is of a custom record type `global_var_type` that in turn contains all of the global variables. In addition to the user program’s own globals, the translation process adds two extra global variables of its own. These variables are used for handling “exceptional exits” (such as those caused by the `break`, `continue` and `return` statements), and for modelling the global heap. The exact names of these variables is not important, here we will refer to them as `global_exn_var`, and `global_heap`.

These two special variables are part of `global_var_type` and so must have Isabelle types themselves. The type of `global_exn_var` is the enumerated type `c_exntype`, defined in the Isabelle theory `CProof` to have three possible values, `Break`, `Return` and `Continue`. The type of the global heap `global_heap` field is a product of underlying heap contents (a map of type `addr → word8`) and a special-purpose data type to store type information about the heap memory (see Harvey Tuch’s PhD thesis [9] for more on this).

28.4.1 Representing Values in Memory

There is one important requirement that must be met by all object types that occur in C programs: they must be representable in memory. Alternatively, it must be possible to encode values of the types in a program as sequences of bytes, and to then decode those same bytes back into the original values. One approach to modelling this fundamental requirement might be to have Isabelle functions that manipulated only lists of bytes. Working at the level of this untyped, and very concrete, view of the program state would be an extremely poor state of affairs (the C programmer would have a more abstract view of the program than the verifier).

Our approach is to use Isabelle type-classes to encode the fact that an Isabelle type can be represented in a consistent amount of “C memory”. When an Isabelle type `'a` is in the class `mem_type`³, it supports functions

```
to_bytes   :: "'a::mem_type => byte list"
from_bytes :: "byte list => 'a::mem_type"
```

as well as a number of other supporting functions that record details like the fields that occur in compound `struct` types, and the (constant) length of the byte-lists that encode the values in the type.

All of the atomic types manipulated by our programs are fixed-width words (*e.g.*, 32 bit words for integers). It is easy to demonstrate that these types are indeed in the `mem_type` class. In particular, we do *not* pretend that programs manipulate infinite precision integers, and then worry about whether or not these integers can be pushed into and pulled out of memory. All of the arithmetic performed by the programs we verify is done at fixed widths, respecting the underlying machine’s operations. Additionally, using

³See `umm_heap/CTypes.thy`.

the techniques from Section 28.5.1, we trap the undefined behaviour caused by overflow on signed values.

28.4.2 Pointers

Pointers are always represented as words of a particular size (regardless of type being pointed to). This is not required by the C standard, which only requires pointers to `void` be capable of storing all other non-function pointer values, and that all function pointer values be inter-convertible. Again, our decision to specialise on particular, and reasonable, target architectures makes life simpler.

Pointers retain type information by using “phantom” type variables. The Isabelle declaration is

```
datatype 'a ptr = Ptr addr
```

Then, if the C program under analysis calls for a variable of type `char *`, the Isabelle environment will include a variable of Isabelle type `byte ptr`. In this way our pointers are typed, even if their underlying encoding makes it trivial to view a pointer to one type as a pointer to another type.

Because the underlying representation is always the same, all pointer types are proved to be in the class `mem_type` once and for all (in theory `CTypes`). Pointers to `void` are represented as values in the Isabelle type `unit Ptr`, where `unit` is the standard singleton type. The `unit` type is not shown to be in the class `mem_type`.

28.4.3 Arrays

C arrays are lists of values of fixed length. A faithful representation of this type requires a novel Isabelle type. We build on Anthony Fox’s implementation of John Harrison’s “finite Cartesian products” idea [4]. Syntactic trickery within Isabelle allows us to write types like

```
nat[10]
```

which is an array of 10 natural numbers. There are operators for updating and indexing into arrays. Note that the type `10` that appears above is an Isabelle *type*, not a term.

If type τ is a `mem_type`, then an array of τ values will also be a `mem_type`, as long as the size of the array is not so large that the array would not fit into memory. This condition is discharged as the `StrictC` program is translated.

For technical reasons due to the implementation of type classes in Isabelle, we need to fix separate limits, ahead of time, on the number of elements in an array and the size of each element. Currently, for 32-bit ARM, our model fixes a maximum of:

```

struct listnode {
    int node_data;
    struct listnode *next;
};

struct node2;
struct node1 {
    struct node2 *data;
    struct node1 *next, *prev;
    int someflag;
};
struct node2 {
    struct node1 *owner;
    char stringdata[100];
};

```

Figure 28.2: Examples of Recursive `struct` Declarations Accepted in StrictC

- 2^{13} (8192) elements in each array; and
- 2^{19} bytes (512 KiB) in each array element

For x86-64, the limits are:

- 2^{20} (1048576) elements in each array; and
- 2^{26} bytes (64 MiB) in each array element

One would prefer to be able to multiply the size of the element type by the number of the elements, but the type system does not permit this (for good technical reasons).

