

Concurrent HOL

Peter Gammie

April 19, 2024

Abstract

This is a simple framework for expressing linear-time properties. It supports the usual programming constructs (including interleaving parallel composition), equational and inequational reasoning about these, compositional assume/guarantee specifications and refinement, and the mixing of specifications and programs, all shallowly embedded in Isabelle/HOL.

Contents

1	Introduction	4
1.1	Road map	4
2	Terminated Aczel sequences	4
2.1	Traces	5
2.2	Combinators on traces	5
2.3	Behaviors	14
2.4	Combinators on behaviors	14
3	Point-free notation	20
4	More lattice	22
4.1	Boolean lattices and implication	25
4.2	Compactness and algebraicity	27
5	Closure operators	30
5.1	Complete lattices and algebraic closures	32
5.2	Closures over powersets	35
5.3	Matroids and antimatroids	37
5.4	Composition	38
5.5	Path independence	38
5.6	Some closures	39
5.6.1	Reflexive, symmetric and transitive closures	39
5.6.2	Relation image	39
5.6.3	Kleene closure	40
6	Galois connections	42
6.1	Some Galois connections	49
7	Heyting algebras	51
7.1	Downwards closure of preorders (downsets)	57
8	Safety logic	59
8.1	Stuttering	60
8.2	The $(\prime a, \prime s, \prime v)$ <i>spec</i> lattice	65
8.3	Irreducible elements	67
8.4	Maps	70
8.5	The idle process	72
8.6	Actions	73
8.7	Operations on return values	78
8.8	Bind	82

8.9	Kleene star	88
8.10	Transition relations	89
8.11	Sequential assertions	96
8.11.1	Preconditions	96
8.11.2	Postconditions	98
8.11.3	Strongest postconditions	101
8.12	Initial steps	102
8.13	Heyting implication	104
8.14	Miscellaneous algebra	106
9	Constructions in the $(\prime a, \prime s, \prime v)$ spec lattice	108
9.1	Constrains-at-most	108
9.2	Abadi and Plotkin's composition principle	112
9.3	Interference closure	113
9.4	The $\prime a$ agent datatype	117
9.5	Parallel composition	119
9.6	Specification Inhabitation	126
10	"Next step" implication ala Abadi and Merz (and Lamport)	130
10.1	Compositionality ala Abadi and Merz (and Lamport)	134
11	Stability	135
12	Refinement	138
12.1	General rules for the $(\prime a, \prime s, \prime v)$ spec lattice	141
12.1.1	Actions	142
12.1.2	Bind	142
12.1.3	Interference	145
12.1.4	Parallel	145
12.2	A relational assume/guarantee program logic for the $(\text{sequential}, \prime s, \prime v)$ spec lattice	146
12.2.1	Stability rules	150
13	A programming language	152
13.1	The $(\prime s, \prime v)$ prog lattice	152
13.2	Morphisms to and from the $(\text{sequential}, \prime s, \prime v)$ spec lattice	152
13.3	Programming language constructs	155
13.3.1	Laws of programming	156
13.4	Refinement for $(\prime s, \prime v)$ prog	165
13.4.1	Introduction rules	165
13.4.2	Galois considerations	166
13.4.3	Rules	166
13.5	A relational assume/guarantee program logic for the $(\prime s, \prime v)$ prog lattice	168
13.5.1	Galois considerations	168
13.5.2	A proof of the parallel rule using Abadi and Plotkin's composition principle	170
13.6	Specification inhabitation	170
14	More combinators	171
15	Structural local state	176
15.1	<i>spec.local</i>	176
15.2	Local state transformations	179
15.2.1	Permuting local actions	183
15.3	<i>spec.localize</i>	185
15.4	<i>spec.local_init</i>	189
15.5	Hoist to $(\prime s, \prime v)$ prog	191
15.6	Refinement rules	194
15.6.1	Data refinement	196
15.7	Assume/guarantee	197
15.8	Specification inhabitation	198

16 A Temporal Logic of Safety (TLS)	198
16.1 Stuttering	199
16.2 The $(\prime a, \prime s, \prime v)$ <i>tls</i> lattice	204
16.3 Irreducible elements	205
16.4 The idle process	207
16.5 Temporal Logic for $(\prime a, \prime s, \prime v)$ <i>tls</i>	207
16.5.1 Leads-to and leads-to-via	220
16.5.2 Fairness	221
16.6 Safety Properties	223
16.7 Maps	229
16.8 Abadi's axioms for TLA	231
16.9 Tweak syntax	232
17 Atomic sections	232
17.1 Inhabitation	235
17.2 Assume/guarantee	236
18 Exceptions	236
19 Assume/Guarantee rule sets	239
19.1 Implicit stabilisation	239
19.1.1 Assume/guarantee rules using implicit stability	241
19.2 Refinement with relational assumes	243
20 Wickerson, Dodds and Parkinson: explicit stabilisation	245
20.1 Assume/Guarantee rules	249
20.2 Examples	250
21 Example: inhabitation	251
22 Example: findP	251
23 Example: data refinement (search)	254
24 Observations about safety closure	260
24.1 Liveness	262
24.2 A Haskell-like <i>Ix</i> class	263
25 A polymorphic heap	266
25.1 References	271
25.2 Arrays	273
25.2.1 Code generation constants: one-dimensional arrays	273
25.2.2 User-facing arrays	274
25.2.3 Stability	278
26 A concurrent variant of Imperative HOL	279
26.1 Code generator setup	281
26.1.1 Haskell	281
26.2 Value-returning parallel	283
27 Total store order (TSO)	284
27.1 References	298
27.2 Inhabitation	298
27.3 Code generator setup for TSO	300
27.3.1 Haskell	300
27.4 A TSO litmus test	302
28 Floyd-Warshall all-pairs shortest paths	302
References	310

1 Introduction

This is a simple framework for expressing linear-time properties. It supports the usual programming constructs (including interleaving parallel composition), equational and inequational reasoning about these, compositional assume/guarantee specifications and refinement, and the mixing of specifications and programs, all shallowly embedded in Isabelle/HOL. The closest extent works to ours are by [Xu and He \(1991, 1994\)](#) and [Dingel \(1996, 2000, 2002\)](#). It is heavily influenced by [Lampert \(1994\)](#).

1.1 Road map

Rather than begin with *a priori* “laws of programming” we take finite and infinite sequences as models of system executions (§16). Also, as transforming realistic concurrent systems while preserving total correctness is too difficult to be usable, we adopt Lampert’s approach to separating liveness and safety properties ([Abadi and Lampert 1991](#)) and do most of our work on safety properties.

The safety model consists of a series of closures (§5) over the powerset lattice of finite, non-empty, terminated “Aczel” sequences (§2), where each transition is ascribed to an agent. The termination marker supports sequential composition. The model of system executions is built similarly.

The *spec* lattice. Firstly and fundamentally we close under prefixes (§7.1), which captures precisely the safety properties (i.e., we identify a safety property with the set of sequences that satisfies it). We also close under stuttering ala Lampert (§8.1) to support refinement and the “laws of programming” (§13.3.1). All properties we consider therefore need to be stuttering invariant which is a mild constraint. We call the set of sets closed in this way the *spec* lattice (§8.2); we can interpret its points as propositions as it is a Heyting algebra. Its chief novelty is that it supports a logical presentation of assume/guarantee reasoning due to Abadi and Plotkin (§13.5.2) where parallel composition (§9.5) is simple (infinitary) conjunction ala [Lampert \(1994\)](#).

This lattice is satisfactory as a logic but deficient as a programming language; see [Zwiers \(1989\)](#) for an extended discussion on this point, and a solution for synchronous message passing. In brief, parallel composition-as-conjunction and the monad laws (§8.8) fail to meet expectations. We therefore look for a stronger closure condition.

The *prog* lattice. We take the view that a concurrent process is a parallel composition of sequential processes where the parallel composition itself yields a sequential process. Abadi and Plotkin’s constrains-at-most (§9.1) closure adds interference to the ends of traces – sufficient to support their circular composition principle (§9.2) – but not their beginnings. Our interference closure (§9.3) makes this symmetric, ensuring that parallel composition conforms to expectations: the monad laws hold as do many of the “laws of programming” (§13.3.1). We define the *prog* type (§13.1) to be the interference-closed specifications. We reason about programs in *prog* using propositions in *spec* via a pair of morphisms that form a Galois connection (§13.2).

Refinement. Abadi and Plotkin’s approach does not support refinement in our setting. We therefore adopt a “next step” implication (§10) and develop a logical account of compositional program refinement (§12). Refinement here is trace inclusion (i.e., the preservation of all safety properties).

Relational assume/guarantee. The definition of relational assume/guarantee in this setting is pleasantly intuitive (§12.2). Its key strength is that program phrases can be abstracted to relational assume/guarantee quadruples that can then be used as program phrases (§13.5). This generalises Morgan’s specification statement to a concurrent setting.

State spaces. As is traditional with shallow embeddings in HOL, we defer state space and value considerations using polymorphism. We develop a mechanism that partially encapsulates local state (§15).

Miscellany. Along the way we assemble some facts about Heyting algebras (§7), and sometimes construct our closures (§5) from Galois connections (§6). We explore the impact of using safety properties and this mix of finite and infinite sequences on TLA (§16).

2 Terminated Aczel sequences

We model a *behavior* of a system as a non-empty finite or infinite sequence of the form $s_0 - a_1 \rightarrow s_1 - a_2 \rightarrow \dots (\rightarrow v)$? where s_i is a state, a_i an agent and v a return value for finite sequences (see §16). A *trace* is a finite sequence $s_0 - a_1 \rightarrow s_1 - a_2 \rightarrow \dots - a_n \rightarrow s_n \rightarrow v$ for $n \geq 0$ with optional return value v (see §8). States, agents and return values are of arbitrary type.

2.1 Traces

$\langle ML \rangle$

```
datatype (aset: 'a, sset: 's, vset: 'v) t =
  T (init: 's) (rest: ('a × 's) list) (term: 'v option)
```

for

```
map: map
pred: pred
rel: rel
```

```
declare trace.t.map-id0[simp]
declare trace.t.map-id0[unfolded id-def, simp]
declare trace.t.map-sel[simp]
declare trace.t.set-map[simp]
declare trace.t.map-comp[unfolded o-def, simp]
declare trace.t.set[simp del]
```

```
instance trace.t :: (countable, countable, countable) countable  $\langle$ proof $\rangle$ 
```

```
lemma split-all[no-atp]: — imitate the setup for 'a × 'b without the automation
  shows (∧x. PROP P x) ≡ (∧s xs v. PROP P (trace.T s xs v))
 $\langle$ proof $\rangle$ 
```

```
lemma split-All[no-atp]:
  shows (∀x. P x) ↔ (∀s xs v. P (trace.T s xs v)) (is ?lhs ↔ ?rhs)
 $\langle$ proof $\rangle$ 
```

```
lemma split-Ex[no-atp]:
  shows (∃x. P x) ↔ (∃s xs v. P (trace.T s xs v)) (is ?lhs ↔ ?rhs)
 $\langle$ proof $\rangle$ 
```

2.2 Combinators on traces

```
definition final' :: 's ⇒ ('a × 's) list ⇒ 's where
  final' s xs = last (s # map snd xs)
```

```
abbreviation (input) final :: ('a, 's, 'v) trace.t ⇒ 's where
  final σ ≡ trace.final' (trace.init σ) (trace.rest σ)
```

```
definition continue :: ('a, 's, 'v) trace.t ⇒ ('a × 's) list × 'v option ⇒ ('a, 's, 'v) trace.t (infixl <@-s> 64)
where
  σ @-s xsv = (case trace.term σ of None ⇒ trace.T (trace.init σ) (trace.rest σ @ fst xsv) (snd xsv) | Some v
  ⇒ σ)
```

```
definition tl :: ('a, 's, 'v) trace.t → ('a, 's, 'v) trace.t where
  tl σ = (case trace.rest σ of [] ⇒ None | x # xs ⇒ Some (trace.T (snd x) xs (trace.term σ)))
```

```
definition dropn :: nat ⇒ ('a, 's, 'v) trace.t → ('a, 's, 'v) trace.t where
  dropn = (˜) trace.tl
```

definition $take :: nat \Rightarrow ('a, 's, 'v) trace.t \Rightarrow ('a, 's, 'v) trace.t$ **where**

$take\ i\ \sigma = (if\ i \leq length\ (trace.rest\ \sigma)\ then\ trace.T\ (trace.init\ \sigma)\ (List.take\ i\ (trace.rest\ \sigma))\ None\ else\ \sigma)$

type-synonym $('a, 's) transitions = ('a \times 's \times 's) list$

primrec $transitions' :: 's \Rightarrow ('a \times 's) list \Rightarrow ('a, 's) trace.transitions$ **where**

$transitions'\ s\ [] = []$

$| transitions'\ s\ (x \# xs) = (fst\ x,\ s,\ snd\ x) \# transitions'\ (snd\ x)\ xs$

abbreviation $(input) transitions :: ('a, 's, 'v) trace.t \Rightarrow ('a, 's) trace.transitions$ **where**

$transitions\ \sigma \equiv trace.transitions'\ (trace.init\ \sigma)\ (trace.rest\ \sigma)$

$\langle ML \rangle$

lemma $simps[simp]$:

shows $trace.final'\ s\ [] = s$

and $trace.final'\ s\ (x \# xs) = trace.final'\ (snd\ x)\ xs$

and $trace.final'\ s\ (xs\ @\ ys) = trace.final'\ (trace.final'\ s\ xs)\ ys$

and $idle: snd\ 'set\ xs \subseteq \{s\} \implies trace.final'\ s\ xs = s$

and $snd\ 'set\ xs \subseteq \{s\} \implies trace.final'\ s\ (xs\ @\ ys) = trace.final'\ s\ ys$

and $snd\ 'set\ ys \subseteq \{trace.final'\ s\ xs\} \implies trace.final'\ s\ (xs\ @\ ys) = trace.final'\ s\ xs$

$\langle proof \rangle$

lemma map :

shows $trace.final'\ (sf\ s)\ (map\ (map-prod\ af\ sf)\ xs) = sf\ (trace.final'\ s\ xs)$

$\langle proof \rangle$

lemma $replicate$:

shows $trace.final'\ s\ (replicate\ i\ as) = (if\ i = 0\ then\ s\ else\ snd\ as)$

$\langle proof \rangle$

lemma $map-idle$:

assumes $(\lambda x. sf\ (snd\ x))\ 'set\ xs \subseteq \{sf\ s\}$

shows $sf\ (trace.final'\ s\ xs) = sf\ s$

$\langle proof \rangle$

$\langle ML \rangle$

lemma $simps[simp]$:

shows $trace.tl\ (trace.T\ s\ []\ v) = None$

and $trace.tl\ (trace.T\ s\ (x \# xs)\ v) = Some\ (trace.T\ (snd\ x)\ xs\ v)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma $dropn-alt-def$:

shows $trace.dropn\ i\ \sigma$

$= (case\ drop\ i\ ((undefined,\ trace.init\ \sigma) \# trace.rest\ \sigma)\ of$

$[] \Rightarrow None$

$| x \# xs \Rightarrow Some\ (trace.T\ (snd\ x)\ xs\ (trace.term\ \sigma)))$

$\langle proof \rangle$

$\langle ML \rangle$

lemma $simps[simp]$:

shows $0: trace.dropn\ 0 = Some$

and $Suc: trace.dropn\ (Suc\ i)\ \sigma = Option.bind\ (trace.tl\ \sigma)\ (trace.dropn\ i)$

and $dropn: Option.bind\ (trace.dropn\ i\ \sigma)\ (trace.dropn\ j) = trace.dropn\ (i + j)\ \sigma$

<proof>

lemma *Suc-right*:

shows $\text{trace.dropn } (Suc\ i)\ \sigma = \text{Option.bind } (\text{trace.dropn } i\ \sigma)\ \text{trace.tl}$

<proof>

lemma *eq-none-length-conv*:

shows $\text{trace.dropn } i\ \sigma = \text{None} \longleftrightarrow \text{length } (\text{trace.rest } \sigma) < i$

<proof>

lemma *eq-some-length-conv*:

shows $(\exists \sigma'. \text{trace.dropn } i\ \sigma = \text{Some } \sigma') \longleftrightarrow i \leq \text{length } (\text{trace.rest } \sigma)$

<proof>

lemma *eq-some-lengthD*:

assumes $\text{trace.dropn } i\ \sigma = \text{Some } \sigma'$

shows $i \leq \text{length } (\text{trace.rest } \sigma)$

<proof>

<ML>

lemma *sel*:

shows $\text{trace.init } (\text{trace.take } i\ \sigma) = \text{trace.init } \sigma$

and $\text{trace.rest } (\text{trace.take } i\ \sigma) = \text{List.take } i\ (\text{trace.rest } \sigma)$

and $\text{trace.term } (\text{trace.take } i\ \sigma) = (\text{if } i \leq \text{length } (\text{trace.rest } \sigma)\ \text{then } \text{None}\ \text{else } \text{trace.term } \sigma)$

<proof>

lemma *0*:

shows $\text{trace.take } 0\ \sigma = \text{trace.T } (\text{trace.init } \sigma)\ []\ \text{None}$

<proof>

lemma *Nil*:

shows $\text{trace.take } i\ (\text{trace.T } s\ []\ \text{None}) = \text{trace.T } s\ []\ \text{None}$

<proof>

lemmas *simps[simp]* =

trace.take.sel

trace.take.0

trace.take.Nil

lemma *map*:

shows $\text{trace.take } i\ (\text{trace.map } af\ sf\ vf\ \sigma) = \text{trace.map } af\ sf\ vf\ (\text{trace.take } i\ \sigma)$

<proof>

lemma *append*:

shows $\text{trace.take } i\ (\text{trace.T } s\ (xs\ @\ ys)\ v) = \text{trace.T } s\ (\text{List.take } i\ (xs\ @\ ys))\ (\text{if } \text{length } (xs\ @\ ys) < i\ \text{then } v\ \text{else } \text{None})$

<proof>

lemma *take*:

shows $\text{trace.take } i\ (\text{trace.take } j\ \sigma) = \text{trace.take } (\min\ i\ j)\ \sigma$

<proof>

lemma *continue*:

shows $\text{trace.take } i\ (\sigma\ @_{-S}\ xsv)$

$= \text{trace.take } i\ \sigma\ @_{-S}\ (\text{List.take } (i - \text{length } (\text{trace.rest } \sigma))\ (\text{fst } xsv)),$

$\text{if } i \leq \text{length } (\text{trace.rest } \sigma) + \text{length } (\text{fst } xsv)\ \text{then } \text{None}\ \text{else } \text{snd } xsv)$

<proof>

lemma *all-iff*:

shows $\text{trace.take } i \ \sigma = \sigma \longleftrightarrow (\text{case } \text{trace.term } \sigma \text{ of } \text{None} \Rightarrow \text{length } (\text{trace.rest } \sigma) \mid \text{Some } - \Rightarrow \text{Suc } (\text{length } (\text{trace.rest } \sigma))) \leq i$ (**is** *?thesis1*)

and $\sigma = \text{trace.take } i \ \sigma \longleftrightarrow (\text{case } \text{trace.term } \sigma \text{ of } \text{None} \Rightarrow \text{length } (\text{trace.rest } \sigma) \mid \text{Some } - \Rightarrow \text{Suc } (\text{length } (\text{trace.rest } \sigma))) \leq i$ (**is** *?thesis2*)

<proof>

lemmas *all* = *iffD2*[*OF trace.take.all-iff(1)*]

lemma *Ex-all*:

shows $\sigma = \text{trace.take } (\text{Suc } (\text{length } (\text{trace.rest } \sigma))) \ \sigma$

<proof>

lemma *replicate*:

shows $\text{trace.take } i \ (\text{trace.T } s \ (\text{replicate } j \ as) \ v)$
 $= \text{trace.T } s \ (\text{replicate } (\text{min } i \ j) \ as) \ (\text{if } i \leq j \ \text{then } \text{None} \ \text{else } v)$

<proof>

<ML>

lemma *sel[simp]*:

shows $\text{trace.init } (\sigma \ @_{-S} \ xs) = \text{trace.init } \sigma$
and $\text{trace.rest } (\sigma \ @_{-S} \ xs) = (\text{case } \text{trace.term } \sigma \text{ of } \text{None} \Rightarrow \text{trace.rest } \sigma \ @ \ \text{fst } xs \mid \text{Some } v \Rightarrow \text{trace.rest } \sigma)$
and $\text{trace.term } (\sigma \ @_{-S} \ xs) = (\text{case } \text{trace.term } \sigma \text{ of } \text{None} \Rightarrow \text{snd } xs \mid \text{Some } v \Rightarrow \text{trace.term } \sigma)$

<proof>

lemma *simps[simp]*:

shows $\text{trace.T } s \ xs \ \text{None} \ @_{-S} \ ysv = \text{trace.T } s \ (xs \ @ \ \text{fst } ysv) \ (\text{snd } ysv)$
and $\text{trace.T } s \ xs \ (\text{Some } v) \ @_{-S} \ ysv = \text{trace.T } s \ xs \ (\text{Some } v)$
and $\sigma \ @_{-S} \ ([], \ \text{None}) = \sigma$

<proof>

lemma *Nil*:

shows $\sigma \ @_{-S} \ ([], \ \text{trace.term } \sigma) = \sigma$
and $\text{trace.T } (\text{trace.init } \sigma) \ [] \ \text{None} \ @_{-S} \ (\text{trace.rest } \sigma, \ \text{trace.term } \sigma) = \sigma$

<proof>

lemma *map*:

shows $\text{trace.map } af \ sf \ vf \ (\sigma \ @_{-S} \ xs) = \text{trace.map } af \ sf \ vf \ \sigma \ @_{-S} \ \text{map-prod } (\text{map } (\text{map-prod } af \ sf)) \ (\text{map-option } vf) \ xs$

<proof>

lemma *eq-trace-conv*:

shows $\sigma \ @_{-S} \ xs = \text{trace.T } s \ xs \ v \longleftrightarrow \text{trace.init } \sigma = s \wedge (\text{case } \text{trace.term } \sigma \text{ of } \text{None} \Rightarrow \text{trace.rest } \sigma \ @ \ \text{fst } xs \mid \text{Some } v' \Rightarrow \text{trace.rest } \sigma = xs \wedge v = \text{Some } v')$

and $\text{trace.T } s \ xs \ v = \sigma \ @_{-S} \ xs \longleftrightarrow \text{trace.init } \sigma = s \wedge (\text{case } \text{trace.term } \sigma \text{ of } \text{None} \Rightarrow \text{trace.rest } \sigma \ @ \ \text{fst } xs \mid \text{Some } v' \Rightarrow \text{trace.rest } \sigma = xs \wedge v = \text{Some } v')$

<proof>

lemma *self-conv*:

shows $(\sigma = \sigma \ @_{-S} \ xs) \longleftrightarrow (\text{case } \text{trace.term } \sigma \text{ of } \text{None} \Rightarrow xs = ([], \ \text{None}) \mid \text{Some } - \Rightarrow \text{True})$
and $(\sigma \ @_{-S} \ xs = \sigma) \longleftrightarrow (\text{case } \text{trace.term } \sigma \text{ of } \text{None} \Rightarrow xs = ([], \ \text{None}) \mid \text{Some } - \Rightarrow \text{True})$

<proof>

lemma *same-eq*:

shows $(\sigma \ @_{-S} \ xs = \sigma \ @_{-S} \ ysv) \longleftrightarrow (\text{case } \text{trace.term } \sigma \text{ of } \text{None} \Rightarrow xs = ysv \mid \text{Some } - \Rightarrow \text{True})$

<proof>

lemma *continue*:

shows $\sigma @-s xsv @-s ysv = \sigma @-s (\text{case } \text{snd } xsv \text{ of } \text{None} \Rightarrow (\text{fst } xsv @ \text{fst } ysv, \text{snd } ysv) \mid \text{Some } - \Rightarrow xsv)$
<proof>

lemma *take-drop-id*:

shows $\text{trace.take } i \sigma @-s \text{case-option } ([], \text{None}) (\lambda \sigma'. (\text{trace.rest } \sigma', \text{trace.term } \sigma')) (\text{trace.dropn } i \sigma) = \sigma$
<proof>

<ML>

Prefix ordering instantiation $\text{trace.t} :: (\text{type}, \text{type}, \text{type}) \text{ order}$
begin

definition *less-eq-t* :: $(\text{'a}, \text{'s}, \text{'v}) \text{ trace.t relp}$ **where**

$\text{less-eq-t } \sigma_1 \sigma_2 \longleftrightarrow (\exists xsv. \sigma_2 = \sigma_1 @-s xsv)$

definition *less-t* :: $(\text{'a}, \text{'s}, \text{'v}) \text{ trace.t relp}$ **where**

$\text{less-t } \sigma_1 \sigma_2 \longleftrightarrow \sigma_1 \leq \sigma_2 \wedge \sigma_1 \neq \sigma_2$

instance

<proof>

end

lemma *less-eqE*[*consumes 1, case-names prefix maximal*]:

assumes $\sigma_1 \leq \sigma_2$

assumes $\llbracket \text{trace.term } \sigma_1 = \text{None}; \text{trace.init } \sigma_1 = \text{trace.init } \sigma_2; \text{prefix } (\text{trace.rest } \sigma_1) (\text{trace.rest } \sigma_2) \rrbracket \Longrightarrow P$

assumes $\bigwedge v. \llbracket \text{trace.term } \sigma_1 = \text{Some } v; \sigma_1 = \sigma_2 \rrbracket \Longrightarrow P$

shows P

<proof>

lemmas *less-eq-extE*[*consumes 1, case-names prefix maximal*]

$= \text{trace.less-eqE}$ [*of trace.T s₁ xs₁ v₁ trace.T s₂ xs₂ v₂, simplified, simplified conj-explode*]

for $s_1 xs_1 v_1 s_2 xs_2 v_2$

lemma *less-eq-self-continue*:

shows $\sigma \leq \sigma @-s xsv$

<proof>

lemma *less-eq-same-append-conv*:

shows $\text{trace.T } s xs v \leq \text{trace.T } s' (xs @ ys) v' \longleftrightarrow s = s' \wedge (\forall v''. v = \text{Some } v'' \longrightarrow ys = [] \wedge v = v')$

<proof>

lemma *less-same-append-conv*:

shows $\text{trace.T } s xs v < \text{trace.T } s' (xs @ ys) v' \longleftrightarrow s = s' \wedge v = \text{None} \wedge (ys \neq [] \vee (\exists v''. v' = \text{Some } v''))$

<proof>

lemma *less-eq-Some*[*simp*]:

shows $\text{trace.T } s xs (\text{Some } v) \leq \sigma \longleftrightarrow \text{trace.init } \sigma = s \wedge \text{trace.rest } \sigma = xs \wedge \text{trace.term } \sigma = \text{Some } v$

<proof>

lemma *less-eq-None*:

shows $\sigma \leq \text{trace.T } s xs \text{None} \longleftrightarrow \text{trace.init } \sigma = s \wedge \text{prefix } (\text{trace.rest } \sigma) xs \wedge \text{trace.term } \sigma = \text{None}$

and $\text{trace.T } s xs \text{None} \leq \sigma \longleftrightarrow \text{trace.init } \sigma = s \wedge \text{prefix } xs (\text{trace.rest } \sigma)$

<proof>

lemma *less*:

shows $\text{trace}.T\ s\ xs\ v < \sigma \iff \text{trace}.init\ \sigma = s \wedge (\exists ys. \text{trace}.rest\ \sigma = xs @ ys \wedge (\text{trace}.term\ \sigma = None \longrightarrow ys \neq [])) \wedge v = None$

and $\sigma < \text{trace}.T\ s\ xs\ v \iff \text{trace}.init\ \sigma = s \wedge (\exists ys. xs = \text{trace}.rest\ \sigma @ ys \wedge (v = None \longrightarrow ys \neq [])) \wedge \text{trace}.term\ \sigma = None$

$\langle \text{proof} \rangle$

lemma *less-eq-take[iff]*:

shows $\text{trace}.take\ i\ \sigma \leq \sigma$

$\langle \text{proof} \rangle$

lemma *less-eq-takeE*:

assumes $\sigma_1 \leq \sigma_2$

obtains i **where** $\sigma_1 = \text{trace}.take\ i\ \sigma_2$

$\langle \text{proof} \rangle$

lemma *less-eq-take-def*:

shows $\sigma_1 \leq \sigma_2 \iff (\exists i. \sigma_1 = \text{trace}.take\ i\ \sigma_2)$

$\langle \text{proof} \rangle$

lemma *less-take-less-eq*:

assumes $\sigma < \text{trace}.take\ (Suc\ i)\ \sigma'$

shows $\sigma \leq \text{trace}.take\ i\ \sigma'$

$\langle \text{proof} \rangle$

lemma *wfP-less*:

shows $wfP\ ((<) :: (-, -, -)\ \text{trace}.t\ \text{relp})$

$\langle \text{proof} \rangle$

lemma *less-eq-same-cases*:

fixes $ys :: (-, -, -)\ \text{trace}.t$

assumes $xs_1 \leq ys$

assumes $xs_2 \leq ys$

shows $xs_1 \leq xs_2 \vee xs_2 \leq xs_1$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *mono*:

assumes $\sigma_1 \leq \sigma_2$

assumes $i \leq j$

shows $\text{trace}.take\ i\ \sigma_1 \leq \text{trace}.take\ j\ \sigma_2$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemmas $\text{map} = \text{trace}.t.\text{map}\text{-comp}[\text{unfolded}\ \text{comp}\text{-def}]$

lemma *monotone*:

shows $\text{mono}\ (\text{trace}.map\ af\ sf\ vf)$

$\langle \text{proof} \rangle$

lemmas $\text{strengthen}[\text{strg}] = \text{st}\text{-monotone}[OF\ \text{trace}.map.\text{monotone}]$

lemmas $\text{mono} = \text{monoD}[OF\ \text{trace}.map.\text{monotone}]$

lemma *monotone-less*:

shows $\text{monotone}\ (<)\ (<)\ (\text{trace}.map\ af\ sf\ vf)$

$\langle \text{proof} \rangle$

lemma *less-eqR*:

assumes $\sigma_1 \leq \text{trace.map af sf vf } \sigma_2$

obtains σ_2' **where** $\sigma_2' \leq \sigma_2$ **and** $\sigma_1 = \text{trace.map af sf vf } \sigma_2'$

<proof>

<ML>

lemmas $\text{eq} = \text{trace.t.rel-eq}$

lemmas $\text{mono} = \text{trace.t.rel-mono-strong[of ar sr vr } \sigma_1 \sigma_2 \text{ ar' sr' vr']}$ **for** $\text{ar sr vr } \sigma_1 \sigma_2 \text{ ar' sr' vr'}$

lemma *strengthen[strg]*:

assumes $\text{st-ord } F \text{ ar ar'}$

assumes $\text{st-ord } F \text{ sr sr'}$

assumes $\text{st-ord } F \text{ vr vr'}$

shows $\text{st-ord } F (\text{trace.rel ar sr vr } \sigma_1 \sigma_2) (\text{trace.rel ar' sr' vr' } \sigma_1 \sigma_2)$

<proof>

lemma *length-rest*:

assumes $\text{trace.rel ar sr vr } \sigma_1 \sigma_2$

shows $\text{length } (\text{trace.rest } \sigma_1)$

$= \text{length } (\text{trace.rest } \sigma_2) \wedge (\forall i < \text{length } (\text{trace.rest } \sigma_1). \text{rel-prod ar sr } (\text{trace.rest } \sigma_1 ! i) (\text{trace.rest } \sigma_2 ! i))$

<proof>

<ML>

lemma *rel*:

assumes $\text{trace.rel ar sr vr } \sigma_1 \sigma_2$

shows $\text{trace.rel ar sr vr } (\text{trace.take } i \sigma_1) (\text{trace.take } i \sigma_2)$

<proof>

<ML>

lemma *prefix-conv*:

shows $\text{prefix } (\text{trace.transitions' } s \text{ xs}) (\text{trace.transitions' } s \text{ ys}) \longleftrightarrow \text{prefix } \text{xs } \text{ys}$

<proof>

lemma *monotone*:

shows $\text{monotone prefix prefix } (\text{trace.transitions' } s)$

<proof>

lemma *append*:

shows $\text{trace.transitions' } s (\text{xs } @ \text{ys}) = \text{trace.transitions' } s \text{xs } @ \text{trace.transitions' } (\text{trace.final' } s \text{xs}) \text{ys}$

<proof>

lemma *eq-Nil-conv*:

shows $\text{trace.transitions' } s \text{xs} = [] \longleftrightarrow \text{xs} = []$

and $[] = \text{trace.transitions' } s \text{xs} \longleftrightarrow \text{xs} = []$

<proof>

lemma *eq-Cons-conv*:

shows $\text{trace.transitions' } s \text{xs} = \text{y } \# \text{ys} \longleftrightarrow (\exists a \text{ s' xs'}. \text{xs} = (a, \text{s'}) \# \text{xs'} \wedge \text{y} = (a, \text{s}, \text{s'}) \wedge \text{ys} = \text{trace.transitions' } \text{s' } \text{xs'})$

and $\text{y } \# \text{ys} = \text{trace.transitions' } s \text{xs} \longleftrightarrow (\exists a \text{ s' xs'}. \text{xs} = (a, \text{s'}) \# \text{xs'} \wedge \text{y} = (a, \text{s}, \text{s'}) \wedge \text{ys} = \text{trace.transitions' } \text{s' } \text{xs'})$

<proof>

lemma *inj-conv*:

shows $\text{trace.transitions}' s xs = \text{trace.transitions}' s ys \longleftrightarrow xs = ys$
<proof>

lemma *continue*:

shows $\text{trace.transitions} (\sigma @_{-s} xsv)$
 $= \text{trace.transitions} \sigma @ (\text{case trace.term } \sigma \text{ of None } \Rightarrow \text{trace.transitions}' (\text{trace.final } \sigma) (\text{fst } xsv) \mid \text{Some } v \Rightarrow$
 $[])$
<proof>

lemma *idle-conv*:

shows $\text{set} (\text{trace.transitions}' s xs) \subseteq \text{UNIV} \times \text{Id} \longleftrightarrow \text{snd } ' \text{ set } xs \subseteq \{s\}$
<proof>

lemma *map*:

shows $\text{trace.transitions}' (sf s) (\text{map} (\text{map-prod } af \ sf) xs)$
 $= \text{map} (\text{map-prod } af (\text{map-prod } sf \ sf)) (\text{trace.transitions}' s xs)$
<proof>

<ML>

lemma *monotone*:

shows *monotone* (\leq) *prefix* trace.transitions
<proof>

lemmas *mono* = *monotoneD*[*OF* $\text{trace.transitions.monotone}$]

lemma *subseq*:

assumes $\sigma \leq \sigma'$
shows *subseq* ($\text{trace.transitions} \sigma$) ($\text{trace.transitions} \sigma'$)
<proof>

<ML>

type-synonym ($'a, 's$) *steps* = ($'a \times 's \times 's$) *set*

<ML>

definition *steps'* :: $'s \Rightarrow ('a \times 's)$ *list* $\Rightarrow ('a, 's)$ *steps* **where**

$\text{steps}' s xs = \text{set} (\text{trace.transitions}' s xs) - \text{UNIV} \times \text{Id}$

abbreviation (*input*) *steps* :: ($'a, 's, 'v$) $\text{trace.t} \Rightarrow ('a, 's)$ *steps* **where**

$\text{steps} \sigma \equiv \text{trace.steps}' (\text{trace.init } \sigma) (\text{trace.rest } \sigma)$

<ML>

lemma *simps*[*simp*]:

shows $\text{trace.steps}' s [] = \{\}$
and $\text{trace.steps}' s ((a, s) \# xs) = \text{trace.steps}' s xs$
and $s \neq \text{snd } x \Longrightarrow \text{trace.steps}' s (x \# xs) = \text{insert} (\text{fst } x, s, \text{snd } x) (\text{trace.steps}' (\text{snd } x) xs)$
and $(a, s', s') \notin \text{trace.steps}' s xs$
and $\text{snd } ' \text{ set } xs \subseteq \{s\} \Longrightarrow \text{trace.steps}' s xs = \{\}$
and $\text{trace.steps}' s [x] = (\text{if } s = \text{snd } x \text{ then } \{\} \text{ else } \{(\text{fst } x, s, \text{snd } x)\})$
<proof>

lemma *Cons-eq-if*:

shows $\text{trace.steps}' s (x \# xs)$
 $= (\text{if } s = \text{snd } x \text{ then } \text{trace.steps}' s xs \text{ else } \text{insert} (\text{fst } x, s, \text{snd } x) (\text{trace.steps}' (\text{snd } x) xs))$

$\langle \text{proof} \rangle$

lemma *stuttering*:

shows $\text{trace.steps}' s xs \subseteq r \cup A \times Id \iff \text{trace.steps}' s xs \subseteq r$

and $\text{trace.steps}' s xs \subseteq A \times Id \cup r \iff \text{trace.steps}' s xs \subseteq r$

$\langle \text{proof} \rangle$

lemma *empty-conv[simp]*:

shows $\text{trace.steps}' s xs = \{\} \iff \text{snd} \text{ ' set } xs \subseteq \{s\} \text{ (is ?thesis1)}$

and $\{\} = \text{trace.steps}' s xs \iff \text{snd} \text{ ' set } xs \subseteq \{s\} \text{ (is ?thesis2)}$

$\langle \text{proof} \rangle$

lemma *append*:

shows $\text{trace.steps}' s (xs @ ys)$

$= \text{trace.steps}' s xs \cup \text{trace.steps}' (\text{trace.final}' s xs) ys$

$\langle \text{proof} \rangle$

lemma *map*:

shows $\text{trace.steps}' (sf s) (\text{map} (\text{map-prod } af sf) xs) = \text{map-prod } af (\text{map-prod } sf sf) \text{ ' trace.steps}' s xs - UNIV \times Id$

and $\text{trace.steps}' s (\text{map} (\text{map-prod } af id) xs) = \text{map-prod } af id \text{ ' trace.steps}' s xs - UNIV \times Id$

$\langle \text{proof} \rangle$

lemma *memberD*:

assumes $(a, s, s') \in \text{trace.steps}' s_0 xs$

shows $(a, s') \in \text{set } xs$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *monotone*:

shows $\text{mono } \text{trace.steps}$

$\langle \text{proof} \rangle$

lemmas $\text{mono} = \text{monoD}[OF \text{ trace.steps.monotone}]$

lemmas $\text{strengthen}[strg] = \text{st-monotone}[OF \text{ trace.steps.monotone}]$

$\langle ML \rangle$

lemma *simps*:

shows $\text{trace.aset} (\text{trace.T } s xs v) = \text{fst} \text{ ' set } xs$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *simps*:

shows $\text{trace.sset} (\text{trace.T } s xs v) = \text{insert } s (\text{snd} \text{ ' set } xs)$

$\langle \text{proof} \rangle$

lemma *dropn-le*:

assumes $\text{trace.dropn } i \sigma = \text{Some } \sigma'$

shows $\text{trace.sset } \sigma' \subseteq \text{trace.sset } \sigma$

$\langle \text{proof} \rangle$

lemma *take-le*:

shows $\text{trace.sset} (\text{trace.take } i \sigma) \subseteq \text{trace.sset } \sigma$

$\langle \text{proof} \rangle$

lemma *mono*:

shows *mono trace.sset*

<proof>

<ML>

2.3 Behaviors

<ML>

datatype (*aset*: 'a, *sset*: 's, *vset*: 'v) *t* =
 B (*init*: 's) (*rest*: ('a × 's, 'v) *tlist*)

for

map: *map*

definition *term* :: ('a, 's, 'v) *behavior.t* ⇒ 'v *option* **where**

term ω = (if *tfinite* (*behavior.rest* ω) then *Some* (*terminal* (*behavior.rest* ω)) else *None*)

declare *behavior.t.map-id0*[*simp*]

declare *behavior.t.map-id0*[*unfolded id-def, simp*]

declare *behavior.t.map-sel*[*simp*]

declare *behavior.t.set-map*[*simp*]

declare *behavior.t.map-comp*[*unfolded o-def, simp*]

declare *behavior.t.set*[*simp del*]

lemma *split-all*[*no-atp*]: — imitate the setup for 'a × 'b without the automation

shows (∧*x*. *PROP P x*) ≡ (∧*s xs*. *PROP P (behavior.B s xs)*)

<proof>

lemma *split-All*[*no-atp*]:

shows (∀*x*. *P x*) ↔ (∀*s xs*. *P (behavior.B s xs)*) (**is** ?*lhs* ↔ ?*rhs*)

<proof>

lemma *split-Ex*[*no-atp*]:

shows (∃*x*. *P x*) ↔ (∃*s xs*. *P (behavior.B s xs)*) (**is** ?*lhs* ↔ ?*rhs*)

<proof>

2.4 Combinators on behaviors

definition *continue* :: ('a, 's, 'v) *trace.t* ⇒ ('a × 's, 'v) *tlist* ⇒ ('a, 's, 'v) *behavior.t* (**infix** <@-*B*> 64) **where**

σ @-*B* *xs* = *behavior.B* (*trace.init* σ) (*tshift2* (*trace.rest* σ, *trace.term* σ) *xs*)

definition *tl* :: ('a, 's, 'v) *behavior.t* → ('a, 's, 'v) *behavior.t* **where**

tl ω = (case *behavior.rest* ω of *TNil v* ⇒ *None* | *TCons x xs* ⇒ *Some (behavior.B (snd x) xs)*)

definition *dropn* :: nat ⇒ ('a, 's, 'v) *behavior.t* → ('a, 's, 'v) *behavior.t* **where**

dropn = (∧) *behavior.tl*

definition *take* :: nat ⇒ ('a, 's, 'v) *behavior.t* ⇒ ('a, 's, 'v) *trace.t* **where**

take i ω = *uncurry* (*trace.T* (*behavior.init* ω)) (*ttake i (behavior.rest* ω))

<ML>

lemma *simps*:

shows *trace.T s xs None* @-*B* *ys* = *behavior.B s (tshift xs ys)*

and *trace.T s xs (Some v)* @-*B* *ys* = *behavior.B s (tshift xs (TNil v))*

and *trace.T s (x # xs) w* @-*B* *ys* = *behavior.B s (TCons x (tshift2 (xs, w) ys))*

<proof>

lemma *sel[simp]*:

shows *init*: $\text{behavior.init } (\sigma @_{-B} xs) = \text{trace.init } \sigma$

and *rest*: $\text{behavior.rest } (\sigma @_{-B} xs) = \text{tshift2 } (\text{trace.rest } \sigma, \text{trace.term } \sigma) xs$

<proof>

lemma *term-None*:

assumes $\text{trace.term } \sigma = \text{None}$

shows $\sigma @_{-B} xs = \text{behavior.B } (\text{trace.init } \sigma) (\text{tshift } (\text{trace.rest } \sigma) xs)$

<proof>

lemma *term-Some*:

assumes $\text{trace.term } \sigma = \text{Some } v$

shows $\sigma @_{-B} xs = \text{behavior.B } (\text{trace.init } \sigma) (\text{tshift } (\text{trace.rest } \sigma) (\text{TNil } v))$

<proof>

lemma *tshift2*:

shows $\sigma @_{-B} \text{tshift2 } xsv ys = (\sigma @_{-S} xsv) @_{-B} ys$

<proof>

<ML>

lemma *TNil*:

shows $\text{behavior.tl } (\text{behavior.B } s (\text{TNil } v)) = \text{None}$

<proof>

lemma *TCons*:

shows $\text{behavior.tl } (\text{behavior.B } s (\text{TCons } x xs)) = \text{Some } (\text{behavior.B } (\text{snd } x) xs)$

<proof>

lemma *eq-None-conv*:

shows $\text{behavior.tl } \omega = \text{None} \iff \text{is-TNil } (\text{behavior.rest } \omega)$

<proof>

lemma *continue-Cons*:

shows $\text{behavior.tl } (\text{trace.T } s (x \# xs) v @_{-B} ys) = \text{Some } (\text{trace.T } (\text{snd } x) xs v @_{-B} ys)$

<proof>

lemmas *simps[simp]* =

behavior.tl.TNil

behavior.tl.TCons

behavior.tl.eq-None-conv

behavior.tl.continue-Cons

lemma *tfiniteD*:

assumes $\text{behavior.tl } \omega = \text{Some } \omega'$

shows $\text{tfinite } (\text{behavior.rest } \omega') \iff \text{tfinite } (\text{behavior.rest } \omega)$

<proof>

<ML>

lemma *dropn-alt-def*:

shows $\text{behavior.dropn } i \omega$

$= (\text{case } \text{tdropn } i (\text{TCons } (\text{undefined}, \text{behavior.init } \omega) (\text{behavior.rest } \omega)) \text{ of}$

$\text{TNil} - \Rightarrow \text{None}$

$| \text{TCons } x xs \Rightarrow \text{Some } (\text{behavior.B } (\text{snd } x) xs))$

<proof>

$\langle ML \rangle$

lemma *simps[simp]*:

shows 0 : $\text{behavior.dropn } 0 \ \omega = \text{Some } \omega$

and *TNil*: $\text{behavior.dropn } i \ (\text{behavior.B } s \ (\text{TNil } v)) = (\text{case } i \ \text{of } 0 \Rightarrow \text{Some } (\text{behavior.B } s \ (\text{TNil } v)) \mid - \Rightarrow \text{None})$

$\langle \text{proof} \rangle$

lemma *TCons*:

shows $\text{behavior.dropn } i \ (\text{behavior.B } s \ (\text{TCons } x \ xs))$

$= (\text{case } i \ \text{of } 0 \Rightarrow \text{Some } (\text{behavior.B } s \ (\text{TCons } x \ xs)) \mid \text{Suc } j \Rightarrow \text{behavior.dropn } j \ (\text{behavior.B } (\text{snd } x) \ xs))$

$\langle \text{proof} \rangle$

lemma *Suc*:

shows $\text{behavior.dropn } (\text{Suc } i) \ \omega = \text{Option.bind } (\text{behavior.tl } \omega) \ (\text{behavior.dropn } i)$

$\langle \text{proof} \rangle$

lemma *bind-tl-commute*:

shows $\text{behavior.tl } \omega \gg= \text{behavior.dropn } i = \text{behavior.dropn } i \ \omega \gg= \text{behavior.tl}$

$\langle \text{proof} \rangle$

lemma *Suc-right*:

shows $\text{behavior.dropn } (\text{Suc } i) \ \omega = \text{Option.bind } (\text{behavior.dropn } i \ \omega) \ \text{behavior.tl}$

$\langle \text{proof} \rangle$

lemma *dropn*:

shows $\text{Option.bind } (\text{behavior.dropn } i \ \omega) \ (\text{behavior.dropn } j) = \text{behavior.dropn } (i + j) \ \omega$

$\langle \text{proof} \rangle$

lemma *add*:

shows $\text{behavior.dropn } (i + j) = (\lambda \omega. \text{Option.bind } (\text{behavior.dropn } i \ \omega) \ (\text{behavior.dropn } j))$

$\langle \text{proof} \rangle$

lemma *tfiniteD*:

assumes $\text{behavior.dropn } i \ \omega = \text{Some } \omega'$

shows $t\text{finite } (\text{behavior.rest } \omega') \longleftrightarrow t\text{finite } (\text{behavior.rest } \omega)$

$\langle \text{proof} \rangle$

lemma *shorterD*:

assumes $\text{behavior.dropn } i \ \omega = \text{Some } \omega'$

assumes $j \leq i$

shows $\exists \omega''. \text{behavior.dropn } j \ \omega = \text{Some } \omega''$

$\langle \text{proof} \rangle$

lemma *eq-None-tlength-conv*:

shows $\text{behavior.dropn } i \ \omega = \text{None} \longleftrightarrow t\text{length } (\text{behavior.rest } \omega) < \text{enat } i$

$\langle \text{proof} \rangle$

lemma *eq-Some-tlength-conv*:

shows $(\exists \omega'. \text{behavior.dropn } i \ \omega = \text{Some } \omega') \longleftrightarrow \text{enat } i \leq t\text{length } (\text{behavior.rest } \omega)$

$\langle \text{proof} \rangle$

lemma *eq-Some-tlengthD*:

assumes $\text{behavior.dropn } i \ \omega = \text{Some } \omega'$

shows $\text{enat } i \leq t\text{length } (\text{behavior.rest } \omega)$

$\langle \text{proof} \rangle$

lemma *tlength-eq-SomeD*:

assumes $enat\ i \leq tlength\ (behavior.rest\ \omega)$
shows $\exists \omega'.\ behavior.dropn\ i\ \omega = Some\ \omega'$
 $\langle proof \rangle$

lemma *eq-Some-tdropnD*:
assumes $behavior.dropn\ i\ \omega = Some\ \omega'$
shows $tdropn\ i\ (behavior.rest\ \omega) = behavior.rest\ \omega'$
 $\langle proof \rangle$

lemma *continue-shorter*:
assumes $i \leq length\ (trace.rest\ \sigma)$
shows $behavior.dropn\ i\ (\sigma\ @-B\ xs) = Option.bind\ (trace.dropn\ i\ \sigma)\ (\lambda \sigma'.\ Some\ (\sigma'\ @-B\ xs))$
 $\langle proof \rangle$

lemma *continue-Some*:
assumes $length\ (trace.rest\ \sigma) < i$
assumes $trace.term\ \sigma = Some\ v$
shows $behavior.dropn\ i\ (\sigma\ @-B\ xs) = None$
 $\langle proof \rangle$

lemma *continue-None*:
assumes $length\ (trace.rest\ \sigma) < i$
assumes $trace.term\ \sigma = None$
shows $behavior.dropn\ i\ (\sigma\ @-B\ xs)$
 $= (case\ tdropn\ (i - Suc\ (length\ (trace.rest\ \sigma)))\ xs\ of$
 $\quad TNil - \Rightarrow None$
 $\quad | TCons\ y\ ys \Rightarrow Some\ (behavior.B\ (snd\ y)\ ys))$
 $\langle proof \rangle$

lemma *continue*:
shows $behavior.dropn\ i\ (\sigma\ @-B\ xs)$
 $= (if\ i \leq length\ (trace.rest\ \sigma)$
 $\quad then\ Option.bind\ (trace.dropn\ i\ \sigma)\ (\lambda \sigma'.\ Some\ (\sigma'\ @-B\ xs))$
 $\quad else\ if\ trace.term\ \sigma = None$
 $\quad then\ case\ tdropn\ (i - Suc\ (length\ (trace.rest\ \sigma)))\ xs\ of$
 $\quad \quad TNil - \Rightarrow None$
 $\quad \quad | TCons\ y\ ys \Rightarrow Some\ (behavior.B\ (snd\ y)\ ys)$
 $\quad else\ None)$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *take*:
shows $trace.take\ i\ (behavior.take\ j\ \omega) = behavior.take\ (min\ i\ j)\ \omega$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *simps[simp]*:
shows $0: behavior.take\ 0\ \omega = trace.T\ (behavior.init\ \omega)\ []\ None$
and $Suc-TNil: behavior.take\ (Suc\ i)\ (behavior.B\ s\ (TNil\ v)) = trace.T\ s\ []\ (Some\ v)$
 $\langle proof \rangle$

lemma *sel[simp]*:
shows $trace.init\ (behavior.take\ i\ \omega) = behavior.init\ \omega$
and $trace.rest\ (behavior.take\ i\ \omega) = fst\ (ttake\ i\ (behavior.rest\ \omega))$
and $trace.term\ (behavior.take\ i\ \omega) = snd\ (ttake\ i\ (behavior.rest\ \omega))$
 $\langle proof \rangle$

lemma *monotone*:

shows $\text{mono } (\lambda i. \text{behavior.take } i \ \omega)$

$\langle \text{proof} \rangle$

lemmas $\text{mono} = \text{monoD}[\text{OF } \text{behavior.take.monotone}]$

lemma *map*:

shows $\text{behavior.take } i \ (\text{behavior.map } af \ sf \ vf \ \omega) = \text{trace.map } af \ sf \ vf \ (\text{behavior.take } i \ \omega)$