28.4.4 C `struct` Types

From the point of view of the translation into Isabelle, `struct` types do not pose any great conceptual difficulties. A `struct` type is clearly very similar to an Isabelle record, which in turn is conceptually the same as a tuple. The first complication that arises is that Isabelle tuples can not be recursive, whereas C `struct` types are often recursive (as when implementing linked structures in the heap).

This required the implementation of an alternative record definition package, allowing (possibly multiple) recursive types. This then allows Isabelle types to be defined that correspond to C declarations such as those shown in Figure 28.2.

Confirming that a `struct` type really is representable in memory, requires the definition of functions for converting Isabelle records into lists of bytes and *vice versa*. The size of the converted value must also be checked to be no bigger than the size of memory. Both of these actions require knowledge of how the fields of the `struct` are laid out in memory, which is in turn a function of the padding that can be inserted between the fields. Such calculations are architecture dependent.

28.5 Translating Expressions

Expression translation is implemented in `expression_translation.ML`.

Fundamental Concepts StrictC expressions are essentially a subset of C expressions, and are fairly easy to translate to corresponding Isabelle expressions that manipulate Isabelle-encoded values. There are two fundamental concepts to grasp of expression translation. First, when being evaluated (“read to determine a value”) a C expression of type τ becomes an Isabelle expression of type `statespace` \rightarrow $\llbracket\tau\rrbracket$, where $\llbracket\tau\rrbracket$ is the translated, Isabelle version of C type τ .

This must be done to make sense of expressions that read memory: an expression such as `s.arrayfld[3]` only has a specific value in the context of a specific state of memory. This use of a “lifted” function space to represent the expression is a standard technique in denotational semantics. In the example given, the value of the expression is a function that looks at the statespace to determine what data is stored at variable `s`. As the statespace evolves, the value returned from this function changes, but the function’s value is the same.

Expressions do not just determine values however, they can also denote an *l-value*, something denoting a “place in memory” that is to be updated. This is done when an address is taken, or when an assignment is to be performed. In a simple language, l-values might only be variable names, but C allows for complicated expressions on both sides of an assignment. The example above (`s.arrayfld[3]`) might just as well be assigned to as read. So, the l-value of an expression e that has type τ will be an Isabelle value of type `statespace` \rightarrow τ \rightarrow `statespace`, a function that takes a new value and a statespace to change, and returns the updated statespace.

Not all expressions are l-values (the expression `3` is not, for example), so the translation of any one expression can return one or two different values, an “r-value” as well as an optional l-value.

In addition, because of the way the translation does not put local variables into memory, the translation provides another separate optional value, that of the expression’s address. If everything did live in memory, all l-values would have addresses, and one could dispense with the separate calculation of l-values. Thus the first three lines of Figure 28.3.

The SML types given to the `lval`, `addr` and `rval` fields in the declaration of `expr_info` are themselves function spaces at the SML level. These function spaces manipulate values of SML type `term`. The way in which typing at the C level is reflected at the Isabelle level is mainly hidden at the SML level, where the programmer just manipulates terms (the Isabelle types are internal to those values).

However, the function spaces *can* be made visible at the SML level. This is done mainly to reduce the number of β -redexes that would otherwise be

```

datatype expr_info =
  EI of {lval   : (term -> term -> term) option,
        addr   : (term -> term) option,
        rval   : term -> term,
        cty    : int ctype,
        ibool   : bool,
        lguard : (term -> term * term) list,
        guard  : (term -> term * term) list,
        left   : SourcePos.t,
        right  : SourcePos.t }

```

Figure 28.3: The `expr_info` type, into which C expressions are translated internally.

created in the resulting term. For example, translating the C expression `x + 3` will first create r-values

$$\begin{aligned} \llbracket x \rrbracket &= (\lambda\sigma. x(\sigma)) \\ \llbracket 3 \rrbracket &= (\lambda\sigma. 3) \end{aligned}$$

where the application of the `x` function to a statespace σ pulls out the value of variable `x` in σ .