$\langle \text{proof} \rangle$

lemma *continue*:

shows $\text{behavior.take } i \ (\sigma \ @_{-B} \ \omega) = \text{trace.take } i \ \sigma \ @_{-S} \ \text{ttake } (i - \text{length } (\text{trace.rest } \sigma)) \ \omega$

$\langle \text{proof} \rangle$

lemma *all-continue*:

assumes $\text{tlength } (\text{behavior.rest } \omega) < \text{enat } i$

shows $\text{behavior.take } i \ \omega \ @_{-S} \ xsv = \text{behavior.take } i \ \omega$

$\langle \text{proof} \rangle$

lemma *continue-same*:

shows $\text{behavior.take } i \ (\text{behavior.take } i \ \omega \ @_{-B} \ xsv) = \text{behavior.take } i \ \omega$

$\langle \text{proof} \rangle$

lemma *trePLICATE*:

shows $\text{behavior.take } i \ (\text{behavior.B } s \ (\text{trePLICATE } j \ as \ v))$

$= \text{trace.T } s \ (\text{List.replicate } (\text{min } i \ j) \ as) \ (\text{if } j < i \ \text{then } \text{Some } v \ \text{else } \text{None})$

$\langle \text{proof} \rangle$

lemma *trepeat*:

shows $\text{behavior.take } i \ (\text{behavior.B } s \ (\text{trepeat } as)) = \text{trace.T } s \ (\text{List.replicate } i \ as) \ \text{None}$

$\langle \text{proof} \rangle$

lemma *tshift*:

shows $\text{behavior.take } i \ (\text{behavior.B } s \ (\text{tshift } xs \ ys)) = \text{trace.take } i \ (\text{trace.T } s \ xs \ \text{None}) \ @_{-S} \ \text{ttake } (i - \text{length } xs)$

$\langle \text{proof} \rangle$

lemma *length*:

shows $\text{length } (\text{trace.rest } (\text{behavior.take } j \ \omega))$

$= (\text{case } \text{tlength } (\text{behavior.rest } \omega) \ \text{of } \text{enat } i \Rightarrow \text{min } i \ j \mid \infty \Rightarrow j)$

$\langle \text{proof} \rangle$

lemma *add*:

shows $\text{behavior.take } (i + j) \ \omega$

$= \text{behavior.take } i \ \omega \ @_{-S} \ (\text{case } \text{behavior.dropn } i \ \omega \ \text{of } \text{Some } \omega' \Rightarrow \text{ttake } j \ (\text{behavior.rest } \omega'))$

$\langle \text{proof} \rangle$

lemma *term-Some-conv*:

shows $\text{trace.term } (\text{behavior.take } j \ \omega) = \text{Some } v$

$\iff (\text{tlength } (\text{behavior.rest } \omega) < \text{enat } j \wedge \text{Some } v = \text{behavior.term } \omega)$

$\langle \text{proof} \rangle$

lemma *dropn*:

assumes $\text{behavior.dropn } i \ \omega = \text{Some } \omega'$

shows $\text{behavior.take } j \ \omega' = \text{the } (\text{trace.dropn } i \ (\text{behavior.take } (i + j) \ \omega))$

$\langle \text{proof} \rangle$

lemma *continue-id*:

assumes $\text{tlength } (\text{behavior.rest } \omega) < \text{enat } i$

shows $\text{behavior.take } i \ \omega @_{-B} \text{xs} = \omega$

$\langle \text{proof} \rangle$

lemma *flat*:

assumes $\text{tlength } (\text{behavior.rest } \omega) < \text{enat } i$

assumes $i \leq j$

shows $\text{behavior.take } i \ \omega = \text{behavior.take } j \ \omega$

$\langle \text{proof} \rangle$

lemma *eqI*:

assumes $\bigwedge i. \text{behavior.take } i \ \omega_1 = \text{behavior.take } i \ \omega_2$

shows $\omega_1 = \omega_2$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *take-drop-shorter*:

assumes $i \leq j$

shows $\text{behavior.take } i \ \omega @_{-S} \text{apfst } (\text{drop } i) (\text{take } j (\text{behavior.rest } \omega)) = \text{behavior.take } j \ \omega$

$\langle \text{proof} \rangle$

lemma *take-drop-id*:

shows $\text{behavior.take } i \ \omega @_{-B} \text{behavior.rest } (\text{the } (\text{behavior.dropn } i \ \omega)) = \omega$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *simps*:

shows $\text{behavior.aset } (\text{behavior.B } s \ \text{xs}) = \text{fst } ' \text{tset } \ \text{xs}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *simps*:

shows $\text{behavior.sset } (\text{behavior.B } s \ \text{xs}) = \text{insert } s \ (\text{snd } ' \text{tset } \ \text{xs})$

$\langle \text{proof} \rangle$

lemma *dropn-le*:

assumes $\text{behavior.dropn } i \ \omega = \text{Some } \omega'$

shows $\text{behavior.sset } \omega' \subseteq \text{behavior.sset } \omega$

$\langle \text{proof} \rangle$

lemma *take-le*:

shows $\text{trace.sset } (\text{behavior.take } i \ \omega) \subseteq \text{behavior.sset } \omega$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *take*:

shows $\text{trace.dropn } i (\text{behavior.take } j \ \omega)$

$= (\text{if } i \leq j \text{ then } \text{Option.bind } (\text{behavior.dropn } i \ \omega) \ (\lambda \omega'. \text{Some } (\text{behavior.take } (j - i) \ \omega'))$
 $\text{else None})$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

3 Point-free notation

Typically we define predicates as functions of a state. The following provide a somewhat comfortable point-free imitation of Isabelle/HOL's operators.

type-synonym $'s \text{ pred} = 's \Rightarrow \text{bool}$

abbreviation (*input*)

$\text{pred-}K :: 'b \Rightarrow 'a \Rightarrow 'b \ (\langle - \rangle)$ **where**
 $\langle f \rangle \equiv \lambda s. f$

abbreviation (*input*)

$\text{pred-not} :: 'a \text{ pred} \Rightarrow 'a \text{ pred} \ (\neg - [40] \ 40)$ **where**
 $\neg a \equiv \lambda s. \neg a \ s$

abbreviation (*input*)

$\text{pred-conj} :: 'a \text{ pred} \Rightarrow 'a \text{ pred} \Rightarrow 'a \text{ pred} \ (\text{infixr } \wedge \ 35)$ **where**
 $a \wedge b \equiv \lambda s. a \ s \wedge b \ s$

abbreviation (*input*)

$\text{pred-disj} :: 'a \text{ pred} \Rightarrow 'a \text{ pred} \Rightarrow 'a \text{ pred} \ (\text{infixr } \vee \ 30)$ **where**
 $a \vee b \equiv \lambda s. a \ s \vee b \ s$

abbreviation (*input*)

$\text{pred-implies} :: 'a \text{ pred} \Rightarrow 'a \text{ pred} \Rightarrow 'a \text{ pred} \ (\text{infixr } \longrightarrow \ 25)$ **where**
 $a \longrightarrow b \equiv \lambda s. a \ s \longrightarrow b \ s$

abbreviation (*input*)

$\text{pred-iff} :: 'a \text{ pred} \Rightarrow 'a \text{ pred} \Rightarrow 'a \text{ pred} \ (\text{infixr } \longleftrightarrow \ 25)$ **where**
 $a \longleftrightarrow b \equiv \lambda s. a \ s \longleftrightarrow b \ s$

abbreviation (*input*)

$\text{pred-eq} :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ pred} \ (\text{infix } = \ 40)$ **where**
 $a = b \equiv \lambda s. a \ s = b \ s$

abbreviation (*input*)

$\text{pred-neq} :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ pred} \ (\text{infix } \neq \ 40)$ **where**
 $a \neq b \equiv \lambda s. a \ s \neq b \ s$

abbreviation (*input*)

$\text{pred-If} :: 'a \text{ pred} \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \ ((\text{If } (-) / \text{Then } (-) / \text{Else } (-)) [0, 0, 10] \ 10)$
where $\text{If } P \ \text{Then } x \ \text{Else } y \equiv \lambda s. \text{if } P \ s \ \text{then } x \ s \ \text{else } y \ s$

abbreviation (*input*)

$\text{pred-less} :: ('a \Rightarrow 'b::\text{ord}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ pred} \ (\text{infix } < \ 40)$ **where**
 $a < b \equiv \lambda s. a \ s < b \ s$

abbreviation (*input*)

$\text{pred-less-eq} :: ('a \Rightarrow 'b::\text{ord}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ pred} \ (\text{infix } \leq \ 40)$ **where**
 $a \leq b \equiv \lambda s. a \ s \leq b \ s$

abbreviation (*input*)

$\text{pred-greater} :: ('a \Rightarrow 'b::\text{ord}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ pred} \ (\text{infix } > \ 40)$ **where**
 $a > b \equiv \lambda s. a \ s > b \ s$

abbreviation (*input*)

$\text{pred-greater-eq} :: ('a \Rightarrow 'b::\text{ord}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ pred} \ (\text{infix } \geq \ 40)$ **where**
 $a \geq b \equiv \lambda s. a \ s \geq b \ s$

abbreviation (*input*)

$\text{pred-plus} :: ('a \Rightarrow 'b::\text{plus}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixl + 65**) **where**
 $a + b \equiv \lambda s. a\ s + b\ s$

abbreviation (*input*)

$\text{pred-minus} :: ('a \Rightarrow 'b::\text{minus}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixl - 65**) **where**
 $a - b \equiv \lambda s. a\ s - b\ s$

abbreviation (*input*)

$\text{pred-times} :: ('a \Rightarrow 'b::\text{times}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixl * 65**) **where**
 $a * b \equiv \lambda s. a\ s * b\ s$

abbreviation (*input*)

$\text{pred-all} :: ('b \Rightarrow 'a\ \text{pred}) \Rightarrow 'a\ \text{pred}$ (**binder \forall 10**) **where**
 $\forall x. P\ x \equiv \lambda s. \forall x. P\ x\ s$

abbreviation (*input*)

$\text{pred-ex} :: ('b \Rightarrow 'a\ \text{pred}) \Rightarrow 'a\ \text{pred}$ (**binder \exists 10**) **where**
 $\exists x. P\ x \equiv \lambda s. \exists x. P\ x\ s$

abbreviation (*input*)

$\text{pred-app} :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (**infixl \$ 100**) **where**
 $f\ \$\ g \equiv \lambda s. f\ s\ (g\ s)$

abbreviation (*input*)

$\text{pred-app}' :: ('b \Rightarrow 'a \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (**infixl \$\$ 100**) **where**
 $f\ \$\$ g \equiv \lambda s. f\ (g\ s)\ s$

abbreviation (*input*)

$\text{pred-member} :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b\ \text{set}) \Rightarrow 'a\ \text{pred}$ (**infix \in 40**) **where**
 $a \in b \equiv \lambda s. a\ s \in b\ s$

abbreviation (*input*)

$\text{pred-subseteq} :: ('a \Rightarrow 'b\ \text{set}) \Rightarrow ('a \Rightarrow 'b\ \text{set}) \Rightarrow 'a\ \text{pred}$ (**infix \subseteq 50**) **where**
 $A \subseteq B \equiv \lambda s. A\ s \subseteq B\ s$

abbreviation (*input*)

$\text{pred-union} :: ('a \Rightarrow 'b\ \text{set}) \Rightarrow ('a \Rightarrow 'b\ \text{set}) \Rightarrow 'a \Rightarrow 'b\ \text{set}$ (**infixl \cup 65**) **where**
 $a \cup b \equiv \lambda s. a\ s \cup b\ s$

abbreviation (*input*)

$\text{pred-inter} :: ('a \Rightarrow 'b\ \text{set}) \Rightarrow ('a \Rightarrow 'b\ \text{set}) \Rightarrow 'a \Rightarrow 'b\ \text{set}$ (**infixl \cap 65**) **where**
 $a \cap b \equiv \lambda s. a\ s \cap b\ s$

abbreviation (*input*)

$\text{pred-conjoin} :: 'a\ \text{pred}\ \text{list} \Rightarrow 'a\ \text{pred}$ **where**
 $\text{pred-conjoin}\ xs \equiv \text{foldr } (\wedge) xs \langle \text{True} \rangle$

abbreviation (*input*)

$\text{pred-disjoin} :: 'a\ \text{pred}\ \text{list} \Rightarrow 'a\ \text{pred}$ **where**
 $\text{pred-disjoin}\ xs \equiv \text{foldr } (\vee) xs \langle \text{False} \rangle$

abbreviation (*input*)

$\text{pred-min} :: ('a \Rightarrow 'b::\text{ord}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ **where**
 $\text{pred-min}\ x\ y \equiv \lambda s. \text{min } (x\ s)\ (y\ s)$

abbreviation (*input*)

$\text{pred-max} :: ('a \Rightarrow 'b::\text{ord}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ **where**
 $\text{pred-max } x \ y \equiv \lambda s. \text{max } (x \ s) \ (y \ s)$

abbreviation (*input*)

$\text{NULL} :: ('a \Rightarrow 'b \text{ option}) \Rightarrow 'a \text{ pred}$ **where**
 $\text{NULL } a \equiv \lambda s. a \ s = \text{None}$

abbreviation (*input*)

$\text{EMPTY} :: ('a \Rightarrow 'b \text{ set}) \Rightarrow 'a \text{ pred}$ **where**
 $\text{EMPTY } a \equiv \lambda s. a \ s = \{\}$

abbreviation (*input*)

$\text{LIST-NULL} :: ('a \Rightarrow 'b \text{ list}) \Rightarrow 'a \text{ pred}$ **where**
 $\text{LIST-NULL } a \equiv \lambda s. a \ s = []$

abbreviation (*input*)

$\text{SIZE} :: ('a \Rightarrow 'b::\text{size}) \Rightarrow 'a \Rightarrow \text{nat}$ **where**
 $\text{SIZE } a \equiv \lambda s. \text{size } (a \ s)$

abbreviation (*input*)

$\text{SET} :: ('a \Rightarrow 'b \text{ list}) \Rightarrow 'a \Rightarrow 'b \text{ set}$ **where**
 $\text{SET } a \equiv \lambda s. \text{set } (a \ s)$

abbreviation (*input*)

$\text{pred-singleton} :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \text{ set}$ **where**
 $\text{pred-singleton } x \equiv \lambda s. \{x \ s\}$

abbreviation (*input*)

$\text{pred-list-nth} :: ('a \Rightarrow 'b \text{ list}) \Rightarrow ('a \Rightarrow \text{nat}) \Rightarrow 'a \Rightarrow 'b$ (**infixl ! 150**) **where**
 $xs \ ! \ i \equiv \lambda s. xs \ s \ ! \ i \ s$

abbreviation (*input*)

$\text{pred-list-append} :: ('a \Rightarrow 'b \text{ list}) \Rightarrow ('a \Rightarrow 'b \text{ list}) \Rightarrow 'a \Rightarrow 'b \text{ list}$ (**infixr @ 65**) **where**
 $xs \ @ \ ys \equiv \lambda s. xs \ s \ @ \ ys \ s$

abbreviation (*input*)

$\text{FST} :: 'a \text{ pred} \Rightarrow ('a \times 'b) \text{ pred}$ **where**
 $\text{FST } P \equiv \lambda s. P \ (\text{fst } s)$

abbreviation (*input*)

$\text{SND} :: 'b \text{ pred} \Rightarrow ('a \times 'b) \text{ pred}$ **where**
 $\text{SND } P \equiv \lambda s. P \ (\text{snd } s)$

abbreviation (*input*)

$\text{pred-pair} :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \times 'c$ (**infixr \otimes 60**) **where**
 $a \ \otimes \ b \equiv \lambda s. (a \ s, b \ s)$

4 More lattice

lemma (**in** *semilattice-sup*) *sup-iff-le*:

shows $x \sqcup y = y \iff x \leq y$

and $y \sqcup x = y \iff x \leq y$

<proof>

lemma (**in** *semilattice-inf*) *inf-iff-le*:

shows $x \sqcap y = x \longleftrightarrow x \leq y$
and $y \sqcap x = x \longleftrightarrow x \leq y$
 $\langle \text{proof} \rangle$

lemma *if-sup-distr*:

fixes $t e :: \text{semilattice-sup}$
shows *if-sup-distrL*: $(\text{if } b \text{ then } t_1 \sqcup t_2 \text{ else } e) = (\text{if } b \text{ then } t_1 \text{ else } e) \sqcup (\text{if } b \text{ then } t_2 \text{ else } e)$
and *if-sup-distrR*: $(\text{if } b \text{ then } t \text{ else } e_1 \sqcup e_2) = (\text{if } b \text{ then } t \text{ else } e_1) \sqcup (\text{if } b \text{ then } t \text{ else } e_2)$
 $\langle \text{proof} \rangle$

lemma *INF-bot*:

assumes $F i = (\perp :: \text{complete-lattice})$
assumes $i \in X$
shows $(\prod_{i \in X} F i) = \perp$
 $\langle \text{proof} \rangle$

lemma *mcont-fun-app-const[cont-intro]*:

shows $mcont \text{ Sup } (\leq) \text{ Sup } (\leq) (\lambda f. f c)$
 $\langle \text{proof} \rangle$

declare *mcont-applyI[cont-intro]*

lemma *INF-rename-bij*:

assumes *bij-betw* $\pi X Y$
shows $(\prod_{y \in Y} F Y y) = (\prod_{x \in X} F (\pi ' X) (\pi x))$
 $\langle \text{proof} \rangle$

lemma *Inf-rename-surj*:

assumes *surj* π
shows $(\prod x. F x) = (\prod x. F (\pi x))$
 $\langle \text{proof} \rangle$

lemma *INF-unwind-index*:

fixes $A :: \text{complete-lattice}$
assumes $i \in I$
shows $(\prod_{x \in I} A x) = A i \sqcap (\prod_{x \in I - \{i\}} A x)$
 $\langle \text{proof} \rangle$

lemma *Sup-fst*:

shows $(\bigsqcup_{x \in X} P (fst x)) = (\bigsqcup_{x \in fst ' X} P x)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *assms-cong*: — simplify assumptions only

assumes $x = x'$
shows $x \leq y \longleftrightarrow x' \leq y$
 $\langle \text{proof} \rangle$

lemma *concl-cong*: — simplify conclusions only

assumes $y = y'$
shows $x \leq y \longleftrightarrow x \leq y'$
 $\langle \text{proof} \rangle$

lemma *subgoal*: — cut for lattice logics

fixes $P :: \text{semilattice-inf}$
assumes $P \leq Q$
assumes $P \sqcap Q \leq R$

shows $P \leq R$

<proof>

<ML>

Logical rules ala HOL lemmas $SupI = Sup-upper$

lemmas $rev-SUPI = SUP-upper2[of\ x\ A\ b\ f\ for\ x\ A\ b\ f]$

lemmas $SUPI = rev-SUPI[rotated]$

lemmas $SUPE = SUP-least[where\ u=z\ for\ z]$

lemmas $SupE = Sup-least$

lemmas $INFI = INF-greatest$

lemmas $InfI = Inf-greatest$

lemmas $infI = semilattice-inf-class.le-infI$

lemma $InfE$:

fixes $R:::complete-lattice$

assumes $P\ x \leq R$

shows $(\bigwedge x. P\ x) \leq R$

<proof>

lemma $INFE$:

fixes $R::'a::complete-lattice$

assumes $P\ x \leq R$

assumes $x \in A$

shows $\bigwedge (P\ 'A) \leq R$

<proof>

lemmas $rev-INFE = INFE[rotated]$

lemma $Inf-inf-distrib$:

fixes $P:::complete-lattice$

shows $(\bigwedge x. P\ x \sqcap Q\ x) = (\bigwedge x. P\ x) \sqcap (\bigwedge x. Q\ x)$

<proof>

lemma $Sup-sup-distrib$:

fixes $P:::complete-lattice$

shows $(\bigsqcup x. P\ x \sqcup Q\ x) = (\bigsqcup x. P\ x) \sqcup (\bigsqcup x. Q\ x)$

<proof>

lemma $Inf-inf$:

fixes $Q :: -:complete-lattice$

shows $(\bigwedge x. P\ x \sqcap Q) = (\bigwedge x. P\ x) \sqcap Q$

<proof>

lemma $inf-Inf$:

fixes $P :: -:complete-lattice$

shows $(\bigwedge x. P \sqcap Q\ x) = P \sqcap (\bigwedge x. Q\ x)$

<proof>

lemma $SUP-sup$:

fixes $Q :: -:complete-lattice$

assumes $X \neq \{\}$

shows $(\bigsqcup x \in X. P\ x \sqcup Q) = (\bigsqcup x \in X. P\ x) \sqcup Q$ (**is** $?lhs = ?rhs$)

<proof>

lemma $sup-SUP$:

fixes $P :: \text{::complete-lattice}$
assumes $X \neq \{\}$
shows $(\bigsqcup x \in X. P \sqcup Q x) = P \sqcup (\bigsqcup x \in X. Q x)$
 $\langle \text{proof} \rangle$

4.1 Boolean lattices and implication

lemma

shows $\text{minus-Not}[simp]: - \text{Not} = id$
and $\text{minus-id}[simp]: - id = \text{Not}$
 $\langle \text{proof} \rangle$

definition $\text{boolean-implication} :: 'a::\text{boolean-algebra} \Rightarrow 'a \Rightarrow 'a$ (**infixr** \longrightarrow_B 60) **where**
 $x \longrightarrow_B y = -x \sqcup y$

definition $\text{boolean-eq} :: 'a::\text{boolean-algebra} \Rightarrow 'a \Rightarrow 'a$ (**infixr** \longleftrightarrow_B 60) **where**
 $x \longleftrightarrow_B y = x \longrightarrow_B y \sqcap y \longrightarrow_B x$

$\langle ML \rangle$

lemma $\text{bool-alt-def}[simp]:$

shows $P \longrightarrow_B Q = (P \longrightarrow Q)$
 $\langle \text{proof} \rangle$

lemma $\text{pred--alt-def}[simp]:$

shows $(P \longrightarrow_B Q) x = (P x \longrightarrow_B Q x)$
 $\langle \text{proof} \rangle$

lemma $\text{set-alt-def}:$

shows $P \longrightarrow_B Q = \{x. x \in P \longrightarrow x \in Q\}$
 $\langle \text{proof} \rangle$

lemma $\text{member}:$

shows $x \in P \longrightarrow_B Q \longleftrightarrow x \in P \longrightarrow x \in Q$
 $\langle \text{proof} \rangle$

lemmas $\text{setI} = \text{iffD2}[OF \text{boolean-implication.member, rule-format}]$

lemma $\text{simps}[simp]:$

shows
 $\top \longrightarrow_B P = P$
 $\perp \longrightarrow_B P = \top$
 $P \longrightarrow_B \top = \top$
 $P \longrightarrow_B P = \top$
 $P \longrightarrow_B \perp = -P$
 $P \longrightarrow_B -P = -P$
 $\langle \text{proof} \rangle$

lemma $\text{Inf-simps}[simp]:$ — Miniscoping: pushing in universal quantifiers.

shows
 $\bigwedge P (Q::\text{::complete-boolean-algebra}). (\bigcap x. P x \longrightarrow_B Q) = ((\bigsqcup x. P x) \longrightarrow_B Q)$
 $\bigwedge P (Q::\text{::complete-boolean-algebra}). (\bigcap x \in X. P x \longrightarrow_B Q) = ((\bigsqcup x \in X. P x) \longrightarrow_B Q)$
 $\bigwedge P (Q::\text{::}\Rightarrow\text{::complete-boolean-algebra}). (\bigcap x. P \longrightarrow_B Q x) = (P \longrightarrow_B (\bigcap x. Q x))$
 $\bigwedge P (Q::\text{::}\Rightarrow\text{::complete-boolean-algebra}). (\bigcap x \in X. P \longrightarrow_B Q x) = (P \longrightarrow_B (\bigcap x \in X. Q x))$
 $\langle \text{proof} \rangle$

lemma $\text{mono}:$

assumes $x' \leq x$

assumes $y \leq y'$
shows $x \longrightarrow_B y \leq x' \longrightarrow_B y'$
 $\langle \text{proof} \rangle$

lemma *strengthen*[*strg*]:
assumes *st-ord* $(\neg F) X X'$
assumes *st-ord* $F Y Y'$
shows *st-ord* $F (X \longrightarrow_B Y) (X' \longrightarrow_B Y')$
 $\langle \text{proof} \rangle$

lemma *eq-conv*:
shows $(P = Q) \longleftrightarrow (P \longrightarrow_B Q) \sqcap (Q \longrightarrow_B P) = \top$
 $\langle \text{proof} \rangle$

lemma *uminus-imp*[*simp*]:
shows $\neg(P \longrightarrow_B Q) = P \sqcap \neg Q$
 $\langle \text{proof} \rangle$

lemma *cases-simp*[*simp*]:
shows $(P \longrightarrow_B Q) \sqcap (\neg P \longrightarrow_B Q) = Q$
 $\langle \text{proof} \rangle$

lemma *conv-sup*:
shows $(P \longrightarrow_B Q) = \neg P \sqcup Q$
 $\langle \text{proof} \rangle$

lemma *infL*:
shows $P \sqcap Q \longrightarrow_B R = P \longrightarrow_B Q \longrightarrow_B R$
 $\langle \text{proof} \rangle$

lemmas *uncurry* = *boolean-implication.infL*[*symmetric*]

lemma *shunt1*:
shows $x \sqcap y \leq z \longleftrightarrow x \leq y \longrightarrow_B z$
 $\langle \text{proof} \rangle$

lemma *shunt2*:
shows $x \sqcap y \leq z \longleftrightarrow y \leq x \longrightarrow_B z$
 $\langle \text{proof} \rangle$

lemma *mp*:
assumes $x \sqcap y \leq z$
shows $x \leq y \longrightarrow_B z$
 $\langle \text{proof} \rangle$

lemma *imp-trivialI*:
assumes $P \sqcap \neg R \leq \neg Q$
shows $P \leq Q \longrightarrow_B R$
 $\langle \text{proof} \rangle$

lemma *shunt-top*:
shows $P \longrightarrow_B Q = \top \longleftrightarrow P \leq Q$
 $\langle \text{proof} \rangle$

lemma *detachment*:
shows $x \sqcap (x \longrightarrow_B y) = x \sqcap y$ (**is** *?thesis1*)
and $(x \longrightarrow_B y) \sqcap x = x \sqcap y$ (**is** *?thesis2*)
 $\langle \text{proof} \rangle$

lemma *discharge*:

assumes $x' \leq x$

shows $x' \sqcap (x \longrightarrow_B y) = x' \sqcap y$ (**is** *?thesis1*)

and $(x \longrightarrow_B y) \sqcap x' = y \sqcap x'$ (**is** *?thesis2*)

<proof>

lemma *trans*:

shows $(x \longrightarrow_B y) \sqcap (y \longrightarrow_B z) \leq (x \longrightarrow_B z)$

<proof>

<ML>

4.2 Compactness and algebraicity

Fundamental lattice concepts drawn from Davey and Priestley (2002).

context *complete-lattice*

begin

definition *compact-points* :: 'a set **where** — Davey and Priestley (2002, Definition 7.15(ii))

$compact_points = \{x. \forall S. x \leq \bigsqcup S \longrightarrow (\exists T \subseteq S. finite\ T \wedge x \leq \bigsqcup T)\}$

lemmas *compact-pointsI* = *subsetD[OF equalityD2[OF compact-points-def], simplified, rule-format]*

lemmas *compact-pointsD* = *subsetD[OF equalityD1[OF compact-points-def], simplified, rule-format]*

lemma *compact-point-bot*:

shows $\perp \in compact_points$

<proof>

lemma *compact-points-sup*: — Davey and Priestley (2002, Lemma 7.16)

assumes $x \in compact_points$

assumes $y \in compact_points$

shows $x \sqcup y \in compact_points$

<proof>

lemma *compact-points-Sup*: — Davey and Priestley (2002, Lemma 7.16)

assumes $X \subseteq compact_points$

assumes *finite* X

shows $\bigsqcup X \in compact_points$

<proof>

lemma *compact-points-are-ccpo-compact*: — converse should hold

assumes $x \in compact_points$

shows *ccpo.compact* *Sup* (\leq) x

<proof>

definition *directed* :: 'a set \Rightarrow bool **where** — Davey and Priestley (2002, Definition 7.7)

$directed\ X \longleftrightarrow X \neq \{\} \wedge (\forall x \in X. \forall y \in X. \exists z \in X. x \leq z \wedge y \leq z)$

lemmas *directedI* = *iffD2[OF directed-def, simplified conj-explode, rule-format]*

lemmas *directedD* = *iffD1[OF directed-def]*

lemma *directed-empty*:

assumes *directed* X

shows $X \neq \{\}$

<proof>

lemma *chain-directed*:

assumes *Complete-Partial-Order.chain* (\leq) Y
assumes $Y \neq \{\}$
shows *directed* Y
 \langle *proof* \rangle

lemma *directed-alt-def*:

shows *directed* $X \iff (\forall Y \subseteq X. \text{finite } Y \implies (\exists x \in X. \forall y \in Y. y \leq x))$ (**is** $?lhs \iff ?rhs$)
 \langle *proof* \rangle

lemma *compact-points-alt-def*: — [Davey and Priestley \(2002, Definition 7.15\(i\)\)](#) (finite points)

shows *compact-points* = $\{x :: 'a. \forall D. \text{directed } D \wedge x \leq \bigsqcup D \implies (\exists d \in D. x \leq d)\}$ (**is** $?lhs = ?rhs$)
 \langle *proof* \rangle

lemmas *compact-points-directedD*

= *subsetD[OF equalityD1[OF compact-points-alt-def], simplified, rule-format, simplified conj-explode, rotated -1]*

end

class *algebraic-lattice* = *complete-lattice* + — [Davey and Priestley \(2002, Definition 7.18\)](#)

assumes *algebraic*: $(x :: 'a) = \bigsqcup (\{Y. Y \leq x\} \cap \text{compact-points})$

begin

lemma *le-compact*:

shows $x \leq y \iff (\forall z \in \text{compact-points}. z \leq x \implies z \leq y)$
 \langle *proof* \rangle

end

lemma (**in** *ccpo*) *compact-alt-def*:

shows *ccpo.compact* $\text{Sup } (\leq) x \iff (\forall Y. Y \neq \{\} \wedge \text{Complete-Partial-Order.chain } (\leq) Y \wedge x \leq \text{Sup } Y \implies (\exists y \in Y. x \leq y))$
 \langle *proof* \rangle

lemma *compact-points-eq-finite-sets*: — [Davey and Priestley \(2002, Examples 7.17\)](#)

shows *compact-points* = *Collect finite* (**is** $?lhs = ?rhs$)
 \langle *proof* \rangle

instance *set* :: (*type*) *algebraic-lattice*

\langle *proof* \rangle

context *semilattice-sup*

begin

definition *sup-irreducible-on* :: $'a \text{ set} \Rightarrow 'a \Rightarrow \text{bool}$ **where** — [Davey and Priestley \(2002, Definition 2.42\)](#)

sup-irreducible-on $A x \iff (\forall y \in A. \forall z \in A. x = y \sqcup z \implies x = y \vee x = z)$

abbreviation *sup-irreducible* :: $'a \Rightarrow \text{bool}$ **where**

sup-irreducible $\equiv \text{sup-irreducible-on UNIV}$

lemma *sup-irreducible-onI*:

assumes $\bigwedge y z. \llbracket y \in A; z \in A; x = y \sqcup z \rrbracket \implies x = y \vee x = z$
shows *sup-irreducible-on* $A x$
 \langle *proof* \rangle

lemma *sup-irreducible-onD*:

assumes *sup-irreducible-on* $A x$
assumes $x = y \sqcup z$

assumes $y \in A$
assumes $z \in A$
shows $x = y \vee x = z$
 ⟨proof⟩

lemma *sup-irreducible-on-less*: — Davey and Priestley (2002, Definition 2.42 (alt))
shows *sup-irreducible-on* A $x \longleftrightarrow (\forall y \in A. \forall z \in A. y < x \wedge z < x \longrightarrow y \sqcup z < x)$
 ⟨proof⟩

end

lemma *sup-irreducible-bot*:
assumes $\perp \in A$
shows *sup-irreducible-on* A $(\perp :: \text{bounded-semilattice-sup-bot})$
 ⟨proof⟩

lemma *sup-irreducible-le-conv*:
fixes $x :: \text{distrib-lattice}$
assumes *sup-irreducible* x
shows $x \leq y \sqcup z \longleftrightarrow x \leq y \vee x \leq z$
 ⟨proof⟩

lemma *set-sup-irreducible*:
shows *sup-irreducible* $X \longleftrightarrow (X = \{\} \vee (\exists y. X = \{y\}))$ (**is** *?lhs* \longleftrightarrow *?rhs*)
 ⟨proof⟩

definition *Sup-irreducible-on* :: $'a :: \text{complete-lattice set} \Rightarrow 'a \Rightarrow \text{bool}$ **where** — Davey and Priestley (2002, Definition 10.26)
 $\text{Sup-irreducible-on } A \ x \longleftrightarrow (\forall S \subseteq A. x = \bigsqcup S \longrightarrow x \in S)$

abbreviation *Sup-irreducible* :: $'a :: \text{complete-lattice} \Rightarrow \text{bool}$ **where**
 $\text{Sup-irreducible} \equiv \text{Sup-irreducible-on UNIV}$

definition *Sup-prime-on* :: $'a :: \text{complete-lattice set} \Rightarrow 'a \Rightarrow \text{bool}$ **where** — Davey and Priestley (2002, Definition 10.26)
 $\text{Sup-prime-on } A \ x \longleftrightarrow (\forall S \subseteq A. x \leq \bigsqcup S \longrightarrow (\exists s \in S. x \leq s))$

abbreviation *Sup-prime* :: $'a :: \text{complete-lattice} \Rightarrow \text{bool}$ **where**
 $\text{Sup-prime} \equiv \text{Sup-prime-on UNIV}$

lemma *Sup-irreducible-onI*:
assumes $\bigwedge S. \llbracket S \subseteq A; x = \bigsqcup S \rrbracket \Longrightarrow x \in S$
shows *Sup-irreducible-on* A x
 ⟨proof⟩

lemma *Sup-irreducible-onD*:
assumes $x = \bigsqcup S$
assumes $S \subseteq A$
assumes *Sup-irreducible-on* A x
shows $x \in S$
 ⟨proof⟩

lemma *Sup-prime-onI*:
assumes $\bigwedge S. \llbracket S \subseteq A; x \leq \bigsqcup S \rrbracket \Longrightarrow \exists s \in S. x \leq s$
shows *Sup-prime-on* A x
 ⟨proof⟩

lemma *Sup-prime-onE*:

assumes *Sup-prime-on* A x
assumes $x \leq \bigsqcup S$
assumes $S \subseteq A$
obtains s **where** $s \in S$ **and** $x \leq s$
 \langle *proof* \rangle

lemma *Sup-prime-on-conv*:
assumes *Sup-prime-on* A x
assumes $S \subseteq A$
shows $x \leq \bigsqcup S \longleftrightarrow (\exists s \in S. x \leq s)$
 \langle *proof* \rangle

lemma *Sup-prime-not-bot*:
assumes *Sup-prime-on* A x
shows $x \neq \perp$
 \langle *proof* \rangle

lemma *Sup-prime-on-imp-Sup-irreducible-on*: — the converse holds in Heyting algebras
assumes *Sup-prime-on* A x
shows *Sup-irreducible-on* A x
 \langle *proof* \rangle

lemma *Sup-irreducible-on-imp-sup-irreducible-on*:
assumes *Sup-irreducible-on* A x
assumes $x \in A$
shows *sup-irreducible-on* A x
 \langle *proof* \rangle

lemma *Sup-prime-is-compact*:
assumes *Sup-prime* x
shows $x \in$ *compact-points*
 \langle *proof* \rangle

5 Closure operators

Our semantic spaces are modelled as lattices arising from the fixed points of various closure operators. We attempt to reduce our proof obligations by defining a locale for Kuratowski's closure axioms, where we do not require strictness (i.e., it need not be the case that the closure maps \perp to \perp). Davey and Priestley (2002, §2.33) term these *topped intersection structures*; see also Pfaltz and Šlapal (2013) for additional useful results.

locale *closure* =
ordering (\leq) ($<$) — We use a partial order as a preorder does not ensure that the closure is idempotent
for *less-eq* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** \leq 50)
and *less* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $<$ 50)
+ **fixes** *cl* :: $'a \Rightarrow 'a$
assumes *cl*: $x \leq cl\ y \longleftrightarrow cl\ x \leq cl\ y$ — All-in-one non-strict Kuratowski axiom
begin

definition *closed* :: $'a$ **set where** — These pre fixed points form a complete lattice ala Tarski/Knaster
closed = $\{x. cl\ x \leq x\}$

lemma *closed-clI*:
assumes *cl* $x \leq x$
shows $x \in$ *closed*
 \langle *proof* \rangle

lemma *expansive*:
shows $x \leq cl\ x$

<proof>

lemma *idempotent[simp]*:

shows $cl (cl x) = cl x$

and $cl \circ cl = cl$

<proof>

lemma *monotone-cl*:

shows $monotone (\leq) (\leq) cl$

<proof>

lemmas *strengthen-cl[strg] = st-monotone[OF monotone-cl]*

lemmas *mono-cl[trans] = monotoneD[OF monotone-cl]*

lemma *least*:

assumes $x \leq y$

assumes $y \in closed$

shows $cl x \leq y$

<proof>

lemma *least-conv*:

assumes $y \in closed$

shows $cl x \leq y \longleftrightarrow x \leq y$

<proof>

lemma *closed[iff]*:

shows $cl x \in closed$

<proof>

lemma *le-closedE*:

assumes $x \leq cl y$

assumes $y \in closed$

shows $x \leq y$

<proof>

lemma *closed-conv*: — Typically used to manifest the closure using *subst*

assumes $X \in closed$

shows $X = cl X$

<proof>

end

lemma (*in ordering*) *closure-axioms-alt-def*: — Equivalence with the Kuratowski closure axioms

shows $closure-axioms (\leq) cl \longleftrightarrow (\forall x. x \leq cl x) \wedge monotone (\leq) (\leq) cl \wedge (\forall x. cl (cl x) = cl x)$

<proof>

lemma (*in ordering*) *closureI*:

assumes $\bigwedge x. x \leq cl x$

assumes $monotone (\leq) (\leq) cl$

assumes $\bigwedge x. cl (cl x) = cl x$

shows $closure (\leq) (<) cl$

<proof>

lemma *closure-inf-closure*:

fixes $cl_1 :: 'a::semilattice-inf \Rightarrow 'a$

assumes $closure-axioms (\leq) cl_1$

assumes $closure-axioms (\leq) cl_2$

shows $closure-axioms (\leq) (\lambda X. cl_1 X \sqcap cl_2 X)$

<proof>

5.1 Complete lattices and algebraic closures

locale *closure-complete-lattice* =
 complete-lattice \sqcap \sqcup (\sqcap) (\leq) ($<$) (\sqcup) \perp \top
+ *closure* (\leq) ($<$) *cl*
 for *less-ega* :: '*a* \Rightarrow '*a* \Rightarrow *bool* (**infix** \leq 50)
 and *lessa* (**infix** $<$ 50)
 and *infa* (**infixl** \sqcap 70)
 and *supa* (**infixl** \sqcup 65)
 and *bota* (\perp)
 and *topa* (\top)
 and *Inf* (\sqcap)
 and *Sup* (\sqcup)
 and *cl* :: '*a* \Rightarrow '*a*

begin

lemma *cl-bot-least*:
 shows *cl* $\perp \leq$ *cl* *X*
<proof>

lemma *cl-Inf-closed*:
 shows *cl* *x* = $\sqcap \{y \in \text{closed}. x \leq y\}$
<proof>

lemma *cl-top*:
 shows *cl* $\top = \top$
<proof>

lemma *closed-top[iff]*:
 shows $\top \in \text{closed}$
<proof>

lemma *Sup-cl-le*:
 shows $\sqcup (\text{cl } 'X) \leq$ *cl* ($\sqcup X$)
<proof>

lemma *sup-cl-le*:
 shows *cl* *x* \sqcup *cl* *y* \leq *cl* (*x* \sqcup *y*)
<proof>

lemma *cl-Inf-le*:
 shows *cl* ($\sqcap X$) \leq $\sqcap (\text{cl } 'X)$
<proof>

lemma *cl-inf-le*:
 shows *cl* (*x* \sqcap *y*) \leq *cl* *x* \sqcap *cl* *y*
<proof>

lemma *closed-Inf*:
 assumes $X \subseteq \text{closed}$
 shows $\sqcap X \in \text{closed}$
<proof>

lemmas *closed-Inf'[intro]* = *closed-Inf[OF subsetI]*

lemma *closed-inf[intro]*:

assumes $P \in \text{closed}$
assumes $Q \in \text{closed}$
shows $P \sqcap Q \in \text{closed}$
 $\langle \text{proof} \rangle$

lemmas $\text{mono2mono}[\text{cont-intro}, \text{partial-function-mono}] = \text{monotone2monotone}[\text{OF monotone-cl}, \text{simplified}]$

definition $\text{dense} :: 'a \text{ set where}$
 $\text{dense} = \{x. \text{cl } x = \top\}$

lemma $\text{dense-top}:$
shows $\top \in \text{dense}$
 $\langle \text{proof} \rangle$

lemma $\text{dense-Sup}:$
assumes $X \subseteq \text{dense}$
assumes $X \neq \{\}$
shows $\bigsqcup X \in \text{dense}$
 $\langle \text{proof} \rangle$

lemma $\text{dense-sup}:$
assumes $P \in \text{dense}$
assumes $Q \in \text{dense}$
shows $P \sqcup Q \in \text{dense}$
 $\langle \text{proof} \rangle$

lemma $\text{dense-le}:$
assumes $P \in \text{dense}$
assumes $P \leq Q$
shows $Q \in \text{dense}$
 $\langle \text{proof} \rangle$

lemma $\text{dense-inf-closed}:$
shows $\text{dense} \cap \text{closed} = \{\top\}$
 $\langle \text{proof} \rangle$

end

locale $\text{closure-complete-lattice-class} =$
 $\text{closure-complete-lattice } (\leq) (<) (\sqcap) (\sqcup) \perp :: - :: \text{complete-lattice } \top \text{ Inf Sup}$

Traditionally closures for logical purposes are taken to be “algebraic”, aka “consequence operators” (Davey and Priestley 2002, Definition 7.12), where *compactness* does the work of the finite/singleton sets.

locale $\text{closure-complete-lattice-algebraic} = \text{— Davey and Priestley (2002, Definition 7.12)}$
 $\text{closure-complete-lattice}$

+ assumes $\text{algebraic-le}: \text{cl } x \leq \bigsqcup (\text{cl } ' (\{y. y \leq x\} \cap \text{compact-points}))$ — The converse is given by monotonicity
begin

lemma $\text{algebraic}:$
shows $\text{cl } x = \bigsqcup (\text{cl } ' (\{y. y \leq x\} \cap \text{compact-points}))$
 $\langle \text{proof} \rangle$

lemma $\text{cont-cl}:$ — Equivalent to *algebraic-le* Davey and Priestley (2002, Theorem 7.14)
shows $\text{cont } \bigsqcup (\leq) \bigsqcup (\leq) \text{cl}$
 $\langle \text{proof} \rangle$

lemma $\text{mcont-cl}:$
shows $\text{mcont } \bigsqcup (\leq) \bigsqcup (\leq) \text{cl}$

<proof>

lemma *mcont2mcont-cl[cont-intro]*:

assumes *mcont luba orda* $\sqcup (\leq) P$

shows *mcont luba orda* $\sqcup (\leq) (\lambda x. cl (P x))$

<proof>

end

locale *closure-complete-lattice-algebraic-class* =

closure-complete-lattice-algebraic $(\leq) (<) (\sqcap) (\sqcup) \perp :: - :: complete-lattice \top Inf Sup$

Our closures often satisfy the stronger condition of *distributivity* (see Scott (1980, §2)).

locale *closure-complete-lattice-distributive* =

closure-complete-lattice

+ **assumes** *cl-Sup-le*: $cl (\sqcup X) \leq \sqcup (cl ' X) \sqcup cl \perp$

begin

lemma *cl-Sup*:

shows $cl (\sqcup X) = \sqcup (cl ' X) \sqcup cl \perp$

<proof>

lemma *cl-Sup-not-empty*:

assumes $X \neq \{\}$

shows $cl (\sqcup X) = \sqcup (cl ' X)$

<proof>

lemma *cl-sup*:

shows $cl (X \sqcup Y) = cl X \sqcup cl Y$

<proof>

lemma *closed-sup[intro]*:

assumes $P \in closed$

assumes $Q \in closed$

shows $P \sqcup Q \in closed$

<proof>

lemma *closed-Sup*: — Alexandrov: https://en.wikipedia.org/wiki/Alexandrov_topology

assumes $X \subseteq closed$

shows $\sqcup X \sqcup cl \perp \in closed$

<proof>

lemmas *closed-Sup'[intro]* = *closed-Sup[OF subsetI]*

lemma *cont-cl*:

shows $cont \sqcup (\leq) \sqcup (\leq) cl$

<proof>

lemma *mcont-cl*:

shows $mcont \sqcup (\leq) \sqcup (\leq) cl$

<proof>

lemma *mcont2mcont-cl[cont-intro]*:

assumes *mcont luba orda* $\sqcup (\leq) F$

shows *mcont luba orda* $\sqcup (\leq) (\lambda x. cl (F x))$

<proof>

lemma *closure-sup-irreducible-on*: — converse requires the closure to be T0

assumes *sup-irreducible-on closed* (*cl x*)
shows *sup-irreducible-on closed* *x*
⟨*proof*⟩

end

locale *closure-complete-lattice-distributive-class* =
closure-complete-lattice-distributive (\leq) ($<$) (\sqcap) (\sqcup) \perp :: - :: *complete-lattice* \top *Inf Sup*

locale *closure-complete-distrib-lattice-distributive-class* =
closure-complete-lattice-distributive (\leq) ($<$) (\sqcap) (\sqcup) \perp :: - :: *complete-distrib-lattice* \top *Inf Sup*
begin

The lattice arising from the closed elements for a distributive closure is completely distributive, i.e., *Inf* and *Sup* distribute. See Davey and Priestley (2002, Section 10.23).

lemma *closed-complete-distrib-lattice-axiomI'*:
assumes $\forall A \in A. \forall x \in A. x \in \text{closed}$
shows $(\sqcap X \in A. \sqcup X \sqcup \text{cl } \perp)$
 $\leq \sqcup (\text{Inf } \{f \text{ ' } A \mid f. (\forall X \subseteq \text{closed}. f X \in \text{closed}) \wedge (\forall Y \in A. f Y \in Y)\}) \sqcup \text{cl } \perp$
⟨*proof*⟩

lemma *closed-complete-distrib-lattice-axiomI[intro]*:
assumes $\forall A \in A. \forall x \in A. x \in \text{closed}$
shows $(\sqcap X \in A. \sqcup X \sqcup \text{cl } \perp)$
 $\leq \sqcup (\text{Inf } \{B. (\exists f. (\forall x. (\forall x \in x. x \in \text{closed}) \longrightarrow f x \in \text{closed}))$
 $\wedge B = f \text{ ' } A \wedge (\forall Y \in A. f Y \in Y)) \wedge (\forall x \in B. x \in \text{closed})\})$
 $\sqcup \text{cl } \perp$
⟨*proof*⟩

lemma *closed-strict-complete-distrib-lattice-axiomI[intro]*:
assumes $\text{cl } \perp = \perp$
assumes $\forall A \in A. \forall x \in A. x \in \text{closed}$
shows $(\sqcap X \in A. \sqcup X)$
 $\leq \sqcup (\text{Inf } \{x. (\exists f. (\forall x. (\forall x \in x. x \in \text{closed}) \longrightarrow f x \in \text{closed}))$
 $\wedge x = f \text{ ' } A \wedge (\forall Y \in A. f Y \in Y)) \wedge (\forall x \in x. x \in \text{closed})\})$
⟨*proof*⟩

end

5.2 Closures over powersets

locale *closure-powerset* =
closure-complete-lattice-class *cl* **for** *cl* :: 'a set \Rightarrow 'a set
begin

lemmas *expansive'* = *subsetD[OF expansive]*

lemma *closedI[intro]*:
assumes $\bigwedge x. x \in \text{cl } X \implies x \in X$
shows $X \in \text{closed}$
⟨*proof*⟩

lemma *closedE*:
assumes $x \in \text{cl } Y$
assumes $Y \in \text{closed}$
shows $x \in Y$
⟨*proof*⟩

lemma *cl-mono*:

assumes $x \in cl\ X$

assumes $X \subseteq Y$

shows $x \in cl\ Y$

<proof>

lemma *cl-bind-le*:

shows $X \gg= cl \circ f \leq cl (X \gg= f)$

<proof>

lemma *pointwise-distributive-iff*:

shows $(\forall X. cl (\bigcup X) = \bigcup (cl \text{ ` } X) \cup cl \{\}) \longleftrightarrow (\forall X. cl X = (\bigcup_{x \in X}. cl \{x\}) \cup cl \{\})$ (**is** *?lhs* \longleftrightarrow *?rhs*)

<proof>

lemma *Sup-prime-on-singleton*:

shows *Sup-prime-on closed* $(cl \{x\})$

<proof>

end

locale *closure-powerset-algebraic* =

closure-powerset

+ *closure-complete-lattice-algebraic-class*

locale *closure-powerset-distributive* =

closure-powerset

+ *closure-complete-distrib-lattice-distributive-class*

begin

lemmas *distributive* = *pointwise-distributive-iff*[*THEN iffD1, rule-format, OF cl-Sup*]

lemma *algebraic-axiom*: — Davey and Priestley (2002, Theorem 7.14)

shows $cl\ x \subseteq \bigcup (cl \text{ ` } (\{y. y \subseteq x\} \cap local.compact-points))$

<proof>

lemma *cl-insert*:

shows $cl (insert\ x\ X) = cl \{x\} \cup cl\ X$

<proof>

lemma *cl-UNION*:

shows $cl (\bigcup_{i \in I}. f\ i) = (\bigcup_{i \in I}. cl (f\ i)) \cup cl \{\}$

<proof>

lemma *closed-UNION*:

assumes $\bigwedge i. i \in I \implies f\ i \in closed$

shows $(\bigcup_{i \in I}. f\ i) \cup cl \{\} \in closed$

<proof>

lemma *sort-of-inverse*: — Pfaltz and Šlapal (2013, Proposition 2.5)

assumes $y \in cl\ X - cl \{\}$

shows $\exists x \in X. y \in cl \{x\}$

<proof>

lemma *cl-diff-le*:

shows $cl\ x - cl\ y \subseteq cl (x - y)$

<proof>

lemma *cl-bind*:

shows $cl (X \gg f) = (X \gg cl \circ f) \cup cl \{\}$
<proof>

lemma *sup-irreducible-on-singleton*:

shows *sup-irreducible-on closed* ($cl \{a\}$)
<proof>

end

5.3 Matroids and antimatroids

The *exchange* axiom characterises *matroids* (see, for instance, §6.1), while the *anti-exchange* axiom characterises *antimatroids* (see e.g. §7.1).

References:

- Pfaltz and Šlapal (2013) provide an overview of these concepts
- <https://en.wikipedia.org/wiki/Antimatroid>

definition *anti-exchange* :: ('a set \Rightarrow 'a set) \Rightarrow bool **where**

anti-exchange $cl \longleftrightarrow (\forall X x y. x \neq y \wedge y \in cl (insert x X) - cl X \longrightarrow x \notin cl (insert y X) - cl X)$

definition *exchange* :: ('a set \Rightarrow 'a set) \Rightarrow bool **where**

exchange $cl \longleftrightarrow (\forall X x y. y \in cl (insert x X) - cl X \longrightarrow x \in cl (insert y X) - cl X)$

lemmas *anti-exchangeI* = *iffD2[OF anti-exchange-def, rule-format]*

lemmas *exchangeI* = *iffD2[OF exchange-def, rule-format]*

lemma *anti-exchangeD*:

assumes $y \in cl (insert x X) - cl X$
assumes $x \neq y$
assumes *anti-exchange* cl
shows $x \notin cl (insert y X) - cl X$
<proof>

lemma *exchange-Image*: — Some matroids arise from equivalence relations. Note $sym r \wedge trans r \longrightarrow Refl r$

shows *exchange* (*Image* r) $\longleftrightarrow sym r \wedge trans r$
<proof>

locale *closure-powerset-distributive-exchange* =

closure-powerset-distributive
+ **assumes** *exchange*: *exchange* cl
begin

lemma *exchange-exchange*:

assumes $x \in cl \{y\}$
assumes $x \notin cl \{\}$
shows $y \in cl \{x\}$
<proof>

lemma *exchange-closed-inter*:

assumes $Q \in closed$
shows $cl P \cap Q = cl (P \cap Q)$ (**is** *?lhs = ?rhs*)
and $Q \cap cl P = cl (P \cap Q)$ (**is** *?thesis1*)
<proof>

lemma *exchange-both-closed-inter*:

assumes $P \in \text{closed}$
assumes $Q \in \text{closed}$
shows $cl (P \cap Q) = P \cap Q$
 $\langle \text{proof} \rangle$

end

lemma *anti-exchange-Image*: — when r is asymmetric on distinct points

shows *anti-exchange* (*Image* r) $\longleftrightarrow (\forall x y. x \neq y \wedge (x, y) \in r \longrightarrow (y, x) \notin r)$
 $\langle \text{proof} \rangle$

locale *closure-powerset-distributive-anti-exchange* =
closure-powerset-distributive
+ **assumes** *anti-exchange*: *anti-exchange* cl

5.4 Composition

Conditions under which composing two closures yields a closure. See also Pfaltz and Šlapal (2013).

lemma *closure-comp*:

assumes *closure lesseqa lessa* cl_1
assumes *closure lesseqa lessa* cl_2
assumes $\bigwedge X. cl_1 (cl_2 X) = cl_2 (cl_1 X)$
shows *closure lesseqa lessa* ($\lambda X. cl_1 (cl_2 X)$)
 $\langle \text{proof} \rangle$

lemma *closure-complete-lattice-comp*:

assumes *closure-complete-lattice Infa Supa infa lesseqa lessa supa bota topa* cl_1
assumes *closure-complete-lattice Infa Supa infa lesseqa lessa supa bota topa* cl_2
assumes $\bigwedge X. cl_1 (cl_2 X) = cl_2 (cl_1 X)$
shows *closure-complete-lattice Infa Supa infa lesseqa lessa supa bota topa* ($\lambda X. cl_1 (cl_2 X)$)
 $\langle \text{proof} \rangle$

lemma *closure-powerset-comp*:

assumes *closure-powerset* cl_1
assumes *closure-powerset* cl_2
assumes $\bigwedge X. cl_1 (cl_2 X) = cl_2 (cl_1 X)$
shows *closure-powerset* ($\lambda X. cl_1 (cl_2 X)$)
 $\langle \text{proof} \rangle$

lemma *closure-powerset-distributive-comp*:

assumes *closure-powerset-distributive* cl_1
assumes *closure-powerset-distributive* cl_2
assumes $\bigwedge X. cl_1 (cl_2 X) = cl_2 (cl_1 X)$
shows *closure-powerset-distributive* ($\lambda X. cl_1 (cl_2 X)$)
 $\langle \text{proof} \rangle$

5.5 Path independence

Pfaltz and Šlapal (2013, Prop 1.1): “an expansive operator is a closure operator iff it is path independent.”

References:

- \$AFP/Stable_Matching/Choice_Functions.thy

context *semilattice-sup*

begin

definition *path-independent* :: ($'a \Rightarrow 'a$) \Rightarrow *bool* **where**

path-independent $f \iff (\forall x y. f (x \sqcup y) = f (f x \sqcup f y))$

lemma *cl-path-independent*:

shows *closure* $(\leq) (<) cl \iff \text{path-independent } cl \wedge (\forall x. x \leq cl x)$ (**is** *?lhs* \iff *?rhs*)
<proof>

end

5.6 Some closures

interpretation *id-cl: closure-powerset-distributive id*
<proof>

5.6.1 Reflexive, symmetric and transitive closures

The reflexive closure *reflcl* is very well behaved. Note the new bottom is *Id*. The reflexive transitive closure *rtrancl* and transitive closure *trancl* are clearly not distributive.

rtrancl is neither matroidal nor antimatroidal.

interpretation *reflcl-cl: closure-powerset-distributive-exchange reflcl*
<proof>

interpretation *symcl-cl: closure-powerset-distributive-exchange* $\lambda X. X \cup X^{-1}$
<proof>

interpretation *trancl-cl: closure-powerset trancl*
<proof>

interpretation *rtrancl-cl: closure-powerset rtrancl*
<proof>

lemma *rtrancl-closed-Id*:
shows $Id \in \text{rtrancl-cl.closed}$
<proof>

lemma *rtrancl-closed-reflcl-closed*:
shows $\text{rtrancl-cl.closed} \subseteq \text{reflcl-cl.closed}$
<proof>

5.6.2 Relation image

lemma *idempotent-Image*:
assumes *refl-on* $Y r$
assumes *trans* r
assumes $X \subseteq Y$
shows $r \text{ `` } r \text{ `` } X = r \text{ `` } X$
<proof>

lemmas *distributive-Image = Image-eq-UN*

lemma *closure-powerset-distributive-ImageI*:
assumes $cl = \text{Image } r$
assumes *refl* r
assumes *trans* r
shows *closure-powerset-distributive* cl
<proof>

lemma *closure-powerset-distributive-exchange-ImageI*:
assumes $cl = \text{Image } r$

assumes *equiv UNIV r* — symmetric, transitive and universal domain
shows *closure-powerset-distributive-exchange cl*
⟨*proof*⟩

interpretation *Image-rtrancl: closure-powerset-distributive Image (r*)*
⟨*proof*⟩

5.6.3 Kleene closure

We define Kleene closure in the traditional way with respect to some axioms that our various lattices satisfy. As trace models are not going to validate $x \cdot \perp = \perp$ (Kozen 1994, Axiom 13), we cannot reuse existing developments of Kleene Algebra (and Concurrent Kleene Algebra (Hoare, Möller, Struth, and Wehrman 2011)). In general it is not distributive.

locale *weak-kleene* =
fixes *unit* :: 'a::complete-lattice (ε)
fixes *comp* :: 'a \Rightarrow 'a \Rightarrow 'a (**infixl** \cdot 60)
assumes *comp-assoc*: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
assumes *weak-comp-unitL*: $\varepsilon \leq x \implies \varepsilon \cdot x = x$
assumes *comp-unitR*: $x \cdot \varepsilon = x$
assumes *comp-supL*: $(x \sqcup y) \cdot z = (x \cdot z) \sqcup (y \cdot z)$
assumes *comp-supR*: $x \cdot (y \sqcup z) = (x \cdot y) \sqcup (x \cdot z)$
assumes *mcont-compL*: *mcont Sup* (\leq) *Sup* (\leq) $(\lambda x. x \cdot y)$
assumes *mcont-compR*: *mcont Sup* (\leq) *Sup* (\leq) $(\lambda y. x \cdot y)$
assumes *comp-botL*: $\perp \cdot x = \perp$

begin

lemma *mcont2mcont-comp*:
assumes *mcont Supa orda Sup* (\leq) *f*
assumes *mcont Supa orda Sup* (\leq) *g*
shows *mcont Supa orda Sup* (\leq) $(\lambda x. f x \cdot g x)$
⟨*proof*⟩

lemma *mono2mono-comp*:
assumes *monotone orda* (\leq) *f*
assumes *monotone orda* (\leq) *g*
shows *monotone orda* (\leq) $(\lambda x. f x \cdot g x)$
⟨*proof*⟩

context
notes *mcont2mcont-comp*[*cont-intro*]
notes *mono2mono-comp*[*cont-intro*, *partial-function-mono*]
notes *st-monotone*[*OF mcont-mono*[*OF mcont-compL*], *strg*]
notes *st-monotone*[*OF mcont-mono*[*OF mcont-compR*], *strg*]
begin

context
notes [*function-internals*] — Exposes the induction rules we need
begin

partial-function (*lfp*) *star* :: 'a \Rightarrow 'a **where**
star $x = (x \cdot \text{star } x) \sqcup \varepsilon$

partial-function (*lfp*) *rev-star* :: 'a \Rightarrow 'a **where**
rev-star $x = (\text{rev-star } x \cdot x) \sqcup \varepsilon$

end

lemmas *parallel-star-induct-1-1* =

parallel-fixp-induct-1-1[*OF*
complete-lattice-partial-function-definitions *complete-lattice-partial-function-definitions*
star.mono *star.mono* *star-def* *star-def*]

lemma *star-bot*:
 shows $star \perp = \varepsilon$
<proof>

lemma *epsilon-star-le*:
 shows $\varepsilon \leq star P$
<proof>

lemma *monotone-star*:
 shows *mono* *star*
<proof>

lemma *expansive-star*:
 shows $x \leq star x$
<proof>

lemma *star-comp-star*:
 shows $star x \cdot star x = star x$ (**is** *?lhs* = *?rhs*)
<proof>

lemma *idempotent-star*:
 shows $star (star x) = star x$ (**is** *?lhs* = *?rhs*)
<proof>

interpretation *star*: *closure-complete-lattice-class* *star*
<proof>

lemma *star-epsilon*:
 shows $star \varepsilon = \varepsilon$
<proof>

lemma *epsilon-rev-star-le*:
 shows $\varepsilon \leq rev-star P$
<proof>

lemma *rev-star-comp-rev-star*:
 shows $rev-star x \cdot rev-star x = rev-star x$ (**is** *?lhs* = *?rhs*)
<proof>

lemma *star-rev-star*:
 shows $star = rev-star$ (**is** *?lhs* = *?rhs*)
<proof>

lemmas *star-fixp-rev-induct* = *rev-star.fixp-induct*[*folded* *star-rev-star*]

interpretation *rev-star*: *closure-complete-lattice-class* *rev-star*
<proof>

lemma *rev-star-bot*:
 shows $rev-star \perp = \varepsilon$
<proof>

lemma *rev-star-epsilon*:
 shows $rev-star \varepsilon = \varepsilon$

<proof>

lemmas *star-unfoldL = star.simps*

lemma *star-unfoldR:*

shows $star\ x = (star\ x \cdot x) \sqcup \varepsilon$

<proof>

lemmas *rev-star-unfoldR = rev-star.simps*

lemma *rev-star-unfoldL:*

shows $rev\text{-}star\ x = (x \cdot rev\text{-}star\ x) \sqcup \varepsilon$

<proof>

lemma *fold-starL:*

shows $x \cdot star\ x \leq star\ x$

<proof>

lemma *fold-starR:*

shows $star\ x \cdot x \leq star\ x$

<proof>

lemma *fold-rev-starL:*

shows $x \cdot rev\text{-}star\ x \leq rev\text{-}star\ x$

<proof>

lemma *fold-rev-starR:*

shows $rev\text{-}star\ x \cdot x \leq rev\text{-}star\ x$

<proof>

declare *star.strengthen-cl[stg] rev-star.strengthen-cl[stg]*

end

end

locale *kleene = weak-kleene +*

assumes *comp-unitL: $\varepsilon \cdot x = x$ — satisfied by ('a, 's, 'v) prog but not ('a, 's, 'v) spec*

6 Galois connections

Here we collect some classical results for Galois connections. These are drawn from [Backhouse \(2000\)](#); [Davey and Priestley \(2002\)](#); [Melton, Schmidt, and Strecker \(1985\)](#); [Müller-Olm \(1997\)](#) amongst others. The canonical reference is likely [Gierz, Hofmann, Keimel, Lawson, Mislove, and Scott \(2003\)](#).

Our focus is on constructing closures (§5) conveniently; we are less interested in the fixed-point story. Many of these results hold for preorders; we simply work with partial orders (via the *ordering* locale). Similarly *conditionally complete lattices* are often sufficient, but for convenience we just assume (unconditional) completeness.

locale *galois =*

orda: ordering less-eqa lessa

+ *ordb: ordering less-eqb lessb*

for *less-eqa (infix \leq_a 50)*

and *lessa (infix $<_a$ 50)*

and *less-eqb (infix \leq_b 50)*

and *lessb (infix $<_b$ 50)*

+ **fixes** *lower :: 'a \Rightarrow 'b*

fixes *upper :: 'b \Rightarrow 'a*

assumes *galois: lower $x \leq_b y \iff x \leq_a upper\ y$*

begin

lemma *monotone-lower*:

shows *monotone* (\leq_a) (\leq_b) *lower*
<proof>

lemma *monotone-upper*:

shows *monotone* (\leq_b) (\leq_a) *upper*
<proof>

lemmas *strengthen-lower*[*strg*] = *st-monotone*[*OF monotone-lower*]

lemmas *strengthen-upper*[*strg*] = *st-monotone*[*OF monotone-upper*]

lemma *upper-lower-expansive*:

shows $x \leq_a$ *upper* (*lower* x)
<proof>

lemma *lower-upper-contractive*:

shows *lower* (*upper* x) \leq_b x
<proof>

lemma *comp-galois*: — Backhouse (2000, Lemma 19). Observe that the roles of upper and lower have swapped.

fixes *less-egc* :: ' $c \Rightarrow 'c \Rightarrow$ *bool* (**infix** \leq_c 50)
fixes *lessc* :: ' $c \Rightarrow 'c \Rightarrow$ *bool* (**infix** $<_c$ 50)
fixes h :: ' $a \Rightarrow 'c$
fixes k :: ' $b \Rightarrow 'c$
assumes *partial-preordering* (\leq_c)
assumes *monotone* (\leq_a) (\leq_c) h
assumes *monotone* (\leq_b) (\leq_c) k
shows $(\forall x. h$ (*upper* x) \leq_c k x) \longleftrightarrow $(\forall x. h$ x \leq_c k (*lower* x))
<proof>

lemma *lower-upper-le-iff*: — Backhouse (2000, Lemma 23)

assumes $\forall x y. lower' x \leq_b y \longleftrightarrow x \leq_a upper' y$
shows $(\forall x. lower' x \leq_b lower x) \longleftrightarrow (\forall y. upper y \leq_a upper' y)$
<proof>

lemma *lower-upper-unique*: — Backhouse (2000, Lemma 24)

assumes $\forall x y. lower' x \leq_b y \longleftrightarrow x \leq_a upper' y$
shows $lower' = lower \longleftrightarrow upper' = upper$
<proof>

lemma *upper-lower-idem*:

shows *upper* (*lower* (*upper* (*lower* x))) = *upper* (*lower* x)
<proof>

lemma *lower-upper-idem*:

shows *lower* (*upper* (*lower* (*upper* x))) = *lower* (*upper* x)
<proof>

lemma *lower-upper-lower*: — Melton et al. (1985, Proposition 1.2(2))

shows $lower \circ upper \circ lower = lower$
and $lower$ (*upper* (*lower* x)) = *lower* x
<proof>

lemma *upper-lower-upper*: — Melton et al. (1985, Proposition 1.2(2))

shows $upper \circ lower \circ upper = upper$
and *upper* (*lower* (*upper* x)) = *upper* x

<proof>

definition $cl :: 'a \Rightarrow 'a$ **where** — The opposite composition yields a kernel operator
 $cl\ x = upper\ (lower\ x)$

lemma *cl-axiom*:

shows $(x \leq_a cl\ y) = (cl\ x \leq_a cl\ y)$

<proof>

sublocale *closure* (\leq_a) $(<_a)$ cl — incorporates definitions and lemmas into this namespace

<proof>

lemma *cl-upper*:

shows $cl\ (upper\ P) = upper\ P$

<proof>

lemma *closed-upper*:

shows $upper\ P \in closed$

<proof>

lemma *inj-lower-iff-surj-upper*:

shows $inj\ lower \longleftrightarrow surj\ upper$

<proof>

lemma *inj-lower-iff-upper-lower-id*:

shows $inj\ lower \longleftrightarrow upper \circ lower = id$

<proof>

lemma *upper-inj-iff-surj-lower*:

shows $inj\ upper \longleftrightarrow surj\ lower$

<proof>

lemma *inj-upper-iff-lower-upper-id*:

shows $inj\ upper \longleftrightarrow lower \circ upper = id$

<proof>

lemma *lower-downset-upper*: — Davey and Priestley (2002, Lemma 7.32): inverse image of lower on a downset is the downset of upper

shows $lower^{-1}\ \{a.\ a \leq_b\ y\} = \{a.\ a \leq_a\ upper\ y\}$

<proof>

lemma *lower-downset*: — Davey and Priestley (2002, Lemma 7.32); equivalent to the Galois axiom

shows $\exists!x.\ lower^{-1}\ \{a.\ a \leq_b\ y\} = \{a.\ a \leq_a\ x\}$

<proof>

end

<ML>

lemma *axioms-alt*:

fixes *less-eqa* (**infix** \leq_a 50)

fixes *less-eqb* (**infix** \leq_b 50)

fixes *lower* :: $'a \Rightarrow 'b$

fixes *upper* :: $'b \Rightarrow 'a$

assumes *oa*: *ordering less-eqa lessa*

assumes *ob*: *ordering less-eqb lessb*

assumes *ul*: $\forall x.\ x \leq_a\ upper\ (lower\ x)$

assumes *lu*: $\forall x.\ lower\ (upper\ x) \leq_b\ x$

assumes *ml*: monotone (\leq_a) (\leq_b) lower
assumes *mu*: monotone (\leq_b) (\leq_a) upper
shows lower $x \leq_b y \iff x \leq_a \text{ upper } y$
 <proof>

lemma *compose*:

fixes *lower*₁ :: 'b \Rightarrow 'c
fixes *lower*₂ :: 'a \Rightarrow 'b
fixes *less-eqa* :: 'a \Rightarrow 'a \Rightarrow bool
assumes *galois less-eqb lessb less-eqc lessc lower*₁ *upper*₁
assumes *galois less-eqa lessa less-eqb lessb lower*₂ *upper*₂
shows *galois less-eqa lessa less-eqc lessc* (*lower*₁ \circ *lower*₂) (*upper*₂ \circ *upper*₁)
 <proof>

locale *complete-lattice* =

cla: complete-lattice *Inf*_a *Sup*_a (\sqcap_a) (\leq_a) $(<_a)$ (\sqcup_a) \perp_a \top_a
 + *clb*: complete-lattice *Inf*_b *Sup*_b (\sqcap_b) (\leq_b) $(<_b)$ (\sqcup_b) \perp_b \top_b
 + *galois* (\leq_a) $(<_a)$ (\leq_b) $(<_b)$ lower upper
for *less-eqa* :: 'a \Rightarrow 'a \Rightarrow bool (**infix** \leq_a 50)
and *lessa* (**infix** $<_a$ 50)
and *infa* (**infixl** \sqcap_a 70)
and *supa* (**infixl** \sqcup_a 65)
and *bota* (\perp_a)
and *topa* (\top_a)
and *Inf*_a *Sup*_a
and *less-eqb* :: 'b \Rightarrow 'b \Rightarrow bool (**infix** \leq_b 50)
and *lessb* (**infix** $<_b$ 50)
and *infb* (**infixl** \sqcap_b 70)
and *supb* (**infixl** \sqcup_b 65)
and *botb* (\perp_b)
and *topb* (\top_b)
and *Inf*_b *Sup*_b
and *lower* :: 'a \Rightarrow 'b
and *upper* :: 'b \Rightarrow 'a

begin

lemma *lower-bot*:

shows lower $\perp_a = \perp_b$
 <proof>

lemmas *mono2mono-lower*[*cont-intro, partial-function-mono*] = *monotone2monotone*[*OF monotone-lower, simplified*]

lemma *lower-Sup*: — Melton et al. (1985, Proposition 1.2(6)): *lower* is always a distributive operation

shows lower $(\text{Sup}_a X) = \text{Sup}_b (\text{lower } ' X)$ (**is** ?lhs = ?rhs)
 <proof>

lemma *lower-SUP*:

shows lower $(\text{Sup}_a (f ' X)) = \text{Sup}_b ((\lambda x. \text{lower } (f x)) ' X)$
 <proof>

lemma *lower-sup*:

shows lower $(X \sqcup_a Y) = \text{lower } X \sqcup_b \text{lower } Y$
 <proof>

lemma *lower-Inf-le*:

shows lower $(\text{Inf}_a X) \leq_b \text{Inf}_b (\text{lower } ' X)$
 <proof>

lemma *lower-INF-le*:

shows $\text{lower } (\text{Inf}_a (f \text{ ' } X)) \leq_b \text{Inf}_b ((\lambda x. \text{lower } (f x)) \text{ ' } X)$
<proof>

lemma *lower-inf-le*:

shows $\text{lower } (x \sqcap_a y) \leq_b \text{lower } x \sqcap_b \text{lower } y$
<proof>

lemma *mcont-lower*: — [Backhouse \(2000\)](#): fixed point theory based on Galois connections is less general than using countable chains

shows $\text{mcont } \text{Sup}_a (\leq_a) \text{Sup}_b (\leq_b) \text{lower}$
<proof>

lemma *mcont2mcont-lower[cont-intro]*:

assumes $\text{mcont } \text{luba } \text{orda } \text{Sup}_a (\leq_a) P$
shows $\text{mcont } \text{luba } \text{orda } \text{Sup}_b (\leq_b) (\lambda x. \text{lower } (P x))$
<proof>

lemma *upper-top*:

shows $\text{upper } \top_b = \top_a$
<proof>

lemma *Sup-upper-le*:

shows $\text{Sup}_a (\text{upper } \text{ ' } X) \leq_a \text{upper } (\text{Sup}_b X)$
<proof>

lemma *sup-upper-le*:

shows $\text{upper } x \sqcup_a \text{upper } y \leq_a \text{upper } (x \sqcup_b y)$
<proof>

lemma *upper-Inf*: — [Melton et al. \(1985, Proposition 1.2\(6\)\)](#)

shows $\text{upper } (\text{Inf}_b X) = \text{Inf}_a (\text{upper } \text{ ' } X)$ (**is** *?lhs = ?rhs*)
<proof>

lemma *upper-INF*:

shows $\text{upper } (\text{Inf}_b (f \text{ ' } X)) = \text{Inf}_a ((\lambda x. \text{upper } (f x)) \text{ ' } X)$
<proof>

lemma *upper-inf*:

shows $\text{upper } (X \sqcap_b Y) = \text{upper } X \sqcap_a \text{upper } Y$
<proof>

In a complete lattice *lower* is determined by *upper* and vice-versa.

lemma *lower-Inf-upper*:

shows $\text{lower } X = \text{Inf}_b \{Y. X \leq_a \text{upper } Y\}$
<proof>

lemma *upper-Sup-lower*:

shows $\text{upper } X = \text{Sup}_a \{Y. \text{lower } Y \leq_b X\}$
<proof>

lemma *upper-downwards-closure-lower*: — [Melton et al. \(1985, Lemma 2.1\)](#)

shows $\text{upper } x = \text{Sup}_a (\text{lower } - \text{ ' } \{y. y \leq_b x\})$
<proof>

sublocale *closure-complete-lattice* $(\leq_a) (<_a) (\sqcap_a) (\sqcup_a) \perp_a \top_a \text{Inf}_a \text{Sup}_a \text{cl}$

<proof>

end

locale *complete-lattice-distributive* =

galois.complete-lattice

+ **assumes** *upper-Sup-le*: $upper (Sup_b X) \leq_a Sup_a (upper \text{ ` } X)$ — Stronger than Scott continuity, which only asks for this for chain or directed X .

begin

lemma *upper-Sup*:

shows $upper (Sup_b X) = Sup_a (upper \text{ ` } X)$

<proof>

lemma *upper-bot*:

shows $upper \perp_b = \perp_a$

<proof>

lemma *upper-sup*:

shows $upper (x \sqcup_b y) = upper x \sqcup_a upper y$

<proof>

lemmas *mono2mono-upper*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF monotone-upper*, *simplified*]

lemma *mcont-upper*:

shows $mcont Sup_b (\leq_b) Sup_a (\leq_a) upper$

<proof>

lemma *mcont2mcont-upper*[*cont-intro*]:

assumes $mcont lub_a orda Sup_b (\leq_b) P$

shows $mcont lub_a orda Sup_a (\leq_a) (\lambda x. upper (P x))$

<proof>

sublocale *closure-complete-lattice-distributive* $(\leq_a) (<_a) (\sqcap_a) (\sqcup_a) \perp_a \top_a Inf_a Sup_a cl$

<proof>

lemma *cl-bot*:

shows $cl \perp_a = \perp_a$

<proof>

lemma *closed-bot*[*iff*]:

shows $\perp_a \in closed$

<proof>

end

locale *complete-lattice-class* =

galois.complete-lattice

$(\leq) (<) (\sqcap) (\sqcup) \perp :: - :: complete-lattice \top Inf Sup$

$(\leq) (<) (\sqcap) (\sqcup) \perp :: - :: complete-lattice \top Inf Sup$

begin

sublocale *closure-complete-lattice-class* cl *<proof>*

end

locale *complete-lattice-distributive-class* =

galois.complete-lattice-distributive

$(\leq) (\lt) (\sqcap) (\sqcup) \perp :: - :: \text{complete-lattice} \top \text{Inf Sup}$

$(\leq) (\lt) (\sqcap) (\sqcup) \perp :: - :: \text{complete-lattice} \top \text{Inf Sup}$

begin

sublocale *galois.complete-lattice-class* $\langle \text{proof} \rangle$

sublocale *closure-complete-lattice-distributive-class* *cl* $\langle \text{proof} \rangle$

end

lemma *existence-lower-preserves-Sup*: — Hoare and He (1987, p8 of Oxford TR PRG-44) amongst others

fixes *lower* :: $-\ :: \text{complete-lattice} \Rightarrow - :: \text{complete-lattice}$

assumes *mono lower*

shows $(\forall x y. \text{lower } x \leq y \iff x \leq \sqcup \{Y. \text{lower } Y \leq y\}) \iff (\forall X. \text{lower } (\sqcup X) \leq \sqcup (\text{lower } 'X))$ (**is** *?lhs*
 \iff *?rhs*)

$\langle \text{proof} \rangle$

lemma *lower-preserves-SupI*:

assumes *mono lower*

assumes $\bigwedge X. \text{lower } (\sqcup X) \leq \sqcup (\text{lower } 'X)$

assumes $\bigwedge x. \text{upper } x = \sqcup \{X. \text{lower } X \leq x\}$

shows *galois.complete-lattice-class lower upper*

$\langle \text{proof} \rangle$

lemma *existence-upper-preserves-Inf*:

fixes *upper* :: $-\ :: \text{complete-lattice} \Rightarrow - :: \text{complete-lattice}$

assumes *mono upper*

shows $(\forall x y. \sqcap \{Y. x \leq \text{upper } Y\} \leq y \iff x \leq \text{upper } y) \iff (\forall X. \sqcap (\text{upper } 'X) \leq \text{upper } (\sqcap X))$ (**is** *?lhs*
 \iff *?rhs*)

$\langle \text{proof} \rangle$

lemma *upper-preserves-InfI*:

assumes *mono upper*

assumes $\bigwedge X. \sqcap (\text{upper } 'X) \leq \text{upper } (\sqcap X)$

assumes $\bigwedge x. \text{lower } x = \sqcap \{X. x \leq \text{upper } X\}$

shows *galois.complete-lattice-class lower upper*

$\langle \text{proof} \rangle$

locale *powerset* =

galois.complete-lattice-class lower upper

for *lower* :: $'a \text{ set} \Rightarrow 'b \text{ set}$

and *upper* :: $'b \text{ set} \Rightarrow 'a \text{ set}$

begin

lemma *lower-insert*:

shows $\text{lower } (\text{insert } x X) = \text{lower } \{x\} \cup \text{lower } X$
 $\langle \text{proof} \rangle$

lemma *lower-distributive*:

shows $\text{lower } X = (\bigcup x \in X. \text{lower } \{x\})$
 $\langle \text{proof} \rangle$

sublocale *closure-powerset* *cl* $\langle \text{proof} \rangle$

end

locale *powerset-distributive* =

galois.powerset

+ *galois.complete-lattice-distributive-class*

begin

lemma *upper-insert*:

shows $upper (insert\ x\ X) = upper\ \{x\} \cup upper\ X$
<proof>

lemma *cl-distributive-axiom*:

shows $cl\ (\bigcup\ X) \subseteq \bigcup\ (cl\ 'X)$
<proof>

sublocale *closure-powerset-distributive cl*

<proof>

end

Müller-Olm (1997, Theorems 3.3.1, 3.3.2): relation image forms a Galois connection. See also Davey and Priestley (2002, Exercise 7.18).

definition $lower_R :: ('a \times 'b)\ set \Rightarrow 'a\ set \Rightarrow 'b\ set$ **where**
 $lower_R\ R\ A = R\ \text{``}\ A$

definition $upper_R :: ('a \times 'b)\ set \Rightarrow 'b\ set \Rightarrow 'a\ set$ **where**
 $upper_R\ R\ B = \{a. \forall b. (a, b) \in R \longrightarrow b \in B\}$

interpretation *relation*: $galois.powerset\ galois.lower_R\ R\ galois.upper_R\ R$
<proof>

context *galois.powerset*

begin

lemma *relations-galois*:

defines $R \equiv \{(a, b). b \in lower\ \{a\}\}$
shows $lower = galois.lower_R\ R$
and $upper = galois.upper_R\ R$
<proof>

end

<ML>

6.1 Some Galois connections

<ML>

locale *complete-lattice-class-monomorphic*

= *galois.complete-lattice-class upper lower*

for $upper :: 'a::complete-lattice \Rightarrow 'a$ **and** $lower :: 'a \Rightarrow 'a$ — Avoid *'a itself* parameters

interpretation *conj-imp*: $galois.complete-lattice-class\ (\sqcap)\ x\ (\longrightarrow_B)\ x$ **for** $x :: -::boolean-algebra$ — Classic example

<proof>

There are very well-behaved Galois connections arising from the image (and inverse image) of sets under a function; stuttering is one instance (§8.1).

locale *image-vimage* =

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

begin

definition $lower :: 'a\ set \Rightarrow 'b\ set$ **where**

lower $X = f \text{ ' } X$

definition *upper* :: 'b set \Rightarrow 'a set **where**

upper $X = f \text{ -' } X$

lemma *upper-empty*[*iff*]:

shows *upper* $\{\}$ = $\{\}$

<proof>

sublocale *galois.powerset-distributive lower upper*

<proof>

abbreviation *equivalent* :: 'a relp **where**

equivalent $x \ y \equiv f \ x = f \ y$

lemma *equiv*:

shows *Equiv-Relations.equivp equivalent*

<proof>

lemma *equiv-cl-singleton*:

assumes *equivalent* $x \ y$

shows *cl* $\{x\} = cl \ \{y\}$

<proof>

lemma *cl-alt-def*:

shows *cl* $X = \{(x, y). \text{equivalent } x \ y\} \text{ `` } X$

<proof>

sublocale *closure-powerset-distributive-exchange cl*

<proof>

lemma *closed-in*:

assumes $x \in P$

assumes *equivalent* $x \ y$

assumes $P: P \in \text{closed}$

shows $y \in P$

<proof>

lemma *clE*:

assumes $x \in cl \ P$

obtains y **where** *equivalent* $y \ x$ **and** $y \in P$

<proof>

lemma *clI*[*intro*]:

assumes $x \in P$

assumes *equivalent* $x \ y$

shows $y \in cl \ P$

<proof>

lemma *closed-diff*[*intro*]:

assumes $X \in \text{closed}$

assumes $Y \in \text{closed}$

shows $X - Y \in \text{closed}$

<proof>

lemma *closed-uminus*[*intro*]:

assumes $X \in \text{closed}$

shows $-X \in \text{closed}$

$\langle proof \rangle$

end

locale *image-vimage-monomorphic*

= *galois.image-vimage f*

for $f :: 'a \Rightarrow 'a$ — Avoid *'a itself* parameters

locale *image-vimage-idempotent*

= *galois.image-vimage-monomorphic* +

assumes *f-idempotent*: $\bigwedge x. f (f x) = f x$

begin

lemma *f-idempotent-comp*:

shows $f \circ f = f$

$\langle proof \rangle$

lemma *idemI*:

assumes $f x \in P$

shows $x \in cl P$

$\langle proof \rangle$

lemma *f-cl*:

shows $f x \in cl P \longleftrightarrow x \in cl P$

$\langle proof \rangle$

lemma *f-closed*:

assumes $P \in closed$

shows $f x \in P \longleftrightarrow x \in P$

$\langle proof \rangle$

lemmas *f-closedI = iffD1[OF f-closed]*

end

$\langle ML \rangle$

7 Heyting algebras

Our (complete) lattices are Heyting algebras. The following development is oriented towards using the derived Heyting implication in a logical fashion. As there are no standard classes for semi-(complete-)lattices we simply work with complete lattices.

References:

- [Esakia, Bezhanishvili, Holliday, and Evseev \(2019\)](#) – fundamental theory
- [van Dalen \(2004, Lemma 5.2.1\)](#) – some equivalences
- <https://en.wikipedia.org/wiki/Pseudocomplement> – properties

class *heyting-algebra* = *complete-lattice* +

assumes *inf-Sup-distrib1*: $\bigwedge Y :: 'a \text{ set}. \bigwedge x :: 'a. x \sqcap (\bigsqcup Y) = (\bigsqcup_{y \in Y} x \sqcap y)$

begin

definition *heyting* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** \longrightarrow_H 53) **where**

$x \longrightarrow_H y = \bigsqcup \{z. x \sqcap z \leq y\}$

lemma *heyting*: — The Galois property for (\sqcap) and \longrightarrow_H

shows $z \leq x \longrightarrow_H y \longleftrightarrow z \sqcap x \leq y$ (**is** ?lhs \longleftrightarrow ?rhs)
(proof)

end

(ML)

context *heyting-algebra*

begin

lemma *commute*:

shows $x \sqcap z \leq y \longleftrightarrow z \leq (x \longrightarrow_H y)$
(proof)

lemmas *uncurry* = *iffD1[OF heyting]*

lemmas *curry* = *iffD2[OF heyting]*

lemma *curry-conv*:

shows $(x \sqcap y \longrightarrow_H z) = (x \longrightarrow_H y \longrightarrow_H z)$
(proof)

lemma *swap*:

shows $P \longrightarrow_H Q \longrightarrow_H R = Q \longrightarrow_H P \longrightarrow_H R$
(proof)

lemma *absorb*:

shows $y \sqcap (x \longrightarrow_H y) = y$
and $(x \longrightarrow_H y) \sqcap y = y$
(proof)

lemma *detachment*:

shows $x \sqcap (x \longrightarrow_H y) = x \sqcap y$ (**is** ?thesis1)
and $(x \longrightarrow_H y) \sqcap x = x \sqcap y$ (**is** ?thesis2)
(proof)

lemma *discharge*:

assumes $x' \leq x$
shows $x' \sqcap (x \longrightarrow_H y) = x' \sqcap y$ (**is** ?thesis1)
and $(x \longrightarrow_H y) \sqcap x' = y \sqcap x'$ (**is** ?thesis2)
(proof)

lemma *trans*:

shows $(x \longrightarrow_H y) \sqcap (y \longrightarrow_H z) \leq x \longrightarrow_H z$
(proof)

lemma *rev-trans*:

shows $(y \longrightarrow_H z) \sqcap (x \longrightarrow_H y) \leq x \longrightarrow_H z$
(proof)

lemma *discard*:

shows $Q \leq P \longrightarrow_H Q$
(proof)

lemma *infR*:

shows $x \longrightarrow_H y \sqcap z = (x \longrightarrow_H y) \sqcap (x \longrightarrow_H z)$
(proof)

lemma *mono*:

assumes $x' \leq x$
assumes $y \leq y'$
shows $x \longrightarrow_H y \leq x' \longrightarrow_H y'$
 $\langle \text{proof} \rangle$

lemma *strengthen*[*strg*]:
assumes *st-ord* $(\neg F) X X'$
assumes *st-ord* $F Y Y'$
shows *st-ord* $F (X \longrightarrow_H Y) (X' \longrightarrow_H Y')$
 $\langle \text{proof} \rangle$

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes *monotone orda* $(\geq) F$
assumes *monotone orda* $(\leq) G$
shows *monotone orda* $(\leq) (\lambda x. F x \longrightarrow_H G x)$
 $\langle \text{proof} \rangle$

lemma *mp*:
assumes $x \leq y \longrightarrow_H z$
assumes $x \leq y$
shows $x \leq z$
 $\langle \text{proof} \rangle$

lemma *botL*:
shows $\perp \longrightarrow_H x = \top$
 $\langle \text{proof} \rangle$

lemma *top-conv*:
shows $x \longrightarrow_H y = \top \longleftrightarrow x \leq y$
 $\langle \text{proof} \rangle$

lemma *refl*[*simp*]:
shows $x \longrightarrow_H x = \top$
 $\langle \text{proof} \rangle$

lemma *topL*[*simp*]:
shows $\top \longrightarrow_H x = x$
 $\langle \text{proof} \rangle$

lemma *topR*[*simp*]:
shows $x \longrightarrow_H \top = \top$
 $\langle \text{proof} \rangle$

lemma *K*[*simp*]:
shows $x \longrightarrow_H (y \longrightarrow_H x) = \top$
 $\langle \text{proof} \rangle$

subclass *distrib-lattice*
 $\langle \text{proof} \rangle$

lemma *supL*:
shows $(x \sqcup y) \longrightarrow_H z = (x \longrightarrow_H z) \sqcap (y \longrightarrow_H z)$
 $\langle \text{proof} \rangle$

subclass (in *complete-distrib-lattice*) *heyting-algebra* $\langle \text{proof} \rangle$

lemma *inf-Sup-distrib*:
shows $x \sqcap \sqcup Y = (\sqcup y \in Y. x \sqcap y)$

and $\bigsqcup Y \sqcap x = (\bigsqcup y \in Y. x \sqcap y)$
 ⟨proof⟩

lemma *inf-SUP-distrib*:

shows $x \sqcap (\bigsqcup i \in I. Y i) = (\bigsqcup i \in I. x \sqcap Y i)$
and $(\bigsqcup i \in I. Y i) \sqcap x = (\bigsqcup i \in I. Y i \sqcap x)$

⟨proof⟩

end

lemma *eq-boolean-implication*: — the implications coincide in *boolean-algebras*

fixes $x :: \text{-}::\text{boolean-algebra}$

shows $x \longrightarrow_H y = x \longrightarrow_B y$

⟨proof⟩

lemmas *simp-thms* =

heyting.botL

heyting.topL

heyting.topR

heyting.refl

lemma *Sup-prime-Sup-irreducible-iff*:

fixes $x :: \text{-}::\text{heyting-algebra}$

shows *Sup-prime* $x \longleftrightarrow$ *Sup-irreducible* x

⟨proof⟩

Logical rules ala HOL **lemma** *bspec*:

fixes $P :: - \Rightarrow (\text{-}::\text{heyting-algebra})$

shows $x \in X \Longrightarrow (\prod x \in X. P x \longrightarrow_H Q x) \sqcap P x \leq Q x$ (**is** $?X \Longrightarrow ?thesis1$)

and $x \in X \Longrightarrow P x \sqcap (\prod x \in X. P x \longrightarrow_H Q x) \leq Q x$ (**is** $- \Longrightarrow ?thesis2$)

and $(\prod x. P x \longrightarrow_H Q x) \sqcap P x \leq Q x$ (**is** $?thesis3$)

and $P x \sqcap (\prod x. P x \longrightarrow_H Q x) \leq Q x$ (**is** $?thesis4$)

⟨proof⟩

lemma *INFL*:

fixes $Q :: \text{-}::\text{heyting-algebra}$

shows $(\prod x \in X. P x \longrightarrow_H Q) = (\bigsqcup x \in X. P x) \longrightarrow_H Q$ (**is** $?lhs = ?rhs$)

⟨proof⟩

lemmas *SUPL* = *heyting.INFL[symmetric]*

lemma *INFR*:

fixes $P :: \text{-}::\text{heyting-algebra}$

shows $(\prod x \in X. P \longrightarrow_H Q x) = (P \longrightarrow_H (\prod x \in X. Q x))$ (**is** $?lhs = ?rhs$)

⟨proof⟩

lemmas *Inf-simps* = — "Miniscoping: pushing in universal quantifiers."

Inf-inf

inf-Inf

INF-inf-const1

INF-inf-const2

heyting.INFL

heyting.INFR

lemma *SUPL-le*:

fixes $Q :: \text{-}::\text{heyting-algebra}$

shows $(\bigsqcup x \in X. P x \longrightarrow_H Q) \leq (\prod x \in X. P x) \longrightarrow_H Q$

⟨proof⟩

lemma *SUPR-le*:

fixes $P :: \text{-}::\text{heyting-algebra}$

shows $(\bigsqcup_{x \in X}. P \longrightarrow_H Q \ x) \leq P \longrightarrow_H (\bigsqcup_{x \in X}. Q \ x)$

$\langle\text{proof}\rangle$

lemma *SUP-inf*:

fixes $Q :: \text{-}::\text{heyting-algebra}$

shows $(\bigsqcup_{x \in X}. P \ x \sqcap Q) = (\bigsqcup_{x \in X}. P \ x) \sqcap Q$

$\langle\text{proof}\rangle$

lemma *inf-SUP*:

fixes $P :: \text{-}::\text{heyting-algebra}$

shows $(\bigsqcup_{x \in X}. P \sqcap Q \ x) = P \sqcap (\bigsqcup_{x \in X}. Q \ x)$

$\langle\text{proof}\rangle$

lemmas *Sup-simps* = — "Miniscoping: pushing in universal quantifiers."

sup-SUP

SUP-sup

heyting.inf-SUP

heyting.SUP-inf

lemma *mcont2mcont-inf[cont-intro]*:

fixes $F :: \text{-} \Rightarrow 'a::\text{heyting-algebra}$

fixes $G :: \text{-} \Rightarrow 'a::\text{heyting-algebra}$

assumes *mcont luba orda Sup* $(\leq) \ F$

assumes *mcont luba orda Sup* $(\leq) \ G$

shows *mcont luba orda Sup* $(\leq) \ (\lambda x. F \ x \sqcap G \ x)$

$\langle\text{proof}\rangle$

lemma *closure-imp-distrib-le*: — [Abadi and Plotkin \(1993, Lemma 3.3\)](#), generalized

fixes $P \ Q :: \text{-}::\text{heyting-algebra}$

assumes *cl: closure-axioms* $(\leq) \ cl$

assumes *cl-inf*: $\bigwedge x \ y. cl \ x \sqcap cl \ y \leq cl \ (x \sqcap y)$

shows $P \longrightarrow_H Q \leq cl \ P \longrightarrow_H cl \ Q$

$\langle\text{proof}\rangle$

$\langle ML \rangle$

Pseudocomplements **definition** *pseudocomplement* $:: 'a::\text{heyting-algebra} \Rightarrow 'a \ (\neg_H - [75] \ 75)$ **where**

$\neg_H x = x \longrightarrow_H \perp$

lemma *pseudocomplementI*:

shows $x \leq \neg_H y \longleftrightarrow x \sqcap y \leq \perp$

$\langle\text{proof}\rangle$

$\langle ML \rangle$

lemma *monotone*:

shows *antimono pseudocomplement*

$\langle\text{proof}\rangle$

lemmas *strengthen[strg]* = *st-monotone[OF pseudocomplement.monotone]*

lemmas *mono* = *monotoneD[OF pseudocomplement.monotone]*

lemmas *mono2mono[cont-intro, partial-function-mono]*

= *monotone2monotone[OF pseudocomplement.monotone, simplified, of orda P for orda P]*

lemma *eq-boolean-negation*: — the negations coincide in *boolean-algebras*

fixes $x :: -::\{\text{boolean-algebra}, \text{heyting-algebra}\}$
shows $\neg_H x = -x$
 $\langle \text{proof} \rangle$

lemma *heyting*:
shows $x \longrightarrow_H \neg_H x = \neg_H x$
 $\langle \text{proof} \rangle$

lemma *Inf*:
shows $x \sqcap \neg_H x = \perp$
and $\neg_H x \sqcap x = \perp$
 $\langle \text{proof} \rangle$

lemma *double-le*:
shows $x \leq \neg_H \neg_H x$
 $\langle \text{proof} \rangle$

interpretation *double*: *closure-complete-lattice-class pseudocomplement* \circ *pseudocomplement*
 $\langle \text{proof} \rangle$

lemma *triple*:
shows $\neg_H \neg_H \neg_H x = \neg_H x$
 $\langle \text{proof} \rangle$

lemma *contrapos-le*:
shows $x \longrightarrow_H y \leq \neg_H y \longrightarrow_H \neg_H x$
 $\langle \text{proof} \rangle$

lemma *sup-inf*: — half of de Morgan
shows $\neg_H (x \sqcup y) = \neg_H x \sqcap \neg_H y$
 $\langle \text{proof} \rangle$

lemma *inf-sup-weak*: — the weakened other half of de Morgan
shows $\neg_H (x \sqcap y) = \neg_H \neg_H (\neg_H x \sqcup \neg_H y)$
 $\langle \text{proof} \rangle$

lemma *fix-triv*:
assumes $x = \neg_H x$
shows $x = y$
 $\langle \text{proof} \rangle$

lemma *double-top*:
shows $\neg_H \neg_H (x \sqcup \neg_H x) = \top$
 $\langle \text{proof} \rangle$

lemma *Inf-inf*:
fixes $P :: - \Rightarrow (-::\text{heyting-algebra})$
shows $(\prod x. P x) \sqcap \neg_H P x = \perp$
 $\langle \text{proof} \rangle$

lemma *SUP-le*: — half of de Morgan
fixes $P :: - \Rightarrow (-::\text{heyting-algebra})$
shows $(\bigsqcup x \in X. P x) \leq \neg_H (\prod x \in X. \neg_H P x)$
 $\langle \text{proof} \rangle$

lemma *SUP-INF-le*:
fixes $P :: - \Rightarrow (-::\text{heyting-algebra})$
shows $(\bigsqcup x \in X. \neg_H P x) \leq \neg_H (\prod x \in X. P x)$

$\langle proof \rangle$

lemma *SUP*:

fixes $P :: - \Rightarrow (-::\text{heyting-algebra})$

shows $\neg_H(\bigsqcup_{x \in X}. P\ x) = (\bigsqcap_{x \in X}. \neg_H P\ x)$

$\langle proof \rangle$

$\langle ML \rangle$

7.1 Downwards closure of preorders (downsets)

A *downset* (also *lower set* and *order ideal*) is a subset of a preorder that is closed under the order relation. (An *ideal* is a downset that is *directed*.) Some results require antisymmetry (a partial order).

References:

- [Vickers \(1989\)](#), early chapters.
- https://en.wikipedia.org/wiki/Alexandrov_topology
- [Abadi and Plotkin \(1991, §3\)](#)

$\langle ML \rangle$

definition $cl :: 'a::preorder\ set \Rightarrow 'a\ set$ **where**

$cl\ P = \{x \mid x\ y.\ y \in P \wedge x \leq y\}$

$\langle ML \rangle$

interpretation *downwards: closure-powerset-distributive downwards.cl* — On preorders

$\langle proof \rangle$

interpretation *downwards: closure-powerset-distributive-anti-exchange (downwards.cl:::order set \Rightarrow -)*

— On partial orders; see [Pfaltz and Šlapal \(2013\)](#)

$\langle proof \rangle$

$\langle ML \rangle$

lemma *cl-empty*:

shows $downwards.cl\ \{\} = \{\}$

$\langle proof \rangle$

lemma *closed-empty[iff]*:

shows $\{\} \in downwards.closed$

$\langle proof \rangle$

lemma *clI[intro]*:

assumes $y \in P$

assumes $x \leq y$

shows $x \in downwards.cl\ P$

$\langle proof \rangle$

lemma *clE*:

assumes $x \in downwards.cl\ P$

obtains y **where** $y \in P$ **and** $x \leq y$

$\langle proof \rangle$

lemma *closed-in*:

assumes $x \in P$

assumes $y \leq x$
assumes $P \in \text{downwards.closed}$
shows $y \in P$
 ⟨proof⟩

lemma *order-embedding*: — On preorders; see Davey and Priestley (2002, §1.35)
fixes $x :: \text{preorder}$
shows $\text{downwards.cl } \{x\} \subseteq \text{downwards.cl } \{y\} \longleftrightarrow x \leq y$
 ⟨proof⟩

The lattice of downsets of a set X is always a *heyting-algebra*.

References:

- Ono (2019, §7.5); uses upsets, points to Stone (1938) as the origin
- Esakia et al. (2019, §2.2)
- https://en.wikipedia.org/wiki/Intuitionistic_logic#Heyting_algebra_semantics

definition $\text{imp} :: 'a::\text{preorder set} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$ **where**
 $\text{imp } P \ Q = \{\sigma. \forall \sigma' \leq \sigma. \sigma' \in P \longrightarrow \sigma' \in Q\}$

lemma *imp-refl*:
shows $\text{downwards.imp } P \ P = \text{UNIV}$
 ⟨proof⟩

lemma *imp-contained*:
assumes $P \subseteq Q$
shows $\text{downwards.imp } P \ Q = \text{UNIV}$
 ⟨proof⟩

lemma *heyting-imp*:
assumes $P \in \text{downwards.closed}$
shows $P \subseteq \text{downwards.imp } Q \ R \longleftrightarrow P \cap Q \subseteq R$
 ⟨proof⟩

lemma *imp-mp'*:
assumes $\sigma \in \text{downwards.imp } P \ Q$
assumes $\sigma \in P$
shows $\sigma \in Q$
 ⟨proof⟩

lemma *imp-mp*:
shows $P \cap \text{downwards.imp } P \ Q \subseteq Q$
and $\text{downwards.imp } P \ Q \cap P \subseteq Q$
 ⟨proof⟩

lemma *imp-contains*:
assumes $X \subseteq Q$
assumes $X \in \text{downwards.closed}$
shows $X \subseteq \text{downwards.imp } P \ Q$
 ⟨proof⟩

lemma *imp-downwards*:
assumes $y \in \text{downwards.imp } P \ Q$
assumes $x \leq y$
shows $x \in \text{downwards.imp } P \ Q$
 ⟨proof⟩

lemma *closed-imp*:

shows $\text{downwards.imp } P \ Q \in \text{downwards.closed}$

<proof>

The set $\text{downwards.imp } P \ Q$ is the greatest downset contained in the Boolean implication $P \longrightarrow_B Q$, i.e., downwards.imp is the *kernel* of (\longrightarrow_B) (Zwiers 1989). Note that “kernel” is a choice or interior function.

lemma *imp-boolean-implication-subseteq*:

shows $\text{downwards.imp } P \ Q \subseteq P \longrightarrow_B Q$

<proof>

lemma *downwards-closed-imp-greatest*:

assumes $R \subseteq P \longrightarrow_B Q$

assumes $R \in \text{downwards.closed}$

shows $R \subseteq \text{downwards.imp } P \ Q$

<proof>

definition *kernel* :: $'a::\text{order set} \Rightarrow 'a \text{ set}$ **where**

$\text{kernel } X = \bigsqcup \{Q \in \text{downwards.closed}. Q \subseteq X\}$

lemma *kernel-def2*:

shows $\text{downwards.kernel } X = \{\sigma. \forall \sigma' \leq \sigma. \sigma' \in X\}$ (**is** *?lhs = ?rhs*)

<proof>

lemma *kernel-contractive*:

shows $\text{downwards.kernel } X \subseteq X$

<proof>

lemma *kernel-idempotent*:

shows $\text{downwards.kernel } (\text{downwards.kernel } X) = \text{downwards.kernel } X$

<proof>

lemma *kernel-monotone*:

shows *mono* downwards.kernel

<proof>

lemma *closed-kernel-conv*:

shows $X \in \text{downwards.closed} \longleftrightarrow \text{downwards.kernel } X = X$

<proof>

lemma *closed-kernel*:

shows $\text{downwards.kernel } X \in \text{downwards.closed}$

<proof>

lemma *kernel-cl*:

shows $\text{downwards.kernel } (\text{downwards.cl } X) = \text{downwards.cl } X$

<proof>

lemma *cl-kernel*:

shows $\text{downwards.cl } (\text{downwards.kernel } X) = \text{downwards.kernel } X$

<proof>

lemma *kernel-boolean-implication*:

fixes $P :: \text{::order}$

shows $\text{downwards.kernel } (P \longrightarrow_B Q) = \text{downwards.imp } P \ Q$

<proof>

<ML>

8 Safety logic

Following Abadi and Lamport (1995); Abadi and Plotkin (1991, 1993) (see also Abadi and Merz (1996, §5.5)), we work in the complete lattice of stuttering-closed safety properties (i.e., stuttering-closed downsets) and use this for logical purposes. We avoid many syntactic issues via a shallow embedding into HOL.