When translating an expression $e_1 + e_2$, one naturally creates the value

$$\lambda\sigma. \llbracket e_1 \rrbracket(\sigma) + \llbracket e_2 \rrbracket(\sigma)$$

If this was done within Isabelle `term` values, the result would be a `term` full of expressions of the form $(\lambda v. M)N$. By lifting the Isabelle λ to the SML level, only one abstraction need to be created, at the very top-level in the translation. Thus, the translation of the addition becomes the SML expression

```
(fn s => mk_plus (rval_of e1 s) (rval_of e2 s))
```

where `rval_of` returns the `rval` field of an `expr_info` and the β -reduction happens within SML.

The fourth line of the record in `expr_info` values stores the type of the expression, something that informs the translation of many different C expressions. For example, additions are not as simple as just presented because of the possibility that one of the arguments might be a pointer. The `left` and `right` fields of the `expr_info` record store the source-code position of the original expression. The two guard fields are explained below in Section 28.5.1.

The `ibool` field of the `expr_info` records whether or not the term being generated in the r-value is of Isabelle's boolean type. This is done so that

translation can avoid some conversions between Isabelle words and booleans. For example, if the expression being translated is `x < 6 && y > 10`, the resulting Isabelle term will include a use of the two comparison operators on words. Strictly, one should then turn the boolean results of these operators into either a one or zero. But, as these results are then combined with boolean conjunction, the values will immediately be converted back into booleans.

Of course, in an expression such as `x && x->fld > 6`, the first operand to the conjunction does not have Isabelle boolean type, and will be converted to a boolean value by comparing it with the null pointer. (In fact, this particular example causes a more complicated translation effect to occur; see Section 28.5.1 below on undefined behaviour.) Dually, if the code puts a boolean value into memory, the translation has to make sure that the Isabelle term has the appropriate word type once more. There are two functions used here:

```
mk_isabool : expr_info -> expr_info
strip_kb   : expr_info -> expr_info
```

The function `mk_isabool` produces an `expr_info` value that does have Isabelle boolean type (it will be the identity function on an r-value that is already known to be boolean). The function `strip_kb` reverses this, turning an Isabelle boolean into an Isabelle word if necessary.

28.5.1 Undefined Behaviour

There are three classes of under-specification in the C standard. Those classed as *implementation defined* are behaviours or values that are supposed to be fixed and documented by particular implementations. For example, the number of bits in an `int` (a number that must be at least 16), is a fixed value that implementations are allowed to choose themselves. Because `StrictC` targets a particular architecture, most implementation-defined aspects of C can be “baked into” the translation. (The design attempts to have the translation sources be easy to modify to account for different architectures; see for example the `ImplementationNumbers` structure in each of the `TargetNumbers.ML` files.)

The second class of under-specification comprises *unspecified* behaviours. The most important of these is the order of evaluation of arguments to operators and function calls. Therefore, one should avoid writing code that depends on unspecified evaluation order. Otherwise, the compiled executable might have different semantics compared to the `Simpl` specification that we generate. For example:

```
int x = f() + g();
```


where `f` and `g` both have side effects, is currently allowed but should be avoided. In the future, we plan to forbid or restrict programs that may have unspecified evaluation order. The standalone analysis tool can check for this problem using the `-embedded_fncalls` option.

The third class of under-specification is *undefined behaviour*. In essence, undefined behaviours are runtime errors (such as dereferencing a null pointer). They are undefined because the standard does not require the implementation to catch them, or to realise that they have occurred. Rather, if an undefined behaviour occurs, the user can no longer rely on their program to do anything sensible. In effect, implementations are given licence to blunder on however they like when an undefined behaviour occurs.

Thus, it is critical that the C code we verify never exhibits any undefined behaviours. We do this with a feature of the Hoare environment called the *guard*. A guard is a boolean condition g attached to a statement s , with the combination written $g \rightarrow s$. If the guard g is true when the combined statement is to be executed, then s is allowed to execute. If it is false, the underlying semantics defines the result to be a “fault”. When a program faults, nothing more can happen, so it has effectively aborted. Verifying a program with guarded statements thus requires proofs that the guards are always true.

Most guards arise in expressions, and the translation process accumulates these in one top-level guard at the statement level. For example, in the statement

```
x = *p >> i;
```

there will be four guards attached to the statement that will need to be discharged: that `p` is not null, that `*p` is not negative, that `i` is not negative, and that `i` is less than 32.

One elegant feature of guards in the Hoare environment is that they can be selectively disabled for the purposes of verification. For example, the C standard’s requirement that right shift operations might not be performed on negative numbers (that it results in undefined behaviour) is too strict, given a particular C compiler and target architecture. In this situation, it is possible to prove Hoare-triples where that particular guard is not used, so that the semantics of $g \rightarrow s$ is simply that of s .