8.1 Stuttering

We define *stuttering equivalence* ala Lamport (1994). This allows any agent to repeat any state at any time. We define a normalisation function (\natural) on $(\iota a, \iota s, \iota v)$ *trace.t* and extract the (matroidal) closure over sets of these from the Galois connection *galois.image-vimage*.

$\langle ML \rangle$

primrec *natural'* :: $\iota s \Rightarrow (\iota a \times \iota s)$ *list* $\Rightarrow (\iota a \times \iota s)$ *list* **where**

natural' s [] = []

| *natural'* s ($x \# xs$) = (if *snd* $x = s$ then *natural'* s xs else $x \# \textit{natural}' (snd\ x)\ xs$)

$\langle ML \rangle$

lemma *natural'[simp]*:

shows *trace.final'* s (*trace.natural'* s xs) = *trace.final'* s xs

$\langle proof \rangle$

lemma *natural'-cong*:

assumes $s = s'$

assumes *trace.natural'* s $xs = \textit{trace.natural}' s\ xs'$

shows *trace.final'* s $xs = \textit{trace.final}' s'\ xs'$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *natural'*:

shows *trace.natural'* s (*trace.natural'* s xs) = *trace.natural'* s xs

$\langle proof \rangle$

lemma *length*:

shows *length* (*trace.natural'* s xs) \leq *length* xs

$\langle proof \rangle$

lemma *subseq*:

shows *subseq* (*trace.natural'* s xs) xs

$\langle proof \rangle$

lemma *remdups-adj*:

shows $s \# \textit{map\ snd}\ (\textit{trace.natural}' s\ xs) = \textit{remdups-adj}\ (s \# \textit{map\ snd}\ xs)$

$\langle proof \rangle$

lemma *append*:

shows *trace.natural'* s ($xs @ ys$) = *trace.natural'* s $xs @ \textit{trace.natural}' (trace.final'\ s\ xs)\ ys$

$\langle proof \rangle$

lemma *eq-Nil-conv*:

shows *trace.natural'* s $xs = [] \iff \textit{snd}\ 'set\ xs \subseteq \{s\}$

and $[] = \textit{trace.natural}' s\ xs \iff \textit{snd}\ 'set\ xs \subseteq \{s\}$

$\langle proof \rangle$

lemma *eq-Cons-conv*:

shows $\text{trace.natural}' s xs = y \# ys$
 $\longleftrightarrow (\exists xs' ys'. xs = xs' @ y \# ys' \wedge \text{snd } ' \text{ set } xs' \subseteq \{s\} \wedge \text{snd } y \neq s \wedge \text{trace.natural}' (\text{snd } y) ys' = ys)$ (**is** ?lhs
 $\longleftrightarrow ?rhs$)
and $y \# ys = \text{trace.natural}' s xs$
 $\longleftrightarrow (\exists xs' ys'. xs = xs' @ y \# ys' \wedge \text{snd } ' \text{ set } xs' \subseteq \{s\} \wedge \text{snd } y \neq s \wedge \text{trace.natural}' (\text{snd } y) ys' = ys)$ (**is**
?thesis1)
⟨proof⟩

lemma *eq-append-conv*:

shows $\text{trace.natural}' s xs = ys @ zs$
 $\longleftrightarrow (\exists ys' zs'. xs = ys' @ zs' \wedge \text{trace.natural}' s ys' = ys \wedge \text{trace.natural}' (\text{trace.final}' s ys) zs' = zs)$ (**is** ?lhs
= ?rhs)
and $ys @ zs = \text{trace.natural}' s xs$
 $\longleftrightarrow (\exists ys' zs'. xs = ys' @ zs' \wedge \text{trace.natural}' s ys' = ys \wedge \text{trace.natural}' (\text{trace.final}' s ys) zs' = zs)$ (**is**
?thesis1)
⟨proof⟩

lemma *replicate*:

shows $\text{trace.natural}' s (\text{replicate } i \text{ as}) = (\text{if } \text{snd } \text{as} = s \vee i = 0 \text{ then } [] \text{ else } [\text{as}])$
⟨proof⟩

lemma *map-natural'*:

shows $\text{trace.natural}' (sf s) (\text{map } (\text{map-prod } af \text{ sf}) (\text{trace.natural}' s xs))$
 $= \text{trace.natural}' (sf s) (\text{map } (\text{map-prod } af \text{ sf}) xs)$
⟨proof⟩

lemma *map-inj-on-sf*:

assumes $\text{inj-on } sf (\text{insert } s (\text{snd } ' \text{ set } xs))$
shows $\text{trace.natural}' (sf s) (\text{map } (\text{map-prod } af \text{ sf}) xs) = \text{map } (\text{map-prod } af \text{ sf}) (\text{trace.natural}' s xs)$
⟨proof⟩

lemma *amap-noop*:

assumes $\text{trace.natural}' s xs = \text{map } (\text{map-prod } af \text{ id}) zs$
shows $\text{trace.natural}' s zs = zs$
⟨proof⟩

lemma *take*:

shows $\exists j \leq \text{length } xs. \text{take } i (\text{trace.natural}' s xs) = \text{trace.natural}' s (\text{take } j xs)$
⟨proof⟩

lemma *idle-prefix*:

assumes $\text{snd } ' \text{ set } xs \subseteq \{s\}$
shows $\text{trace.natural}' s (xs @ ys) = \text{trace.natural}' s ys$
⟨proof⟩

lemma *prefixE*:

assumes $\text{trace.natural}' s ys = \text{trace.natural}' s (xs @ xsrest)$
obtains $xs' xs'rest$ **where** $\text{trace.natural}' s xs = \text{trace.natural}' s xs'$ **and** $ys = xs' @ xs'rest$
⟨proof⟩

lemma *aset-conv*:

shows $a \in \text{trace.aset } (\text{trace.T } s (\text{trace.natural}' s xs) v)$
 $\longleftrightarrow (\exists s' s''. (a, s', s'') \in \text{set } (\text{trace.transitions}' s xs) \wedge s' \neq s'')$
⟨proof⟩

⟨ML⟩

definition $\text{natural} :: ('a, 's, 'v) \text{trace.t} \Rightarrow ('a, 's, 'v) \text{trace.t}$ (\natural) **where**

$\Downarrow \sigma = \text{trace}.T (\text{trace}.init \sigma) (\text{trace}.natural' (\text{trace}.init \sigma) (\text{trace}.rest \sigma)) (\text{trace}.term \sigma)$

$\langle ML \rangle$

lemma *sel[simp]*:

shows $\text{trace}.init (\Downarrow \sigma) = \text{trace}.init \sigma$
and $\text{trace}.rest (\Downarrow \sigma) = \text{trace}.natural' (\text{trace}.init \sigma) (\text{trace}.rest \sigma)$
and $\text{trace}.term (\Downarrow \sigma) = \text{trace}.term \sigma$

$\langle proof \rangle$

lemma *simps*:

shows $\Downarrow (\text{trace}.T s [] v) = \text{trace}.T s [] v$
and $\Downarrow (\text{trace}.T s ((a, s) \# xs) v) = \Downarrow (\text{trace}.T s xs v)$
and $\Downarrow (\text{trace}.T s (\text{trace}.natural' s xs) v) = \Downarrow (\text{trace}.T s xs v)$

$\langle proof \rangle$

lemma *idempotent[simp]*:

shows $\Downarrow (\Downarrow \sigma) = \Downarrow \sigma$

$\langle proof \rangle$

lemma *idle*:

assumes $snd \text{ ' set } xs \subseteq \{s\}$
shows $\Downarrow (\text{trace}.T s xs v) = \text{trace}.T s [] v$

$\langle proof \rangle$

lemma *trace-conv*:

shows $\Downarrow (\text{trace}.T s xs v) = \Downarrow \sigma \longleftrightarrow \text{trace}.init \sigma = s \wedge \text{trace}.natural' s xs = \text{trace}.natural' s (\text{trace}.rest \sigma) \wedge \text{trace}.term \sigma = v$
and $\Downarrow \sigma = \Downarrow (\text{trace}.T s xs v) \longleftrightarrow \text{trace}.init \sigma = s \wedge \text{trace}.natural' s xs = \text{trace}.natural' s (\text{trace}.rest \sigma) \wedge \text{trace}.term \sigma = v$

$\langle proof \rangle$

lemma *map-natural*:

shows $\Downarrow (\text{trace}.map \text{ af } sf \text{ vf } (\Downarrow \sigma)) = \Downarrow (\text{trace}.map \text{ af } sf \text{ vf } \sigma)$

$\langle proof \rangle$

lemma *continue*:

shows $\Downarrow (\sigma @_{-S} xsv) = \Downarrow \sigma @_{-S} (\text{trace}.natural' (\text{trace}.final \sigma) (\text{fst } xsv), \text{snd } xsv)$

$\langle proof \rangle$

lemma *replicate*:

shows $\Downarrow (\text{trace}.T s (\text{replicate } i \text{ as}) v)$
 $= (\text{trace}.T s (\text{if } snd \text{ as} = s \vee i = 0 \text{ then } [] \text{ else } [\text{as}]) v)$

$\langle proof \rangle$

lemma *monotone*:

shows $\text{mono } \Downarrow$

$\langle proof \rangle$

lemmas *strengthen[strg] = st-monotone[OF trace.natural.monotone]*

lemmas *mono = monotoneD[OF trace.natural.monotone]*

lemmas *mono2mono[cont-intro, partial-function-mono]*

$= \text{monotone2monotone}[OF \text{ trace}.natural.monotone, \text{simplified}, \text{of orda } P \text{ for orda } P]$

lemma *less-eqE*:

assumes $t \leq u$

assumes $\Downarrow u' = \Downarrow u$

obtains t' **where** $\Downarrow t = \Downarrow t'$ **and** $t' \leq u'$

$\langle proof \rangle$

lemma *less-eq-natural*:

assumes $\sigma_1 \leq \natural\sigma_2$

shows $\natural\sigma_1 = \sigma_1$

$\langle proof \rangle$

lemma *map-le*:

assumes $\natural\sigma_1 \leq \natural\sigma_2$

shows $\natural(\text{trace.map af sf vf } \sigma_1) \leq \natural(\text{trace.map af sf vf } \sigma_2)$

$\langle proof \rangle$

lemma *map-inj-on-sf*:

assumes *inj-on sf* ($\text{trace.sset } \sigma$)

shows $\natural(\text{trace.map af sf vf } \sigma) = \text{trace.map af sf vf } (\natural\sigma)$

$\langle proof \rangle$

lemma *take*:

shows $\exists j. \natural(\text{trace.take } i \sigma) = \text{trace.take } j (\natural\sigma)$

$\langle proof \rangle$

lemma *take-natural*:

shows $\natural(\text{trace.take } i (\natural\sigma)) = \text{trace.take } i (\natural\sigma)$

$\langle proof \rangle$

lemma *takeE*:

shows $\llbracket \sigma_1 = \natural(\text{trace.take } i \sigma_2); \bigwedge j. \llbracket \sigma_1 = \text{trace.take } j (\natural\sigma_2) \rrbracket \implies \text{thesis} \rrbracket \implies \text{thesis}$

and $\llbracket \natural(\text{trace.take } i \sigma_2) = \sigma_1; \bigwedge j. \llbracket \sigma_1 = \text{trace.take } j (\natural\sigma_2) \rrbracket \implies \text{thesis} \rrbracket \implies \text{thesis}$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *natural-conv*:

shows $a \in \text{trace.aset } (\natural\sigma) \iff (\exists s s'. (a, s, s') \in \text{trace.steps } \sigma)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *natural'[simp]*:

shows $\text{trace.sset } (\text{trace.T } s_0 (\text{trace.natural}' s_0 xs) v) = \text{trace.sset } (\text{trace.T } s_0 xs v)$

$\langle proof \rangle$

lemma *natural[simp]*:

shows $\text{trace.sset } (\natural\sigma) = \text{trace.sset } \sigma$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *natural[simp]*:

shows $\text{trace.vset } (\natural\sigma) = \text{trace.vset } \sigma$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *natural*:

shows $\exists j \leq \text{Suc } (\text{length } (\text{trace.rest } \sigma)). \text{trace.take } i (\natural\sigma) = \natural(\text{trace.take } j \sigma)$

$\langle proof \rangle$

lemma *naturalE*:

shows $\llbracket \sigma_1 = \text{trace.take } i \ (\natural\sigma_2); \bigwedge j. \llbracket j \leq \text{Suc } (\text{length } (\text{trace.rest } \sigma_2)); \sigma_1 = \natural(\text{trace.take } j \ \sigma_2) \rrbracket \implies \text{thesis} \rrbracket \implies \text{thesis}$

and $\llbracket \text{trace.take } i \ (\natural\sigma_2) = \sigma_1; \bigwedge j. \llbracket j \leq \text{Suc } (\text{length } (\text{trace.rest } \sigma_2)); \natural(\text{trace.take } j \ \sigma_2) = \sigma_1 \rrbracket \implies \text{thesis} \rrbracket \implies \text{thesis}$

<proof>

<ML>

lemma *steps'-alt-def*:

shows $\text{trace.steps}' s \ xs = \text{set } (\text{trace.transitions}' s \ (\text{trace.natural}' s \ xs))$

<proof>

<ML>

lemma *natural'*:

shows $\text{trace.steps}' s \ (\text{trace.natural}' s \ xs) = \text{trace.steps}' s \ xs$

<proof>

lemma *asetD*:

assumes $\text{trace.steps } \sigma \subseteq r$

shows $\forall a. a \in \text{trace.aset } (\natural\sigma) \longrightarrow a \in \text{fst } ' r$

<proof>

lemma *range-initE*:

assumes $\text{trace.steps}' s_0 \ xs \subseteq \text{range } af \times \text{range } sf \times \text{range } sf$

assumes $(a, s, s') \in \text{trace.steps}' s_0 \ xs$

obtains s_0' **where** $s_0 = sf \ s_0'$

<proof>

lemma *map-range-conv*:

shows $\text{trace.steps}' (sf \ s) \ xs \subseteq \text{range } af \times \text{range } sf \times \text{range } sf$

$\longleftrightarrow (\exists xs'. \text{trace.natural}' (sf \ s) \ xs = \text{map } (\text{map-prod } af \ sf) \ xs') \ (\text{is } ?lhs \longleftrightarrow ?rhs)$

<proof>

lemma *step-conv*:

shows $\text{trace.steps}' s \ xs = \{x\}$

$\longleftrightarrow \text{fst } (\text{snd } x) = s \wedge \text{fst } (\text{snd } x) \neq \text{snd } (\text{snd } x)$

$\wedge (\exists ys \ zs. \text{snd } ' \text{set } ys \subseteq \{s\} \wedge \text{snd } ' \text{set } zs \subseteq \{\text{snd } (\text{snd } x)\})$

$\wedge xs = ys \ @ \ [(\text{fst } x, \text{snd } (\text{snd } x))] \ @ \ zs \ (\text{is } ?lhs \longleftrightarrow ?rhs)$

<proof>

<ML>

interpretation *stuttering*: *galois.image-vimage-idempotent* \natural

<proof>

abbreviation *stuttering-equiv-syn* :: $('a, 's, 'v) \text{ trace.t} \Rightarrow ('a, 's, 'v) \text{ trace.t} \Rightarrow \text{bool}$ (**infix** \simeq_S 50) **where**

$\sigma_1 \simeq_S \sigma_2 \equiv \text{trace.stuttering.equivalent } \sigma_1 \ \sigma_2$

<ML>

lemma *cl*:

shows $\text{trace.stuttering.cl } (\text{downwards.cl } P) = \text{downwards.cl } (\text{trace.stuttering.cl } P) \ (\text{is } ?lhs = ?rhs)$

<proof>

lemma *closed*:

assumes $P \in \text{downwards.closed}$

shows $\text{trace.stuttering.cl } P \in \text{downwards.closed}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *downwards-imp*: — Abadi and Plotkin (1993, p13)

assumes $P \in \text{trace.stuttering.closed}$

assumes $Q \in \text{trace.stuttering.closed}$

shows $\text{downwards.imp } P Q \in \text{trace.stuttering.closed}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *simps*:

shows $\text{snd ' set } xs \subseteq \{s\} \implies \text{trace.T } s (xs @ ys) v \simeq_S \text{trace.T } s ys v$

and $\text{snd ' set } ys \subseteq \{\text{trace.final' } s xs\} \implies \text{trace.T } s (xs @ ys) v \simeq_S \text{trace.T } s xs v$

and $\text{snd ' set } xs \subseteq \{\text{snd } x\} \implies \text{trace.T } s (x \# xs @ ys) v \simeq_S \text{trace.T } s (x \# ys) v$

$\langle \text{proof} \rangle$

lemma *append-cong*:

assumes $s = s'$

assumes $\text{trace.natural' } s xs = \text{trace.natural' } s xs'$

assumes $\text{trace.natural' } (\text{trace.final' } s xs) ys = \text{trace.natural' } (\text{trace.final' } s xs) ys'$

assumes $v = v'$

shows $\text{trace.T } s (xs @ ys) v \simeq_S \text{trace.T } s' (xs' @ ys') v'$

$\langle \text{proof} \rangle$

lemma *E*:

assumes $\text{trace.T } s xs v \simeq_S \text{trace.T } s' xs' v'$

obtains $\text{trace.natural' } s xs = \text{trace.natural' } s' xs'$ **and** $s = s'$ **and** $v = v'$

$\langle \text{proof} \rangle$

lemma *append-conv*:

shows $\text{trace.T } s (xs @ ys) v \simeq_S \sigma$

$\iff (\exists xs' ys'. \sigma = \text{trace.T } s (xs' @ ys') v \wedge \text{trace.natural' } s xs = \text{trace.natural' } s xs'$
 $\wedge \text{trace.natural' } (\text{trace.final' } s xs) ys = \text{trace.natural' } (\text{trace.final' } s xs) ys') \text{ (is ?thesis1)}$

and $\sigma \simeq_S \text{trace.T } s (xs @ ys) v$

$\iff (\exists xs' ys'. \sigma = \text{trace.T } s (xs' @ ys') v \wedge \text{trace.natural' } s xs = \text{trace.natural' } s xs'$
 $\wedge \text{trace.natural' } (\text{trace.final' } s xs) ys = \text{trace.natural' } (\text{trace.final' } s xs) ys') \text{ (is ?thesis2)}$

$\langle \text{proof} \rangle$

lemma *map*:

assumes $\sigma_1 \simeq_S \sigma_2$

shows $\text{trace.map af sf vf } \sigma_1 \simeq_S \text{trace.map af sf vf } \sigma_2$

$\langle \text{proof} \rangle$

lemma *steps*:

assumes $\sigma_1 \simeq_S \sigma_2$

shows $\text{trace.steps } \sigma_1 = \text{trace.steps } \sigma_2$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

8.2 The ('a, 's, 'v) spec lattice

Our workhorse lattice consists of all sets of traces that are downwards and stuttering closed. This combined closure is neither matroidal nor antimatroidal (§5.3).

We define the lattice as a type and instantiate the relevant type classes. In the following read $P \leq Q$ ($P \subseteq Q$ in

the powerset model) as “Q follows from P” or “P entails Q”.

$\langle ML \rangle$

definition $cl :: ('a, 's, 'v) \text{ trace.t set} \Rightarrow ('a, 's, 'v) \text{ trace.t set}$ **where**
 $cl P = \text{downwards.cl } (\text{trace.stuttering.cl } P)$

$\langle ML \rangle$

interpretation $\text{spec: closure-powerset-distributive raw.spec.cl}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma empty[simp] :
shows $\text{raw.spec.cl } \{\} = \{\}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma I :

assumes $P \in \text{downwards.closed}$
assumes $P \in \text{trace.stuttering.closed}$
shows $P \in \text{raw.spec.closed}$

$\langle \text{proof} \rangle$

lemma empty[intro] :

shows $\{\} \in \text{raw.spec.closed}$

$\langle \text{proof} \rangle$

lemma downwards-closed :

assumes $P \in \text{raw.spec.closed}$
shows $P \in \text{downwards.closed}$

$\langle \text{proof} \rangle$

lemma stuttering-closed :

assumes $P \in \text{raw.spec.closed}$
shows $P \in \text{trace.stuttering.closed}$

$\langle \text{proof} \rangle$

lemma downwards-imp :

assumes $P \in \text{raw.spec.closed}$
assumes $Q \in \text{raw.spec.closed}$
shows $\text{downwards.imp } P Q \in \text{raw.spec.closed}$

$\langle \text{proof} \rangle$

lemma $\text{heyting-downwards-imp}$:

assumes $P \in \text{raw.spec.closed}$
shows $P \subseteq \text{downwards.imp } Q R \iff P \cap Q \subseteq R$

$\langle \text{proof} \rangle$

lemma takeE :

assumes $\sigma \in P$
assumes $P \in \text{raw.spec.closed}$
shows $\text{trace.take } i \sigma \in P$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

```

typedef ('a, 's, 'v) spec = raw.spec.closed :: ('a, 's, 'v) trace.t set set
morphisms unMkS MkS
⟨proof⟩

```

```

setup-lifting type-definition-spec

```

```

instantiation spec :: (type, type, type) complete-distrib-lattice
begin

```

```

lift-definition bot-spec :: ('a, 's, 'v) spec is empty ⟨proof⟩
lift-definition top-spec :: ('a, 's, 'v) spec is UNIV ⟨proof⟩
lift-definition sup-spec :: ('a, 's, 'v) spec ⇒ ('a, 's, 'v) spec ⇒ ('a, 's, 'v) spec is sup ⟨proof⟩
lift-definition inf-spec :: ('a, 's, 'v) spec ⇒ ('a, 's, 'v) spec ⇒ ('a, 's, 'v) spec is inf ⟨proof⟩
lift-definition less-eq-spec :: ('a, 's, 'v) spec ⇒ ('a, 's, 'v) spec ⇒ bool is less-eq ⟨proof⟩
lift-definition less-spec :: ('a, 's, 'v) spec ⇒ ('a, 's, 'v) spec ⇒ bool is less ⟨proof⟩
lift-definition Inf-spec :: ('a, 's, 'v) spec set ⇒ ('a, 's, 'v) spec is Inf ⟨proof⟩
lift-definition Sup-spec :: ('a, 's, 'v) spec set ⇒ ('a, 's, 'v) spec is λX. Sup X ⊔ raw.spec.cl {} ⟨proof⟩

```

```

instance
⟨proof⟩

```

```

end

```

```

declare

```

```

  SUPE[where 'a=(('a, 's, 'v) spec, intro!)]
  SupE[where 'a=(('a, 's, 'v) spec, intro!)]
  Sup-le-iff[where 'a=(('a, 's, 'v) spec, simp)]
  SupI[where 'a=(('a, 's, 'v) spec, intro)]
  SUPI[where 'a=(('a, 's, 'v) spec, intro)]
  rev-SUPI[where 'a=(('a, 's, 'v) spec, intro?)]
  INFE[where 'a=(('a, 's, 'v) spec, intro)]

```

Observations about this type:

- it is not a BNF (datatype) as it uses the powerset
- it fails to be T0 or sober due to the lack of limit points (completeness) in ('a, 's, 'v) trace.t
 - also stuttering closure precludes T0
- the complete-distrib-lattice instance shows that arbitrary/infinitary Sups and Infs distribute
 - in other words: safety properties are closed under arbitrary intersections and unions
 - in other words: Alexandrov
- conclude: the lack of limit points makes this model easier to work in and adds expressivity
 - see §24 for further discussion

⟨ML⟩

```

lemmas antisym = antisym[where 'a=(('a, 's, 'v) spec)]
lemmas eq-iff = order.eq-iff[where 'a=(('a, 's, 'v) spec)]

```

⟨ML⟩

8.3 Irreducible elements

The irreducible elements of $(\prime a, \prime s, \prime v)$ *trace.t* are the closures of singletons.

$\langle ML \rangle$

definition *singleton* $:: (\prime a, \prime s, \prime v)$ *trace.t* $\Rightarrow (\prime a, \prime s, \prime v)$ *trace.t set* **where**
singleton $\sigma = \text{raw.spec.cl } \{\sigma\}$

lemma *singleton-le-conv*:

shows $\text{raw.singleton } \sigma_1 \leq \text{raw.singleton } \sigma_2 \iff \Downarrow \sigma_1 \leq \Downarrow \sigma_2$ (**is** *?lhs* \iff *?rhs*)
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lift-definition *singleton* $:: (\prime a, \prime s, \prime v)$ *trace.t* $\Rightarrow (\prime a, \prime s, \prime v)$ *spec* $(\Downarrow -)$ **is** *raw.singleton*
 $\langle \text{proof} \rangle$

abbreviation *singleton-trace-syn* $:: \prime s \Rightarrow (\prime a \times \prime s)$ *list* $\Rightarrow \prime v$ *option* $\Rightarrow (\prime a, \prime s, \prime v)$ *spec* $(\Downarrow -, -, -)$ **where**
 $\langle s, xs, v \rangle \equiv \langle \text{trace.T } s \ xs \ v \rangle$

$\langle ML \rangle$

lemma *Sup-prime*:

shows *Sup-prime* $\langle \sigma \rangle$
 $\langle \text{proof} \rangle$

lemma *nchotomy*:

shows $\exists X \in \text{raw.spec.closed. } x = \bigsqcup (\text{spec.singleton } \prime X)$
 $\langle \text{proof} \rangle$

lemmas *exhaust* = *bexE[OF spec.singleton.nchotomy]*

lemma *collapse[simp]*:

shows $\bigsqcup (\text{spec.singleton } \prime \{\sigma. \langle \sigma \rangle \leq P\}) = P$
 $\langle \text{proof} \rangle$

lemmas *not-bot* = *Sup-prime-not-bot[OF spec.singleton.Sup-prime]* — Non-triviality

$\langle ML \rangle$

lemma *singleton-le-ext-conv*:

shows $P \leq Q \iff (\forall \sigma. \langle \sigma \rangle \leq P \longrightarrow \langle \sigma \rangle \leq Q)$ (**is** *?lhs* \iff *?rhs*)
 $\langle \text{proof} \rangle$

lemmas *singleton-le-conv* = *raw.singleton-le-conv[transferred]*

lemmas *singleton-le-extI* = *iffD2[OF spec.singleton-le-ext-conv, rule-format]*

lemma *singleton-eq-conv[simp]*:

shows $\langle \sigma \rangle = \langle \sigma' \rangle \iff \sigma \simeq_S \sigma'$
 $\langle \text{proof} \rangle$

lemma *singleton-cong*:

assumes $\sigma \simeq_S \sigma'$
shows $\langle \sigma \rangle = \langle \sigma' \rangle$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

named-theorems *le-conv* < *simplification rules for* $\langle \sigma \rangle \leq \text{const} \dots$ >

lemmas *antisym* = *antisym*[*OF spec.singleton-le-extI spec.singleton-le-extI*]

lemmas *top* = *spec.singleton.collapse*[*of* \top , *simplified, symmetric*]

lemma *monotone*:

shows *mono spec.singleton*

<*proof*>

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.singleton.monotone*]

lemmas *mono* = *monoD*[*OF spec.singleton.monotone*]

lemmas *mono2mono*[*cont-intro, partial-function-mono*]
= *monotone2monotone*[*OF spec.singleton.monotone, simplified*]

lemma *simps*[*simp*]:

shows $\langle \natural\sigma \rangle = \langle \sigma \rangle$

and $\langle s, xs, v \rangle \leq \langle s, \text{trace.natural}' s xs, v \rangle$

and *snd* ' *set* $xs \subseteq \{s\} \implies \langle s, xs @ ys, v \rangle = \langle s, ys, v \rangle$

and *snd* ' *set* $ys \subseteq \{\text{trace.final}' s xs\} \implies \langle s, xs @ ys, v \rangle = \langle s, xs, v \rangle$

and *snd* ' *set* $xs \subseteq \{\text{snd } x\} \implies \langle s, x \# xs @ ys, v \rangle = \langle s, x \# ys, v \rangle$

and $\langle s, (a, s) \# xs, v \rangle = \langle s, xs, v \rangle$

<*proof*>

lemma *Cons*: — self-applies, not usable by *simp*

assumes *snd* ' *set* $as \subseteq \{s'\}$

shows $\langle s, (a, s') \# as, v \rangle = \langle s, [(a, s')], v \rangle$

<*proof*>

lemmas *Sup-irreducible* = *iffD1*[*OF heyting.Sup-prime-Sup-irreducible-iff spec.singleton.Sup-prime*]

lemmas *sup-irreducible* = *Sup-irreducible-on-imp-sup-irreducible-on*[*OF spec.singleton.Sup-irreducible, simplified*]

lemmas *Sup-leE*[*elim*] = *Sup-prime-onE*[*OF spec.singleton.Sup-prime, simplified*]

lemmas *sup-le-conv*[*simp*] = *sup-irreducible-le-conv*[*OF spec.singleton.sup-irreducible*]

lemmas *Sup-le-conv*[*simp*] = *Sup-prime-on-conv*[*OF spec.singleton.Sup-prime, simplified*]

lemmas *compact-point* = *Sup-prime-is-compact*[*OF spec.singleton.Sup-prime*]

lemmas *compact*[*cont-intro*] = *compact-points-are-ccpo-compact*[*OF spec.singleton.compact-point*]

lemma *Inf*:

shows $\bigcap (\text{spec.singleton } ' X) = \bigsqcup (\text{spec.singleton } ' \{\sigma. \forall \sigma_1 \in X. \sigma \leq \natural\sigma_1\})$

<*proof*>

lemmas *inf* = *spec.singleton.Inf*[**where** $X = \{\sigma_1, \sigma_2\}$, *simplified*] **for** $\sigma_1 \sigma_2$

lemma *less-eq-Some*[*simp*]:

shows $\langle s, xs, \text{Some } v \rangle \leq \langle \sigma \rangle$

$\longleftrightarrow \text{trace.term } \sigma = \text{Some } v \wedge \text{trace.init } \sigma = s \wedge \text{trace.natural}' s (\text{trace.rest } \sigma) = \text{trace.natural}' s xs$

<*proof*>

lemma *less-eq-None*:

shows [*iff*]: $\langle s, xs, \text{None} \rangle \leq \langle s, xs, v' \rangle$

<*proof*>

lemma *map-cong*:

assumes $\bigwedge a. a \in \text{trace.aset } (\natural\sigma') \implies af a = af' a$

assumes $\bigwedge x. x \in \text{trace.sset } (\natural\sigma') \implies sf x = sf' x$

assumes $\bigwedge v. v \in \text{trace.vset } (\natural\sigma') \implies vf v = vf' v$

assumes $\natural\sigma = \natural\sigma'$

shows $\langle \text{trace.map } af sf vf \sigma \rangle = \langle \text{trace.map } af' sf' vf' \sigma' \rangle$

$\langle \text{proof} \rangle$

lemma *map-le*:

assumes $\langle \sigma \rangle \leq \langle \sigma' \rangle$

shows $\langle \text{trace.map af sf vf } \sigma \rangle \leq \langle \text{trace.map af sf vf } \sigma' \rangle$

$\langle \text{proof} \rangle$

lemma *takeI*:

assumes $\langle \sigma \rangle \leq P$

shows $\langle \text{trace.take } i \sigma \rangle \leq P$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemmas *assms-cong* = *order.assms-cong*[**where** $'a=('a, 's, 'v) \text{ spec}$]

lemmas *concl-cong* = *order.concl-cong*[**where** $'a=('a, 's, 'v) \text{ spec}$]

declare *spec.singleton.transfer*[*transfer-rule del*]

$\langle \text{ML} \rangle$

8.4 Maps

Lift *trace.map* to the $('a, 's, 'v) \text{ spec}$ lattice via image and inverse image.

Note that the image may yield a set that is not stuttering closed (i.e., we need to close the obvious model-level definition of *spec.map* under stuttering) as arbitrary *sf* may introduce stuttering not present in *P*. In contrast the inverse image preserves stuttering. These issues are elided here through the use of *spec.singleton*.

$\langle \text{ML} \rangle$

definition *map* :: $('a \Rightarrow 'b) \Rightarrow ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('b, 't, 'w) \text{ spec}$ **where**
 $\text{map af sf vf } P = \bigsqcup (\text{spec.singleton } ' \text{ trace.map af sf vf } ' \{ \sigma. \langle \sigma \rangle \leq P \})$

definition *invmap* :: $('a \Rightarrow 'b) \Rightarrow ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('b, 't, 'w) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**
 $\text{invmap af sf vf } P = \bigsqcup (\text{spec.singleton } ' \text{ trace.map af sf vf } -' \{ \sigma. \langle \sigma \rangle \leq P \})$

abbreviation *amap* :: $('a \Rightarrow 'b) \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('b, 's, 'v) \text{ spec}$ **where**

$\text{amap af} \equiv \text{spec.map af id id}$

abbreviation *ainvmap* :: $('a \Rightarrow 'b) \Rightarrow ('b, 's, 'v) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$\text{ainvmap af} \equiv \text{spec.invmap af id id}$

abbreviation *smap* :: $('s \Rightarrow 't) \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 't, 'v) \text{ spec}$ **where**

$\text{smap sf} \equiv \text{spec.map id sf id}$

abbreviation *sinvmap* :: $('s \Rightarrow 't) \Rightarrow ('a, 't, 'v) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$\text{sinvmap sf} \equiv \text{spec.invmap id sf id}$

abbreviation *vmap* :: $('v \Rightarrow 'w) \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 's, 'w) \text{ spec}$ **where** — aka *liftM*

$\text{vmap vf} \equiv \text{spec.map id id vf}$

abbreviation *vinvmap* :: $('v \Rightarrow 'w) \Rightarrow ('a, 's, 'w) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$\text{vinvmap vf} \equiv \text{spec.invmap id id vf}$

interpretation *map-invmap*: *galois.complete-lattice-distributive-class*

spec.map af sf vf

spec.invmap af sf vf **for** *af sf vf*

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *map-le-conv*[*spec.singleton.le-conv*]:

shows $\langle \sigma \rangle \leq \text{spec.map af sf vf } P \iff (\exists \sigma'. \langle \sigma' \rangle \leq P \wedge \langle \sigma \rangle \leq \langle \text{trace.map af sf vf } \sigma' \rangle)$

⟨proof⟩

lemma *invmap-le-conv*[*spec.singleton.le-conv*]:

shows $\langle \sigma \rangle \leq \text{spec.invmap af sf vf } P \longleftrightarrow \langle \text{trace.map af sf vf } \sigma \rangle \leq P$

⟨proof⟩

⟨ML⟩

lemmas *bot* = *spec.map-invmap.lower-bot*

lemmas *monotone* = *spec.map-invmap.monotone-lower*

lemmas *mono* = *monotoneD[OF spec.map.monotone]*

lemmas *Sup* = *spec.map-invmap.lower-Sup*

lemmas *sup* = *spec.map-invmap.lower-sup*

lemmas *Inf-le* = *spec.map-invmap.lower-Inf-le* — Converse does not hold

lemmas *inf-le* = *spec.map-invmap.lower-inf-le* — Converse does not hold

lemmas *invmap-le* = *spec.map-invmap.lower-upper-contractive*

lemma *singleton*:

shows *spec.map af sf vf* $\langle \sigma \rangle = \langle \text{trace.map af sf vf } \sigma \rangle$

⟨proof⟩

lemma *top*:

assumes *surj af*

assumes *surj sf*

assumes *surj vf*

shows *spec.map af sf vf* $\top = \top$

⟨proof⟩

lemma *id*:

shows *spec.map id id id* $P = P$

and *spec.map* $(\lambda x. x) (\lambda x. x) (\lambda x. x) P = P$

⟨proof⟩

lemma *comp*:

shows *spec.map af sf vf* \circ *spec.map ag sg vg* = *spec.map* $(af \circ ag) (sf \circ sg) (vf \circ vg)$ (**is** *?lhs = ?rhs*)

and *spec.map af sf vf* $(\text{spec.map ag sg vg } P) = \text{spec.map } (\lambda a. af (ag a)) (\lambda s. sf (sg s)) (\lambda v. vf (vg v)) P$ (**is** *?thesis1*)

⟨proof⟩

lemmas *map* = *spec.map.comp*

lemma *inf-distr*:

shows *spec.map af sf vf* $P \sqcap Q = \text{spec.map af sf vf } (P \sqcap \text{spec.invmap af sf vf } Q)$ (**is** *?lhs = ?rhs*)

and $Q \sqcap \text{spec.map af sf vf } P = \text{spec.map af sf vf } (\text{spec.invmap af sf vf } Q \sqcap P)$ (**is** *?thesis1*)

⟨proof⟩

⟨ML⟩

lemma *comp*:

shows *spec.smap sf* \circ *spec.smap sg* = *spec.smap* $(sf \circ sg)$

and *spec.smap sf* $(\text{spec.smap sg } P) = \text{spec.smap } (\lambda s. sf (sg s)) P$

⟨proof⟩

⟨ML⟩

lemmas *bot* = *spec.map-invmap.upper-bot*

lemmas *top* = *spec.map-invmap.upper-top*

lemmas *monotone* = *spec.map-invmap.monotone-upper*

lemmas *mono* = *monotoneD[OF spec.invmap.monotone]*

lemmas *Sup* = *spec.map-invmap.upper-Sup*

lemmas *sup* = *spec.map-invmap.upper-sup*

lemmas *Inf* = *spec.map-invmap.upper-Inf*

lemmas *inf* = *spec.map-invmap.upper-inf*

lemma *singleton*:

shows *spec.invmap af sf vf* $\langle\sigma\rangle = \bigsqcup (\text{spec.singleton } \{ \sigma'. \langle\text{trace.map af sf vf } \sigma'\rangle \leq \langle\sigma\rangle \})$
<proof>

lemma *id*:

shows *spec.invmap id id id* $P = P$
and *spec.invmap* $(\lambda x. x) (\lambda x. x) (\lambda x. x) P = P$
<proof>

lemma *comp*:

shows *spec.invmap af sf vf* (*spec.invmap ag sg vg* P) = *spec.invmap* $(\lambda x. ag (af x)) (\lambda s. sg (sf s)) (\lambda v. vg (vf v)) P$ (**is** *?lhs* $P = ?rhs P$)
and *spec.invmap af sf vf* \circ *spec.invmap ag sg vg* = *spec.invmap* $(ag \circ af) (sg \circ sf) (vg \circ vf)$ (**is** *?thesis1*)
<proof>

lemmas *invmap* = *spec.invmap.comp*

lemma *invmap-inf-distr-le*:

fixes *af* :: $'a \Rightarrow 'b$
fixes *sf* :: $'s \Rightarrow 't$
fixes *vf* :: $'v \Rightarrow 'w$
shows *spec.invmap af sf vf* $P \sqcap Q \leq \text{spec.invmap af sf vf } (P \sqcap \text{spec.map af sf vf } Q)$
and $Q \sqcap \text{spec.invmap af sf vf } P \leq \text{spec.invmap af sf vf } (\text{spec.map af sf vf } Q \sqcap P)$
<proof>

<ML>

lemma *invmap-le*: — *af = id* in *spec.invmap*

shows *spec.amap af* (*spec.invmap id sf vf* P) $\leq \text{spec.invmap id sf vf } (\text{spec.amap af } P)$
<proof>

lemma *surj-invmap*: — *af = id* in *spec.invmap*

fixes $P :: ('a, 't, 'w) \text{ spec}$
fixes *af* :: $'a \Rightarrow 'b$
fixes *sf* :: $'s \Rightarrow 't$
fixes *vf* :: $'v \Rightarrow 'w$
assumes *surj af*
shows *spec.amap af* (*spec.invmap id sf vf* P) = *spec.invmap id sf vf* (*spec.amap af* P) (**is** *?lhs = ?rhs*)
<proof>

<ML>

8.5 The idle process

As observed by [Abadi and Plotkin \(1991\)](#), many laws require the processes involved to accept all initial states (see, for instance, §8.8). We call the minimal such process *spec.idle*. It is also the lower bound on specification by transition relation (§8.10).

⟨ML⟩

definition *idle* :: ('a, 's, 'v) spec **where**
idle = (⊔ s. ⟨s, [], None⟩)

named-theorems *idle-le* < rules for <spec.idle ≤ const ...> >

⟨ML⟩

lemma *idle-le-conv*[*spec.singleton.le-conv*]:
shows ⟨σ⟩ ≤ *spec.idle* ↔ *trace.steps* σ = {} ∧ *trace.term* σ = None
 ⟨proof⟩

⟨ML⟩

lemma *minimal-le*:
shows ⟨s, [], None⟩ ≤ *spec.idle*
 ⟨proof⟩

lemma *map-le*[*spec.idle-le*]:
assumes *spec.idle* ≤ P
assumes *surj sf*
shows *spec.idle* ≤ *spec.map af sf vf P*
 ⟨proof⟩

lemma *invmap-le*:
assumes *spec.idle* ≤ P
shows *spec.idle* ≤ *spec.invmap af sf vf P*
 ⟨proof⟩

⟨ML⟩

lemma *cl-alt-def*:
shows *spec.map-invmap.cl - - - af sf vf P*
 = ⊔ {⟨σ⟩ | σ σ'. ⟨σ'⟩ ≤ P ∧ ⟨*trace.map af sf vf* σ⟩ ≤ ⟨*trace.map af sf vf* σ'⟩} (**is** ?lhs = ?rhs)
 ⟨proof⟩

lemma *cl-le-conv*[*spec.singleton.le-conv*]:
shows ⟨σ⟩ ≤ *spec.map-invmap.cl - - - af sf vf P* ↔ ⟨*trace.map af sf vf* σ⟩ ≤ *spec.map af sf vf P*
 ⟨proof⟩

⟨ML⟩

8.6 Actions

Our primitive actions are arbitrary relations on the state, labelled by the agent performing the state transition and a value to return.

For refinement purposes we need *idle* ≤ *action a F*; see §12.1.1.

⟨ML⟩

definition *action* :: ('v × 'a × 's × 's) set ⇒ ('a, 's, 'v) spec **where**
action F = (⊔ (v, a, s, s') ∈ F. ⟨s, [(a, s')], Some v⟩) ⊔ *spec.idle*

definition $guard :: ('s \Rightarrow bool) \Rightarrow ('a, 's, unit) \text{ spec where}$
 $guard\ g = \text{spec.action } (\{\} \times UNIV \times \text{Diag } g)$

definition $return :: 'v \Rightarrow ('a, 's, 'v) \text{ spec where}$
 $return\ v = \text{spec.action } (\{v\} \times UNIV \times Id)$

abbreviation $(input) \text{ read} :: ('s \Rightarrow 'v \text{ option}) \Rightarrow ('a, 's, 'v) \text{ spec where}$
 $read\ f \equiv \text{spec.action } \{(v, a, s, s) \mid a\ s\ v.\ f\ s = \text{Some } v\}$

abbreviation $(input) \text{ write} :: 'a \Rightarrow ('s \Rightarrow 's) \Rightarrow ('a, 's, unit) \text{ spec where}$
 $write\ a\ f \equiv \text{spec.action } \{((, a, s, f\ s) \mid s.\ \text{True})\}$

lemma $action-le[case-names\ idle\ step]:$

assumes $\text{spec.idle} \leq P$

assumes $\bigwedge v\ a\ s\ s'. (v, a, s, s') \in F \implies \langle s, [(a, s'), \text{Some } v] \rangle \leq P$

shows $\text{spec.action } F \leq P$

$\langle proof \rangle$

$\langle ML \rangle$

lemma $action-le[\text{spec.idle-le}]:$

shows $\text{spec.idle} \leq \text{spec.action } F$

$\langle proof \rangle$

lemma $guard-le[\text{spec.idle-le}]:$

shows $\text{spec.idle} \leq \text{spec.guard } g$

$\langle proof \rangle$

lemma $return-le[\text{spec.idle-le}]:$

shows $\text{spec.idle} \leq \text{spec.return } v$

$\langle proof \rangle$

$\langle ML \rangle$

lemma $action-le:$

fixes $F :: ('v \times 'a \times 's \times 's) \text{ set}$

shows $\text{spec.map } af\ sf\ vf\ (\text{spec.action } F) \leq \text{spec.action } (\text{map-prod } vf\ (\text{map-prod } af\ (\text{map-prod } sf\ sf)))\ 'F$

$\langle proof \rangle$

lemma $action:$

fixes $F :: ('v \times 'a \times 's \times 's) \text{ set}$

shows $\text{spec.map } af\ sf\ vf\ (\text{spec.action } F) \sqcup \text{spec.idle}$

$= \text{spec.action } (\text{map-prod } vf\ (\text{map-prod } af\ (\text{map-prod } sf\ sf)))\ 'F$ (**is** ?lhs = ?rhs)

$\langle proof \rangle$

lemma $surj-sf-action:$

assumes $surj\ sf$

shows $\text{spec.map } af\ sf\ vf\ (\text{spec.action } F) = \text{spec.action } (\text{map-prod } vf\ (\text{map-prod } af\ (\text{map-prod } sf\ sf)))\ 'F$

$\langle proof \rangle$

$\langle ML \rangle$

lemma $empty:$

shows $\text{spec.action } \{\} = \text{spec.idle}$

$\langle proof \rangle$

lemma $idleI:$

assumes $snd \text{ ' set } xs \subseteq \{s\}$
shows $\langle s, xs, None \rangle \leq spec.action F$
 $\langle proof \rangle$

lemma stepI:
assumes $(v, a, s, s') \in F$
assumes $\forall v''. w = Some v'' \longrightarrow v'' = v$
shows $\langle s, [(a, s')], w \rangle \leq spec.action F$
 $\langle proof \rangle$

lemma stutterI:
assumes $(v, a, s, s) \in F$
shows $\langle s, [], Some v \rangle \leq spec.action F$
 $\langle proof \rangle$

lemma stutter-stepI:
assumes $(v, a, s, s) \in F$
shows $\langle s, [(b, s)], Some v \rangle \leq spec.action F$
 $\langle proof \rangle$

lemma stutter-stepsI:
assumes $(v, a, s, s) \in F$
assumes $snd \text{ ' set } xs \subseteq \{s\}$
shows $\langle s, xs, Some v \rangle \leq spec.action F$
 $\langle proof \rangle$

lemma monotone:
shows $mono spec.action$
 $\langle proof \rangle$

lemmas $strengthen[strg] = st-monotone[OF spec.action.monotone]$
lemmas $mono = monotoneD[OF spec.action.monotone]$
lemmas $mono2mono[cont-intro, partial-function-mono]$
 $= monotone2monotone[OF spec.action.monotone, simplified]$

lemma Sup:
shows $spec.action (\bigcup X) = (\bigsqcup_{F \in X. spec.action F}) \sqcup spec.idle$
 $\langle proof \rangle$

lemma
shows $SUP: spec.action (\bigcup_{x \in X. F x}) = (\bigsqcup_{x \in X. spec.action (F x)}) \sqcup spec.idle$
and $SUP-not-empty: X \neq \{\} \implies spec.action (\bigcup_{x \in X. F x}) = (\bigsqcup_{x \in X. spec.action (F x)})$
 $\langle proof \rangle$

lemma sup:
shows $spec.action (F \cup G) = spec.action F \sqcup spec.action G$
 $\langle proof \rangle$

lemma Inf-le:
shows $spec.action (\bigcap Fs) \leq \prod (spec.action \text{ ' } Fs)$
 $\langle proof \rangle$

lemma inf-le:
shows $spec.action (F \cap G) \leq spec.action F \sqcap spec.action G$
 $\langle proof \rangle$

lemma stutter-agents-le:

assumes $\llbracket A \neq \{\}; r \neq \{\} \rrbracket \implies B \neq \{\}$
assumes $r \subseteq Id$
shows $spec.action (\{v\} \times A \times r) \leq spec.action (\{v\} \times B \times r)$
 $\langle proof \rangle$

lemma *read-agents*:

assumes $A \neq \{\}$
assumes $B \neq \{\}$
assumes $r \subseteq Id$
shows $spec.action (\{v\} \times A \times r) = spec.action (\{v\} \times B \times r)$
 $\langle proof \rangle$

lemma *invmap-le*: — A typical refinement

fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$
shows $spec.action (map-prod vf (map-prod af (map-prod sf sf))) - ' F \leq spec.invmap af sf vf (spec.action F)$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *action-le-conv*:

shows $\langle \sigma \rangle \leq spec.action F$
 $\longleftrightarrow (trace.steps \sigma = \{\} \wedge case-option True (\lambda v. \exists a. (v, a, trace.init \sigma, trace.init \sigma) \in F) (trace.term \sigma))$
 $\vee (\exists x \in F. trace.steps \sigma = \{snd x\} \wedge case-option True ((=) (fst x)) (trace.term \sigma))$ (**is** $?lhs \longleftrightarrow ?rhs$)
 $\langle proof \rangle$

lemma *action-Some-leE*:

assumes $\langle \sigma \rangle \leq spec.action F$
assumes $trace.term \sigma = Some v$
obtains x
where $x \in F$
and $trace.init \sigma = fst (snd (snd x))$
and $trace.final \sigma = snd (snd (snd x))$
and $trace.steps \sigma \subseteq \{snd x\}$
and $v = fst x$
 $\langle proof \rangle$

lemma *action-not-idle-leE*:

assumes $\langle \sigma \rangle \leq spec.action F$
assumes $\not\vdash \sigma \neq trace.T (trace.init \sigma) \square None$
obtains x
where $x \in F$
and $trace.init \sigma = fst (snd (snd x))$
and $trace.final \sigma = snd (snd (snd x))$
and $trace.steps \sigma \subseteq \{snd x\}$
and $case-option True ((=) (fst x)) (trace.term \sigma)$
 $\langle proof \rangle$

lemma *action-not-idle-le-splitE*:

assumes $\langle \sigma \rangle \leq spec.action F$
assumes $\not\vdash \sigma \neq trace.T (trace.init \sigma) \square None$
obtains $(return) v a$
where $(v, a, trace.init \sigma, trace.init \sigma) \in F$
and $trace.steps \sigma = \{\}$
and $trace.term \sigma = Some v$
 $| (step) v a ys zs$
where $(v, a, trace.init \sigma, trace.final \sigma) \in F$

and $trace.init\ \sigma \neq trace.final\ \sigma$
and $snd\ 'set\ ys \subseteq \{trace.init\ \sigma\}$
and $snd\ 'set\ zs \subseteq \{trace.final\ \sigma\}$
and $trace.rest\ \sigma = ys\ @\ [(a,\ trace.final\ \sigma)]\ @\ zs$
and $case-option\ True\ ((=)\ v)\ (trace.term\ \sigma)$

$\langle proof \rangle$

lemma *guard-le-conv[spec.singleton.le-conv]*:

shows $\langle \sigma \rangle \leq spec.guard\ g \longleftrightarrow trace.steps\ \sigma = \{\} \wedge (case-option\ True\ \langle g\ (trace.init\ \sigma) \rangle\ (trace.term\ \sigma))$

$\langle proof \rangle$

lemma *return-le-conv[spec.singleton.le-conv]*:

shows $\langle \sigma \rangle \leq spec.return\ v$
 $\longleftrightarrow trace.steps\ \sigma = \{\} \wedge (case-option\ True\ ((=)\ v)\ (trace.term\ \sigma))$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *mono-stronger*:

assumes $\bigwedge v\ a\ s\ s'. \llbracket (v,\ a,\ s,\ s') \in F; s \neq s' \rrbracket \implies (v,\ a,\ s,\ s') \in F'$
assumes $\bigwedge v\ a\ s. (v,\ a,\ s,\ s) \in F \implies \exists a'. (v,\ a',\ s,\ s) \in F'$
shows $spec.action\ F \leq spec.action\ F'$

$\langle proof \rangle$

lemma *cong*:

assumes $\bigwedge v\ a\ s\ s'. s \neq s' \implies (v,\ a,\ s,\ s') \in F \longleftrightarrow (v,\ a,\ s,\ s') \in F'$
assumes $\bigwedge v\ a\ s. (v,\ a,\ s,\ s) \in F \implies \exists a'. (v,\ a',\ s,\ s) \in F'$
assumes $\bigwedge v\ a\ s. (v,\ a,\ s,\ s) \in F' \implies \exists a'. (v,\ a',\ s,\ s) \in F$
shows $spec.action\ F = spec.action\ F'$

$\langle proof \rangle$

lemma *le-actionD*:

assumes $spec.action\ F \leq spec.action\ F'$
shows $\llbracket (v,\ a,\ s,\ s') \in F; s \neq s' \rrbracket \implies (v,\ a,\ s,\ s') \in F'$
and $(v,\ a,\ s,\ s) \in F \implies \exists a'. (v,\ a',\ s,\ s) \in F'$

$\langle proof \rangle$

lemma *eq-action-conv*:

shows $spec.action\ F = spec.action\ F'$
 $\longleftrightarrow (\forall v\ a\ s\ s'. s \neq s' \longrightarrow (v,\ a,\ s,\ s') \in F \longleftrightarrow (v,\ a,\ s,\ s') \in F')$
 $\wedge (\forall v\ a\ s. (v,\ a,\ s,\ s) \in F \longrightarrow (\exists a'. (v,\ a',\ s,\ s) \in F'))$
 $\wedge (\forall v\ a\ s. (v,\ a,\ s,\ s) \in F' \longrightarrow (\exists a'. (v,\ a',\ s,\ s) \in F))$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *return-alt-def*:

assumes $A \neq \{\}$
shows $spec.return\ v = spec.action\ (\{v\} \times A \times Id)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *cong*:

assumes $\bigwedge v\ a\ s\ s'. (v,\ a,\ s,\ s') \in F \implies s' = s$
assumes $\bigwedge v\ s. v \in fst\ 'F \implies \exists a. (v,\ a,\ s,\ s) \in F$
shows $spec.action\ F = \bigsqcup (spec.return\ 'fst\ 'F) \sqcup spec.idle$

$\langle proof \rangle$

lemma *action-le*:

assumes $Id \subseteq snd \text{ ' } snd \text{ ' } F$

shows $spec.return () \leq spec.action F$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *alt-def*:

assumes $A \neq \{\}$

shows $spec.guard g = spec.action (\{\}) \times A \times Diag g$

$\langle proof \rangle$

lemma *bot*:

shows $spec.guard \perp = spec.idle$

and $spec.guard \langle False \rangle = spec.idle$

$\langle proof \rangle$

lemma *top*:

shows $spec.guard \top = spec.return ()$

and $spec.guard \langle True \rangle = spec.return ()$

$\langle proof \rangle$

lemma *monotone*:

shows $mono spec.guard$

$\langle proof \rangle$

lemmas $strengthen[strg] = st-monotone[OF spec.guard.monotone]$

lemmas $mono = monotoneD[OF spec.guard.monotone]$

lemmas $mono2mono[cont-intro, partial-function-mono] = monotone2monotone[OF spec.guard.monotone, simplified]$

lemma *Sup*:

shows $spec.guard (\bigsqcup X) = \bigsqcup (spec.guard \text{ ' } X) \sqcup spec.idle$

$\langle proof \rangle$

lemma *sup*:

shows $spec.guard (g \sqcup h) = spec.guard g \sqcup spec.guard h$

$\langle proof \rangle$

lemma *return-le*:

shows $spec.guard g \leq spec.return ()$

$\langle proof \rangle$

lemma *guard-less*: — Non-triviality

assumes $g < g'$

shows $spec.guard g < spec.guard g'$

$\langle proof \rangle$

lemma *cong*:

assumes $\bigwedge v a s s'. (v, a, s, s') \in F \implies s' = s$

shows $spec.action F = spec.guard (\lambda s. s \in fst \text{ ' } snd \text{ ' } snd \text{ ' } F) \text{ (is ?lhs = ?rhs)}$

$\langle proof \rangle$

lemma *action-le*:

assumes $Diag g \subseteq snd \text{ ' } snd \text{ ' } F$

shows $spec.guard g \leq spec.action F$

$\langle proof \rangle$

⟨ML⟩

8.7 Operations on return values

For various purposes, including defining a history-respecting sequential composition (bind, see §8.8), we use a Galois pair of operations that saturate or eradicate return values.

⟨ML⟩

definition *none* :: ('a, 's, 'v) spec ⇒ ('a, 's, 'w) spec **where**
none P = $\bigsqcup \{ \langle s, xs, None \rangle \mid s \ xs \ v. \langle s, xs, v \rangle \leq P \}$

definition *all* :: ('a, 's, 'v) spec ⇒ ('a, 's, 'w) spec **where**
all P = $\bigsqcup \{ \langle s, xs, v \rangle \mid s \ xs \ v. \langle s, xs, None \rangle \leq P \}$

⟨ML⟩

interpretation *term*: *galois.complete-lattice-distributive-class spec.term.none spec.term.all*
⟨proof⟩

⟨ML⟩

lemma *none-le-conv*[*spec.singleton.le-conv*]:

shows $\langle \sigma \rangle \leq \text{spec.term.none } P \iff \text{trace.term } \sigma = \text{None} \wedge \langle \text{trace.init } \sigma, \text{trace.rest } \sigma, \text{None} \rangle \leq P$ (**is** ?lhs \iff ?rhs)
⟨proof⟩

lemma *all-le-conv*[*spec.singleton.le-conv*]:

shows $\langle \sigma \rangle \leq \text{spec.term.all } P \iff (\exists w. \langle \text{trace.init } \sigma, \text{trace.rest } \sigma, w \rangle \leq P)$ (**is** ?lhs \iff ?rhs)
⟨proof⟩

⟨ML⟩

lemma *singleton*:

shows $\text{spec.term.none } \langle \sigma \rangle = \langle \text{trace.init } \sigma, \text{trace.rest } \sigma, \text{None} \rangle$
⟨proof⟩

lemmas *bot*[*simp*] = *spec.term.lower-bot*

lemmas *monotone* = *spec.term.monotone-lower*

lemmas *mono* = *monotoneD*[*OF spec.term.none.monotone*]

lemmas *Sup* = *spec.term.lower-Sup*

lemmas *sup* = *spec.term.lower-sup*

lemmas *Inf-le* = *spec.term.lower-Inf-le*

lemma *Inf-not-empty*:

assumes $X \neq \{\}$
shows $\text{spec.term.none } (\bigsqcap X) = (\bigsqcap x \in X. \text{spec.term.none } x)$
⟨proof⟩

lemma *inf*:

shows $\text{spec.term.none } (P \sqcap Q) = \text{spec.term.none } P \sqcap \text{spec.term.none } Q$
and $\text{spec.term.none } (Q \sqcap P) = \text{spec.term.none } Q \sqcap \text{spec.term.none } P$
⟨proof⟩

lemma *inf-unit*:

fixes $P Q :: (-, -, \text{unit}) \text{spec}$
shows $\text{spec.term.none } (P \sqcap Q) = \text{spec.term.none } P \sqcap Q$ (**is** *?thesis1* $P Q$)
and $\text{spec.term.none } (P \sqcap Q) = P \sqcap \text{spec.term.none } Q$ (**is** *?thesis2*)
 $\langle \text{proof} \rangle$

lemma *idempotent[simp]*:
shows $\text{spec.term.none } (\text{spec.term.none } P) = \text{spec.term.none } P$
 $\langle \text{proof} \rangle$

lemma *contractive[iff]*:
shows $\text{spec.term.none } P \leq P$
 $\langle \text{proof} \rangle$

lemma *map-gen*:
fixes $vf :: 'v \Rightarrow 'w$
fixes $vf' :: 'a \Rightarrow 'b$ — arbitrary type
shows $\text{spec.term.none } (\text{spec.map } af \text{ sf } vf \ P) = \text{spec.map } af \text{ sf } vf' (\text{spec.term.none } P)$ (**is** *?lhs = ?rhs*)
 $\langle \text{proof} \rangle$

lemmas $\text{map} = \text{spec.term.none.map-gen}[\text{where } vf'=id]$ — *simp-friendly*

lemma *invmap-gen*:
fixes $vf :: 'v \Rightarrow 'w$
fixes $vf' :: 'a \Rightarrow 'b$ — arbitrary type
shows $\text{spec.term.none } (\text{spec.invmap } af \text{ sf } vf \ P) = \text{spec.invmap } af \text{ sf } vf' (\text{spec.term.none } P)$ (**is** *?lhs = ?rhs*)
 $\langle \text{proof} \rangle$

lemmas $\text{invmap} = \text{spec.term.none.invmap-gen}[\text{where } vf'=id]$ — *simp-friendly*

lemma *idle*:
shows $\text{spec.term.none } \text{spec.idle} = \text{spec.idle}$
 $\langle \text{proof} \rangle$

lemma *return*:
shows $\text{spec.term.none } (\text{spec.return } v) = \text{spec.idle}$
 $\langle \text{proof} \rangle$

lemma *guard*:
shows $\text{spec.term.none } (\text{spec.guard } g) = \text{spec.idle}$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *none-all-le*:
shows $\text{spec.term.none } P \leq \text{spec.term.all } P$
 $\langle \text{proof} \rangle$

lemma *none-all[simp]*:
shows $\text{spec.term.none } (\text{spec.term.all } P) = \text{spec.term.none } P$
 $\langle \text{proof} \rangle$

lemma *all-none[simp]*:
shows $\text{spec.term.all } (\text{spec.term.none } P) = \text{spec.term.all } P$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemmas $\text{bot}[\text{simp}] = \text{spec.term.upper-bot}$

lemmas *top* = *spec.term.upper-top*

lemmas *monotone* = *spec.term.monotone-upper*

lemmas *mono* = *monotoneD[OF spec.term.all.monotone]*

lemma *expansive*:

shows $P \leq \text{spec.term.all } P$

<proof>

lemmas *Sup* = *spec.term.upper-Sup*

lemmas *sup* = *spec.term.upper-sup*

lemmas *Inf* = *spec.term.upper-Inf*

lemmas *inf* = *spec.term.upper-inf*

lemmas *singleton* = *spec.term.all-def[where P= $\langle\sigma\rangle$]* **for** σ

lemma *monomorphic*:

shows *spec.term.cl* - = *spec.term.all*

<proof>

lemma *closed-conv*:

assumes $P \in \text{spec.term.closed}$ -

shows $P = \text{spec.term.all } P$

<proof>

lemma *closed[iff]*:

shows $\text{spec.term.all } P \in \text{spec.term.closed}$ -

<proof>

lemma *idempotent[simp]*:

shows $\text{spec.term.all } (\text{spec.term.all } P) = \text{spec.term.all } P$

<proof>

lemma *map*: — *vf* = *id* on the RHS

fixes $vf :: 'v \Rightarrow 'w$

shows $\text{spec.term.all } (\text{spec.map af sf vf } P) = \text{spec.map af sf id } (\text{spec.term.all } P)$ (**is** *?lhs* = *?rhs*)

<proof>

lemma *invmap*: — *vf* = *id* on the RHS

fixes $vf :: 'v \Rightarrow 'w$

shows $\text{spec.term.all } (\text{spec.invmap af sf vf } P) = \text{spec.invmap af sf id } (\text{spec.term.all } P)$ (**is** *?lhs* = *?rhs*)

<proof>

lemma *vmap-unit-absorb*:

shows $\text{spec.vmap } \langle()\rangle (\text{spec.term.all } P) = \text{spec.term.all } P$ (**is** *?lhs* = *?rhs*)

<proof>

lemma *vmap-unit*:

shows $\text{spec.vmap } \langle()\rangle (\text{spec.term.all } P) = \text{spec.term.all } (\text{spec.vmap } \langle()\rangle P)$

<proof>

lemma *idle*:

shows $\text{spec.term.all spec.idle} = (\bigsqcup v. \text{spec.return } v)$ (**is** *?lhs* = *?rhs*)

<proof>

lemma *action*:

fixes $F :: ('v \times 'a \times 's \times 's)$ set
shows $\text{spec.term.all} (\text{spec.action } F) = \text{spec.action} (UNIV \times \text{snd } 'F) \sqcup (\bigsqcup v. \text{spec.return } v)$ (**is** ?lhs = ?rhs)
 ⟨proof⟩

lemma *return*:
shows $\text{spec.term.all} (\text{spec.return } v) = (\bigsqcup v. \text{spec.return } v)$
 ⟨proof⟩

lemma *guard*:
shows $\text{spec.term.all} (\text{spec.guard } g) = (\bigsqcup v. \text{spec.return } v)$
 ⟨proof⟩

⟨ML⟩

lemma *none-le-conv[spec.idle-le]*:
shows $\text{spec.idle} \leq \text{spec.term.none } P \longleftrightarrow \text{spec.idle} \leq P$
 ⟨proof⟩

lemma *all-le-conv[spec.idle-le]*:
shows $\text{spec.idle} \leq \text{spec.term.all } P \longleftrightarrow \text{spec.idle} \leq P$
 ⟨proof⟩

⟨ML⟩

lemma *return-unit*:
shows $\text{spec.return } () \in \text{spec.term.closed}$ -
 ⟨proof⟩

lemma *none-inf*:
fixes $P :: ('a, 's, 'v)$ spec
fixes $Q :: ('a, 's, 'w)$ spec
assumes $P \in \text{spec.term.closed}$ -
shows $P \sqcap \text{spec.term.none } Q = \text{spec.term.none} (\text{spec.term.none } P \sqcap Q)$ (**is** ?lhs = ?rhs)
and $\text{spec.term.none } Q \sqcap P = \text{spec.term.none} (Q \sqcap \text{spec.term.none } P)$ (**is** ?thesis1)
 ⟨proof⟩

lemma *none-inf-monomorphic*:
fixes $P :: ('a, 's, 'v)$ spec
fixes $Q :: ('a, 's, 'v)$ spec
assumes $P \in \text{spec.term.closed}$ -
shows $P \sqcap \text{spec.term.none } Q = \text{spec.term.none} (P \sqcap Q)$ (**is** ?thesis1)
and $\text{spec.term.none } Q \sqcap P = \text{spec.term.none} (Q \sqcap P)$ (**is** ?thesis2)
 ⟨proof⟩

lemma *singleton-le-extI*:
assumes $Q \in \text{spec.term.closed}$ -
assumes $\bigwedge s xs. \langle s, xs, \text{None} \rangle \leq P \implies \langle s, xs, \text{None} \rangle \leq Q$
shows $P \leq Q$
 ⟨proof⟩

⟨ML⟩

8.8 Bind

We define monadic *bind* in terms of bi-strict *continue*. The latter supports left and right residuals (see, amongst many others, [Hoare and He \(1987\)](#); [Hoare, He, and Sanders \(1987b\)](#); [Pratt \(1990\)](#)), whereas *bind* encodes the non-retractability of observable actions, i.e., $\text{spec.term.none } f \leq f \ggg g$, which defeats a general right residual.

It is tempting to write this in a more direct style (using *case-option*) but the set comprehension syntax is not

friendly to strengthen/monotonicity facts.

$\langle ML \rangle$

definition *continue* :: $(\prime a, \prime s, \prime v) \text{ spec} \Rightarrow (\prime v \Rightarrow (\prime a, \prime s, \prime w) \text{ spec}) \Rightarrow (\prime a, \prime s, \prime w) \text{ spec}$ **where**
continue $f g =$
 $\sqcup \{ \langle \text{trace.init } \sigma_f, \text{trace.rest } \sigma_f @ \text{trace.rest } \sigma_g, \text{trace.term } \sigma_g \rangle$
 $|\sigma_f \sigma_g v. \langle \sigma_f \rangle \leq f \wedge \text{trace.init } \sigma_g = \text{trace.final } \sigma_f \wedge \text{trace.term } \sigma_f = \text{Some } v \wedge \langle \sigma_g \rangle \leq g v \}$

definition *bind* :: $(\prime a, \prime s, \prime v) \text{ spec} \Rightarrow (\prime v \Rightarrow (\prime a, \prime s, \prime w) \text{ spec}) \Rightarrow (\prime a, \prime s, \prime w) \text{ spec}$ **where**
bind $f g = \text{spec.term.none } f \sqcup \text{spec.continue } f g$

adhoc-overloading

Monad-Syntax.bind spec.bind

$\langle ML \rangle$

lemma *continue-le-conv*:

shows $\langle \sigma \rangle \leq \text{spec.continue } f g$
 $\longleftrightarrow (\exists xs ys v w. \langle \text{trace.init } \sigma, xs, \text{Some } v \rangle \leq f$
 $\wedge \langle \text{trace.final}' (\text{trace.init } \sigma) xs, ys, w \rangle \leq g v$
 $\wedge \sigma \leq \text{trace.T } (\text{trace.init } \sigma) (xs @ ys) w)$ **(is ?lhs \longleftrightarrow ?rhs)**

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *mono*:

assumes $f \leq f'$
assumes $\bigwedge v. g v \leq g' v$
shows $\text{spec.continue } f g \leq \text{spec.continue } f' g'$

$\langle \text{proof} \rangle$

lemma *strengthen[stg]*:

assumes *st-ord* $F f f'$
assumes $\bigwedge x. \text{st-ord } F (g x) (g' x)$
shows *st-ord* $F (\text{spec.continue } f g) (\text{spec.continue } f' g')$

$\langle \text{proof} \rangle$

lemma *mono2mono[cont-intro, partial-function-mono]*:

assumes *monotone orda* $(\leq) f$
assumes $\bigwedge x. \text{monotone orda } (\leq) (\lambda y. g y x)$
shows *monotone orda* $(\leq) (\lambda x. \text{spec.continue } (f x) (g x))$

$\langle \text{proof} \rangle$

definition *resL* :: $(\prime v \Rightarrow (\prime a, \prime s, \prime w) \text{ spec}) \Rightarrow (\prime a, \prime s, \prime w) \text{ spec} \Rightarrow (\prime a, \prime s, \prime v) \text{ spec}$ **where**
resL $g P = \sqcup \{ f. \text{spec.continue } f g \leq P \}$

definition *resR* :: $(\prime a, \prime s, \prime v) \text{ spec} \Rightarrow (\prime a, \prime s, \prime w) \text{ spec} \Rightarrow (\prime v \Rightarrow (\prime a, \prime s, \prime w) \text{ spec})$ **where**
resR $f P = \sqcup \{ g. \text{spec.continue } f g \leq P \}$

interpretation *L*: *galois.complete-lattice-class* $\lambda f. \text{spec.continue } f g \text{spec.continue.resL } g$ **for** g

$\langle \text{proof} \rangle$

interpretation *R*: *galois.complete-lattice-class* $\lambda g. \text{spec.continue } f g \text{spec.continue.resR } f$

for $f :: (\prime a, \prime s, \prime v) \text{ spec}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *bind-le-conv*:

shows $\langle \sigma \rangle \leq \text{spec.bind } f \ g \iff \langle \sigma \rangle \leq \text{spec.term.none } f \vee \langle \sigma \rangle \leq \text{spec.continue } f \ g$
 ⟨proof⟩

lemma *bind-le[consumes 1]*:

assumes $\langle \sigma \rangle \leq f \ggg g$

obtains

(incomplete) $\langle \sigma \rangle \leq \text{spec.term.none } f$

| (continue) $\sigma_f \ \sigma_g \ v_f$

where $\langle \sigma_f \rangle \leq f$ **and** $\text{trace.final } \sigma_f = \text{trace.init } \sigma_g$ **and** $\text{trace.term } \sigma_f = \text{Some } v_f$

and $\langle \sigma_g \rangle \leq g \ v_f$ **and** $\not\vdash \sigma_g \neq \text{trace.T } (\text{trace.init } \sigma_g) \ \square \ \text{None}$

and $\sigma = \text{trace.T } (\text{trace.init } \sigma_f) \ (\text{trace.rest } \sigma_f \ @ \ \text{trace.rest } \sigma_g) \ (\text{trace.term } \sigma_g)$

⟨proof⟩

⟨ML⟩

lemma *bind-le[case-names incomplete continue]*:

assumes $\text{spec.term.none } f \leq P$

assumes $\bigwedge \sigma_f \ \sigma_g \ v. \ [\langle \sigma_f \rangle \leq f; \text{trace.init } \sigma_g = \text{trace.final } \sigma_f; \text{trace.term } \sigma_f = \text{Some } v; \langle \sigma_g \rangle \leq g \ v;$

$\not\vdash \sigma_g \neq \text{trace.T } (\text{trace.init } \sigma_g) \ \square \ \text{None}]$

$\implies \langle \text{trace.init } \sigma_f, \text{trace.rest } \sigma_f \ @ \ \text{trace.rest } \sigma_g, \text{trace.term } \sigma_g \rangle \leq P$

shows $f \ggg g \leq P$

⟨proof⟩

⟨ML⟩

definition *resL* :: $(v \Rightarrow (a, s, w) \text{ spec}) \Rightarrow (a, s, w) \text{ spec} \Rightarrow (a, s, v) \text{ spec}$ **where**

$\text{resL } g \ P = \bigsqcup \{f. f \ggg g \leq P\}$

lemma *incompleteI*:

assumes $\langle s, xs, \text{None} \rangle \leq f$

shows $\langle s, xs, \text{None} \rangle \leq f \ggg g$

⟨proof⟩

lemma *continueI*:

assumes $f: \langle s, xs, \text{Some } v \rangle \leq f$

assumes $g: \langle \text{trace.final}' \ s \ xs, ys, w \rangle \leq g \ v$

shows $\langle s, xs \ @ \ ys, w \rangle \leq f \ggg g$

⟨proof⟩

lemma *singletonL*:

shows $\langle \sigma \rangle \ggg g$

$= \text{spec.term.none } \langle \sigma \rangle$

$\sqcup \bigsqcup \{ \langle \text{trace.init } \sigma, \text{trace.rest } \sigma \ @ \ \text{trace.rest } \sigma_g, \text{trace.term } \sigma_g \rangle \mid \sigma_g.$

$\text{trace.final } \sigma = \text{trace.init } \sigma_g \wedge (\exists v. \text{trace.term } \sigma = \text{Some } v \wedge \langle \sigma_g \rangle \leq g \ v) \}$ (**is** ?lhs = ?rhs)

⟨proof⟩

lemma *mono*:

assumes $f \leq f'$

assumes $\bigwedge v. g \ v \leq g' \ v$

shows $\text{spec.bind } f \ g \leq \text{spec.bind } f' \ g'$

⟨proof⟩

lemma *strengthen[strg]*:

assumes $\text{st-ord } F \ f \ f'$

assumes $\bigwedge x. \text{st-ord } F \ (g \ x) \ (g' \ x)$

shows $\text{st-ord } F \ (\text{spec.bind } f \ g) \ (\text{spec.bind } f' \ g')$

<proof>

lemma *mono2mono*[*cont-intro, partial-function-mono*]:

assumes *monotone orda* (\leq) *f*

assumes $\bigwedge x. \text{monotone orda } (\leq) (\lambda y. g y x)$

shows *monotone orda* (\leq) ($\lambda x. \text{spec.bind } (f x) (g x)$)

<proof>

interpretation *L: galois.complete-lattice-class* $\lambda f. f \ggg g \text{ spec.bind.resL } g$ **for** *g*

<proof>

lemmas *SUPL = spec.bind.L.lower-SUP*

lemmas *SupL = spec.bind.L.lower-Sup*

lemmas *supL = spec.bind.L.lower-sup*[*of f₁ f₂ g*] **for** *f₁ f₂ g*

lemmas *INFL-le = spec.bind.L.lower-INF-le*

lemmas *InfL-le = spec.bind.L.lower-Inf-le*

lemmas *infL-le = spec.bind.L.lower-inf-le*[*of f₁ f₂ g*] **for** *f₁ f₂ g*

lemma *SUPR*:

shows *spec.bind f* ($\lambda v. \bigsqcup_{x \in X}. g x v$) = ($\bigsqcup_{x \in X}. f \ggg g x$) \sqcup ($f \ggg \perp$) (**is ?thesis1**) — *Sup over ('a, 's, 'v) spec*

and *spec.bind f* ($\bigsqcup_{x \in X}. g x$) = ($\bigsqcup_{x \in X}. f \ggg g x$) \sqcup ($f \ggg \perp$) (**is ?thesis2**) — *Sup over functions*
<proof>

lemma *SUPR-not-empty*:

assumes $X \neq \{\}$

shows *spec.bind f* ($\lambda v. \bigsqcup_{x \in X}. g x v$) = ($\bigsqcup_{x \in X}. f \ggg g x$)

<proof>

lemmas *supR = spec.bind.SUPR-not-empty*[**where** $g=id$ **and** $X=\{g_1, g_2\}$ **for** $g_1 g_2$, *simplified*]

lemma *InfR-le*:

shows *spec.bind f* ($\lambda v. \prod_{x \in X}. g x v$) \leq ($\prod_{x \in X}. f \ggg g x$)

<proof>

lemma *infR-le*:

shows *spec.bind f* ($g_1 \sqcap g_2$) \leq ($f \ggg g_1$) \sqcap ($f \ggg g_2$)

and *spec.bind f* ($\lambda v. g_1 v \sqcap g_2 v$) \leq ($f \ggg g_1$) \sqcap ($f \ggg g_2$)

<proof>

lemma *Inf-le*:

shows *spec.bind* ($\prod_{x \in X}. f x$) ($\lambda v. (\prod_{x \in X}. g x v)$) \leq ($\prod_{x \in X}. \text{spec.bind } (f x) (g x)$)

<proof>

lemma *inf-le*:

shows *spec.bind* ($f_1 \sqcap f_2$) ($\lambda v. g_1 v \sqcap g_2 v$) \leq *spec.bind f₁ g₁ \sqcap spec.bind f₂ g₂*

<proof>

lemma *mcont2mcont*[*cont-intro*]:

assumes *mcont luba orda Sup* (\leq) *f*

assumes $\bigwedge v. \text{mcont luba orda Sup } (\leq) (\lambda x. g x v)$

shows *mcont luba orda Sup* (\leq) ($\lambda x. \text{spec.bind } (f x) (g x)$)

<proof>

lemmas *botL[simp] = spec.bind.L.lower-bot*

lemma *botR*:

shows $f \gg \perp = \text{spec.term.none } f$
 $\langle \text{proof} \rangle$

lemma *eq-bot-conv*:

shows $\text{spec.bind } f g = \perp \longleftrightarrow f = \perp$
 $\langle \text{proof} \rangle$

lemma *idleL[simp]*:

shows $\text{spec.idle} \gg g = \text{spec.idle}$
 $\langle \text{proof} \rangle$

lemma *idleR*:

shows $f \gg \text{spec.idle} = f \gg \perp$ (**is** $?lhs = ?rhs$)
 $\langle \text{proof} \rangle$

lemmas *ifL = if-distrib*[**where** $f = \lambda f. \text{spec.bind } f g$ **for** g]

$\langle ML \rangle$

lemma *bind-le-conv[spec.idle-le]*:

shows $\text{spec.idle} \leq f \gg g \longleftrightarrow \text{spec.idle} \leq f$ (**is** $?lhs \longleftrightarrow ?rhs$)
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *bindL-le[iff]*:

shows $\text{spec.term.none } f \leq f \gg g$
 $\langle \text{proof} \rangle$

lemma *bind*:

shows $\text{spec.term.none } (f \gg g) = f \gg (\lambda v. \text{spec.term.none } (g v))$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *bind*:

shows $\text{spec.term.all } (f \gg g) = \text{spec.term.all } f \sqcup (f \gg (\lambda v. \text{spec.term.all } (g v)))$ (**is** $?lhs = ?rhs$)
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

The monad laws for (\gg) . $\langle ML \rangle$

lemma *bind*:

fixes $f :: (-, -, -) \text{ spec}$
shows $f \gg g \gg h = f \gg (\lambda v. g v \gg h)$ (**is** $?lhs = ?rhs$)
 $\langle \text{proof} \rangle$

lemmas *assoc = spec.bind.bind*

lemma *returnL-le*:

shows $g v \leq \text{spec.return } v \gg g$ (**is** $?lhs \leq ?rhs$)
 $\langle \text{proof} \rangle$

lemma *returnL*:

assumes $\text{spec.idle} \leq g v$
shows $\text{spec.return } v \gg g = g v$
 $\langle \text{proof} \rangle$

lemma *returnR[simp]*:

shows $f \ggg \text{spec.return} = f$ (**is** $?lhs = ?rhs$)
<proof>

lemma *return*: — Does not require $\text{spec.idle} \leq g \ v$

fixes $f :: ('a, 's, 'v) \text{spec}$
fixes $g :: 'v \Rightarrow ('a, 's, 'w) \text{spec}$
shows $f \ggg (\lambda v. \text{spec.return } x \ggg g \ v) = f \ggg (\lambda v. g \ v \ x)$ (**is** $?lhs = ?rhs$)
<proof>

<ML>

lemma *noneL[simp]*:

shows $\text{spec.term.none } f \ggg g = \text{spec.term.none } f$
<proof>

<ML>

lemma *bind-le*: — Converse does not hold: it may be that no final states of f satisfy g

fixes $f :: ('a, 's, 'v) \text{spec}$
fixes $g :: 'v \Rightarrow ('a, 's, 'w) \text{spec}$
fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'w \Rightarrow 'x$
shows $\text{spec.map } af \ sf \ vf \ (f \ggg g) \leq \text{spec.map } af \ sf \ id \ f \ggg (\lambda v. \text{spec.map } af \ sf \ vf \ (g \ v))$
<proof>

lemma *bind-inj-sf*:

fixes $f :: ('a, 's, 'x) \text{spec}$
fixes $g :: 'x \Rightarrow ('a, 's, 'v) \text{spec}$
assumes *inj sf*
shows $\text{spec.map } af \ sf \ vf \ (f \ggg g) = \text{spec.map } af \ sf \ id \ f \ggg (\lambda v. \text{spec.map } af \ sf \ vf \ (g \ v))$ (**is** $?lhs = ?rhs$)
<proof>

<ML>

lemma *eq-return*: — generalizes *spec.bind.returnR*

shows $\text{spec.vmap } vf \ P = P \ggg \text{spec.return} \circ vf$ (**is** $?thesis1$)
and $\text{spec.vmap } vf \ P = P \ggg (\lambda v. \text{spec.return} \ (vf \ v))$ (**is** $?lhs = ?rhs$) — useful for flip/symmetric
<proof>

lemma *unitL*: — monomorphise ignored return values

shows $f \ggg g = \text{spec.vmap } \langle () \rangle \ f \ggg g$
<proof>

<ML>

lemma *bind*:

fixes $f :: ('b, 't, 'v) \text{spec}$
fixes $g :: 'v \Rightarrow ('b, 't, 'x) \text{spec}$
fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'w \Rightarrow 'x$
shows $\text{spec.invmap } af \ sf \ vf \ (f \ggg g) = \text{spec.invmap } af \ sf \ id \ f \ggg (\lambda v. \text{spec.invmap } af \ sf \ vf \ (g \ v))$ (**is** $?lhs = ?rhs$)
<proof>

lemma *split-invmap*:

shows $\text{spec.invmap } af \text{ } sf \text{ } vf \text{ } P = \text{spec.invmap } af \text{ } sf \text{ } id \text{ } P \gg (\lambda v. \bigsqcup v' \in vf - \{v\}. \text{spec.return } v')$ (**is** $?lhs = ?rhs$)
<proof>

<ML>

lemma *return-const*:

assumes $V \neq \{\}$

assumes $W \neq \{\}$

shows $\text{spec.action } (V \times F) = \text{spec.action } (W \times F) \gg (\bigsqcup v \in V. \text{spec.return } v)$ (**is** $?lhs = ?rhs$)
<proof>

<ML>

lemma *bind-all-return*:

assumes $f \in \text{spec.term.closed}$ -

shows $f \gg (\bigsqcup \text{range spec.return}) = \text{spec.term.all } f$ (**is** $?lhs = ?rhs$)
<proof>

<ML>

8.9 Kleene star

We instantiate the generic Kleene locale with monomorphic $\text{spec.return } ()$. The polymorphic $(\bigsqcup v. \text{spec.return } v)$ fails the *comp-unitR* axiom ($\varepsilon \leq x \implies x \cdot \varepsilon = x$).

<ML>

interpretation *kleene*: *weak-kleene* $\text{spec.return } () \lambda x y. \text{spec.bind } x \langle y \rangle$

<proof>

<ML>

lemmas $\text{star-le}[\text{spec.idle-le}] = \text{order.trans}[OF \text{spec.idle.return-le spec.kleene.epsilon-star-le}]$

lemmas $\text{rev-star-le}[\text{spec.idle-le}] = \text{spec.idle.kleene.star-le}[\text{unfolded spec.kleene.star-rev-star}]$

<ML>

lemmas $\text{star-le} = \text{spec.kleene.epsilon-star-le}$

lemmas $\text{rev-star-le} = \text{spec.return.kleene.star-le}[\text{unfolded spec.kleene.star-rev-star}]$

<ML>

lemma *star-idle*:

shows $\text{spec.kleene.star spec.idle} = \text{spec.return } ()$
<proof>

lemmas $\text{rev-star-idle} = \text{spec.kleene.star-idle}[\text{unfolded spec.kleene.star-rev-star}]$

<ML>

lemma *star-closed-le*:

fixes $P :: (-, -, \text{unit}) \text{spec}$

assumes $P \in \text{spec.term.closed}$ -

shows $\text{spec.term.all } (\text{spec.kleene.star } P) \leq \text{spec.kleene.star } P$ (**is** $- \leq ?rhs$)
<proof>

$\langle ML \rangle$

lemma *star*:

assumes $P \in \text{spec.term.closed}$ -

shows $\text{spec.kleene.star } P \in \text{spec.term.closed}$ -

$\langle \text{proof} \rangle$

$\langle ML \rangle$

8.10 Transition relations

Using spec.kleene.star we can specify the transitions each agent is allowed to perform. These constraints ((\sqcap) $\text{spec.rel } r$) distribute through all program constructs (for suitable r).

Observations:

- the Galois connection between spec.rel and spec.steps is much easier to show in the powerset model
 - see [van Staden \(2015, Footnote 2\)](#)
- most useful facts about spec.steps depend on the model

$\langle ML \rangle$

definition $\text{act} :: ('a, 's) \text{ steps} \Rightarrow ('a, 's, \text{unit}) \text{ spec}$ **where** — lift above spec.return to ease some proofs

$\text{act } r = \text{spec.action } (\{\{\}\} \times (r \cup \text{UNIV} \times \text{Id}))$

abbreviation $\text{monomorphic} :: ('a, 's) \text{ steps} \Rightarrow ('a, 's, \text{unit}) \text{ spec}$ **where**

$\text{monomorphic } r \equiv \text{spec.kleene.star } (\text{spec.rel.act } r)$

lemma *act-alt-def*:

shows $\text{spec.rel.act } r = \text{spec.action } (\{\{\}\} \times r) \sqcup \text{spec.return } ()$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

definition $\text{rel} :: ('a, 's) \text{ steps} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$\text{rel } r = \text{spec.term.all } (\text{spec.rel.monomorphic } r)$

definition $\text{steps} :: ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 's) \text{ steps}$ **where**

$\text{steps } P = \bigcap \{r. P \leq \text{spec.rel } r\}$

$\langle ML \rangle$

lemma *monotone*:

shows $\text{mono } \text{spec.rel.act}$

$\langle \text{proof} \rangle$

lemmas $\text{strengthen}[strg] = \text{st-monotone}[OF \text{spec.rel.act.monotone}]$

lemmas $\text{mono} = \text{monotoneD}[OF \text{spec.rel.act.monotone}]$

lemma *empty*:

shows $\text{spec.rel.act } \{\} = \text{spec.return } ()$

$\langle \text{proof} \rangle$

lemma *UNIV*:

shows $\text{spec.rel.act } \text{UNIV} = \text{spec.action } (\{\{\}\} \times \text{UNIV})$

$\langle \text{proof} \rangle$

lemma *sup*:

shows $\text{spec.rel.act } (r \cup s) = \text{spec.rel.act } r \sqcup \text{spec.rel.act } s$
<proof>

lemma *stutter*:

shows $\text{spec.rel.act } (UNIV \times Id) = \text{spec.return } ()$
<proof>

<ML>

lemma *act-mono*:

shows $\text{spec.term.all } (\text{spec.rel.act } r) = \text{spec.rel.act } r$
<proof>

<ML>

lemma *rel*:

shows $\text{spec.term.all } (\text{spec.rel } r) = \text{spec.rel } r$
<proof>

<ML>

lemma *act*:

shows $\text{spec.rel.act } r \in \text{spec.term.closed}$ -
<proof>

<ML>

lemma *rel*:

shows $\text{spec.rel } r \in \text{spec.term.closed}$ -
<proof>

<ML>

lemma *inf-none-rel*: — polymorphic constants

shows $\text{spec.term.none } (\text{spec.rel } r :: ('a, 's, 'w) \text{ spec}) \sqcap \text{spec.term.none } P$
 $= \text{spec.rel } r \sqcap (\text{spec.term.none } P :: ('a, 's, 'v) \text{ spec})$ (**is** *?thesis1*)
and $\text{spec.term.none } P \sqcap \text{spec.term.none } (\text{spec.rel } r :: ('a, 's, 'w) \text{ spec})$
 $= \text{spec.term.none } P \sqcap (\text{spec.rel } r :: ('a, 's, 'v) \text{ spec})$ (**is** *?thesis2*)
<proof>

lemma *inf-rel*:

shows $\text{spec.term.none } P \sqcap \text{spec.rel } r = \text{spec.term.none } (P \sqcap \text{spec.rel } r)$ (**is** *?thesis1*)
and $\text{spec.rel } r \sqcap \text{spec.term.none } P = \text{spec.term.none } (\text{spec.rel } r \sqcap P)$ (**is** *?thesis2*)
<proof>

<ML>

lemma *act-le*:

shows $\text{spec.return } () \leq \text{spec.rel.act } r$
<proof>

<ML>

lemma *rel-le*:

shows $\text{spec.return } v \leq \text{spec.rel } r$
<proof>

lemma *Sup-rel-le*:

shows $\sqcup \text{range spec.return} \leq \text{spec.rel } r$
<proof>

<ML>

lemmas $\text{act-le}[\text{spec.idle-le}] = \text{order.trans}[OF \text{spec.idle.return-le spec.return.rel.act-le}]$

<ML>

lemmas $\text{rel-le}[\text{spec.idle-le}] = \text{order.trans}[OF \text{spec.idle.return-le spec.return.rel-le}]$

<ML>

lemma *le-conv[spec.singleton.le-conv]*:

shows $\langle \sigma \rangle \leq \text{spec.rel.act } r \longleftrightarrow \text{trace.steps } \sigma = \{ \} \vee (\exists x \in r. \text{trace.steps } \sigma = \{ x \})$
<proof>

<ML>

lemma *le-steps*:

assumes $\text{trace.steps } \sigma \subseteq r$
shows $\langle \sigma \rangle \leq \text{spec.rel.monomorphic } r$
<proof>

<ML>

lemmas $\text{mono-le} = \text{spec.kleene.expansive-star}$

<ML>

lemma *alt-def*:

shows $\text{spec.rel.monomorphic } r = \sqcup (\text{spec.singleton } ' \{ \sigma. \text{trace.steps } \sigma \subseteq r \})$ (**is** *?lhs = ?rhs*)
<proof>

<ML>

lemma *monomorphic-le-conv[spec.singleton.le-conv]*:

shows $\langle \sigma \rangle \leq \text{spec.rel.monomorphic } r \longleftrightarrow \text{trace.steps } \sigma \subseteq r$
<proof>

<ML>

lemma *rel-le-conv[spec.singleton.le-conv]*:

shows $\langle \sigma \rangle \leq \text{spec.rel } r \longleftrightarrow \text{trace.steps } \sigma \subseteq r$
<proof>

<ML>

interpretation *rel*: *galois.complete-lattice-class spec.steps spec.rel*

<proof>

lemma *rel-alt-def*:

shows $\text{spec.rel } r = \sqcup (\text{spec.singleton } ' \{ \sigma. \text{trace.steps } \sigma \subseteq r \})$
<proof>

<ML>

lemma *unit-rel*:

shows $\text{spec.vmap } \langle () \rangle (\text{spec.rel } r) = \text{spec.rel } r$
<proof>

<ML>

lemma *monomorphic-conv*: — if the return type is *unit*

shows $\text{spec.rel } r = \text{spec.rel.monomorphic } r$
<proof>

lemma *monomorphic-act-le*: — *unit* return type

shows $\text{spec.rel.act } r \leq \text{spec.rel } r$
<proof>

lemma *empty*:

shows $\text{spec.rel } \{\} = (\bigsqcup v. \text{spec.return } v)$
<proof>

lemmas $UNIV = \text{spec.rel.upper-top}$

lemmas $top = \text{spec.rel.UNIV}$

lemmas $INF = \text{spec.rel.upper-INF}$

lemmas $Inf = \text{spec.rel.upper-Inf}$

lemmas $inf = \text{spec.rel.upper-inf}$

lemmas $Sup-le = \text{spec.rel.Sup-upper-le}$

lemmas $sup-le = \text{spec.rel.sup-upper-le}$ — Converse does not hold: the RHS allows interleaving of r and s steps

lemma *reflcl*:

shows $\text{spec.rel } (r \cup A \times Id) = \text{spec.rel } r$
and $\text{spec.rel } (A \times Id \cup r) = \text{spec.rel } r$
<proof>

lemma *minus-Id*:

shows $\text{spec.rel } (r - A \times Id) = \text{spec.rel } r$
<proof>

lemma *Id*:

shows $\text{spec.rel } (A \times Id) = (\bigsqcup v. \text{spec.return } v)$
<proof>

lemmas $monotone = \text{spec.rel.monotone-upper}$

lemmas $mono = \text{monotoneD}[OF \text{spec.rel.monotone}, \text{of } r \ r' \text{ for } r \ r']$

lemma *mono-reflcl*:

assumes $r \subseteq s \cup UNIV \times Id$
shows $\text{spec.rel } r \leq \text{spec.rel } s$
<proof>

lemma *unfoldL*:

shows $\text{spec.rel } r = \text{spec.rel.act } r \gg \text{spec.rel } r$ (**is** $?lhs = ?rhs$)
<proof>

lemma *foldR*: — arbitrary interstitial return type

shows $\text{spec.rel } r \gg \text{spec.rel.act } r = \text{spec.rel } r$ (**is** $?lhs = ?rhs$)
<proof>

lemma *wind-bind*: — arbitrary interstitial return type

shows $\text{spec.rel } r \gg \text{spec.rel } r = \text{spec.rel } r$ (**is** $?lhs = ?rhs$)
 ⟨proof⟩

lemma *wind-bind-leading*: — arbitrary interstitial return type
assumes $r' \subseteq r$
shows $\text{spec.rel } r' \gg \text{spec.rel } r = \text{spec.rel } r$ (**is** $?lhs = ?rhs$)
 ⟨proof⟩

lemma *wind-bind-trailing*: — arbitrary interstitial return type
assumes $r' \subseteq r$
shows $\text{spec.rel } r \gg \text{spec.rel } r' = \text{spec.rel } r$ (**is** $?lhs = ?rhs$)
 ⟨proof⟩

Interstitial unit, for unfolding

lemmas $\text{unwind-bind} = \text{spec.rel.wind-bind}[\mathbf{where } 'd=\text{unit}, \text{symmetric}]$
lemmas $\text{unwind-bind-leading} = \text{spec.rel.wind-bind-leading}[\mathbf{where } 'd=\text{unit}, \text{symmetric}]$
lemmas $\text{unwind-bind-trailing} = \text{spec.rel.wind-bind-trailing}[\mathbf{where } 'd=\text{unit}, \text{symmetric}]$

⟨ML⟩

lemma *rel*:
shows $\text{spec.invmap } af \ sf \ vf \ (\text{spec.rel } r) = \text{spec.rel } (\text{map-prod } af \ (\text{map-prod } sf \ sf) - ' (r \cup \text{UNIV} \times \text{Id}))$
 ⟨proof⟩

lemma *range*:
shows $\text{spec.invmap } af \ sf \ vf \ P = \text{spec.invmap } af \ sf \ vf \ (P \sqcap \text{spec.rel } (\text{range } af \times \text{range } sf \times \text{range } sf))$
 ⟨proof⟩

⟨ML⟩

lemma *inf-rel*:
shows $\text{spec.map } af \ sf \ vf \ P \sqcap \text{spec.rel } r$
 $= \text{spec.map } af \ sf \ vf \ (P \sqcap \text{spec.rel } (\text{map-prod } af \ (\text{map-prod } sf \ sf) - ' (r \cup \text{UNIV} \times \text{Id})))$
and $\text{spec.rel } r \sqcap \text{spec.map } af \ sf \ vf \ P$
 $= \text{spec.map } af \ sf \ vf \ (\text{spec.rel } (\text{map-prod } af \ (\text{map-prod } sf \ sf) - ' (r \cup \text{UNIV} \times \text{Id})) \sqcap P)$
 ⟨proof⟩

⟨ML⟩

lemma *rel-le*:
fixes $F :: ('v \times 'a \times 's \times 's)$ set
fixes $r :: ('a, 's)$ steps
assumes $\bigwedge v \ a \ s \ s'. (v, a, s, s') \in F \implies (a, s, s') \in r \vee s = s'$
shows $\text{spec.action } F \leq \text{spec.rel } r$
 ⟨proof⟩

⟨ML⟩

lemma *star-le*:
assumes $S \leq \text{spec.rel } r$
shows $\text{spec.kleene.star } S \leq \text{spec.rel } r$
 ⟨proof⟩

⟨ML⟩

lemma *relL-le*:
shows $g \ x \leq \text{spec.rel } r \gg g$
 ⟨proof⟩

lemma *relR-le*:

shows $f \leq f \ggg \text{spec.rel } r$
<proof>

lemma *inf-rel*:

shows $(f \ggg g) \sqcap \text{spec.rel } r = (\text{spec.rel } r \sqcap f) \ggg (\lambda x. \text{spec.rel } r \sqcap g x)$ (**is** *?thesis1*)
and $\text{spec.rel } r \sqcap (f \ggg g) = (\text{spec.rel } r \sqcap f) \ggg (\lambda x. \text{spec.rel } r \sqcap g x)$ (**is** *?lhs = ?rhs*)
<proof>

lemma *inf-rel-distr-le*:

shows $(f \sqcap \text{spec.rel } r) \ggg (\lambda v. g_1 v \sqcap g_2) \leq (f \ggg g_1) \sqcap (\text{spec.rel } r \ggg (\lambda :: \text{unit}. g_2))$
<proof>

<ML>

lemma *inf-rel*:

shows $\langle \sigma \rangle \sqcap \text{spec.rel } r = \bigsqcup (\text{spec.singleton } ' \{ \sigma'. \sigma' \leq \sigma \wedge \text{trace.steps } \sigma' \subseteq r \})$ (**is** *?lhs = ?rhs*)
and $\text{spec.rel } r \sqcap \langle \sigma \rangle = \bigsqcup (\text{spec.singleton } ' \{ \sigma'. \sigma' \leq \sigma \wedge \text{trace.steps } \sigma' \subseteq r \})$ (**is** *?thesis2*)
<proof>

<ML>

lemma *inf-rel*:

fixes $F :: ('v \times 'a \times 's \times 's)$ *set*
fixes $r :: ('a, 's)$ *steps*
assumes $\bigwedge a. \text{refl } (r \text{ `` } \{a\})$
shows $\text{spec.action } F \sqcap \text{spec.rel } r = \text{spec.action } (F \cap \text{UNIV} \times r)$ (**is** *?lhs = ?rhs*)
and $\text{spec.rel } r \sqcap \text{spec.action } F = \text{spec.action } (F \cap \text{UNIV} \times r)$ (**is** *?thesis1*)
<proof>

lemma *inf-rel-reflcl*:

shows $\text{spec.action } F \sqcap \text{spec.rel } r = \text{spec.action } (F \cap \text{UNIV} \times (r \cup \text{UNIV} \times \text{Id}))$
and $\text{spec.rel } r \sqcap \text{spec.action } F = \text{spec.action } (F \cap \text{UNIV} \times (r \cup \text{UNIV} \times \text{Id}))$
<proof>

<ML>

lemma *inf-rel*:

shows $\text{spec.rel } r \sqcap \text{spec.return } v = \text{spec.return } v$
and $\text{spec.return } v \sqcap \text{spec.rel } r = \text{spec.return } v$
<proof>

<ML>

lemma *inf-rel*:

shows $\text{spec.kleene.star } P \sqcap \text{spec.rel } r = \text{spec.kleene.star } (P \sqcap \text{spec.rel } r)$ (**is** *?lhs = ?rhs*)
<proof>

<ML>

lemma *simps[simp]*:

shows $(a, s, s) \notin \text{spec.steps } P$
<proof>

lemma *member-conv*:

shows $x \in \text{spec.steps } P \longleftrightarrow (\exists \sigma. \langle \sigma \rangle \leq P \wedge x \in \text{trace.steps } \sigma)$
<proof>

$\langle ML \rangle$

lemma *none*:

shows $spec.steps (spec.term.none P) = spec.steps P$

$\langle proof \rangle$

lemma *all*:

shows $spec.steps (spec.term.all P) = spec.steps P$

$\langle proof \rangle$

$\langle ML \rangle$

lemmas $bot = spec.rel.lower-bot$

lemmas $monotone = spec.rel.monotone-lower$

lemmas $mono = monotoneD[OF spec.steps.monotone]$

lemmas $Sup = spec.rel.lower-Sup$

lemmas $sup = spec.rel.lower-sup$

lemmas $Inf-le = spec.rel.lower-Inf-le$

lemmas $inf-le = spec.rel.lower-inf-le$

lemma *singleton*:

shows $spec.steps \langle \sigma \rangle = trace.steps \sigma$

$\langle proof \rangle$

lemma *idle*:

shows $spec.steps spec.idle = \{\}$

$\langle proof \rangle$

lemma *action*:

shows $spec.steps (spec.action F) = snd ' F - UNIV \times Id$

$\langle proof \rangle$

lemma *return*:

shows $spec.steps (spec.return v) = \{\}$

$\langle proof \rangle$

lemma *bind-le*: — see $spec.steps.bind$

shows $spec.steps (f \ggg g) \subseteq spec.steps f \cup (\bigcup v. spec.steps (g v))$

$\langle proof \rangle$

lemma *kleene-star*:

shows $spec.steps (spec.kleene.star P) = spec.steps P$ (**is** $?lhs = ?rhs$)

$\langle proof \rangle$

lemma *map*:

shows $spec.steps (spec.map af sf vf P)$

$= map-prod af (map-prod sf sf) ' spec.steps P - UNIV \times Id$

$\langle proof \rangle$

lemma *invmap-le*:

shows $spec.steps (spec.invmap af sf vf P)$

$\subseteq map-prod af (map-prod sf sf) - ' (spec.steps (P \sqcap spec.rel (range af \times range sf \times range sf)) \cup UNIV \times Id) - UNIV \times Id$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *monomorphic*:

fixes $r :: ('a, 's) \text{ steps}$

shows $\text{spec.steps } (\text{spec.rel.monomorphic } r) = r - \text{UNIV} \times \text{Id}$ (**is** $?lhs = ?rhs$)

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *rel*:

fixes $r :: ('a, 's) \text{ steps}$

shows $\text{spec.steps } (\text{spec.rel } r) = r - \text{UNIV} \times \text{Id}$

$\langle \text{proof} \rangle$

lemma *top*:

shows $\text{spec.steps } \top = \text{UNIV} \times - \text{Id}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

8.11 Sequential assertions

We specify sequential behavior with preconditions and postconditions.