28.6 Concrete Syntax: Parsing and Lexing

The grammar for `StrictC` is given in the file `StrictC.grm`, which is given as input to the standard tool `mlyacc`. The format of an `mlyacc` file looks quite similar to the format accepted by `yacc`. A typical grammar rule is

```
init_declarator_list
: init_declarator
```

```

        ([init_declarator])
    | init_declarator YCOMMA init_declarator_list
        (init_declarator :: init_declarator_list)

```

where a non-terminal appears before a colon, and multiple possible right-hand sides are separated by the pipe or vertical bar character. The code for a production appears in parentheses after each right-hand-side. The code's convention is to have token names (*e.g.*, YCOMMA above) be upper-case.

Apart from the changes that turn assignment expressions into statements, the grammar for `StrictC` attempts to be as close as possible to the grammar of the C standard. In general, the names for non-terminals in `StrictC.grm` are taken from the standard, so it should be fairly clear how the standard's grammar has been mapped into `StrictC.grm`.

28.6.1 Lexing and typedef Names

The standard problem in lexing and parsing C is that the grammar is ambiguous: the non-terminal *typedef-name* is defined to simply be the same as an *identifier*. When the parser encounters

```
x * f(y);
```

it can't know if this is meant to be a multiplication of variable `x` by a function call expression, or whether it is the (prototype) declaration of a function called `f` taking an argument of type `y` and returning a value of type `x`.

To resolve this, the lexer must be able to classify identifier tokens as normal identifiers or *typedef-names*. The `StrictC` translator's approach to this problem is not typical, because of the strange way in which `mlyacc` combines handling of side effects with its error correction. Normally, one would have some sort of updatable symbol table that the parser would write to when it encountered a `typedef` declaration. The lexer would read from the same table as it encountered identifiers, allowing appropriate categorisation.

The approach taken in the `StrictC` translator is to have the lexer do all of the work, without reference to the parser (see `StrictC.lex`). This *is* possible, but is also convoluted, and involves many updatable variables that are internal to the lexer. The basic idea is that when the lexer sees a `typedef` token, it switches to the TDEF state. When an identifier is seen in this state, the identifier is added to the list of *typedef-names*, and lexing can continue. The complications arrive in declarations like

```
typedef struct s { int fld1; char fld2; } s_t;
```

where the identifiers `s`, `fld1` and `fld2` have to be ignored, and `s_t` taken as the new *typedef-name*. This requires the lexer to handle the matching brace characters, and partly motivates the requirement that `typedef` declarations all occur at the top-level (and not be nested).

28.6.2 GCC `__attribute__` Declarations

The parser handles, but mostly ignores, various gcc-specific extensions, such as `__attribute__`. These are tricky to parse (something admitted by the relevant gcc documentation): users are given almost unlimited liberty to put their `__attribute__` strings anywhere within a declaration.

For example, these three

```
int f(int) __attribute__((__const__));
__attribute__((__const__)) int f(int);
int __attribute__((__const__)) f(int);
```

are all supposed to parse successfully (and have the same meaning). Making this work is rather involved. Most attributes are ignored, but the standalone analysis tool does check that `const` and `pure` attributes are reasonable, given what it knows of how functions may modify and read the global state.

Bibliography

- [1] D. Cock. Bitfields and tagged unions in C: Verification through automatic generation. In B. Beckert and G. Klein, editors, *Proc, 5th VERIFY*, volume 372 of *CEUR Workshop Proceedings*, pages 44–55, Sydney, Australia, Aug. 2008.
- [2] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. 7406:99–115, Aug. 2012.
- [3] D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don’t sweat the small stuff: Formal verification of C code without the pain. pages 429–439, June 2014.
- [4] J. Harrison. A HOL theory of Euclidean space. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference*, volume 3603 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2005.
- [5] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. <http://www.in.tum.de/~nipkow/LNCS2283/>.
- [6] M. Norrish. *C Formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998. Also published as Technical Report 453, available from <http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-453.pdf>.
- [7] M. Norrish. C-to-Isabelle parser, version 1.13.0, May 2013. Accessed May 2016.
- [8] N. Schirmer. A sequential imperative programming language syntax, semantics, hoare logics and verification environment. *Archive of Formal Proofs*, Feb. 2008. Formal proof development.
- [9] H. Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, Aug 2008.