8.11.1 Preconditions

$\langle ML \rangle$

definition $\text{pre} :: 's \text{ pred} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$\text{pre } P = \sqcup (\text{spec.singleton } \{ \sigma. P (\text{trace.init } \sigma) \})$

$\langle ML \rangle$

lemma $\text{pre-le-conv}[\text{spec.singleton.le-conv}]$:

shows $\langle \sigma \rangle \leq \text{spec.pre } P \longleftrightarrow P (\text{trace.init } \sigma)$

$\langle \text{proof} \rangle$

lemma *inf-pre*:

shows $\text{spec.pre } P \sqcap \langle \sigma \rangle = (\text{if } P (\text{trace.init } \sigma) \text{ then } \langle \sigma \rangle \text{ else } \perp)$ (**is** $?thesis1$)

and $\langle \sigma \rangle \sqcap \text{spec.pre } P = (\text{if } P (\text{trace.init } \sigma) \text{ then } \langle \sigma \rangle \text{ else } \perp)$ (**is** $?thesis2$)

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma $\text{pre-le-conv}[\text{spec.idle-le}]$:

shows $\text{spec.idle} \leq (\text{spec.pre } P :: ('a, 's, 'v) \text{ spec}) \longleftrightarrow P = \top$ (**is** $?lhs \longleftrightarrow ?rhs$)

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *pre*:

shows $\text{spec.term.all } (\text{spec.pre } P) = \text{spec.pre } P$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *pre*:

shows $\text{spec.pre } P \in \text{spec.term.closed}$ -

$\langle proof \rangle$

lemma *none-inf-pre*:

fixes $P :: 's \text{ pred}$

fixes $Q :: ('a, 's, 'v) \text{ spec}$

shows $\text{spec.term.none } (Q \sqcap \text{spec.pre } P) = (\text{spec.term.none } Q \sqcap \text{spec.pre } P :: ('a, 's, 'v) \text{ spec})$ (**is** *?lhs = ?rhs*)

and $\text{spec.term.none } (\text{spec.pre } P \sqcap Q) = (\text{spec.pre } P \sqcap \text{spec.term.none } Q :: ('a, 's, 'v) \text{ spec})$ (**is** *?thesis2*)

$\langle proof \rangle$

$\langle ML \rangle$

lemma *bot[iff]*:

shows $\text{spec.pre } \langle False \rangle = \perp$

and $\text{spec.pre } \perp = \perp$

$\langle proof \rangle$

lemma *top[iff]*:

shows $\text{spec.pre } \langle True \rangle = \top$

and $\text{spec.pre } \top = \top$

$\langle proof \rangle$

lemma *top-conv*:

shows $\text{spec.pre } P = (\top :: ('a, 's, 'v) \text{ spec}) \longleftrightarrow P = \top$

$\langle proof \rangle$

lemma *K*:

shows $\text{spec.pre } \langle P \rangle = (\text{if } P \text{ then } \top \text{ else } \perp)$

$\langle proof \rangle$

lemma *monotone*:

shows *mono spec.pre*

$\langle proof \rangle$

lemmas *strengthen[strg] = st-monotone[OF spec.pre.monotone]*

lemmas *mono = monotoneD[OF spec.pre.monotone]*

lemma *SUP*:

shows $\text{spec.pre } (\bigsqcup x \in X. P x) = (\bigsqcup x \in X. \text{spec.pre } (P x))$

$\langle proof \rangle$

lemma *Sup*:

shows $\text{spec.pre } (\bigsqcup X) = (\bigsqcup x \in X. \text{spec.pre } x)$

$\langle proof \rangle$

lemma *Bex*:

shows $\text{spec.pre } (\lambda s. \exists x \in X. P x s) = (\bigsqcup x \in X. \text{spec.pre } (P x))$

$\langle proof \rangle$

lemma *Ex*:

shows $\text{spec.pre } (\lambda s. \exists x. P x s) = (\bigsqcup x. \text{spec.pre } (P x))$

$\langle proof \rangle$

lemma

shows *disj: spec.pre (P \vee Q) = spec.pre P \sqcup spec.pre Q*

and *sup: spec.pre (P \sqcup Q) = spec.pre P \sqcup spec.pre Q*

$\langle proof \rangle$

lemma *INF*:

shows $spec.pre (\prod x \in X. P x) = (\prod x \in X. spec.pre (P x))$
 ⟨proof⟩

lemma *Inf*:

shows $spec.pre (\prod X) = (\prod x \in X. spec.pre x)$
 ⟨proof⟩

lemma *Ball*:

shows $spec.pre (\lambda s. \forall x \in X. P x s) = (\prod x \in X. spec.pre (P x))$
 ⟨proof⟩

lemma *All*:

shows $spec.pre (\lambda s. \forall x. P x s) = (\prod x. spec.pre (P x))$
 ⟨proof⟩

lemma *inf*:

shows *conj*: $spec.pre (P \wedge Q) = spec.pre P \sqcap spec.pre Q$
and $spec.pre (P \sqcap Q) = spec.pre P \sqcap spec.pre Q$
 ⟨proof⟩

lemma *inf-action-le*: — Converse does not hold

shows $spec.pre P \sqcap spec.action F \leq spec.action (UNIV \times UNIV \times Collect P \times UNIV \cap F)$ (**is** *?lhs ≤ ?rhs*)
and $spec.action F \sqcap spec.pre P \leq spec.action (F \cap UNIV \times UNIV \times Collect P \times UNIV)$ (**is** *?thesis2*)
 ⟨proof⟩

⟨ML⟩

lemma *pre*:

shows $spec.invmap af sf vf (spec.pre P) = spec.pre (\lambda s. P (sf s))$
 ⟨proof⟩

⟨ML⟩

lemma *inf-pre*:

shows $spec.pre P \sqcap (f \ggg g) = (spec.pre P \sqcap f) \ggg g$ (**is** *?lhs = ?rhs*)
and $(f \ggg g) \sqcap spec.pre P = (f \sqcap spec.pre P) \ggg g$ (**is** *?thesis1*)
 ⟨proof⟩

⟨ML⟩

lemma *pre*:

assumes $P s_0$
shows $spec.steps (spec.pre P :: ('a, 's, 'v) spec) = UNIV \times - Id$
 ⟨proof⟩

⟨ML⟩

8.11.2 Postconditions

Unlike $spec.pre$ $spec.post$ can be expressed in terms of other constants.

⟨ML⟩

definition $act :: ('v \Rightarrow 's pred) \Rightarrow ('v \times 'a \times 's \times 's)$ set **where**

$act Q = \{(v, a, s, s') \mid v a s s'. Q v s'\}$

⟨ML⟩

lemma $simps[simp]$:

shows $\text{spec.post.act } \langle\langle \text{False} \rangle\rangle = \{\}$
and $\text{spec.post.act } \langle\perp\rangle = \{\}$
and $\text{spec.post.act } \perp = \{\}$
and $\text{spec.post.act } \langle\langle \text{True} \rangle\rangle = \text{UNIV}$
and $\text{spec.post.act } \langle\top\rangle = \text{UNIV}$
and $\text{spec.post.act } \top = \text{UNIV}$
and $\text{spec.post.act } (Q \sqcup Q') = \text{spec.post.act } Q \cup \text{spec.post.act } Q'$
and $\text{spec.post.act } (\bigsqcup X) = (\bigcup x \in X. \text{spec.post.act } x)$
and $\text{spec.post.act } (\lambda v. \bigsqcup x \in Y. R \ x \ v) = (\bigcup x \in Y. \text{spec.post.act } (R \ x))$
 $\langle\text{proof}\rangle$

lemma *monotone*:
shows *mono spec.post.act*
 $\langle\text{proof}\rangle$

lemmas *strengthen[strg] = st-monotone[OF spec.post.act.monotone]*
lemmas *mono = monotoneD[OF spec.post.act.monotone]*
 $\langle\text{ML}\rangle$

definition *post* :: $('v \Rightarrow 's \text{ pred}) \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**
 $\text{post } Q = \top \gg (\lambda :: \text{unit}. \text{spec.action } (\text{spec.post.act } Q))$
 $\langle\text{ML}\rangle$

lemma *post-le-conv[spec.singleton.le-conv]*:
fixes $Q :: 'v \Rightarrow 's \text{ pred}$
shows $\langle\sigma\rangle \leq \text{spec.post } Q$
 $\longleftrightarrow (\text{case trace.term } \sigma \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } v \Rightarrow Q \ v \ (\text{trace.final } \sigma)) \text{ (is ?lhs } \longleftrightarrow \text{ ?rhs)}$
 $\langle\text{proof}\rangle$
 $\langle\text{ML}\rangle$

lemma *post-le[spec.idle-le]*:
shows $\text{spec.idle} \leq \text{spec.post } Q$
 $\langle\text{proof}\rangle$
 $\langle\text{ML}\rangle$

lemma *post-le*:
shows $\text{spec.term.none } P \leq \text{spec.post } Q$
 $\langle\text{proof}\rangle$

lemma *post*:
shows $\text{spec.term.none } (\text{spec.post } Q :: ('a, 's, 'v) \text{ spec})$
 $= \text{spec.term.none } (\top :: ('a, 's, \text{unit}) \text{ spec})$
 $\langle\text{proof}\rangle$
 $\langle\text{ML}\rangle$

lemma *post*:
shows $\text{spec.term.all } (\text{spec.post } Q) = \top$
 $\langle\text{proof}\rangle$
 $\langle\text{ML}\rangle$

lemma *bot[iff]*:
shows $\text{spec.post } \langle\langle \text{False} \rangle\rangle = \text{spec.term.none } (\top :: (-, -, \text{unit}) \text{ spec})$

and $\text{spec.post } \langle \perp \rangle = \text{spec.term.none } (\top :: (-, -, \text{unit}) \text{ spec})$
and $\text{spec.post } \perp = \text{spec.term.none } (\top :: (-, -, \text{unit}) \text{ spec})$
 $\langle \text{proof} \rangle$

lemma monotone:
shows mono spec.post
 $\langle \text{proof} \rangle$

lemmas $\text{strengthen}[\text{strg}] = \text{st-monotone}[\text{OF spec.post.monotone}]$
lemmas $\text{mono} = \text{monotoneD}[\text{OF spec.post.monotone}]$

lemma SUP-not-empty:
fixes $X :: 'a \text{ set}$
fixes $Q :: 'a \Rightarrow 'v \Rightarrow 's \text{ pred}$
assumes $X \neq \{\}$
shows $\text{spec.post } (\lambda v. \bigsqcup_{x \in X}. Q \ x \ v) = (\bigsqcup_{x \in X}. \text{spec.post } (Q \ x))$
 $\langle \text{proof} \rangle$

lemma disj:
shows $\text{spec.post } (Q \sqcup Q') = \text{spec.post } Q \sqcup \text{spec.post } Q'$
and $\text{spec.post } (\lambda rv. Q \ rv \sqcup Q' \ rv) = \text{spec.post } Q \sqcup \text{spec.post } Q'$
and $\text{spec.post } (\lambda rv. Q \ rv \vee Q' \ rv) = \text{spec.post } Q \sqcup \text{spec.post } Q'$
 $\langle \text{proof} \rangle$

lemma INF:
shows $\text{spec.post } (\prod_{x \in X}. Q \ x) = (\prod_{x \in X}. \text{spec.post } (Q \ x))$
and $\text{spec.post } (\lambda v. \prod_{x \in X}. Q \ x \ v) = (\prod_{x \in X}. \text{spec.post } (Q \ x))$
and $\text{spec.post } (\lambda v \ s. \prod_{x \in X}. Q \ x \ v \ s) = (\prod_{x \in X}. \text{spec.post } (Q \ x))$
 $\langle \text{proof} \rangle$

lemma Inf:
shows $\text{spec.post } (\prod X) = (\prod_{x \in X}. \text{spec.post } x)$
 $\langle \text{proof} \rangle$

lemma Ball:
shows $\text{spec.post } (\lambda v \ s. \forall x \in X. Q \ x \ v \ s) = (\prod_{x \in X}. \text{spec.post } (Q \ x))$
 $\langle \text{proof} \rangle$

lemma All:
shows $\text{spec.post } (\lambda v \ s. \forall x. Q \ x \ v \ s) = (\prod x. \text{spec.post } (Q \ x))$
 $\langle \text{proof} \rangle$

lemma inf:
shows $\text{spec.post } (Q \sqcap Q') = \text{spec.post } Q \sqcap \text{spec.post } Q'$
and $\text{spec.post } (\lambda rv. Q \ rv \sqcap Q' \ rv) = \text{spec.post } Q \sqcap \text{spec.post } Q'$
and $\text{conj: spec.post } (\lambda rv. Q \ rv \wedge Q' \ rv) = \text{spec.post } Q \sqcap \text{spec.post } Q'$
 $\langle \text{proof} \rangle$

lemma top[iff]:
shows $\text{spec.post } \langle \langle \text{True} \rangle \rangle = \top$
and $\text{spec.post } \langle \top \rangle = \top$
and $\text{spec.post } \top = \top$
 $\langle \text{proof} \rangle$

lemma top-conv:
shows $\text{spec.post } Q = (\top :: ('a, 's, 'v) \text{ spec}) \longleftrightarrow Q = \top$
 $\langle \text{proof} \rangle$

lemma K:

shows $\text{spec.post } (\lambda - . Q) = (\text{if } Q \text{ then } \top \text{ else } \top \gg= (\lambda :: \text{unit}. \perp))$
<proof>

<ML>

lemma bind-post-pre:

shows $f \sqcap \text{spec.post } Q \gg= g = f \gg= (\lambda v. g \ v \sqcap \text{spec.pre } (Q \ v))$ (**is** ?lhs = ?rhs)
and $\text{spec.post } Q \sqcap f \gg= g = f \gg= (\lambda v. \text{spec.pre } (Q \ v) \sqcap g \ v)$ (**is** ?thesis1)
<proof>

<ML>

lemma post:

shows $\text{spec.invmap } af \ sf \ vf \ (\text{spec.post } Q) = \text{spec.post } (\lambda v \ s. Q \ (vf \ v) \ (sf \ s))$
<proof>

<ML>

lemma post-le-conv:

shows $\text{spec.action } F \leq \text{spec.post } Q \longleftrightarrow (\forall v \ a \ s \ s'. (v, a, s, s') \in F \longrightarrow Q \ v \ s')$
<proof>

<ML>

lemma post-le:

assumes $\bigwedge v. g \ v \leq \text{spec.post } Q$
shows $f \gg= g \leq \text{spec.post } Q$
<proof>

lemma inf-post:

shows $(f \gg= g) \sqcap \text{spec.post } Q = f \gg= (\lambda v. g \ v \sqcap \text{spec.post } Q)$ (**is** ?lhs = ?rhs)
and $\text{spec.post } Q \sqcap (f \gg= g) = f \gg= (\lambda v. \text{spec.post } Q \sqcap g \ v)$ (**is** ?thesis2)
<proof>

lemma mono-stronger:

assumes $f: f \leq f' \sqcap \text{spec.post } Q$
assumes $g: \bigwedge v. g \ v \sqcap \text{spec.pre } (Q \ v) \leq g' \ v$
shows $\text{spec.bind } f \ g \leq \text{spec.bind } f' \ g'$
<proof>

<ML>

8.11.3 Strongest postconditions

<ML>

definition *strongest* :: $(\ 'a, \ 's, \ 'v) \text{ spec} \Rightarrow \ 'v \Rightarrow \ 's \text{ pred}$ **where**
 $\text{strongest } P = \bigcap \{ Q. P \leq \text{spec.post } Q \}$

interpretation *strongest*: *galois.complete-lattice-class spec.post.strongest spec.post*
<proof>

lemma strongest-alt-def:

shows $\text{spec.post.strongest } P = (\lambda v \ s. \exists \sigma. \langle \sigma \rangle \leq P \wedge \text{trace.term } \sigma = \text{Some } v \wedge \text{trace.final } \sigma = s)$ (**is** ?lhs = ?rhs)
<proof>

$\langle ML \rangle$

lemma *singleton*:

shows $spec.post.strongest \langle \sigma \rangle$

$= (\lambda v s. case\ trace.term\ \sigma\ of\ None \Rightarrow False \mid Some\ v' \Rightarrow v' = v \wedge trace.final\ \sigma = s)$

$\langle proof \rangle$

lemmas *monotone* = $spec.post.strongest.monotone-lower$

lemmas *mono* = $monoD[OF\ spec.post.strongest.monotone]$

lemmas *Sup* = $spec.post.strongest.lower-Sup$

lemmas *sup* = $spec.post.strongest.lower-sup$

lemma *top[iff]*:

shows $spec.post.strongest \top = \top$

$\langle proof \rangle$

lemma *action*:

shows $spec.post.strongest (spec.action\ F) = (\lambda v s'. \exists a s. (v, a, s, s') \in F)$

$\langle proof \rangle$

lemma *return*:

shows $spec.post.strongest (spec.return\ v) = (\lambda v' s. v' = v)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *none*:

shows $spec.post.strongest (spec.term.none\ P) = \perp$

$\langle proof \rangle$

lemma *all*:

assumes $spec.idle \leq P$

shows $spec.post.strongest (spec.term.all\ P) = \top$

$\langle proof \rangle$

lemma *closed*:

assumes $spec.idle \leq P$

assumes $P \in spec.term.closed -$

shows $spec.post.strongest\ P = \top$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *bind*:

shows $spec.post.strongest (f \ggg g)$

$= spec.post.strongest (\bigsqcup v. spec.pre (spec.post.strongest\ f\ v) \sqcap g\ v) \text{ (is ?lhs = ?rhs)}$

$\langle proof \rangle$

lemma *rel*:

shows $spec.post.strongest (spec.rel\ r) = \top$

$\langle proof \rangle$

lemma *pre*:

shows $spec.post.strongest (spec.pre\ P) = (\lambda v s'. \exists s. P\ s)$

$\langle proof \rangle$

lemma *post*:

shows $spec.post.strongest (spec.post\ Q) = Q$

$\langle proof \rangle$

$\langle ML \rangle$

8.12 Initial steps

The initial transition of a process.

$\langle ML \rangle$

definition *initial-steps* :: $(\prime a, \prime s, \prime v)$ *spec* \Rightarrow $(\prime a, \prime s)$ *steps* **where**
initial-steps $P = \{(a, s, s'). \langle s, [(a, s')], None \rangle \leq P\}$

$\langle ML \rangle$

lemma *steps-le*:

shows *spec.initial-steps* $P \subseteq$ *spec.steps* $P \cup UNIV \times Id$

$\langle proof \rangle$

lemma *galois*:

shows $r \subseteq$ *spec.initial-steps* $P \wedge$ *spec.idle* $\leq P \iff$ *spec.action* $(\{\{\}\} \times r) \gg= \perp \leq P$ (**is** ?lhs \iff ?rhs)

$\langle proof \rangle$

lemma *bot*:

shows *spec.initial-steps* $\perp = \{\}$

$\langle proof \rangle$

lemma *top*:

shows *spec.initial-steps* $\top = UNIV$

$\langle proof \rangle$

lemma *monotone*:

shows *mono spec.initial-steps*

$\langle proof \rangle$

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.initial-steps.monotone*]

lemmas *mono* = *monotoneD*[*OF spec.initial-steps.monotone*]

lemma *Sup*:

shows *spec.initial-steps* $(\bigsqcup X) = \bigcup (\textit{spec.initial-steps } \prime X)$

$\langle proof \rangle$

lemma *Inf*:

shows *spec.initial-steps* $(\bigsqcap X) = \bigcap (\textit{spec.initial-steps } \prime X)$

$\langle proof \rangle$

lemma *idle*:

shows *spec.initial-steps spec.idle* = $UNIV \times Id$

$\langle proof \rangle$

lemma *action*:

shows *spec.initial-steps (spec.action F)* = *snd* $\prime F \cup UNIV \times Id$

$\langle proof \rangle$

lemma *return*:

shows *spec.initial-steps (spec.return v)* = $UNIV \times Id$

$\langle proof \rangle$

lemma *bind*:

shows $\text{spec.initial-steps } (f \ggg g)$
 $= \text{spec.initial-steps } f$
 $\cup \text{spec.initial-steps } (\sqcup v. \text{spec.pre } (\text{spec.post.strongest } (f \sqcap \text{spec.return } v) v) \sqcap g v)$ (**is** $?lhs = ?rhs$)
 $\langle \text{proof} \rangle$

lemma *rel*:

shows $\text{spec.initial-steps } (\text{spec.rel } r) = r \cup UNIV \times Id$
 $\langle \text{proof} \rangle$

lemma *pre*:

shows $\text{spec.initial-steps } (\text{spec.pre } P) = UNIV \times Pre P$
 $\langle \text{proof} \rangle$

lemma *post*:

shows $\text{spec.initial-steps } (\text{spec.post } Q) = UNIV$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *none*:

shows $\text{spec.initial-steps } (\text{spec.term.none } P) = \text{spec.initial-steps } P$
 $\langle \text{proof} \rangle$

lemma *all*:

shows $\text{spec.initial-steps } (\text{spec.term.all } P) = \text{spec.initial-steps } P$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

8.13 Heyting implication

$\langle ML \rangle$

lemma *heyting-le-conv*:

shows $\langle \sigma \rangle \leq P \longrightarrow_H Q \iff (\forall \sigma' \leq \sigma. \langle \sigma' \rangle \leq P \longrightarrow \langle \sigma' \rangle \leq Q)$ (**is** $?lhs \iff ?rhs$)
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

Connect the generic definition of Heyting implication to a concrete one in the model.

lift-definition *heyting* :: $(\text{'a}, \text{'s}, \text{'v}) \text{spec} \Rightarrow (\text{'a}, \text{'s}, \text{'v}) \text{spec} \Rightarrow (\text{'a}, \text{'s}, \text{'v}) \text{spec}$ **is**

downwards.imp

$\langle \text{proof} \rangle$

lemma *heyting-alt-def*:

shows $(\longrightarrow_H) = (\text{spec.heyting} :: \text{-}\Rightarrow\text{-}\Rightarrow (\text{'a}, \text{'s}, \text{'v}) \text{spec})$
 $\langle \text{proof} \rangle$

declare *spec.heyting.transfer*[*transfer-rule del*]

$\langle ML \rangle$

lemma *transfer-alt*[*transfer-rule*]:

shows $\text{rel-fun } (\text{pcr-spec } (=) (=) (=)) (\text{rel-fun } (\text{pcr-spec } (=) (=) (=)) (\text{pcr-spec } (=) (=) (=))) \text{downwards.imp}$
 (\longrightarrow_H)

$\langle \text{proof} \rangle$

An example due to [Abadi and Merz \(1995, p504\)](#) where the (TLA) model validates a theorem that is not intuitionistically valid. This is “some kind of linearity” and intuitively encodes disjunction elimination.

lemma *linearity*:

fixes $Q :: (-, -, -)$ *spec*

shows $((P \longrightarrow_H Q) \longrightarrow_H R) \sqcap ((Q \longrightarrow_H P) \longrightarrow_H R) \leq R$

\langle *proof* \rangle

lemma *SupR*:

fixes $P :: (-, -, -)$ *spec*

assumes $X \neq \{\}$

shows $P \longrightarrow_H (\bigsqcup_{x \in X}. Q\ x) = (\bigsqcup_{x \in X}. P \longrightarrow_H Q\ x)$ (**is** $?lhs = ?rhs$)

\langle *proof* \rangle

lemma *cont*:

fixes $P :: (-, -, -)$ *spec*

shows $cont\ Sup (\leq)\ Sup (\leq) ((\longrightarrow_H)\ P)$

\langle *proof* \rangle

lemma *mcont*:

fixes $P :: (-, -, -)$ *spec*

shows $mcont\ Sup (\leq)\ Sup (\leq) ((\longrightarrow_H)\ P)$

\langle *proof* \rangle

lemmas $mcont2mcont[cont-intro] = mcont2mcont[OF\ spec.heyting.mcont, of\ luba\ orda\ Q\ P]$ **for** $luba\ orda\ Q\ P$

lemma *non-triv*:

shows $P \longrightarrow_H \perp \leq P \longleftrightarrow spec.idle \leq P$ (**is** $?lhs \longleftrightarrow ?rhs$)

\langle *proof* \rangle

lemma *post*:

shows $spec.post\ Q \longrightarrow_H spec.post\ Q' = spec.post\ (\lambda v\ s. Q\ v\ s \longrightarrow Q'\ v\ s)$ (**is** $?lhs = ?rhs$)

\langle *proof* \rangle

\langle *ML* \rangle

lemma *heyting*:

shows $spec.invmap\ af\ sf\ vf\ (P \longrightarrow_H Q) = spec.invmap\ af\ sf\ vf\ P \longrightarrow_H spec.invmap\ af\ sf\ vf\ Q$ (**is** $?lhs = ?rhs$)

\langle *proof* \rangle

\langle *ML* \rangle

lemma *heyting-noneL-allR-mono*:

fixes $P :: (-, -, 'v)$ *spec*

fixes $Q :: (-, -, 'v)$ *spec*

shows $spec.term.none\ P \longrightarrow_H Q = P \longrightarrow_H spec.term.all\ Q$ (**is** $?lhs = ?rhs$)

\langle *proof* \rangle

\langle *ML* \rangle

lemma *heyting*: — polymorphic *spec.term.all*

fixes $P :: (-, -, 'v)$ *spec*

fixes $Q :: (-, -, 'v)$ *spec*

shows $(spec.term.all\ (P \longrightarrow_H Q) :: (-, -, 'w)\ spec)$

$= spec.term.none\ P \longrightarrow_H spec.term.all\ Q$ (**is** $?lhs = ?rhs$)

\langle *proof* \rangle

\langle *ML* \rangle

lemma *heyting-le*:

shows $spec.term.none\ (P \longrightarrow_H Q) \leq spec.term.all\ P \longrightarrow_H spec.term.none\ Q$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *heyting*:

assumes $Q \in \text{spec.term.closed}$ -

shows $P \longrightarrow_H Q \in \text{spec.term.closed}$ -

$\langle \text{proof} \rangle$

$\langle ML \rangle$

8.14 Miscellaneous algebra

$\langle ML \rangle$

lemma *bind*:

shows $\text{spec.steps } f \gg= g$

$= \text{spec.steps } f \cup (\bigcup v. \text{spec.steps } (\text{spec.pre } (\text{spec.post.strongest } f v) \sqcap g v))$ (**is** $?lhs = ?rhs$)

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *idle*:

shows $\text{spec.map } af \ sf \ vf \ \text{spec.idle} = \text{spec.pre } (\lambda s. s \in \text{range } sf) \sqcap \text{spec.idle}$ (**is** $?lhs = ?rhs$)

$\langle \text{proof} \rangle$

lemma *return*:

fixes $F :: ('v \times 'a \times 's \times 's)$ *set*

shows $\text{spec.map } af \ sf \ vf \ (\text{spec.return } v)$

$= \text{spec.pre } (\lambda s. s \in \text{range } sf) \sqcap \text{spec.return } (vf v)$ (**is** $?lhs = ?rhs$)

$\langle \text{proof} \rangle$

lemma *kleene-star-le*:

fixes $P :: ('a, 's, \text{unit})$ *spec*

fixes $af :: 'a \Rightarrow 'b$

fixes $sf :: 's \Rightarrow 't$

fixes $vf :: \text{unit} \Rightarrow \text{unit}$

shows $\text{spec.map } af \ sf \ vf \ (\text{spec.kleene.star } P) \leq \text{spec.kleene.star } (\text{spec.map } af \ sf \ vf \ P)$ (**is** $- \leq ?rhs$)

$\langle \text{proof} \rangle$

lemma *rel-le*:

shows $\text{spec.map } af \ sf \ vf \ (\text{spec.rel } r) \leq \text{spec.rel } (\text{map-prod } af \ (\text{map-prod } sf \ sf) \ ' r)$

$\langle \text{proof} \rangle$

General lemmas for spec.map are elusive. We relate it to spec.rel , spec.pre and spec.post under a somewhat weak constraint. Intuitively we ask that, for distinct representations (s_0 and s_0') of an abstract state ($sf \ s_0$ where $sf \ s_0' = sf \ s_0$), if agent a can evolve s_0 to s_1 according to r ($(a, s_0, s_1) \in r$) then there is an agent a' where $af \ a' = af \ a$ that can evolve s_0' to an s_1' which represents the same abstract state ($sf \ s_1' = sf \ s_1$).

All injective sf satisfy this condition.

context

fixes $af :: 'a \Rightarrow 'b$

fixes $sf :: 's \Rightarrow 't$

fixes $vf :: 'v \Rightarrow 'w$

begin

context

fixes $r :: ('a, 's)$ *steps*

assumes *step-cong*: $\forall a \ s_0 \ s_1 \ s_0'. (a, s_0, s_1) \in r \wedge sf \ s_1 \neq sf \ s_0 \wedge sf \ s_0' = sf \ s_0$

$$\longrightarrow (\exists a' s_1'. af a' = af a \wedge sf s_1' = sf s_1 \wedge (a', s_0', s_1') \in r)$$

begin

private lemma *map-relE[consumes 1]*:

fixes $xs :: ('b \times 't) \text{ list}$

assumes $trace.steps' s xs \subseteq map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf)\ 'r$

obtains $(Idle)\ snd\ 'set\ xs \subseteq \{s\}$

| $(Step)\ s'\ xs'$

where $sf\ s' = s$

and $trace.natural'\ s\ xs = map\ (map\text{-}prod\ af\ sf)\ xs'$

and $trace.steps'\ s'\ xs' \subseteq r$

$\langle proof \rangle$

lemma *rel*:

shows $spec.map\ af\ sf\ vf\ (spec.rel\ r)$

$= spec.rel\ (map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf)\ 'r)$

$\sqcap spec.pre\ (\lambda s. s \in range\ sf)$

$\sqcap spec.post\ (\lambda v\ s. v \in range\ vf)\ (\mathbf{is}\ ?lhs = ?rhs)$

$\langle proof \rangle$

lemma *pre*:

fixes $P :: 't\ pred$

shows $spec.map\ af\ sf\ vf\ (spec.pre\ (\lambda s. P\ (sf\ s)))$

$= spec.pre\ (\lambda s. P\ s \wedge s \in range\ sf) \sqcap spec.post\ (\lambda v\ s. s \in range\ sf \longrightarrow v \in range\ vf)$

$\sqcap spec.rel\ (range\ af \times range\ sf \times range\ sf)\ (\mathbf{is}\ ?lhs = ?rhs)$

$\langle proof \rangle$

lemma *post*:

fixes $Q :: 'w \Rightarrow 't\ pred$

shows $spec.map\ af\ sf\ vf\ (spec.post\ (\lambda v\ s. Q\ (vf\ v)\ (sf\ s)))$

$= spec.pre\ (\lambda s. s \in range\ sf) \sqcap spec.post\ (\lambda v\ s. s \in range\ sf \longrightarrow Q\ v\ s \wedge v \in range\ vf)$

$\sqcap spec.rel\ (range\ af \times range\ sf \times range\ sf)\ (\mathbf{is}\ ?lhs = ?rhs)$

$\langle proof \rangle$

end

end

$\langle ML \rangle$

lemma *idle*:

shows $spec.invmap\ af\ sf\ vf\ spec.idle$

$= spec.term.none\ (spec.rel\ (UNIV \times map\text{-}prod\ sf\ sf - 'Id) :: ('a, 's, unit)\ spec)\ (\mathbf{is}\ ?lhs = ?rhs)$

$\langle proof \rangle$

lemma *inf-rel*:

shows $spec.rel\ (map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf) - '(r \cup UNIV \times Id)) \sqcap spec.invmap\ af\ sf\ vf\ P = spec.invmap\ af\ sf\ vf\ (spec.rel\ r \sqcap P)$

and $spec.invmap\ af\ sf\ vf\ P \sqcap spec.rel\ (map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf) - '(r \cup UNIV \times Id)) = spec.invmap\ af\ sf\ vf\ (P \sqcap spec.rel\ r)$

$\langle proof \rangle$

lemma *action*: — (* could restrict the stuttering expansion to *range af* or an arbitrary element of that

fixes $af :: 'a \Rightarrow 'b$

fixes $sf :: 's \Rightarrow 't$

fixes $vf :: 'v \Rightarrow 'w$

fixes $F :: ('w \times 'b \times 't \times 't)\ set$

defines $F' \equiv map\text{-}prod\ id\ (map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf))$

$$- ' (F \cup \{(v, a', s, s) \mid v a a' s. (v, a, s, s) \in F \wedge \neg \text{surj } af\})$$

shows $\text{spec.invmap } af \text{ } sf \text{ } vf \text{ } (\text{spec.action } F)$
 $= \text{spec.rel } (UNIV \times \text{map-prod } sf \text{ } sf \text{ } - ' \text{ } Id)$
 $\gg= (\lambda :: \text{unit. spec.action } F')$
 $\gg= (\lambda v. \text{spec.rel } (UNIV \times \text{map-prod } sf \text{ } sf \text{ } - ' \text{ } Id)$
 $\gg= (\lambda :: \text{unit. } \sqcup v' \in vf \text{ } - ' \{v\}. \text{spec.return } v')$ (**is** $?lhs = ?rhs$)

$\langle \text{proof} \rangle$

lemma *return*:

fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$
fixes $F :: ('w \times 'b \times 't \times 't) \text{ set}$
shows $\text{spec.invmap } af \text{ } sf \text{ } vf \text{ } (\text{spec.return } v)$
 $= \text{spec.rel } (UNIV \times \text{map-prod } sf \text{ } sf \text{ } - ' \text{ } Id) \gg= (\lambda :: \text{unit. } \sqcup v' \in vf \text{ } - ' \{v\}. \text{spec.return } v')$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

9 Constructions in the ($'a$, $'s$, $'v$) *spec* lattice

9.1 Constrains-at-most

Abadi and Plotkin (1993, §3.1) require that processes to be composed in parallel *constrain at most* (CAM) distinct sets of agents: intuitively each process cannot block other processes from taking steps after any of its transitions. We model this as a closure.

See §9.2 for a discussion of their composition rules.

Observations:

- the sense of the relation r here is inverted wrt Abadi/Plotkin
- this is a key ingredient in interference closure (§9.3)
- this closure is antimatroidal

$\langle ML \rangle$

definition $cl :: ('a, 's) \text{ steps} \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$$cl \ r \ P = P \sqcup \text{spec.term.none } (\text{spec.term.all } P \gg= (\lambda :: \text{unit. spec.rel } r :: (-, -, \text{unit}) \text{ spec}))$$

$\langle ML \rangle$

lemma *cl*:

shows $\text{spec.term.none } (\text{spec.cam.cl } r \ P) = \text{spec.cam.cl } r \ (\text{spec.term.none } P)$

$\langle \text{proof} \rangle$

lemma *cl-rel-wind*:

fixes $P :: ('a, 's, 'v) \text{ spec}$
shows $\text{spec.cam.cl } r \ P \gg= \text{spec.term.none } (\text{spec.rel } r :: ('a, 's, 'w) \text{ spec})$
 $= \text{spec.term.none } (\text{spec.cam.cl } r \ P)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *cl-le*: — Converse does not hold

shows $\text{spec.cam.cl } r \ (\text{spec.term.all } P) \leq \text{spec.term.all } (\text{spec.cam.cl } r \ P)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

interpretation *cam: closure-complete-distrib-lattice-distributive-class spec.cam.cl r for r :: ('a, 's) steps*
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *bot[simp]:*
shows *spec.cam.cl r $\perp = \perp$*
 $\langle proof \rangle$

lemma *mono:*
fixes *r :: ('a, 's) steps*
assumes *r \subseteq r'*
assumes *P \leq P'*
shows *spec.cam.cl r P \leq spec.cam.cl r' P'*
 $\langle proof \rangle$

declare *spec.cam.strengthen-cl[strg del]*

lemma *strengthen[strg]:*
assumes *st-ord F r r'*
assumes *st-ord F P P'*
shows *st-ord F (spec.cam.cl r P) (spec.cam.cl r' P')*
 $\langle proof \rangle$

lemma *Sup:*
shows *spec.cam.cl r ($\bigsqcup X$) = ($\bigsqcup P \in X$. spec.cam.cl r P)*
 $\langle proof \rangle$

lemmas *sup = spec.cam.cl.Sup[where X={P, Q} for P Q, simplified]*

lemma *rel-empty:*
shows *spec.cam.cl {} P = P*
 $\langle proof \rangle$

lemma *rel-reflcl:*
shows *spec.cam.cl (r \cup A \times Id) P = spec.cam.cl r P*
and *spec.cam.cl (A \times Id \cup r) P = spec.cam.cl r P*
 $\langle proof \rangle$

lemma *rel-minus-Id:*
shows *spec.cam.cl (r - UNIV \times Id) P = spec.cam.cl r P*
 $\langle proof \rangle$

lemma *Inf:*
shows *spec.cam.cl r ($\prod X$) = \prod (spec.cam.cl r ' X) (is ?lhs = ?rhs)*
 $\langle proof \rangle$

lemmas *inf = spec.cam.cl.Inf[where X={P, Q} for P Q, simplified]*

lemma *idle:*
shows *spec.cam.cl r spec.idle = spec.term.none (spec.rel r :: (-, -, unit) spec)*
 $\langle proof \rangle$

lemma *bind:*
shows *spec.cam.cl r (f \ggg g) = spec.cam.cl r f \ggg (λv . spec.cam.cl r (g v))*
 $\langle proof \rangle$

lemma *action*:

fixes $r :: ('a, 's)$ *steps*

fixes $F :: ('v \times 'a \times 's \times 's)$ *set*

shows $\text{spec.cam.cl } r (\text{spec.action } F)$
 $= \text{spec.action } F$

$\sqcup \text{spec.term.none } (\text{spec.action } F \gg (\text{spec.rel } r :: (-, -, \text{unit}) \text{ spec}))$

$\sqcup \text{spec.term.none } (\text{spec.rel } r :: (-, -, \text{unit}) \text{ spec})$

$\langle \text{proof} \rangle$

lemma *return*:

shows $\text{spec.cam.cl } r (\text{spec.return } v) = \text{spec.return } v \sqcup \text{spec.term.none } (\text{spec.rel } r :: (-, -, \text{unit}) \text{ spec})$

$\langle \text{proof} \rangle$

lemma *rel-le*:

assumes $r \subseteq r' \vee r' \subseteq r$

shows $\text{spec.cam.cl } r (\text{spec.rel } r') \leq \text{spec.rel } (r \cup r')$

$\langle \text{proof} \rangle$

lemma *rel*:

assumes $r \subseteq r'$

shows $\text{spec.cam.cl } r (\text{spec.rel } r') = \text{spec.rel } r'$

$\langle \text{proof} \rangle$

lemma *inf-rel*:

fixes $r :: ('a, 's)$ *steps*

fixes $s :: ('a, 's)$ *steps*

fixes $P :: ('a, 's, 'v)$ *spec*

shows $\text{spec.rel } r \sqcap \text{spec.cam.cl } r' P = \text{spec.cam.cl } (r \cap r') (\text{spec.rel } r \sqcap P)$ (**is** *?thesis1*)

and $\text{spec.cam.cl } r' P \sqcap \text{spec.rel } r = \text{spec.cam.cl } (r \cap r') (\text{spec.rel } r \sqcap P)$ (**is** *?thesis2*)

$\langle \text{proof} \rangle$

lemma *bind-return*:

shows $\text{spec.cam.cl } r (f \gg \text{spec.return } v) = \text{spec.cam.cl } r f \gg \text{spec.return } v$

$\langle \text{proof} \rangle$

lemma *heyting-le*:

shows $\text{spec.cam.cl } r (P \longrightarrow_H Q) \leq P \longrightarrow_H \text{spec.cam.cl } r Q$

$\langle \text{proof} \rangle$

lemma *pre*:

shows $\text{spec.cam.cl } r (\text{spec.pre } P) = \text{spec.pre } P$

$\langle \text{proof} \rangle$

lemma *post*:

shows $\text{spec.cam.cl } r (\text{spec.post } Q) = \text{spec.post } Q$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *empty*:

shows $\text{spec.cam.closed } \{\} = \text{UNIV}$

$\langle \text{proof} \rangle$

lemma *antimonotone*:

shows *antimono spec.cam.closed*

$\langle \text{proof} \rangle$

lemmas *strengthen*[*strg*] = *st-ord-antimono*[*OF spec.cam.closed.antimonotone*]
lemmas *antimono* = *antimonoD*[*OF spec.cam.closed.antimonotone, of r r' for r r'*]

lemma *reflcl*:

shows *spec.cam.closed* ($r \cup A \times Id$) = *spec.cam.closed* r
 \langle *proof* \rangle

\langle *ML* \rangle

lemma *none*:

assumes $P \in \text{spec.cam.closed } r$
shows *spec.term.none* $P \in \text{spec.cam.closed } r$
 \langle *proof* \rangle

\langle *ML* \rangle

lemma *bind*:

fixes $f :: ('a, 's, 'v) \text{ spec}$
fixes $g :: 'v \Rightarrow ('a, 's, 'w) \text{ spec}$
assumes $f \in \text{spec.cam.closed } r$
assumes $\bigwedge x. g \ x \in \text{spec.cam.closed } r$
shows $f \gg g \in \text{spec.cam.closed } r$
 \langle *proof* \rangle

lemma *rel*[*intro*]:

assumes $r \subseteq r'$
shows *spec.rel* $r' \in \text{spec.cam.closed } r$
 \langle *proof* \rangle

lemma *pre*[*intro*]:

shows *spec.pre* $P \in \text{spec.cam.closed } r$
 \langle *proof* \rangle

lemma *post*[*intro*]:

shows *spec.post* $Q \in \text{spec.cam.closed } r$
 \langle *proof* \rangle

lemma *heyting*[*intro*]:

assumes $Q \in \text{spec.cam.closed } r$
shows $P \longrightarrow_H Q \in \text{spec.cam.closed } r$
 \langle *proof* \rangle

lemma *snoc-conv*:

fixes $P :: ('a, 's, 'v) \text{ spec}$
assumes $P \in \text{spec.cam.closed } r$
assumes $(fst \ x, trace.final' \ s \ xs, snd \ x) \in r \cup UNIV \times Id$
shows $\langle s, xs @ [x], None \rangle \leq P \longleftrightarrow \langle s, xs, None \rangle \leq P$ (**is** *?lhs* \longleftrightarrow *?rhs*)
 \langle *proof* \rangle

\langle *ML* \rangle

lemma *cl*:

fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$
fixes $r :: ('b, 't) \text{ steps}$
fixes $P :: ('b, 't, 'w) \text{ spec}$
shows *spec.invmap* $af \ sf \ vf$ (*spec.cam.cl* $r \ P$)

$= \text{spec.cam.cl } (\text{map-prod } af \ (\text{map-prod } sf \ sf) \ - \ ' (r \cup \text{UNIV} \times \text{Id})) \ (\text{spec.invmap } af \ sf \ vf \ P)$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *cl-le*:

fixes $af :: 'a \Rightarrow 'b$

fixes $sf :: 's \Rightarrow 't$

fixes $vf :: 'v \Rightarrow 'w$

fixes $r :: ('a, 's) \text{ steps}$

fixes $P :: ('a, 's, 'v) \text{ spec}$

shows $\text{spec.map } af \ sf \ vf \ (\text{spec.cam.cl } r \ P)$

$\leq \text{spec.cam.cl } (\text{map-prod } af \ (\text{map-prod } sf \ sf) \ ' r) \ (\text{spec.map } af \ sf \ vf \ P)$

$\langle \text{proof} \rangle$

lemma *cl-inj-sf*:

fixes $af :: 'a \Rightarrow 'b$

fixes $sf :: 's \Rightarrow 't$

fixes $vf :: 'v \Rightarrow 'w$

fixes $r :: ('a, 's) \text{ steps}$

fixes $P :: ('a, 's, 'v) \text{ spec}$

assumes *inj sf*

shows $\text{spec.map } af \ sf \ vf \ (\text{spec.cam.cl } r \ P)$

$= \text{spec.cam.cl } (\text{map-prod } af \ (\text{map-prod } sf \ sf) \ ' r) \ (\text{spec.map } af \ sf \ vf \ P)$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

9.2 Abadi and Plotkin's composition principle

Abadi and Plotkin (1991, 1993) develop a theory of circular reasoning about Heyting implication for safety properties under the mild condition that each is CAM-closed with respect to the other.

$\langle \text{ML} \rangle$

abbreviation *ap-cam-cl* $:: 'a \text{ set} \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$\text{ap-cam-cl } as \equiv \text{spec.cam.cl } ((-as) \times \text{UNIV})$

abbreviation (*input*) *ap-cam-closed* $:: 'a \text{ set} \Rightarrow ('a, 's, 'v) \text{ spec set}$ **where**

$\text{ap-cam-closed } as \equiv \text{spec.cam.closed } ((-as) \times \text{UNIV})$

lemma *composition-principle-1*:

fixes $P :: ('a, 's, 'v) \text{ spec}$

assumes $P \in \text{spec.ap-cam-closed } as$

assumes $P \in \text{spec.term.closed } -$

assumes $\text{spec.idle} \leq P$

shows $\text{spec.ap-cam-cl } (-as) \ P \longrightarrow_H P \leq P$ (**is** $?lhs \leq ?rhs$)

$\langle \text{proof} \rangle$

lemma *composition-principle-half*: — Abadi and Plotkin (1993, §3.1(4)) — cleaner than in Abadi and Plotkin (1991, §3.1)

assumes $M_1 \in \text{spec.ap-cam-closed } a_1$

assumes $M_2 \in \text{spec.ap-cam-closed } a_2$

assumes $M_1 \in \text{spec.term.closed } -$

assumes $\text{spec.idle} \leq M_1$

assumes $a_1 \cap a_2 = \{\}$

shows $(M_1 \longrightarrow_H M_2) \sqcap (M_2 \longrightarrow_H M_1) \leq M_1$

$\langle \text{proof} \rangle$

theorem *composition-principle*: — Abadi and Plotkin (1993, §3.1(3))

assumes $M_1 \in \text{spec.ap-cam-closed } a_1$
assumes $M_2 \in \text{spec.ap-cam-closed } a_2$
assumes $M_1 \in \text{spec.term.closed -}$
assumes $M_2 \in \text{spec.term.closed -}$
assumes $\text{spec.idle} \leq M_1$
assumes $\text{spec.idle} \leq M_2$
assumes $a_1 \cap a_2 = \{\}$
shows $(M_1 \longrightarrow_H M_2) \sqcap (M_2 \longrightarrow_H M_1) \leq M_1 \sqcap M_2$

<proof>

An infinitary variant can be established in essentially the same way as *spec.composition-principle-1*.

lemma *ag-circular*:

fixes $P_s :: 'a \Rightarrow ('a, 's, 'v) \text{ spec}$
assumes *cam-closed*: $\bigwedge a. a \in as \implies P_s a \in \text{spec.ap-cam-closed } \{a\}$
assumes *term-closed*: $\bigwedge a. a \in as \implies P_s a \in \text{spec.term.closed -}$
assumes *idle*: $\bigwedge a. a \in as \implies \text{spec.idle} \leq P_s a$
shows $(\prod a \in as. (\prod a' \in as - \{a\}. P_s a')) \longrightarrow_H P_s a \leq (\prod a \in as. P_s a)$ (**is** ?lhs \leq ?rhs)

<proof>

<ML>

9.3 Interference closure

We add environment interference to the beginnings and ends of behaviors for two reasons:

- it ensures the wellformedness of parallel composition as conjunction (see §9.5)
- it guarantees the monad laws hold (see §13.3.1)
 - *spec.cam.cl* by itself is too weak to justify these

We use this closure to build the program sublattice of the $('a, 's, 'v) \text{ spec}$ lattice (see §13).

Observations:

- if processes are made out of actions then it is not necessary to apply *spec.cam.cl*

<ML>

definition $cl :: ('a, 's) \text{ steps} \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$cl \ r \ P = \text{spec.rel } r \gg\gg (\lambda :: \text{unit}. \text{spec.cam.cl } r \ P) \gg\gg (\lambda v. \text{spec.rel } r \gg\gg (\lambda :: \text{unit}. \text{spec.return } v))$

<ML>

interpretation *interference: closure-complete-distrib-lattice-distributive-class* $\text{spec.interference.cl } r$
for $r :: ('a, 's) \text{ steps}$

<proof>

<ML>

lemma *cl*:

shows $\text{spec.term.none } (\text{spec.interference.cl } r \ P) = \text{spec.interference.cl } r \ (\text{spec.term.none } P)$

<proof>

<ML>

lemma *rel-le*:

assumes $P \in \text{spec.interference.closed } r$

shows $\text{spec.term.none } (\text{spec.rel } r) \leq P$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *cl-le*: — Converse does not hold

shows $\text{spec.interference.cl } r (\text{spec.term.all } P) \leq \text{spec.term.all } (\text{spec.interference.cl } r P)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *cl*:

shows $\text{spec.interference.cl } r P \in \text{spec.cam.closed } r$

$\langle \text{proof} \rangle$

lemma *closed-subseteq*:

shows $\text{spec.interference.closed } r \subseteq \text{spec.cam.closed } r$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *mono*:

assumes $r \subseteq r'$

assumes $P \leq P'$

shows $\text{spec.interference.cl } r P \leq \text{spec.interference.cl } r' P'$

$\langle \text{proof} \rangle$

declare $\text{spec.interference.strengthen-cl}[\text{strg del}]$

lemma *strengthen*[*strg*]:

assumes $\text{st-ord } F r r'$

assumes $\text{st-ord } F P P'$

shows $\text{st-ord } F (\text{spec.interference.cl } r P) (\text{spec.interference.cl } r' P')$

$\langle \text{proof} \rangle$

lemma *bot*:

shows $\text{spec.interference.cl } r \perp = \text{spec.term.none } (\text{spec.rel } r :: (-, -, \text{unit}) \text{ spec})$

$\langle \text{proof} \rangle$

lemmas $\text{Sup} = \text{spec.interference.cl-Sup}$

lemmas $\text{sup} = \text{spec.interference.cl-sup}$

lemma *idle*:

shows $\text{spec.interference.cl } r \text{spec.idle} = \text{spec.term.none } (\text{spec.rel } r :: (-, -, \text{unit}) \text{ spec})$

$\langle \text{proof} \rangle$

lemma *rel-empty*:

assumes $\text{spec.idle} \leq P$

shows $\text{spec.interference.cl } \{\} P = P$

$\langle \text{proof} \rangle$

lemma *rel-reflcl*:

shows $\text{spec.interference.cl } (r \cup A \times \text{Id}) P = \text{spec.interference.cl } r P$

and $\text{spec.interference.cl } (A \times \text{Id} \cup r) P = \text{spec.interference.cl } r P$

$\langle \text{proof} \rangle$

lemma *rel-minus-Id*:

shows $\text{spec.interference.cl } (r - \text{UNIV} \times \text{Id}) P = \text{spec.interference.cl } r P$

$\langle \text{proof} \rangle$

lemma *inf-rel*:

shows $\text{spec.interference.cl } s \sqcap P \sqcap \text{spec.rel } r = \text{spec.interference.cl } (r \sqcap s) (\text{spec.rel } r \sqcap P)$

and $\text{spec.rel } r \sqcap \text{spec.interference.cl } s \sqcap P = \text{spec.interference.cl } (r \sqcap s) (\text{spec.rel } r \sqcap P)$

$\langle \text{proof} \rangle$

lemma *bindL*:

assumes $f \in \text{spec.interference.closed } r$

shows $\text{spec.interference.cl } r (f \ggg g) = f \ggg (\lambda v. \text{spec.interference.cl } r (g v))$

$\langle \text{proof} \rangle$

lemma *bindR*:

assumes $\bigwedge v. g v \in \text{spec.interference.closed } r$

shows $\text{spec.interference.cl } r (f \ggg g) = \text{spec.interference.cl } r f \ggg g$ (**is** ?lhs = ?rhs)

$\langle \text{proof} \rangle$

lemma *bind-conv*:

assumes $f \in \text{spec.interference.closed } r$

assumes $\forall x. g x \in \text{spec.interference.closed } r$

shows $\text{spec.interference.cl } r (f \ggg g) = f \ggg g$

$\langle \text{proof} \rangle$

lemma *action*:

shows $\text{spec.interference.cl } r (\text{spec.action } F)$

$= \text{spec.rel } r \ggg (\lambda :: \text{unit}. \text{spec.action } F \ggg (\lambda v. \text{spec.rel } r \ggg (\lambda :: \text{unit}. \text{spec.return } v)))$

$\langle \text{proof} \rangle$

lemma *return*:

shows $\text{spec.interference.cl } r (\text{spec.return } v) = \text{spec.rel } r \ggg (\lambda :: \text{unit}. \text{spec.return } v)$

$\langle \text{proof} \rangle$

lemma *bind-return*:

shows $\text{spec.interference.cl } r (f \gg \text{spec.return } v) = \text{spec.interference.cl } r f \gg \text{spec.return } v$

$\langle \text{proof} \rangle$

lemma *rel*: — complicated by polymorphic *spec.rel*

assumes $r \subseteq r' \vee r' \subseteq r$

shows $\text{spec.interference.cl } r (\text{spec.rel } r') = \text{spec.rel } (r \cup r')$ (**is** ?lhs = ?rhs)

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *cl-le[spec.idle-le]*:

shows $\text{spec.idle} \leq \text{spec.interference.cl } r P$

$\langle \text{proof} \rangle$

lemma *closed-le[spec.idle-le]*:

assumes $P \in \text{spec.interference.closed } r$

shows $\text{spec.idle} \leq P$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *cl-sf-id*:

shows $\text{spec.map } af \text{ id } vf (\text{spec.interference.cl } r P)$

$= \text{spec.interference.cl } (\text{map-prod af id } ' r) (\text{spec.map af id vf } P)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *cl*:

fixes $as :: 'b \text{ set}$

fixes $af :: 'a \Rightarrow 'b$

fixes $sf :: 's \Rightarrow 't$

fixes $vf :: 'v \Rightarrow 'w$

fixes $r :: ('b, 't) \text{ steps}$

fixes $P :: ('b, 't, 'w) \text{ spec}$

shows $\text{spec.invmap af sf vf } (\text{spec.interference.cl } r P)$

$= \text{spec.interference.cl } (\text{map-prod af } (\text{map-prod sf sf}) - ' (r \cup \text{UNIV} \times \text{Id})) (\text{spec.invmap af sf vf } P)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *antimonotone*:

shows $\text{antimono spec.interference.closed}$

$\langle \text{proof} \rangle$

lemmas $\text{strengthen}[strg] = \text{st-ord-antimono}[OF \text{ spec.interference.closed.antimonotone}]$

lemmas $\text{antimono} = \text{antimonoD}[OF \text{ spec.interference.closed.antimonotone}]$

lemma *Sup'*:

assumes $X \subseteq \text{spec.interference.closed } r$

shows $\bigsqcup X \sqcup \text{spec.term.none } (\text{spec.rel } r :: (-, -, \text{unit}) \text{ spec}) \in \text{spec.interference.closed } r$

$\langle \text{proof} \rangle$

lemma *Sup-not-empty*:

assumes $X \subseteq \text{spec.interference.closed } r$

assumes $X \neq \{\}$

shows $\bigsqcup X \in \text{spec.interference.closed } r$

$\langle \text{proof} \rangle$

lemma *rel*:

assumes $r' \subseteq r$

shows $\text{spec.rel } r \in \text{spec.interference.closed } r'$

$\langle \text{proof} \rangle$

lemma *bind-relL*:

fixes $P :: ('a, 's, 'v) \text{ spec}$

assumes $P \in \text{spec.interference.closed } r$

shows $\text{spec.rel } r \gg (\lambda :: \text{unit}. P) = P$

$\langle \text{proof} \rangle$

lemma *bind-relR*:

assumes $P \in \text{spec.interference.closed } r$

shows $P \gg (\lambda v. \text{spec.rel } r \gg (\lambda :: \text{unit}. Q v)) = P \gg Q$

$\langle \text{proof} \rangle$

lemma *bind-rel-unitR*:

assumes $P \in \text{spec.interference.closed } r$

shows $P \gg (\text{spec.rel } r :: (-, -, \text{unit}) \text{ spec}) = P$

$\langle \text{proof} \rangle$

lemma *bind-rel-botR*:

assumes $P \in \text{spec.interference.closed } r$
shows $P \ggg (\lambda v. \text{spec.rel } r \ggg (\lambda :: \text{unit. } \perp)) = P \ggg \perp$
 $\langle \text{proof} \rangle$

lemma *bind[intro]*:
fixes $f :: ('a, 's, 'v) \text{ spec}$
fixes $g :: 'v \Rightarrow ('a, 's, 'w) \text{ spec}$
assumes $f \in \text{spec.interference.closed } r$
assumes $\bigwedge x. g \ x \in \text{spec.interference.closed } r$
shows $(f \ggg g) \in \text{spec.interference.closed } r$
 $\langle \text{proof} \rangle$

lemma *kleene-star*:
assumes $P \in \text{spec.interference.closed } r$
assumes $\text{spec.return } () \leq P$
shows $\text{spec.kleene.star } P \in \text{spec.interference.closed } r$
 $\langle \text{proof} \rangle$

lemma *map-sf-id*:
fixes $af :: 'a \Rightarrow 'b$
fixes $vf :: 'v \Rightarrow 'w$
assumes $P \in \text{spec.interference.closed } r$
shows $\text{spec.map } af \ id \ vf \ P \in \text{spec.interference.closed } (\text{map-prod } af \ id \ ' r)$
 $\langle \text{proof} \rangle$

lemma *invmap*:
fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$
assumes $P \in \text{spec.interference.closed } r$
shows $\text{spec.invmap } af \ sf \ vf \ P \in \text{spec.interference.closed } (\text{map-prod } af \ (\text{map-prod } sf \ sf) - ' r)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *none*:
assumes $P \in \text{spec.interference.closed } r$
shows $\text{spec.term.none } P \in \text{spec.interference.closed } r$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

9.4 The 'a agent datatype

For compositionality we often wish to designate a specific agent as the environment.

datatype *'a agent* = *proc* (*the-agent*: 'a) | *env*
type-synonym *sequential* = *unit agent* — Sequential programs (§13)
abbreviation *self* :: *sequential where self* \equiv *proc* ()

declare *agent.map-id[simp]*
declare *agent.map-id0[simp]*
declare *agent.map-id0[unfolded id-def, simp]*
declare *agent.map-comp[unfolded comp-def, simp]*

lemma *env-not-in-range-proc[iff]*:
shows $\text{env} \notin \text{range } \text{proc}$
 $\langle \text{proof} \rangle$

lemma *range-proc-conv[simp]*:
 shows $x \in \text{range proc} \longleftrightarrow x \neq \text{env}$
 ⟨proof⟩

lemma *inj-proc[iff]*:
 shows *inj proc*
 ⟨proof⟩

lemma *surj-the-inv-proc[iff]*:
 shows *surj (the-inv proc)*
 ⟨proof⟩

lemma *the-inv-proc[simp]*:
 shows *the-inv proc (proc a) = a*
 ⟨proof⟩

lemma *uminus-env-range-proc[simp]*:
 shows $-\{\text{env}\} = \text{range proc}$
 ⟨proof⟩

lemma *env-range-proc-UNIV[simp]*:
 shows *insert env (range proc) = UNIV*
 ⟨proof⟩

⟨ML⟩

lemma *not-conv[simp]*:
 shows $a \neq \text{env} \longleftrightarrow a = \text{self}$
 and $a \neq \text{self} \longleftrightarrow a = \text{env}$
 ⟨proof⟩

lemma *range-proc-self[simp]*:
 shows $\text{range proc} = \{\text{self}\}$
 ⟨proof⟩

lemma *UNIV*:
 shows $\text{UNIV} = \{\text{env}, \text{self}\}$
 ⟨proof⟩

lemma *rev-UNIV[simp]*:
 shows $\{\text{env}, \text{self}\} = \text{UNIV}$
 and $\{\text{self}, \text{env}\} = \text{UNIV}$
 ⟨proof⟩

lemma *uminus-self-env[simp]*:
 shows $-\{\text{self}\} = \{\text{env}\}$
 ⟨proof⟩

⟨ML⟩

lemma *eq-conv*:
 shows $\text{map-agent } f \ x = \text{env} \longleftrightarrow x = \text{env}$
 and $\text{env} = \text{map-agent } f \ x \longleftrightarrow x = \text{env}$
 and $\text{map-agent } f \ x = \text{proc } a \longleftrightarrow (\exists a'. x = \text{proc } a' \wedge a = f \ a')$
 and $\text{proc } a = \text{map-agent } f \ x \longleftrightarrow (\exists a'. x = \text{proc } a' \wedge a = f \ a')$
 ⟨proof⟩

lemma *surj*:

fixes $\pi :: 'a \Rightarrow 'b$
assumes *surj* π
shows *surj* (*map-agent* π)
 \langle *proof* \rangle

lemma *bij*:
fixes $\pi :: 'a \Rightarrow 'b$
assumes *bij* π
shows *bij* (*map-agent* π)
 \langle *proof* \rangle

\langle *ML* \rangle

definition *swap-env-self-fn* :: *sequential* \Rightarrow *sequential* **where**
swap-env-self-fn $a = (\text{case } a \text{ of } \text{proc } () \Rightarrow \text{env} \mid \text{env} \Rightarrow \text{self})$

lemma *swap-env-self-fn-simps*:
shows *swap-env-self-fn* *self* = *env*
swap-env-self-fn *env* = *self*
 \langle *proof* \rangle

lemma *bij-swap-env-self-fn*:
shows *bij* *swap-env-self-fn*
 \langle *proof* \rangle

lemma *swap-env-self-fn-vimage-singleton*:
shows *swap-env-self-fn* -' {*env*} = {*self*}
and *swap-env-self-fn* -' {*self*} = {*env*}
 \langle *proof* \rangle

\langle *ML* \rangle

abbreviation *swap-env-self* :: (*sequential*, '*s*', '*v*') *spec* \Rightarrow (*sequential*, '*s*', '*v*') *spec* **where**
swap-env-self $\equiv \text{spec.} \text{amap } \text{swap-env-self-fn}$

\langle *ML* \rangle

9.5 Parallel composition

We compose a collection of programs (*sequential*, '*s*', '*v*') *spec* in parallel by mapping these into the (*'a agent*, '*s*', '*v*') *spec* lattice, taking the infimum, and mapping back.

definition *toConcurrent-fn* :: '*a*' \Rightarrow '*a*' \Rightarrow *sequential* **where**
toConcurrent-fn = $(\lambda a a'. \text{if } a' = a \text{ then } \text{self} \text{ else } \text{env})$

definition *toSequential-fn* :: '*a agent*' \Rightarrow *sequential* **where**
toSequential-fn = *map-agent* \langle () \rangle

lemma *toSequential-fn-alt-def*:
shows *toSequential-fn* = $(\lambda x. \text{case } x \text{ of } \text{proc } x \Rightarrow \text{self} \mid \text{env} \Rightarrow \text{env})$
 \langle *proof* \rangle

\langle *ML* \rangle

abbreviation *toConcurrent* :: '*a*' \Rightarrow (*sequential*, '*s*', '*v*') *spec* \Rightarrow (*'a agent*, '*s*', '*v*') *spec* **where**
toConcurrent $a \equiv \text{spec.} \text{ainvmap } (\text{toConcurrent-fn } (\text{proc } a))$

abbreviation *toSequential* :: (*'a agent*, '*s*', '*v*') *spec* \Rightarrow (*sequential*, '*s*', '*v*') *spec* **where**
toSequential $\equiv \text{spec.} \text{amap } \text{toSequential-fn}$

definition *Parallel* :: 'a set \Rightarrow ('a \Rightarrow (sequential, 's, unit) spec) \Rightarrow (sequential, 's, unit) spec **where**

Parallel as Ps = spec.toSequential (spec.rel (insert env (proc ' as) \times UNIV) \sqcap ($\prod_{a \in as}$. spec.toConcurrent a (Ps a)))

definition *parallel* :: (sequential, 's, unit) spec \Rightarrow (sequential, 's, unit) spec \Rightarrow (sequential, 's, unit) spec **where**

parallel P Q = spec.Parallel UNIV ($\lambda a::\text{bool}$. if a then P else Q)

ad hoc-overloading

Parallel spec.Parallel

ad hoc-overloading

parallel spec.parallel

lemma *parallel-alt-def*:

shows spec.parallel P Q = spec.toSequential (spec.toConcurrent True P \sqcap spec.toConcurrent False Q)
 <proof>

<ML>

lemma *simps[simp]*:

shows toConcurrent-fn (proc a) (proc a) = self
and toConcurrent-fn (proc a) env = env
and toConcurrent-fn a' a'' = self \longleftrightarrow a'' = a'
and self = toConcurrent-fn a' a'' \longleftrightarrow a'' = a'
and toConcurrent-fn a' a'' = env \longleftrightarrow a'' \neq a'
and env = toConcurrent-fn a' a'' \longleftrightarrow a'' \neq a'
and toConcurrent-fn (proc a) (map-agent <a> x) = map-agent <()> x
 <proof>

lemma *inj-map-agent*:

assumes inj-on f (insert x (set-agent a))
shows toConcurrent-fn (proc (f x)) (map-agent f a) = toConcurrent-fn (proc x) a
 <proof>

lemma *inv-into-map-agent*:

fixes f :: 'a \Rightarrow 'b
fixes a :: 'b agent
fixes x :: 'a
assumes inj-on f as
assumes x \in as
assumes a \in insert env ((λx . proc (f x)) ' as)
shows toConcurrent-fn (proc x) (map-agent (inv-into as f) a) = toConcurrent-fn (proc (f x)) a
 <proof>

lemma *vimage-sequential[simp]*:

shows toConcurrent-fn (proc a) - ' {self} = {proc a}
and toConcurrent-fn (proc a) - ' {env} = -{proc a}
 <proof>

<ML>

lemma *simps[simp]*:

shows toSequential-fn env = env
and toSequential-fn (proc x) = self
and toSequential-fn (map-agent f a) = toSequential-fn a
and trace.map toSequential-fn id id σ = σ
and trace.map toSequential-fn (λx . x) (λx . x) σ = σ
and (λx . if x = self then self else env) = id

$\langle proof \rangle$

lemma *eq-conv*:

shows $toSequential\text{-}fn\ x = env \longleftrightarrow x = env$

and $toSequential\text{-}fn\ x = self \longleftrightarrow (\exists a. x = proc\ a)$

$\langle proof \rangle$

lemma *surj*:

shows $surj\ toSequential\text{-}fn$

$\langle proof \rangle$

lemma *image[simp]*:

assumes $as \neq \{\}$

shows $toSequential\text{-}fn\ \text{' } proc\ \text{' } as = \{self\}$

$\langle proof \rangle$

lemma *vimage-sequential[simp]*:

shows $toSequential\text{-}fn\ \text{' } \{env\} = \{env\}$

and $toSequential\text{-}fn\ \text{' } \{self\} = range\ proc$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *toSequential-fn-eq-toConcurrent-fn-conv*:

shows $toSequential\text{-}fn\ a = toConcurrent\text{-}fn\ a'\ a'' \longleftrightarrow (case\ a\ of\ env \Rightarrow a'' \neq a' \mid proc\ - \Rightarrow a'' = a')$

and $toConcurrent\text{-}fn\ a'\ a'' = toSequential\text{-}fn\ a \longleftrightarrow (case\ a\ of\ env \Rightarrow a'' \neq a' \mid proc\ - \Rightarrow a'' = a')$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *interference*:

shows $spec.toSequential\ (spec.rel\ (\{env\} \times r)) = spec.rel\ (\{env\} \times r)$

$\langle proof \rangle$

lemma *interference-inf-toConcurrent*:

fixes $a :: 'a$

fixes $P :: (sequential, 's, 'v)\ spec$

shows $spec.toSequential\ (spec.rel\ (\{env, proc\ a\} \times UNIV) \sqcap spec.toConcurrent\ a\ P) = P\ (\mathbf{is}\ ?lhs = ?rhs)$

and $spec.toSequential\ (spec.toConcurrent\ a\ P \sqcap spec.rel\ (\{env, proc\ a\} \times UNIV)) = P\ (\mathbf{is}\ ?thesis1)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *interference*:

shows $spec.toConcurrent\ a\ (spec.rel\ (\{env\} \times UNIV)) = spec.rel\ ((-\ \{proc\ a\}) \times UNIV)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *Parallel-le[spec.idle-le]*:

assumes $\bigwedge a. a \in as \implies spec.idle \leq Ps\ a$

shows $spec.idle \leq spec.Parallel\ as\ Ps$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *cong*:

assumes $as = as'$

assumes $\bigwedge a. a \in as' \implies Ps\ a = Ps'\ a$
shows $spec.Parallel\ as\ Ps = spec.Parallel\ as'\ Ps'$
 $\langle proof \rangle$

lemma no-agents:
shows $spec.Parallel\ \{\}\ Ps = spec.rel\ (\{env\} \times UNIV)$
 $\langle proof \rangle$

lemma singleton-agents:
shows $spec.Parallel\ \{a\}\ Ps = Ps\ a$
 $\langle proof \rangle$

lemma bot:
assumes $Ps\ a = \perp$
assumes $a \in as$
shows $spec.Parallel\ as\ Ps = \perp$
 $\langle proof \rangle$

lemma top:
shows $spec.Parallel\ as\ \top = (if\ as = \{\}\ then\ spec.rel\ (\{env\} \times UNIV)\ else\ \top)$
 $\langle proof \rangle$

lemma mono:
assumes $\bigwedge a. a \in as \implies Ps\ a \leq Ps'\ a$
shows $spec.Parallel\ as\ Ps \leq spec.Parallel\ as\ Ps'$
 $\langle proof \rangle$

lemma strengthen[stg]:
assumes $\bigwedge a. a \in as \implies st\text{-ord}\ F\ (Ps\ a)\ (Ps'\ a)$
shows $st\text{-ord}\ F\ (spec.Parallel\ as\ Ps)\ (spec.Parallel\ as\ Ps')$
 $\langle proof \rangle$

lemma mono2mono[cont-intro, partial-function-mono]:
fixes $Ps :: 'a \Rightarrow 'b \Rightarrow (sequential, 's, unit)\ spec$
assumes $\bigwedge a. a \in as \implies monotone\ orda\ (\leq)\ (Ps\ a)$
shows $monotone\ orda\ (\leq)\ (\lambda x :: 'b. spec.Parallel\ as\ (\lambda a. Ps\ a\ x))$
 $\langle proof \rangle$

lemma invmap: — af = id in spec.invmap
shows $spec.invmap\ id\ sf\ vf\ (spec.Parallel\ UNIV\ Ps) = spec.Parallel\ UNIV\ (spec.invmap\ id\ sf\ vf \circ Ps)$
 $\langle proof \rangle$

lemma discard-interference:
assumes $\bigwedge a. a \in bs \implies Ps\ a = spec.rel\ (\{env\} \times UNIV)$
shows $spec.Parallel\ as\ Ps = spec.Parallel\ (as - bs)\ Ps$
 $\langle proof \rangle$

lemma rename-UNIV-aux:
fixes $f :: 'a \Rightarrow 'b$
assumes $inj\text{-on}\ f\ as$
shows $spec.toSequential\ (spec.rel\ (insert\ env\ (proc\ 'as) \times UNIV))$
 $\quad \sqcap\ (\bigsqcap\ a \in as. spec.toConcurrent\ a\ (Ps\ a))$
 $= spec.toSequential\ (spec.rel\ (insert\ env\ (proc\ 'f\ 'as) \times UNIV))$
 $\quad \sqcap\ (\bigsqcap\ a \in as. spec.toConcurrent\ (f\ a)\ (Ps\ a))\ (is\ ?lhs = ?rhs)$
 $\langle proof \rangle$

lemma *rename-UNIV*: — expand the set of agents to *UNIV*

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

assumes *inj-on f as*

shows *spec.Parallel as Ps*

= *spec.Parallel (UNIV :: 'b set)*

($\lambda b. \text{if } b \in f \text{ 'as then } Ps \text{ (inv-into as f b) else spec.rel (\{env\} \times UNIV)$)

(**is** ?lhs = *spec.Parallel - ?f*)

<proof>

lemma *rename*:

fixes $\pi :: 'a \Rightarrow 'b$

fixes $P_s :: 'b \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec}$

assumes *bij-betw π as bs*

shows *spec.Parallel as (Ps \circ π) = spec.Parallel bs Ps*

<proof>

lemma *rename-cong*:

fixes $\pi :: 'a \Rightarrow 'b$

fixes $P_s :: 'a \Rightarrow (-, -, -) \text{ spec}$

fixes $P_s' :: 'b \Rightarrow (-, -, -) \text{ spec}$

assumes *bij-betw π as bs*

assumes $\bigwedge a. a \in as \implies P_s a = P_s' (\pi a)$

shows *spec.Parallel as Ps = spec.Parallel bs Ps'*

<proof>

lemma *inf-pre*:

assumes $as \neq \{\}$

shows *spec.Parallel as Ps \sqcap spec.pre P = ($\|i \in as. P_s i \sqcap \text{spec.pre P}$) (is ?thesis1)*

and *spec.pre P \sqcap spec.Parallel as Ps = ($\|i \in as. \text{spec.pre P} \sqcap P_s i$) (is ?thesis2)*

<proof>

lemma *inf-post*:

assumes $as \neq \{\}$

shows *spec.Parallel as Ps \sqcap spec.post Q = spec.Parallel as ($\lambda i. P_s i \sqcap \text{spec.post Q}$) (is ?thesis1)*

and *spec.post Q \sqcap spec.Parallel as Ps = spec.Parallel as ($\lambda i. \text{spec.post Q} \sqcap P_s i$) (is ?thesis2)*

<proof>

lemma *unwind*:

— All other processes begin with interference

assumes $b: \bigwedge b. b \in as - \{a\} \implies \text{spec.rel (\{env\} \times UNIV) \ggg (\lambda :: \text{unit}. P_s b) \leq P_s b}$

assumes $a: f \ggg g \leq P_s a$ — The selected process starts with *f*

assumes $a \in as$

shows $f \ggg (\lambda v. \text{spec.Parallel as (Ps(a := g v))}) \leq \text{spec.Parallel as Ps}$

<proof>

lemma *inf-rel*:

fixes $as :: 'a \text{ set}$

fixes $r :: 's \text{ rel}$

shows *spec.rel (\{env\} \times UNIV \cup \{self\} \times r) \sqcap spec.Parallel as Ps*

= *spec.Parallel as ($\lambda a. \text{spec.rel (\{env\} \times UNIV \cup \{self\} \times r) \sqcap P_s a$) (is ?lhs = ?rhs)*

and *spec.Parallel as Ps \sqcap spec.rel (\{env\} \times UNIV \cup \{self\} \times r)*

= *spec.Parallel as ($\lambda a. P_s a \sqcap \text{spec.rel (\{env\} \times UNIV \cup \{self\} \times r)$) (is ?thesis1)*

<proof>

lemma *flatten*:

fixes $as :: 'a \text{ set}$

fixes $a :: 'a$

fixes $bs :: 'b \text{ set}$

fixes $P_s :: 'a \Rightarrow (\text{sequential}, 's, \text{unit}) \text{spec}$
fixes $P_{s'} :: 'b \Rightarrow (\text{sequential}, 's, \text{unit}) \text{spec}$
assumes $P_s a = \text{spec.Parallel } bs \ P_{s'}$
assumes $a \in as$
shows $\text{spec.Parallel } as \ P_s = \text{spec.Parallel } ((as - \{a\}) <+> bs) \ (\text{case-sum } P_s \ P_{s'}) \ (\text{is } ?lhs = ?rhs)$
 <proof>

<ML>

lemma *Parallel-some-agents:*

assumes $\bigwedge a. a \in bs \implies P_s a = \text{spec.term.none } (P_{s'} a)$
assumes $as \cap bs \neq \{\}$
shows $\text{spec.Parallel } as \ P_s = \text{spec.term.none } (\|a \in as. \text{if } a \in as \cap bs \text{ then } P_{s'} a \text{ else } P_s a)$
 <proof>

lemma *Parallel-not-empty:*

assumes $as \neq \{\}$
shows $\text{spec.term.none } (\text{Parallel } as \ P_s) = \text{Parallel } as \ (\text{spec.term.none } \circ P_s)$
 <proof>

lemma *parallel:*

shows $\text{spec.term.none } (P \parallel Q) = \text{spec.term.none } P \parallel \text{spec.term.none } Q$
 <proof>

lemma

shows *parallelL:* $\text{spec.term.none } P \parallel Q = \text{spec.term.none } (P \parallel Q)$
and *parallelR:* $P \parallel \text{spec.term.none } Q = \text{spec.term.none } (P \parallel Q)$
 <proof>

<ML>

lemma *Parallel:*

shows $\text{spec.term.all } (\text{spec.Parallel } as \ P_s) = \text{spec.Parallel } as \ (\text{spec.term.all } \circ P_s)$
 <proof>

<ML>

lemma *parallel-le:*

assumes $\text{spec.idle} \leq P$
assumes $\text{spec.idle} \leq Q$
shows $\text{spec.idle} \leq P \parallel Q$
 <proof>

<ML>

lemma *parallel: — af = id in spec.invmap*

shows $\text{spec.invmap } id \ sf \ vf \ (\text{spec.parallel } P \ Q)$
 $= \text{spec.parallel } (\text{spec.invmap } id \ sf \ vf \ P) \ (\text{spec.invmap } id \ sf \ vf \ Q)$
 <proof>

<ML>

lemma *bot:*

shows *botL:* $\text{spec.parallel } \perp \ P = \perp$
and *botR:* $\text{spec.parallel } P \ \perp = \perp$
 <proof>

lemma *commute:*

shows $\text{spec.parallel } P \ Q = \text{spec.parallel } Q \ P$
 $\langle \text{proof} \rangle$

lemma *mono*:

assumes $P \leq P'$
assumes $Q \leq Q'$
shows $\text{spec.parallel } P \ Q \leq \text{spec.parallel } P' \ Q'$
 $\langle \text{proof} \rangle$

lemma *strengthen[stg]*:

assumes $\text{st-ord } F \ P \ P'$
assumes $\text{st-ord } F \ Q \ Q'$
shows $\text{st-ord } F \ (\text{spec.parallel } P \ Q) \ (\text{spec.parallel } P' \ Q')$
 $\langle \text{proof} \rangle$

lemma *mono2mono[cont-intro, partial-function-mono]*:

assumes $\text{monotone orda } (\leq) \ F$
assumes $\text{monotone orda } (\leq) \ G$
shows $\text{monotone orda } (\leq) \ (\lambda f. \text{spec.parallel } (F \ f) \ (G \ f))$
 $\langle \text{proof} \rangle$

lemma *Sup*:

fixes $P_s :: (\text{sequential}, 's, \text{unit}) \text{ spec set}$
shows $\text{SupL: } \bigsqcup P_s \parallel Q = (\bigsqcup P \in P_s. P \parallel Q)$
and $\text{SupR: } Q \parallel \bigsqcup P_s = (\bigsqcup P \in P_s. Q \parallel P)$
 $\langle \text{proof} \rangle$

lemma *sup*:

fixes $P :: (\text{sequential}, 's, \text{unit}) \text{ spec}$
shows $\text{supL: } (P \sqcup Q) \parallel R = (P \parallel R) \sqcup (Q \parallel R)$
and $\text{supR: } P \parallel (Q \sqcup R) = (P \parallel Q) \sqcup (P \parallel R)$
 $\langle \text{proof} \rangle$

We can residuate (\parallel) but not *prog.parallel* (see §13.3) as the latter is not strict. Intuitively any realistic modelling of parallel composition will be non-strict as the divergence of one process should not block the progress of others, and incorporating such interference may preclude the implementation of a specification via this residuation.

References:

- [Hayes \(2016, Law 23\)](#): residuate parallel
- [van Staden \(2015, Lemma 6\)](#) who cites [Armstrong, Gomes, and Struth \(2014\)](#)

definition $\text{res} :: (\text{sequential}, 's, \text{unit}) \text{ spec} \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec} \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec}$ **where**
 $\text{res } S \ i = \bigsqcup \{P. P \parallel i \leq S\}$

interpretation res : *galois.complete-lattice-class* $\lambda S. \text{spec.parallel } S \ i \ \lambda S. \text{spec.parallel.res } S \ i$ **for** i — [Hayes \(2016, Law 23 \(rely refinement\)\)](#)

$\langle \text{proof} \rangle$

lemma *mcont2mcont[cont-intro]*:

assumes $\text{mcont luba orda Sup } (\leq) \ P$
assumes $\text{mcont luba orda Sup } (\leq) \ Q$
shows $\text{mcont luba orda Sup } (\leq) \ (\lambda x. \text{spec.parallel } (P \ x) \ (Q \ x))$
 $\langle \text{proof} \rangle$

lemma *inf-rel*:

shows $\text{spec.rel } (\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r) \sqcap (P \parallel Q)$
 $= (\text{spec.rel } (\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r) \sqcap P) \parallel (\text{spec.rel } (\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r) \sqcap Q)$

and $(P \parallel Q) \sqcap \text{spec.rel } (\{env\} \times UNIV \cup \{self\} \times r)$
 $= (\text{spec.rel } (\{env\} \times UNIV \cup \{self\} \times r) \sqcap P) \parallel (\text{spec.rel } (\{env\} \times UNIV \cup \{self\} \times r) \sqcap Q)$
 $\langle \text{proof} \rangle$

lemma *assoc*:

shows $\text{spec.parallel } P (\text{spec.parallel } Q R) = \text{spec.parallel } (\text{spec.parallel } P Q) R$ (**is** $?lhs = ?rhs$)
 $\langle \text{proof} \rangle$

lemma *bind-botR*:

shows $\text{spec.parallel } (P \gg \perp) Q = \text{spec.parallel } P Q \gg \perp$
and $\text{spec.parallel } P (Q \gg \perp) = \text{spec.parallel } P Q \gg \perp$
 $\langle \text{proof} \rangle$

lemma *interference*:

shows *interferenceL*: $\text{spec.rel } (\{env\} \times UNIV) \parallel c = c$
and *interferenceR*: $c \parallel \text{spec.rel } (\{env\} \times UNIV) = c$
 $\langle \text{proof} \rangle$

lemma *unwindL*:

assumes $\text{spec.rel } (\{env\} \times UNIV) \gg (\lambda::\text{unit}. Q) \leq Q$ — All other processes begin with interference
assumes $f \gg g \leq P$ — The selected process starts with action f
shows $f \gg (\lambda v. g v \parallel Q) \leq P \parallel Q$
 $\langle \text{proof} \rangle$

lemma *unwindR*:

assumes $\text{spec.rel } (\{env\} \times UNIV) \gg (\lambda::\text{unit}. P) \leq P$ — All other processes begin with interference
assumes $f \gg g \leq Q$ — The selected process starts with action f
shows $f \gg (\lambda v. P \parallel g v) \leq P \parallel Q$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *toConcurrent-gen*:

fixes $P :: (\text{sequential}, 's, 'v) \text{ spec}$
fixes $a :: 'a$
assumes $P: P \in \text{spec.interference.closed } (\{env\} \times r)$
shows $\text{spec.toConcurrent } a P \in \text{spec.interference.closed } ((-\{\text{proc } a\}) \times r)$
 $\langle \text{proof} \rangle$

lemma *toConcurrent*:

fixes $P :: (\text{sequential}, 's, 'v) \text{ spec}$
fixes $a :: 'a$
assumes $P: P \in \text{spec.interference.closed } (\{env\} \times r)$
shows $\text{spec.toConcurrent } a P \in \text{spec.interference.closed } (\{env\} \times r)$
 $\langle \text{proof} \rangle$

lemma *toSequential*:

fixes $P :: ('a \text{ agent}, 's, 'v) \text{ spec}$
assumes $P \in \text{spec.interference.closed } (\{env\} \times r)$
shows $\text{spec.toSequential } P \in \text{spec.interference.closed } (\{env\} \times r)$
 $\langle \text{proof} \rangle$

lemma *Parallel*:

assumes $\bigwedge a. Ps a \in \text{spec.interference.closed } (\{env\} \times UNIV)$
shows $\text{spec.Parallel } as Ps \in \text{spec.interference.closed } (\{env\} \times UNIV)$
 $\langle \text{proof} \rangle$

lemma *parallel*:

assumes $P \in \text{spec.interference.closed } (\{env\} \times UNIV)$
assumes $Q \in \text{spec.interference.closed } (\{env\} \times UNIV)$
shows $P \parallel Q \in \text{spec.interference.closed } (\{env\} \times UNIV)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

9.6 Specification Inhabitation

Given that \perp satisfies any specification S , we may wish to show that a specific trace σ is allowed by S .

The strategy is to compute the allowed transitions from a given initial state and possibly a return value. We almost always discard the closures we've added for various kinds of compositionality.

References:

- Similar to how [van Staden \(2014, §3.3\)](#) models a small-step operational semantics.
 - i.e., we can (semantically) define something like an LTS, which is compositional wrt parallel
 - a bit like a resumption or a residual
- Similar to [Hoare, He, and Sampaio \(2000\)](#)

TODO:

- often want transitive variants of these rules
- automate: only stop when there's a scheduling decision to be made

definition $\text{inhabits} :: ('a, 's, 'w) \text{spec} \Rightarrow 's \Rightarrow ('a \times 's) \text{list} \Rightarrow ('a, 's, 'w) \text{spec} \Rightarrow \text{bool}$ (-/ --, -->/ - [50, 0, 0, 50] 50) **where**

$S -s, xs \rightarrow T \longleftrightarrow \langle s, xs, \text{Some } () \rangle \gg T \leq S$

$\langle ML \rangle$

lemma *incomplete*:

assumes $S -s, xs \rightarrow S'$

shows $\langle s, xs, \text{None} \rangle \leq S$

$\langle \text{proof} \rangle$

lemma *complete*:

assumes $S -s, xs \rightarrow \text{spec.return } v$

shows $\langle s, xs, \text{Some } v \rangle \leq S$

$\langle \text{proof} \rangle$

lemmas $I = \text{inhabits.complete inhabits.incomplete}$

lemma *mono*:

assumes $S \leq S'$

assumes $T' \leq T$

assumes $S -s, xs \rightarrow T$

shows $S' -s, xs \rightarrow T'$

$\langle \text{proof} \rangle$

lemma *strengthen[strg]*:

assumes $\text{st-ord } F S S'$

assumes $\text{st-ord } (\neg F) T T'$

shows $\text{st } F (\longrightarrow) (S -s, xs \rightarrow T) (S' -s, xs \rightarrow T')$

$\langle \text{proof} \rangle$

lemma *pre*:

assumes $S -s, xs' \rightarrow T$

assumes $T' \leq T$

assumes $xs = xs'$

shows $S -s, xs \rightarrow T'$

\langle *proof* \rangle

lemma *tau*:

assumes $spec.idle \leq S$

shows $S -s, [] \rightarrow S$

\langle *proof* \rangle

lemma *trans*:

assumes $R -s, xs \rightarrow S$

assumes $S -trace.final' s xs, ys \rightarrow T$

shows $R -s, xs @ ys \rightarrow T$

\langle *proof* \rangle

lemma *Sup*:

assumes $P -s, xs \rightarrow P'$

assumes $P \in X$

shows $\bigsqcup X -s, xs \rightarrow P'$

\langle *proof* \rangle

lemma *supL*:

assumes $P -s, xs \rightarrow P'$

shows $P \sqcup Q -s, xs \rightarrow P'$

\langle *proof* \rangle

lemma *supR*:

assumes $Q -s, xs \rightarrow Q'$

shows $P \sqcup Q -s, xs \rightarrow Q'$

\langle *proof* \rangle

lemma *inf*:

assumes $P -s, xs \rightarrow P'$

assumes $Q -s, xs \rightarrow Q'$

shows $P \sqcap Q -s, xs \rightarrow P' \sqcap Q'$

\langle *proof* \rangle

lemma *infL*:

assumes $P -s, xs \rightarrow R$

assumes $Q -s, xs \rightarrow R$

shows $P \sqcap Q -s, xs \rightarrow R$

\langle *proof* \rangle

\langle *ML* \rangle

lemma *bind*:

assumes $f -s, xs \rightarrow f'$

shows $f \gg g -s, xs \rightarrow f' \gg g$

\langle *proof* \rangle

lemmas $bind' = inhabits.trans[OF inhabits.spec.bind]$

lemma *parallelL*:

assumes $P -s, xs \rightarrow P'$

assumes $spec.rel (\{env\} \times UNIV) \gg (\lambda::unit. Q) \leq Q$

shows $P \parallel Q -s, xs \rightarrow P' \parallel Q$
 $\langle proof \rangle$

lemma parallelR:

assumes $Q -s, xs \rightarrow Q'$

assumes $spec.rel (\{env\} \times UNIV) \gg= (\lambda::unit. P) \leq P$

shows $P \parallel Q -s, xs \rightarrow P \parallel Q'$

$\langle proof \rangle$

lemmas parallelL' = inhabits.trans[OF inhabits.spec.parallelL]

lemmas parallelR' = inhabits.trans[OF inhabits.spec.parallelR]

$\langle ML \rangle$

lemma step:

assumes $(v, a, s, s') \in F$

shows $spec.action F -s, [(a, s')] \rightarrow spec.return v$

$\langle proof \rangle$

lemma stutter:

assumes $(v, a, s, s) \in F$

shows $spec.action F -s, [] \rightarrow spec.return v$

$\langle proof \rangle$

$\langle ML \rangle$

lemma map:

fixes $af :: 'a \Rightarrow 'b$

fixes $sf :: 's \Rightarrow 't$

fixes $vf :: 'v \Rightarrow 'w$

assumes $P -s, xs \rightarrow spec.return v$

shows $spec.map af sf vf P -sf s, map (map-prod af sf) xs \rightarrow spec.return (vf v)$

$\langle proof \rangle$

lemma invmap:

fixes $af :: 'a \Rightarrow 'b$

fixes $sf :: 's \Rightarrow 't$

fixes $vf :: 'v \Rightarrow 'w$

assumes $P -sf s, map (map-prod af sf) xs \rightarrow P'$

shows $spec.invmap af sf vf P -s, xs \rightarrow spec.invmap af sf vf P'$

$\langle proof \rangle$

$\langle ML \rangle$

lemma step:

assumes $P -s, xs \rightarrow P'$

shows $spec.term.none P -s, xs \rightarrow spec.term.none P'$

$\langle proof \rangle$

$\langle ML \rangle$

lemma step:

assumes $P -s, xs \rightarrow P'$

shows $spec.term.all P -s, xs \rightarrow spec.term.all P'$

$\langle proof \rangle$

lemma term:

assumes $spec.idle \leq P$

shows $spec.term.all P -s, [] \rightarrow spec.return v$
<proof>

<ML>

lemma step:

assumes $P -s, xs \rightarrow P'$

shows $spec.kleene.star P -s, xs \rightarrow P' \gg spec.kleene.star P$
<proof>

lemma term:

shows $spec.kleene.star P -s, [] \rightarrow spec.return ()$
<proof>

<ML>

lemma rel:

assumes $trace.steps' s xs \subseteq r$

shows $spec.rel r -s, xs \rightarrow spec.rel r$
<proof>

lemma rel-term:

assumes $trace.steps' s xs \subseteq r$

shows $spec.rel r -s, xs \rightarrow spec.return v$
<proof>

lemma step:

assumes $(a, s, s') \in r$

shows $spec.rel r -s, [(a, s')] \rightarrow spec.rel r$
<proof>

lemma term:

shows $spec.rel r -s, [] \rightarrow spec.return v$
<proof>

<ML>

10 “Next step” implication ala Abadi and Merz (and Lamport)

As was apparently well-known in the mid-1990s (see, e.g., [Xu, Cau, and Collette \(1994, §4\)](#) and the references therein), Heyting implication is inadequate for a general refinement story. (We show it is strong enough for a relational assume/guarantee program logic; see §9.2, §12.2 and §13.5.2. In our setting it fails to generalize (at least) because the composition theorem for Heyting implication (§9.2) is not termination sensitive.)

We therefore follow [Abadi and Lamport \(1995\)](#) by developing a stronger implication $P \longrightarrow_+ Q$ with the intuitive semantics that the consequent Q holds for at least one step beyond the antecedent P . This is some kind of step indexing.

Here we sketch the relevant parts of [Abadi and Merz \(1995, 1996\)](#), the latter of which has a fuller story, including a formal account of the logical core of TLA and the (implicit) observation that infinitary parallel composition poses no problem for safety properties (see the comments under Theorem 5.2 and §5.5). [Abadi and Lamport \(1995\)](#); [Cau and Collette \(1996\)](#); [Xu et al. \(1994\)](#) provide further background; [Jonsson and Tsay \(1996, Appendix B\)](#) provide a more syntactic account.

Observations:

- The hypothesis P is always a safety property here
- TLA does not label transitions or have termination markers
- Abadi/Cau/Collette/Lamport/Merz/Xu/... avoid naming this operator

Further references:

- [Maier \(2001\)](#)

definition *next-imp* :: 'a::preorder set \Rightarrow 'a set \Rightarrow 'a set **where** — [Abadi and Merz \(1995, §2\)](#)
next-imp P Q = $\{\sigma. \forall \sigma' \leq \sigma. (\forall \sigma'' < \sigma'. \sigma'' \in P) \longrightarrow \sigma' \in Q\}$

$\langle ML \rangle$

lemma *downwards-closed*:

assumes P \in *downwards.closed*

shows *next-imp* P Q \in *downwards.closed*

$\langle proof \rangle$

lemma *mono*:

assumes $x' \leq x$

assumes $y \leq y'$

shows *next-imp* x y \leq *next-imp* x' y'

$\langle proof \rangle$

lemma *strengthen[strg]*:

assumes *st-ord* (\neg F) X X'

assumes *st-ord* F Y Y'

shows *st-ord* F (*next-imp* X Y) (*next-imp* X' Y')

$\langle proof \rangle$

lemma *minimal*:

assumes *trace.T* s xs v \in *next-imp* P Q

shows *trace.T* s [] None \in Q

$\langle proof \rangle$

lemma *alt-def*: — This definition coincides with [Cau and Collette \(1996\)](#), [Abadi and Lamport \(1995, §3.5.3\)](#)

assumes P \in *downwards.closed*

shows *next-imp* P Q

= $\{\sigma. \text{trace.T } (\text{trace.init } \sigma) [] \text{None} \in Q$

$\wedge (\forall i. \text{trace.take } i \sigma \in P \longrightarrow \text{trace.take } (\text{Suc } i) \sigma \in Q)\}$ (**is** ?lhs = ?rhs)

$\langle proof \rangle$

[Abadi and Lamport \(1995, §3.5.3\)](#) assert but do not prove the following connection with Heyting implication. [Abadi and Merz \(1995\)](#) do. See also [Abadi and Merz \(1996, §5.3 and §5.5\)](#).

lemma *Abadi-Merz-Prop-1-subseteq*: — First half of [Abadi and Merz \(1995, Proposition 1\)](#)

fixes P :: 'a::preorder set

assumes P \in *downwards.closed*

assumes wf: wfP (($<$) :: 'a relp)

shows *next-imp* P Q \subseteq *downwards.imp* (*downwards.imp* Q P) Q (**is** ?lhs \subseteq ?rhs)

$\langle proof \rangle$

The converse holds if either Q is a safety property or the order is partial.

lemma *Abadi-Merz-Prop1*: — [Abadi and Merz \(1995, Proposition 1\)](#) and [Abadi and Merz \(1996, Proposition 5.2\)](#)

fixes P :: 'a::preorder set

assumes P \in *downwards.closed*

assumes Q \in *downwards.closed*

assumes wf: wfP (($<$) :: 'a relp)

shows *next-imp* P Q = *downwards.imp* (*downwards.imp* Q P) Q (**is** ?lhs = ?rhs)

$\langle proof \rangle$

lemma *Abadi-Lamport-Lemma6*: — [Abadi and Lamport \(1995, Lemma 6\)](#) (no proof given there)

fixes P :: 'a::order set

assumes $P \in \text{downwards.closed}$
assumes $\text{wf}: \text{wfP} (< :: 'a \text{ relp})$
shows $\text{next-imp } P \ Q = \text{downwards.imp } (\text{downwards.imp } Q \ P) \ Q$ (**is** $?lhs = ?rhs$)
 $\langle \text{proof} \rangle$

lemmas $\text{downwards-imp} = \text{next-imp.Abadi-Lamport-Lemma6}[\text{OF} - \text{trace.wfP-less}]$

lemma *boolean-implication-le*:
assumes $P \in \text{downwards.closed}$
shows $\text{next-imp } P \ Q \subseteq P \longrightarrow_B Q$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lift-definition $\text{next-imp} :: ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ (**infixr** \longrightarrow_+ 61) **is**
 Next-Imp.next-imp
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *heyting*: — fundamental
shows $P \longrightarrow_+ Q = (Q \longrightarrow_H P) \longrightarrow_H Q$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *next-imp-le-conv*:
fixes $P :: ('a, 's, 'v) \text{ spec}$
shows $\langle \sigma \rangle \leq P \longrightarrow_+ Q \longleftrightarrow (\forall \sigma' \leq \sigma. (\forall \sigma'' < \sigma'. \langle \sigma'' \rangle \leq P) \longrightarrow \langle \sigma' \rangle \leq Q)$ (**is** $?lhs \longleftrightarrow ?rhs$)
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *mono*:
assumes $x' \leq x$
assumes $y \leq y'$
shows $x \longrightarrow_+ y \leq x' \longrightarrow_+ y'$
 $\langle \text{proof} \rangle$

lemma *strengthen[stg]*:
assumes $\text{st-ord } (\neg F) \ X \ X'$
assumes $\text{st-ord } F \ Y \ Y'$
shows $\text{st-ord } F \ (X \longrightarrow_+ Y) \ (X' \longrightarrow_+ Y')$
 $\langle \text{proof} \rangle$

lemma *idempotentR*:
shows $P \longrightarrow_+ (P \longrightarrow_+ Q) = P \longrightarrow_+ Q$
 $\langle \text{proof} \rangle$

lemma *contains*:
assumes $X \leq Q$
shows $X \leq P \longrightarrow_+ Q$
 $\langle \text{proof} \rangle$

interpretation *closure*: *closure-complete-lattice-class* (\longrightarrow_+) P **for** P
 $\langle \text{proof} \rangle$

lemma *InfR*:

shows $x \longrightarrow_+ \sqcap X = \sqcap ((\longrightarrow_+) x \text{ ' } X)$
 ⟨proof⟩

lemma *SupR-not-empty*:

fixes $P :: (-, -, -)$ *spec*

assumes $X \neq \{\}$

shows $P \longrightarrow_+ (\sqcup x \in X. Q x) = (\sqcup x \in X. P \longrightarrow_+ Q x)$ (**is** *?lhs = ?rhs*)

⟨proof⟩

lemma *cont*:

shows $\text{cont } \text{Sup} (\leq) \text{Sup} (\leq) ((\longrightarrow_+) P)$

⟨proof⟩

lemma *mcont*:

shows $\text{mcont } \text{Sup} (\leq) \text{Sup} (\leq) ((\longrightarrow_+) P)$

⟨proof⟩

lemmas $\text{mcont2mcont}[\text{cont-intro}] = \text{mcont2mcont}[\text{OF spec.next-imp.mcont, of luba orda } Q P]$ **for** *luba orda* $Q P$

lemma *botL*:

assumes $\text{spec.idle} \leq P$

shows $\perp \longrightarrow_+ P = \top$

⟨proof⟩

lemma *topL[simp]*:

shows $\top \longrightarrow_+ P = P$

⟨proof⟩

lemmas $\text{topR}[\text{simp}] = \text{spec.next-imp.closure.cl-top}$

lemma *refl*:

shows $P \longrightarrow_+ P \leq P$

⟨proof⟩

lemma *heyting-le*:

shows $P \longrightarrow_+ Q \leq P \longrightarrow_H Q$

⟨proof⟩

lemma *discharge*:

shows $P \sqcap (P \sqcap Q \longrightarrow_+ R) = P \sqcap (Q \longrightarrow_+ R)$ (**is** *?thesis1* $P Q$)

and $(P \sqcap Q \longrightarrow_+ R) \sqcap P = P \sqcap (Q \longrightarrow_+ R)$ (**is** *?thesis2*)

and $Q \sqcap (P \sqcap Q \longrightarrow_+ R) = Q \sqcap (P \longrightarrow_+ R)$ (**is** *?thesis3*)

and $(P \sqcap Q \longrightarrow_+ R) \sqcap Q = Q \sqcap (P \longrightarrow_+ R)$ (**is** *?thesis4*)

⟨proof⟩

lemma *detachment*:

shows $x \sqcap (x \longrightarrow_+ y) \leq y$

and $(x \longrightarrow_+ y) \sqcap x \leq y$

⟨proof⟩

lemma *infR*:

shows $P \longrightarrow_+ Q \sqcap R = (P \longrightarrow_+ Q) \sqcap (P \longrightarrow_+ R)$

⟨proof⟩

lemma *supL-le*:

shows $x \sqcup y \longrightarrow_+ z \leq (x \longrightarrow_+ z) \sqcup (y \longrightarrow_+ z)$

⟨proof⟩

lemma *heytingL*:

shows $(P \longrightarrow_H Q) \sqcap (Q \longrightarrow_+ R) \leq P \longrightarrow_+ R$
 $\langle \text{proof} \rangle$

lemma *heytingR*:

shows $(P \longrightarrow_+ Q) \sqcap (Q \longrightarrow_H R) \leq P \longrightarrow_+ R$
 $\langle \text{proof} \rangle$

lemma *heytingL-distrib*:

shows $P \longrightarrow_H (Q \longrightarrow_+ R) = (P \sqcap Q) \longrightarrow_+ (P \longrightarrow_H R)$
 $\langle \text{proof} \rangle$

lemma *trans*:

shows $(P \longrightarrow_+ Q) \sqcap (Q \longrightarrow_+ R) \leq P \longrightarrow_+ R$
 $\langle \text{proof} \rangle$

lemma *rev-trans*:

shows $(Q \longrightarrow_+ R) \sqcap (P \longrightarrow_+ Q) \leq P \longrightarrow_+ R$
 $\langle \text{proof} \rangle$

lemma

assumes $x' \leq x$
shows *discharge-leL*: $x' \sqcap (x \longrightarrow_+ y) = x' \sqcap y$ (**is** *?thesis1*)
and *discharge-leR*: $(x \longrightarrow_+ y) \sqcap x' = y \sqcap x'$ (**is** *?thesis2*)
 $\langle \text{proof} \rangle$

lemma *invmap*:

shows $\text{spec.invmap af sf vf } (P \longrightarrow_+ Q) = \text{spec.invmap af sf vf } P \longrightarrow_+ \text{spec.invmap af sf vf } Q$
 $\langle \text{proof} \rangle$

lemma *Abadi-Lamport-Lemma7*:

assumes $Q \sqcap R \leq P$
shows $P \longrightarrow_+ Q \leq R \longrightarrow_+ Q$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *next-imp*:

shows $\text{spec.term.none } (P \longrightarrow_+ Q) \leq \text{spec.term.all } P \longrightarrow_+ \text{spec.term.none } Q$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *next-imp*:

shows $\text{spec.term.all } (P \longrightarrow_+ Q) = \text{spec.term.all } P \longrightarrow_+ \text{spec.term.all } Q$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *next-imp*:

assumes $Q \in \text{spec.term.closed}$ -
shows $P \longrightarrow_+ Q \in \text{spec.term.closed}$ -
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *next-imp-eq-heyting*:

assumes $\text{spec.idle} \leq R$

shows $Q \sqcap \text{spec.pre } P \longrightarrow_+ R = \text{spec.pre } P \longrightarrow_H (Q \longrightarrow_+ R)$ (**is** *?lhs = ?rhs*)
and $\text{spec.pre } P \sqcap Q \longrightarrow_+ R = \text{spec.pre } P \longrightarrow_H (Q \longrightarrow_+ R)$ (**is** *?thesis1*)
 <proof>

<ML>

10.1 Compositionality ala Abadi and Merz (and Lamport)

The main theorem for this implication (Abadi and Merz (1995, Theorem 4) and Abadi and Merz (1996, Corollary 5.1)) shows how to do assumption/commitment proofs for TLA considered as an algebraic logic. See also Cau and Collette (1996).

<ML>

lemma *Abadi-Lamport-Lemma5:*

shows $(\prod_{i \in I}. P \ i \longrightarrow_+ Q \ i) \leq (\prod_{i \in I}. P \ i) \longrightarrow_+ (\prod_{i \in I}. Q \ i)$
 <proof>

lemma *Abadi-Merz-Prop2-1:*

shows $(P \longrightarrow_+ Q) \sqcap (P \longrightarrow_+ (Q \longrightarrow_H R)) \leq P \longrightarrow_+ R$
 <proof>

lemma *Abadi-Merz-Theorem3-5:*

shows $P \longrightarrow_H (Q \longrightarrow_H R) \leq (R \longrightarrow_+ Q) \longrightarrow_H (P \longrightarrow_+ Q)$
 <proof>

theorem *Abadi-Merz-Theorem4:*

shows $(A \sqcap (\prod_{i \in I}. C \ s \ i) \longrightarrow_H (\prod_{i \in I}. A \ s \ i))$
 $\sqcap (A \longrightarrow_+ ((\prod_{i \in I}. C \ s \ i) \longrightarrow_H C))$
 $\sqcap (\prod_{i \in I}. A \ s \ i \longrightarrow_+ C \ s \ i)$
 $\leq A \longrightarrow_+ C$ (**is** *?lhs ≤ ?rhs*)
 <proof>

<ML>

11 Stability

The essence of rely/guarantee reasoning is that sequential assertions must be *stable* with respect to interfering transitions as expressed in a *rely* relation. Formally an assertion P is stable if it becomes no less true for each transition in the rely r . This is essentially monotonicity, or that the extension of P is r -closed.

References:

- Vafeiadis (2008, §3.1.3) has a def for stability in terms of separation logic

definition *stable* :: 'a rel \Rightarrow 'a pred \Rightarrow bool **where**

stable r $P = \text{monotone } (\lambda x \ y. (x, y) \in r) (\leq) P$

<ML>

named-theorems *intro stability intro rules*

lemma *singleton-conv:*

shows *stable* $\{(s, s')\} P \iff (P \ s \longrightarrow P \ s')$
 <proof>

lemma *closed:*

shows *stable* r $P \iff r$ “ *Collect* $P \subseteq \text{Collect } P$ ”
 <proof>

lemma *rtrancl-conv*:

shows $stable (r^*) = stable r$
 $\langle proof \rangle$

lemma *reflcl-conv*:

shows $stable (r^-) = stable r$
 $\langle proof \rangle$

lemma *empty[stable.intro]*:

shows $stable \{ \} P$
 $\langle proof \rangle$

lemma [*stable.intro*]:

shows $Id: stable Id P$
and $Id\text{-fst}: \bigwedge P. stable (Id \times_R A) (\lambda s. P (fst s))$
and $Id\text{-fst-fst-snd}: \bigwedge P. stable (Id \times_R Id \times_R A) (\lambda s. P (fst s) (fst (snd s)))$
 $\langle proof \rangle$

lemma *UNIV*:

shows $stable UNIV P \longleftrightarrow (\exists c. P = \langle c \rangle)$
 $\langle proof \rangle$

lemma *antimono-rel*:

shows $antimono (\lambda r. stable r P)$
 $\langle proof \rangle$

lemmas *strengthen-rel[strg] = st-ord-antimono[OF stable.antimono-rel, unfolded le-bool-def]*

lemma *infI*:

assumes $stable r P$
shows $infI1: stable (r \cap s) P$
and $infI2: stable (s \cap r) P$
 $\langle proof \rangle$

lemma *UNION-conv*:

shows $stable (\bigcup_{x \in X}. r x) P \longleftrightarrow (\forall x \in X. stable (r x) P)$
 $\langle proof \rangle$

lemmas *UNIONI[stable.intro] = iffD2[OF stable.UNION-conv, rule-format]*

lemma *Union-conv*:

shows $stable (\bigcup X) P \longleftrightarrow (\forall x \in X. stable x P)$
 $\langle proof \rangle$

lemma *union-conv*:

shows $stable (r \cup s) P \longleftrightarrow stable r P \wedge stable s P$
 $\langle proof \rangle$

lemmas *UnionI[stable.intro] = iffD2[OF stable.Union-conv, rule-format]*

lemmas *unionI[stable.intro] = iffD2[OF stable.union-conv, rule-format, unfolded conj-explode]*

Properties of stable with respect to the predicate **lemma** *const[stable.intro]*:

shows $stable r \langle c \rangle$
and $stable r \perp$
and $stable r \top$
 $\langle proof \rangle$

lemma *conjI*[*stable.intro*]:

assumes *stable r P*
assumes *stable r Q*
shows *stable r (P ∧ Q)*

⟨*proof*⟩

lemma *disjI*[*stable.intro*]:

assumes *stable r P*
assumes *stable r Q*
shows *stable r (P ∨ Q)*

⟨*proof*⟩

lemma *implies-constI*[*stable.intro*]:

assumes $P \implies \text{stable } r \ Q$
shows *stable r (λs. P → Q s)*

⟨*proof*⟩

lemma *allI*[*stable.intro*]:

assumes $\bigwedge x. \text{stable } r \ (P \ x)$
shows *stable r (∀x. P x)*

⟨*proof*⟩

lemma *ballI*[*stable.intro*]:

assumes $\bigwedge x. x \in X \implies \text{stable } r \ (P \ x)$
shows *stable r (λs. ∀x∈X. P x s)*

⟨*proof*⟩

lemma *stable-relprod-fstI*[*stable.intro*]:

assumes *stable r P*
shows *stable (r ×_R s) (λs. P (fst s))*

⟨*proof*⟩

lemma *stable-relprod-sndI*[*stable.intro*]:

assumes *stable s P*
shows *stable (r ×_R s) (λs. P (snd s))*

⟨*proof*⟩

lemma *local-only*: — for predicates that are insensitive to the shared state

assumes $\bigwedge ls \ s \ s'. P \ (ls, \ s) \longleftrightarrow P \ (ls, \ s')$
shows *stable (Id ×_R UNIV) P*

⟨*proof*⟩

lemma *Id-on-proj*:

assumes $\bigwedge v. \text{stable } Id_f \ (\lambda s. P \ v \ s)$
shows *stable Id_f (λs. P (f s) s)*

⟨*proof*⟩

lemma *if-const-conv*:

shows *stable r (if c then P else Q) ↔ stable r (λs. c → P s) ∧ stable r (λs. ¬c → Q s)*

⟨*proof*⟩

lemma *ifI*[*stable.intro*]:

assumes *stable r (λs. c s → P s)*
assumes *stable r (λs. ¬c s → Q s)*
shows *stable r (λs. if c s then P s else Q s)*

⟨*proof*⟩

lemma *ifI2*[*stable.intro*]:

assumes *stable* r ($\lambda s. c\ s \longrightarrow P\ s\ s$)
assumes *stable* r ($\lambda s. \neg c\ s \longrightarrow Q\ s\ s$)
shows *stable* r ($\lambda s. (\text{if } c\ s \text{ then } P\ s \text{ else } Q\ s)\ s$)
 $\langle \text{proof} \rangle$

lemma *case-optionI*[*stable.intro*]:
assumes *stable* r ($\lambda s. \text{opt } s = \text{None} \longrightarrow \text{none } s$)
assumes $\bigwedge v. \text{stable } r$ ($\lambda s. \text{opt } s = \text{Some } v \longrightarrow \text{some } v\ s$)
shows *stable* r ($\lambda s. \text{case opt } s \text{ of } \text{None} \Rightarrow \text{none } s \mid \text{Some } v \Rightarrow \text{some } v\ s$)
 $\langle \text{proof} \rangle$

lemma *case-optionI2*[*stable.intro*]:
assumes $\text{opt} = \text{None} \Longrightarrow \text{stable } r\ \text{none}$
assumes $\bigwedge v. \text{opt} = \text{Some } v \Longrightarrow \text{stable } r\ (\text{some } v)$
shows *stable* r ($\text{case opt of } \text{None} \Rightarrow \text{none} \mid \text{Some } v \Rightarrow \text{some } v$)
 $\langle \text{proof} \rangle$

In practice the following rules are often too weak

lemma *impliesI*:
assumes *stable* r ($\neg P$)
assumes *stable* r Q
shows *stable* r ($P \longrightarrow Q$)
 $\langle \text{proof} \rangle$

lemma *exI*:
assumes $\bigwedge x. \text{stable } r\ (P\ x)$
shows *stable* r ($\exists x. P\ x$)
 $\langle \text{proof} \rangle$

lemma *bexI*:
assumes $\bigwedge x. x \in X \Longrightarrow \text{stable } r\ (P\ x)$
shows *stable* r ($\lambda s. \exists x \in X. P\ x\ s$)
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

12 Refinement

We develop a refinement story for the $(\prime a, \prime s, \prime v)$ *spec* lattice.

References:

- Vafeiadis (2008) (RGsep, program logic) and Liang, Feng, and Fu (2014) (RGsim, refinement)
- Armstrong et al. (2014)
- van Staden (2015)

definition *refinement* $:: \prime s\ \text{pred} \Rightarrow (\prime a, \prime s, \prime v)\ \text{spec} \Rightarrow (\prime a, \prime s, \prime v)\ \text{spec} \Rightarrow (\prime v \Rightarrow \prime s\ \text{pred}) \Rightarrow (\prime a, \prime s, \prime v)\ \text{spec}$ ($\{\!\{-}\!\}$, \vdash , \vdash , $\{\!\{-}\!\}$ [0,0,0,0] 100) **where**
 $\{\!\{P}\!\}, A \vdash G, \{\!\{Q}\!\} = \text{spec.pre } P \sqcap A \longrightarrow_+ G \sqcap \text{spec.post } Q$

An intuitive gloss on the proposition $c \leq \{\!\{P}\!\}, A \vdash G, \{\!\{Q}\!\}$ is: assuming the precondition P holds and all steps conform to the process A , then c is a refinement of G and satisfies the postcondition Q .

Observations:

- We use *next-imp* here; (\longrightarrow_H) is (only) enough for an assume/guarantee program logic (see §12.2)
- A is arbitrary but is intended to constrain only *env* steps

– similarly termination can depend on A : a parallel composition can only terminate if all of its constituent processes terminate

- As $P \longrightarrow_+ Q$ implies $idle \leq Q$, in practice $idle \leq G$
- see §13.4.1 for some introduction rules

$\langle ML \rangle$

lemma E :

assumes $c \leq \{P\}$, $A \Vdash G$, $\{Q\}$
obtains $c \leq spec.pre\ P \sqcap A \longrightarrow_+ G$
and $c \leq spec.pre\ P \sqcap A \longrightarrow_+ spec.post\ Q$

$\langle proof \rangle$

lemma $pre-post-cong$:

assumes $P = P'$
assumes $Q = Q'$
shows $\{P\}$, $A \Vdash G$, $\{Q\} = \{P'\}$, $A \Vdash G$, $\{Q'\}$

$\langle proof \rangle$

lemma top :

shows $\{P\}$, $A \Vdash \top$, $\{\top\} = \top$
and $\{P\}$, $A \Vdash \top$, $\{\langle \top \rangle\} = \top$
and $\{P\}$, $A \Vdash \top$, $\{\lambda -. True\} = \top$

$\langle proof \rangle$

lemma $mcont2mcont[cont-intro]$:

assumes $mcont\ luba\ orda\ Sup\ (\leq)\ G$
shows $mcont\ luba\ orda\ Sup\ (\leq)\ (\lambda x. \{P\}, A \Vdash G\ x, \{Q\})$

$\langle proof \rangle$

lemma $mono$:

assumes $P' \leq P$
assumes $A' \leq A$
assumes $G \leq G'$
assumes $Q \leq Q'$
shows $\{P\}$, $A \Vdash G$, $\{Q\} \leq \{P'\}$, $A' \Vdash G'$, $\{Q'\}$

$\langle proof \rangle$

lemma $strengthen[strg]$:

assumes $st-ord\ (\neg F)\ P\ P'$
assumes $st-ord\ (\neg F)\ A\ A'$
assumes $st-ord\ F\ G\ G'$
assumes $st-ord\ F\ Q\ Q'$
shows $st-ord\ F\ (\{P\}, A \Vdash G, \{Q\})\ (\{P'\}, A' \Vdash G', \{Q'\})$

$\langle proof \rangle$

lemma $mono-stronger$:

assumes $P' \leq P$
assumes $spec.pre\ P' \sqcap A' \leq A$
assumes $spec.pre\ P' \sqcap G \leq A' \longrightarrow_+ G'$
assumes $Q \leq Q'$
assumes $spec.idle \leq G'$
shows $\{P\}$, $A \Vdash G$, $\{Q\} \leq \{P'\}$, $A' \Vdash G'$, $\{Q'\}$

$\langle proof \rangle$

lemmas $pre-ag = order.trans[OF\ -\ refinement.mono[OF\ order.refl\ -\ order.refl],\ of\ c]$ **for** c

lemmas *pre-a* = *refinement.pre-ag*[*OF* - - *order.refl*]

lemmas *pre-g* = *refinement.pre-ag*[*OF* - *order.refl*]

lemma *pre*:

assumes $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$

assumes $\bigwedge s. P' s \implies P s$

assumes $A' \leq A$

assumes $G \leq G'$

assumes $\bigwedge s v. Q s v \implies Q' s v$

shows $c \leq \{\!|P'|\!\}, A' \Vdash G', \{\!|Q'|\!\}$

<proof>

lemmas *pre-pre-post* = *refinement.pre*[*OF* - - *order.refl* *order.refl*, *of c*] **for** *c*

lemma *pre-imp*:

assumes $\bigwedge s. P s \implies P' s$

assumes $c \leq \{\!|P'|\!\}, A \Vdash G, \{\!|Q|\!\}$

shows $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$

<proof>

lemmas *pre-pre* = *refinement.pre-imp*[*rotated*]

lemma *post-imp*:

assumes $\bigwedge v s. Q v s \implies R v s$

assumes $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$

shows $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|R|\!\}$

<proof>

lemmas *pre-post* = *refinement.post-imp*[*rotated*]

lemmas *strengthen-post* = *refinement.pre-post*

lemma *pre-inf-assume*:

shows $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} = \{\!|P|\!\}, A \sqcap \text{spec.pre } P \Vdash G, \{\!|Q|\!\}$

<proof>

lemma *pre-assume-absorb*:

assumes $A \leq \text{spec.pre } P$

shows $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} = \{\!|\top|\!\}, A \Vdash G, \{\!|Q|\!\}$

<proof>

lemmas *sup* = *sup-least*[**where** $x = \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$] **for** $A G P Q$

lemma

shows *supRL*: $c \leq \{\!|P|\!\}, A \Vdash G_1, \{\!|Q|\!\} \implies c \leq \{\!|P|\!\}, A \Vdash G_1 \sqcup G_2, \{\!|Q|\!\}$

and *supRR*: $c \leq \{\!|P|\!\}, A \Vdash G_2, \{\!|Q|\!\} \implies c \leq \{\!|P|\!\}, A \Vdash G_1 \sqcup G_2, \{\!|Q|\!\}$

<proof>

lemma *infR-conv*:

shows $\{\!|P|\!\}, A \Vdash G_1 \sqcap G_2, \{\!|Q_1 \sqcap Q_2|\!\} = \{\!|P|\!\}, A \Vdash G_1, \{\!|Q_1|\!\} \sqcap \{\!|P|\!\}, A \Vdash G_2, \{\!|Q_2|\!\}$

<proof>

lemma *inf-le*:

shows $X \sqcap \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} \leq \{\!|P|\!\}, X \sqcap A \Vdash X \sqcap G, \{\!|Q|\!\}$

<proof>

lemma *heyting-le*:

shows $\{\!|P|\!\}, A \sqcap B \Vdash B \longrightarrow_H G, \{\!|Q|\!\} \leq B \longrightarrow_H \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$

<proof>

lemma *heyting-pre*:

assumes *spec.idle* $\leq G$

shows *spec.pre* $P \longrightarrow_H \{\!\{P'\}\!\}, A \Vdash G, \{\!\{Q\}\!\} = \{\!\{P \wedge P'\}\!\}, A \Vdash G, \{\!\{Q\}\!\}$

\langle *proof* \rangle

lemma *sort-of-refl*:

assumes $c \leq \{\!\{P\}\!\}, A \Vdash G, \{\!\{Q\}\!\}$

shows $c \leq \{\!\{P\}\!\}, A \Vdash c, \{\!\{Q\}\!\}$

\langle *proof* \rangle

lemma *gen-asm-base*:

assumes $P \implies c \leq \{\!\{P' \wedge P''\}\!\}, A \Vdash G, \{\!\{Q\}\!\}$

assumes *spec.idle* $\leq G$

shows $c \leq \{\!\{P' \wedge \langle P \rangle \wedge P''\}\!\}, A \Vdash G, \{\!\{Q\}\!\}$

\langle *proof* \rangle

lemmas *gen-asm =*

refinement.gen-asm-base[**where** $P' = \langle True \rangle$ **and** $P'' = \langle True \rangle$, *simplified*]

refinement.gen-asm-base[**where** $P' = \langle True \rangle$, *simplified*]

refinement.gen-asm-base[**where** $P'' = \langle True \rangle$, *simplified*]

refinement.gen-asm-base

lemma *post-conj*:

assumes $c \leq \{\!\{P\}\!\}, A \Vdash G, \{\!\{Q\}\!\}$

assumes $c \leq \{\!\{P\}\!\}, A \Vdash G, \{\!\{Q'\}\!\}$

shows $c \leq \{\!\{P\}\!\}, A \Vdash G, \{\!\{\lambda rv. Q \ rv \wedge Q' \ rv\}\!\}$

\langle *proof* \rangle

lemma *conj-lift*:

assumes $c \leq \{\!\{P\}\!\}, A \Vdash G, \{\!\{Q\}\!\}$

assumes $c \leq \{\!\{P'\}\!\}, A \Vdash G, \{\!\{Q'\}\!\}$

shows $c \leq \{\!\{P \wedge P'\}\!\}, A \Vdash G, \{\!\{\lambda rv. Q \ rv \wedge Q' \ rv\}\!\}$

\langle *proof* \rangle

lemma *drop-imp*:

assumes $c \leq \{\!\{P\}\!\}, A \Vdash G, \{\!\{Q\}\!\}$

shows $c \leq \{\!\{P\}\!\}, A \Vdash G, \{\!\{\lambda rv. Q' \ rv \longrightarrow Q \ rv\}\!\}$

\langle *proof* \rangle

lemma *prop*:

shows $c \leq \{\!\{\langle P \rangle\}\!\}, A \Vdash c, \{\!\{\lambda v. \langle P \rangle\}\!\}$

\langle *proof* \rangle

lemma *name-pre-state*:

assumes $\bigwedge s. P \ s \implies c \leq \{\!\{ (=) \ s \}\!\}, A \Vdash G, \{\!\{Q\}\!\}$

assumes *spec.idle* $\leq G$

shows $c \leq \{\!\{P\}\!\}, A \Vdash G, \{\!\{Q\}\!\}$ (**is** *?lhs* \leq *?rhs*)

\langle *proof* \rangle

\langle *ML* \rangle

12.1 Geenral rules for the (*'a*, *'s*, *'v*) *spec* lattice

\langle *ML* \rangle

lemma *refinement*:

shows *spec.term.all* ($\{\!\{P\}\!\}, A \Vdash G, \{\!\{Q\}\!\}) = \{\!\{P\}\!\}, \text{spec.term.all } A \Vdash \text{spec.term.all } G, \{\!\{\top\}\!\}$

$\langle proof \rangle$

$\langle ML \rangle$

lemma refinement-le:

shows $spec.term.none (\{P\}, A \Vdash G, \{Q\}) \leq \{P\}, spec.term.all A \Vdash spec.term.all G, \{\perp\}$

$\langle proof \rangle$

$\langle ML \rangle$

lemma refinement:

fixes $af :: 'a \Rightarrow 'b$

fixes $sf :: 's \Rightarrow 't$

fixes $vf :: 'v \Rightarrow 'w$

fixes $A :: ('b, 't, 'w) spec$

fixes $G :: ('b, 't, 'w) spec$

fixes $P :: 't pred$

fixes $Q :: 'w \Rightarrow 't pred$

shows $spec.invmap af sf vf (\{P\}, A \Vdash G, \{Q\})$

$= (\{\lambda s. P (sf s)\}, spec.invmap af sf vf A \Vdash spec.invmap af sf vf G, \{\lambda v s. Q (vf v) (sf s)\})$

$\langle proof \rangle$

$\langle ML \rangle$

12.1.1 Actions

Actions are anchored at the start of a trace, and therefore must be an initial step of the assume A . However by the semantics of (\longrightarrow_+) we may only know that that initial state of the trace is acceptable to A when showing that F -steps are F' -steps (the second assumption). This hypothesis is vacuous when $idle \leq A$.

$\langle ML \rangle$

lemma action:

fixes $F :: ('v \times 'a \times 's \times 's) set$

assumes $\bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in F; (a, s, s') \in spec.initial-steps A \rrbracket \Longrightarrow Q v s'$

assumes $\bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in F; (a, s, s) \in spec.initial-steps A \rrbracket \Longrightarrow (v, a, s, s') \in F'$

shows $spec.action F \leq \{P\}, A \Vdash spec.action F', \{Q\}$

$\langle proof \rangle$

lemma return:

shows $spec.return v \leq \{Q v\}, A \Vdash spec.return v, \{Q\}$

$\langle proof \rangle$

lemma action-rel:

fixes $F :: ('v \times 'a \times 's \times 's) set$

assumes $\bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in F; (a, s, s') \in spec.initial-steps A \rrbracket \Longrightarrow Q v s'$

assumes $\bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in F; (a, s, s) \in spec.initial-steps A; s \neq s' \rrbracket \Longrightarrow (a, s, s') \in r$

shows $spec.action F \leq \{P\}, A \Vdash spec.rel r, \{Q\}$

$\langle proof \rangle$

$\langle ML \rangle$

12.1.2 Bind

Consider showing $f \ggg g \leq f' \ggg g'$ under the assume A and pre/post conditions P/Q .

The tricky part is to residuate the assume A wrt the process f for use in the refinement proof of g .

- we want to preserve as much of the structure of A as possible

- intuition: we want all the ways a trace can continue in A having started with a terminating trace in f
- intuitively a right residual for (\gg) should do the job
 - however unlike [Hoare and He \(1987\)](#) we have no chance of a right residual for (\gg) as we use traces (they use relations)
 - * i.e., if it is not the case that $f \gg \perp \leq A$ then there is no continuation h such that $f \gg h \leq A$.
 - * also such a residual h has arbitrary behavior starting from states that f cannot reach
 - i.e., for traces $\neg\sigma \leq f \langle\sigma\rangle \gg h \leq A$ need not hold
 - and the user-provided assertions may be too weak to correct for this
- in practice the termination information in the assume A is not useful

We therefore define an ad hoc residual that does the trick.

See [Emerson \(1983, §4\)](#) for some related concerns.

$\langle ML \rangle$

definition $res :: ('a, 's, 'v) spec \Rightarrow ('a, 's, 'w) spec \Rightarrow 'v \Rightarrow ('a, 's, 'w) spec$ **where**
 $res\ f\ A\ v = \sqcup \{ \langle trace.final'\ s\ xs,\ ys,\ w \rangle \mid s\ xs\ ys\ w.\ \langle s,\ xs,\ Some\ v \rangle \leq f \wedge \langle s,\ xs\ @\ ys,\ None \rangle \leq A \}$

$\langle ML \rangle$

lemma $res\text{-}le\text{-}conv[spec.singleton.le\text{-}conv]$:

shows $\langle\sigma\rangle \leq refinement.spec.bind.res\ f\ A\ v$

$\longleftrightarrow (\exists s\ xs.\ \langle s,\ xs,\ Some\ v \rangle \leq f$
 $\wedge trace.init\ \sigma = trace.final'\ s\ xs$
 $\wedge \langle s,\ xs\ @\ trace.rest\ \sigma,\ None \rangle \leq A)$ (**is** $?lhs \longleftrightarrow ?rhs$)

$\langle proof \rangle$

$\langle ML \rangle$

lemma $resL$:

shows $refinement.spec.bind.res\ (spec.term.none\ f)\ A\ v = \perp$

$\langle proof \rangle$

lemma $resR$:

shows $refinement.spec.bind.res\ f\ (spec.term.none\ A)\ v = refinement.spec.bind.res\ f\ A\ v$

$\langle proof \rangle$

$\langle ML \rangle$

lemma $resR\text{-}mono$:

shows $refinement.spec.bind.res\ f\ (spec.term.all\ A)\ v = refinement.spec.bind.res\ f\ A\ v$

$\langle proof \rangle$

lemma res :

shows $spec.term.all\ (refinement.spec.bind.res\ f\ A\ v) = refinement.spec.bind.res\ f\ A\ v$

$\langle proof \rangle$

$\langle ML \rangle$

lemma res :

shows $refinement.spec.bind.res\ f\ A\ v \in spec.term.closed$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *bot*:

shows *botL*: $\text{refinement.spec.bind.res } \perp = \perp$

and *botR*: $\text{refinement.spec.bind.res } f \perp = \perp$

$\langle \text{proof} \rangle$

lemma *mono*:

assumes $f \leq f'$

assumes $A \leq A'$

shows $\text{refinement.spec.bind.res } f A v \leq \text{refinement.spec.bind.res } f' A' v$

$\langle \text{proof} \rangle$

lemma *strengthen[stg]*:

assumes *st-ord* $F f f'$

assumes *st-ord* $F A A'$

shows *st-ord* $F (\text{refinement.spec.bind.res } f A v) (\text{refinement.spec.bind.res } f' A' v)$

$\langle \text{proof} \rangle$

lemma *mono2mono[cont-intro, partial-function-mono]*:

assumes *monotone orda* $(\leq) f$

assumes *monotone orda* $(\leq) A$

shows *monotone orda* $(\leq) (\lambda x. \text{refinement.spec.bind.res } (f x) (A x) v)$

$\langle \text{proof} \rangle$

lemma *SupL*:

shows $\text{refinement.spec.bind.res } (\bigsqcup X) A v = (\bigsqcup x \in X. \text{refinement.spec.bind.res } x A v)$

$\langle \text{proof} \rangle$

lemma *SupR*:

shows $\text{refinement.spec.bind.res } f (\bigsqcup X) v = (\bigsqcup x \in X. \text{refinement.spec.bind.res } f x v)$

$\langle \text{proof} \rangle$

lemma *InfL-le*:

shows $\text{refinement.spec.bind.res } (\bigsqcap X) A v \leq (\bigsqcap x \in X. \text{refinement.spec.bind.res } x A v)$

$\langle \text{proof} \rangle$

lemma *InfR-le*:

shows $\text{refinement.spec.bind.res } f (\bigsqcap X) v \leq (\bigsqcap x \in X. \text{refinement.spec.bind.res } f x v)$

$\langle \text{proof} \rangle$

lemma *mcont2mcont[cont-intro]*:

assumes *mcont luba orda Sup* $(\leq) f$

assumes *mcont luba orda Sup* $(\leq) A$

shows *mcont luba orda Sup* $(\leq) (\lambda x. \text{refinement.spec.bind.res } (f x) (A x) v)$

$\langle \text{proof} \rangle$

lemma *returnL*:

assumes *spec.idle* $\leq A$

shows $\text{refinement.spec.bind.res } (\text{spec.return } v) A v = \text{spec.term.all } A$ (**is** *?lhs = ?rhs*)

$\langle \text{proof} \rangle$

lemma *rel-le*:

assumes $r \subseteq r'$

shows $\text{refinement.spec.bind.res } f (\text{spec.rel } r) v \leq \text{spec.rel } r'$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *res-le*: — we can always discard the extra structure

shows $spec.steps (refinement.spec.bind.res f A v) \subseteq spec.steps A$
 ⟨proof⟩

⟨ML⟩

A refinement rule for (\gg). The function vf abstracts interstitial monadic return values.

⟨ML⟩

lemma *bind-abstract*:

fixes $f :: ('a, 's, 'v) spec$

fixes $f' :: ('a, 's, 'v') spec$

fixes $g :: 'v \Rightarrow ('a, 's, 'w) spec$

fixes $g' :: 'v' \Rightarrow ('a, 's, 'w) spec$

fixes $vf :: 'v \Rightarrow 'v'$

assumes $g: \bigwedge v. g v \leq \{\{Q' (vf v)\}\}, refinement.spec.bind.res (spec.pre P \sqcap spec.term.all A \sqcap f') A (vf v) \Vdash g' (vf v), \{\{Q\}\}$

assumes $f: f \leq \{\{P\}\}, spec.term.all A \Vdash spec.vinvmmap vf f', \{\{\lambda v. Q' (vf v)\}\}$

shows $f \gg g \leq \{\{P\}\}, A \Vdash f' \gg g', \{\{Q\}\}$

⟨proof⟩

lemmas $bind = refinement.spec.bind-abstract[\mathbf{where} \text{ } vf=id, \text{ simplified } spec.invmmap.id, \text{ simplified}]$

12.1.3 Interference

lemma *rel-mono*:

assumes $r \subseteq r'$

assumes $stable (snd \text{ ' } (spec.steps A \sqcap r)) P$

shows $spec.rel r \leq \{\{P\}\}, A \Vdash spec.rel r', \{\{\lambda -:unit. P\}\}$

⟨proof⟩

⟨ML⟩

12.1.4 Parallel

Our refinement rule for *Parallel* does not constrain the constituent processes in any way, unlike Abadi and Plotkin's proposed rule (see §9.2).

⟨ML⟩

definition — roughly the *Parallel* construction with roles reversed

$env-hyp :: ('a \Rightarrow 's \text{ pred}) \Rightarrow (sequential, 's, unit) spec \Rightarrow 'a \text{ set} \Rightarrow ('a \Rightarrow (sequential, 's, unit) spec) \Rightarrow 'a \Rightarrow (sequential, 's, unit) spec$

where

$env-hyp P A as Ps a =$

$spec.pre (\sqcap (P \text{ ' } as))$

$\sqcap spec.amap (toConcurrent-fn (proc a))$

$(spec.rel ((\{env\} \cup proc \text{ ' } as) \times UNIV))$

$\sqcap (\sqcap i \in as. spec.toConcurrent i (Ps i))$

$\sqcap spec.ainvmmap toSequential-fn A)$

⟨ML⟩

lemma *mono*:

assumes $\bigwedge a. a \in as \Longrightarrow P a \leq P' a$

assumes $A \leq A'$

assumes $\bigwedge a. a \in as \Longrightarrow Ps a \leq Ps' a$

shows $refinement.spec.env-hyp P A as Ps a \leq refinement.spec.env-hyp P' A' as Ps' a$

⟨proof⟩

lemma *strengthen*[*strg*]:

assumes $\bigwedge a. a \in as \implies st\text{-ord } F (P a) (P' a)$

assumes $st\text{-ord } F A A'$

assumes $\bigwedge a. a \in as \implies st\text{-ord } F (Ps a) (Ps' a)$

shows $st\text{-ord } F (refinement.spec.env\text{-hyp } P A as Ps a) (refinement.spec.env\text{-hyp } P' A' as Ps' a)$

<proof>

<ML>

lemma *Parallel*:

fixes $A :: (sequential, 's, unit) spec$

fixes $Q :: 'a \Rightarrow 's \Rightarrow bool$

fixes $Ps :: 'a \Rightarrow (sequential, 's, unit) spec$

fixes $Ps' :: 'a \Rightarrow (sequential, 's, unit) spec$

assumes $\bigwedge a. a \in as \implies Ps a \leq \{\!\{P a}\!\}, refinement.spec.env\text{-hyp } P A as Ps' a \Vdash Ps' a, \{\!\{\lambda v. Q a}\!\}$

shows $spec.Parallel as Ps \leq \{\!\{\bigcap_{a \in as} P a}\!\}, A \Vdash spec.Parallel as Ps', \{\!\{\lambda v. \bigcap_{a \in as} Q a}\!\}$

<proof>

<ML>

12.2 A relational assume/guarantee program logic for the *(sequential, 's, 'v) spec* lattice

Here we develop an assume/guarantee story based on abstracting processes (represented as safety properties) to binary relations.

Observations:

- this can be seen as a reconstruction of the algebraic account given by [van Staden \(2015\)](#) in our setting
- we show Heyting implication suffices for relations (see *ag.refinement*)
 - the processes' agent type is required to be *sequential*
- we use predicates and not relations for pre/post assertions
 - we can use the metalanguage to do some relational reasoning; see, for example, *ag.name-pre-state*
- *Id* is the smallest significant assume and guarantee relation here; processes can always stutter any state

<ML>

abbreviation *(input) assm* $:: 's rel \Rightarrow (sequential, 's, 'v) spec$ **where**

$assm A \equiv spec.rel (\{env\} \times A \cup \{self\} \times UNIV)$

abbreviation *(input) guar* $:: 's rel \Rightarrow (sequential, 's, 'v) spec$ **where**

$guar G \equiv spec.rel (\{env\} \times UNIV \cup \{self\} \times G)$

<ML>

definition *ag* $:: 's pred \Rightarrow 's rel \Rightarrow 's rel \Rightarrow ('v \Rightarrow 's pred) \Rightarrow (sequential, 's, 'v) spec (\{\!\{-\}\!\}, -/\vdash -, \{\!\{-\}\!\} [0,0,0,0])$ **where**

$\{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\} = spec.pre P \sqcap ag.assm A \longrightarrow_H ag.guar G \sqcap spec.post Q$

<ML>

lemma *ag*: — Note $af = id$

fixes $sf :: 's \Rightarrow 't$

fixes $vf :: 'v \Rightarrow 'w$

fixes $A :: 't rel$

fixes $G :: 't rel$

fixes $P :: 't \text{ pred}$
fixes $Q :: 'w \Rightarrow 't \text{ pred}$
shows $\text{spec.invmap id sf vf } (\{P\}, A \vdash G, \{Q\}) = \{\lambda s. P (sf s)\}, \text{inv-image } (A^=) sf \vdash \text{inv-image } (G^=) sf, \{\lambda v$
 $s. Q (vf v) (sf s)\}$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma refinement:

shows $\{P\}, A \vdash G, \{Q\} = \{P\}, \text{ag.assm } A \vdash \text{ag.guar } G, \{Q\}$
 $\langle \text{proof} \rangle$

lemma E:

assumes $c \leq \{P\}, A \vdash G, \{Q\}$
obtains $c \leq \text{spec.pre } P \sqcap \text{ag.assm } A \longrightarrow_H \text{ag.guar } G$
and $c \leq \text{spec.pre } P \sqcap \text{ag.assm } A \longrightarrow_H \text{spec.post } Q$
 $\langle \text{proof} \rangle$

lemma pre-post-cong:

assumes $P = P'$
assumes $Q = Q'$
shows $\{P\}, A \vdash G, \{Q\} = \{P'\}, A \vdash G, \{Q'\}$
 $\langle \text{proof} \rangle$

lemma pre-bot:

shows $\{\perp\}, A \vdash G, \{Q\} = \top$
and $\{\langle \perp \rangle\}, A \vdash G, \{Q\} = \top$
and $\{\langle \text{False} \rangle\}, A \vdash G, \{Q\} = \top$
 $\langle \text{proof} \rangle$

lemma post-top:

shows $\{P\}, A \vdash \text{UNIV}, \{\top\} = \top$
and $\{P\}, A \vdash \text{UNIV}, \{\langle \top \rangle\} = \top$
and $\{P\}, A \vdash \text{UNIV}, \{\lambda -. \text{True}\} = \top$
 $\langle \text{proof} \rangle$

lemma mono:

assumes $P' \leq P$
assumes $A' \leq A$
assumes $G \leq G'$
assumes $Q \leq Q'$
shows $\{P\}, A \vdash G, \{Q\} \leq \{P'\}, A' \vdash G', \{Q'\}$
 $\langle \text{proof} \rangle$

lemma strengthen[strg]:

assumes $\text{st-ord } (\neg F) P P'$
assumes $\text{st-ord } (\neg F) A A'$
assumes $\text{st-ord } F G G'$
assumes $\text{st-ord } F Q Q'$
shows $\text{st-ord } F (\{P\}, A \vdash G, \{Q\}) (\{P'\}, A' \vdash G', \{Q'\})$
 $\langle \text{proof} \rangle$

lemma strengthen-pre:

assumes $\text{st-ord } (\neg F) P P'$
shows $\text{st-ord } F (\{P\}, A \vdash G, \{Q\}) (\{P'\}, A' \vdash G, \{Q\})$
 $\langle \text{proof} \rangle$

lemmas $\text{pre-ag} = \text{order.trans}[\text{OF} - \text{ag.mono}[\text{OF order.refl} - - \text{order.refl}], \text{of } c]$ **for** c

lemmas *pre-a* = *ag.pre-ag*[*OF* - - *order.refl*]

lemmas *pre-g* = *ag.pre-ag*[*OF* - - *order.refl*]

lemma *pre*:

assumes $c \leq \{P\}, A \vdash G, \{Q\}$

assumes $\bigwedge s. P' s \implies P s$

assumes $A' \subseteq A$

assumes $G \subseteq G'$

assumes $\bigwedge v s. Q v s \implies Q' v s$

shows $c \leq \{P'\}, A' \vdash G', \{Q'\}$

<proof>

lemmas *pre-pre-post* = *ag.pre*[*OF* - - *order.refl order.refl*, of *c*] **for** *c*

lemma *pre-imp*:

assumes $\bigwedge s. P s \implies P' s$

assumes $c \leq \{P'\}, A \vdash G, \{Q\}$

shows $c \leq \{P\}, A \vdash G, \{Q\}$

<proof>

lemmas *pre-pre* = *ag.pre-imp*[*rotated*]

lemma *post-imp*:

assumes $\bigwedge v s. Q v s \implies Q' v s$

assumes $c \leq \{P\}, A \vdash G, \{Q\}$

shows $c \leq \{P\}, A \vdash G, \{Q'\}$

<proof>

lemmas *pre-post* = *ag.post-imp*[*rotated*]

lemmas *strengthen-post* = *ag.pre-post*

lemmas *reflcl-ag* = *spec.invmap.ag*[**where** *sf=id* **and** *vf=id*, *simplified spec.invmap.id*, *simplified*]

lemma

shows *reflcl-a*: $\{P\}, A \vdash G, \{Q\} = \{P\}, A^= \vdash G, \{Q\}$

and *reflcl-g*: $\{P\}, A \vdash G, \{Q\} = \{P\}, A \vdash G^=, \{Q\}$

<proof>

lemma *gen-asm-base*:

assumes $P \implies c \leq \{P' \wedge P''\}, A \vdash G, \{Q\}$

shows $c \leq \{P' \wedge \langle P \rangle \wedge P''\}, A \vdash G, \{Q\}$

<proof>

lemmas *gen-asm* =

ag.gen-asm-base[**where** $P'=\langle True \rangle$ **and** $P''=\langle True \rangle$, *simplified*]

ag.gen-asm-base[**where** $P'=\langle True \rangle$, *simplified*]

ag.gen-asm-base[**where** $P''=\langle True \rangle$, *simplified*]

ag.gen-asm-base

lemma *post-conj*:

assumes $c \leq \{P\}, A \vdash G, \{Q\}$

assumes $c \leq \{P\}, A \vdash G, \{Q'\}$

shows $c \leq \{P\}, A \vdash G, \{\lambda v. Q v \wedge Q' v\}$

<proof>

lemma *pre-disj*:

assumes $c \leq \{P\}, A \vdash G, \{Q\}$

assumes $c \leq \{P'\}, A \vdash G, \{Q\}$

shows $c \leq \{\!\{P \vee P'\}\!\}, A \vdash G, \{\!\{Q\}\!\}$
 $\langle proof \rangle$

lemma *drop-imp*:

assumes $c \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$
shows $c \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{\lambda v. Q' v \longrightarrow Q v\}\!\}$
 $\langle proof \rangle$

lemma *prop*:

shows $c \leq \{\!\{\langle P \rangle\}\!\}, A \vdash UNIV, \{\!\{\lambda v. \langle P \rangle\}\!\}$
 $\langle proof \rangle$

lemma *name-pre-state*:

assumes $\bigwedge s. P s \implies c \leq \{\!\{ (=) s \}\!\}, A \vdash G, \{\!\{Q\}\!\}$
shows $c \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$
 $\langle proof \rangle$

lemma *conj-lift*:

assumes $c \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$
assumes $c \leq \{\!\{P'\}\!\}, A \vdash G, \{\!\{Q'\}\!\}$
shows $c \leq \{\!\{P \wedge P'\}\!\}, A \vdash G, \{\!\{\lambda v. Q v \wedge Q' v\}\!\}$
 $\langle proof \rangle$

lemma *disj-lift*:

assumes $c \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$
assumes $c \leq \{\!\{P'\}\!\}, A \vdash G, \{\!\{Q'\}\!\}$
shows $c \leq \{\!\{P \vee P'\}\!\}, A \vdash G, \{\!\{\lambda v. Q v \vee Q' v\}\!\}$
 $\langle proof \rangle$

lemma *all-lift*:

assumes $\bigwedge x. c \leq \{\!\{P x\}\!\}, A \vdash G, \{\!\{Q x\}\!\}$
shows $c \leq \{\!\{\forall x. P x\}\!\}, A \vdash G, \{\!\{\lambda v. \forall x. Q x v\}\!\}$
 $\langle proof \rangle$

lemma *interference-le*:

shows $spec.rel (\{env\} \times UNIV) \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{\top\}\!\}$
and $spec.rel (\{env\} \times UNIV) \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{\lambda -. \top\}\!\}$
and $spec.rel (\{env\} \times UNIV) \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{\lambda -. True\}\!\}$
 $\langle proof \rangle$

lemma *assm-heyting*:

fixes $Q :: 'v \Rightarrow 's\ pred$
shows $ag.assm\ r \longrightarrow_H \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\} = \{\!\{P\}\!\}, A \cap r \vdash G, \{\!\{Q\}\!\}$
 $\langle proof \rangle$

lemma *augment-a*: — instantiate A'

assumes $c \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$
shows $c \leq \{\!\{P\}\!\}, A \cap A' \vdash G, \{\!\{Q\}\!\}$
 $\langle proof \rangle$

lemma *augment-post*: — instantiate Q

assumes $c \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{\lambda v. Q' v \wedge Q v\}\!\}$
shows $c \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q'\}\!\}$
 $\langle proof \rangle$

lemma *augment-post-imp*: — instantiate Q

assumes $c \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{\lambda v. (Q v \longrightarrow Q' v) \wedge Q v\}\!\}$
shows $c \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q'\}\!\}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *ag-le*:

shows *spec.term.none* ($\{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}) \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{\perp\}\!\}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemmas *none-inteference* =

order.trans[*OF spec.term.none.mono*,

OF ag.interference-le(1) ag.pre-post[**where** $Q'=Q$ **for** Q , *OF spec.term.none.ag-le, simplified*]]

$\langle ML \rangle$

lemma *bind*:

assumes $g: \bigwedge v. g v \leq \{\!\{Q' v\}\!\}, A \vdash G, \{\!\{Q\}\!\}$

assumes $f: f \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q'\}\!\}$

shows $f \ggg g \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$

$\langle \text{proof} \rangle$

lemma *action*:

fixes $F :: ('v \times \text{sequential} \times 's \times 's)$ *set*

assumes $Q: \bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in F \rrbracket \implies Q v s'$

assumes $G: \bigwedge v s s'. \llbracket P s; (v, \text{self}, s, s') \in F; s \neq s' \rrbracket \implies (s, s') \in G$

shows *spec.action* $F \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$

$\langle \text{proof} \rangle$

lemma *return*:

shows *spec.return* $v \leq \{\!\{Q v\}\!\}, A \vdash G, \{\!\{Q\}\!\}$

$\langle \text{proof} \rangle$

lemma *Parallel-assm*:

shows *refinement.spec.env-hyp* $P (ag.assm A) as (ag.guar \circ G) a \leq ag.assm (A \cup \bigcup (G \text{ ' } (as - \{a\})))$

$\langle \text{proof} \rangle$

lemma *Parallel-guar*:

shows *spec.Parallel* $as (ag.guar \circ G) = ag.guar (\bigcup a \in as. G a)$

$\langle \text{proof} \rangle$

lemma *Parallel*:

fixes $A :: 's \text{ rel}$

fixes $G :: 'a \Rightarrow 's \text{ rel}$

fixes $Q :: 'a \Rightarrow 's \Rightarrow \text{bool}$

fixes $P_s :: 'a \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec}$

assumes *proc-ag*: $\bigwedge a. a \in as \implies P_s a \leq \{\!\{P a\}\!\}, A \cup (\bigcup a' \in as - \{a\}. G a') \vdash G a, \{\!\{\lambda v. Q a\}\!\}$

shows *spec.Parallel* $as P_s \leq \{\!\{\bigwedge a \in as. P a\}\!\}, A \vdash \bigcup a \in as. G a, \{\!\{\lambda rv. \bigwedge a \in as. Q a\}\!\}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

12.2.1 Stability rules

$\langle ML \rangle$

lemma *stable-pre-post*:

fixes $S :: ('a, 's, 'v) \text{ spec}$

assumes $stable (snd \text{ ' } r) P$
assumes $spec.steps S \subseteq r$
shows $S \leq spec.pre P \longrightarrow_H spec.post \langle P \rangle$
 $\langle proof \rangle$

lemma *pre-post-stable*:

fixes $P :: 's \Rightarrow bool$
assumes $stable (snd \text{ ' } r) P$
shows $spec.rel r \leq spec.pre P \longrightarrow_H spec.post \langle P \rangle$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *stable-lift*:

assumes $stable (A \cup G) P'$ — anything stable over $A \cup G$ is invariant
shows $\{\!\{P \wedge P'\}\!\}, A \vdash G, \{\!\{\lambda v. P' \longrightarrow Q v\}\!\} \leq \{\!\{P \wedge P'\}\!\}, A \vdash G, \{\!\{\lambda v. Q v \wedge P'\}\!\}$
 $\langle proof \rangle$

lemma *stable-augment-base*:

assumes $c \leq \{\!\{P \wedge P'\}\!\}, A \vdash G, \{\!\{\lambda v. P' \longrightarrow Q v\}\!\}$
assumes $stable (A \cup G) P'$ — anything stable over $A \cup G$ is invariant
shows $c \leq \{\!\{P \wedge P'\}\!\}, A \vdash G, \{\!\{\lambda v. Q v \wedge P'\}\!\}$
 $\langle proof \rangle$

lemma *stable-augment*:

assumes $c \leq \{\!\{P'\}\!\}, A \vdash G, \{\!\{Q'\}\!\}$
assumes $\bigwedge v s. \llbracket P s; Q' v s \rrbracket \Longrightarrow Q v s$
assumes $stable (A \cup G) P$
shows $c \leq \{\!\{P' \wedge P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$
 $\langle proof \rangle$

lemma *stable-augment-post*:

assumes $c \leq \{\!\{P'\}\!\}, A \vdash G, \{\!\{Q'\}\!\}$ — resolve before application
assumes $\bigwedge v. stable (A \cup G) (Q' v \longrightarrow Q v)$
shows $c \leq \{\!\{(\forall v. Q' v \longrightarrow Q v) \wedge P'\}\!\}, A \vdash G, \{\!\{Q\}\!\}$
 $\langle proof \rangle$

lemma *stable-augment-frame*: — anything stable over $A \cup G$ is invariant

assumes $c \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$
assumes $stable (A \cup G) P'$
shows $c \leq \{\!\{P \wedge P'\}\!\}, A \vdash G, \{\!\{\lambda v. Q v \wedge P'\}\!\}$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *stable-interference*:

assumes $stable (A \cap r) P$
shows $spec.rel (\{env\} \times r) \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{\langle P \rangle\}\!\}$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *closed-ag*:

shows $\{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\} \in spec.cam.closed (\{env\} \times r)$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *cl-ag-le*:

assumes P : *stable* ($A \cap r$) P

assumes Q : $\bigwedge v$. *stable* ($A \cap r$) ($Q v$)

shows *spec.interference.cl* ($\{\text{env}\} \times r$) ($\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$) \leq $\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$

\langle *proof* \rangle

lemma *closed-ag*:

assumes P : *stable* ($A \cap r$) P

assumes Q : $\bigwedge v$. *stable* ($A \cap r$) ($Q v$)

shows $\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$ \in *spec.interference.closed* ($\{\text{env}\} \times r$)

\langle *proof* \rangle

\langle *ML* \rangle

13 A programming language

The $(\prime a, \prime s, \prime v)$ *spec* lattice of §8.2 is adequate for logic but is deficient as a programming language. In particular we wish to interpret the parallel composition as intersection (§9.5) which requires processes to contain enough interference opportunities. Similarly we want the customary “laws of programming” (Hoare, Hayes, He, Morgan, Roscoe, Sanders, Sørensen, Spivey, and Sufrin 1987a) to hold without side conditions.

These points are discussed at some length by Zwiers (1989, §3.2) and also Foster, Baxter, Cavalcanti, Woodcock, and Zeyda (2020, Lemma 6.7).

Our $(\prime v, \prime s)$ *prog* lattice (§13.1) therefore handles the common case of the familiar constructs for sequential programming, and we lean on our $(\prime a, \prime s, \prime v)$ *spec* lattice for other constructions such as interleaving parallel composition (§9.5) and local state (§15). It allows arbitrary interference by the environment before and after every program action.

13.1 The $(\prime s, \prime v)$ *prog* lattice

According to Müller-Olm (1997, §2.1), $(\prime s, \prime v)$ *prog* is a *sub-lattice* of $(\prime a, \prime s, \prime v)$ *spec* as the corresponding (\sqcap) and (\sqcup) operations coincide. However it is not a *complete* sub-lattice as *Sup* in $(\prime s, \prime v)$ *prog* needs to account for the higher bottom of that lattice.

typedef $(\prime s, \prime v)$ *prog* = *spec.interference.closed* ($\{\text{env}\} \times UNIV$) :: (*sequential*, $\prime s, \prime v$) *spec set*

morphisms *p2s Abs-t*

\langle *proof* \rangle

hide-const (*open*) *p2s*

setup-lifting *type-definition-prog*

instantiation *prog* :: (*type, type*) *complete-distrib-lattice*

begin

lift-definition *bot-prog* :: $(\prime s, \prime v)$ *prog is spec.interference.cl* ($\{\text{env}\} \times UNIV$) \perp \langle *proof* \rangle

lift-definition *top-prog* :: $(\prime s, \prime v)$ *prog is* \top \langle *proof* \rangle

lift-definition *sup-prog* :: $(\prime s, \prime v)$ *prog* \Rightarrow $(\prime s, \prime v)$ *prog* \Rightarrow $(\prime s, \prime v)$ *prog is sup* \langle *proof* \rangle

lift-definition *inf-prog* :: $(\prime s, \prime v)$ *prog* \Rightarrow $(\prime s, \prime v)$ *prog* \Rightarrow $(\prime s, \prime v)$ *prog is inf* \langle *proof* \rangle

lift-definition *less-eq-prog* :: $(\prime s, \prime v)$ *prog* \Rightarrow $(\prime s, \prime v)$ *prog* \Rightarrow *bool is less-eq* \langle *proof* \rangle

lift-definition *less-prog* :: $(\prime s, \prime v)$ *prog* \Rightarrow $(\prime s, \prime v)$ *prog* \Rightarrow *bool is less* \langle *proof* \rangle

lift-definition *Inf-prog* :: $(\prime s, \prime v)$ *prog set* \Rightarrow $(\prime s, \prime v)$ *prog is Inf* \langle *proof* \rangle

lift-definition *Sup-prog* :: $(\prime s, \prime v)$ *prog set* \Rightarrow $(\prime s, \prime v)$ *prog is* λX . *Sup* $X \sqcup$ *spec.interference.cl* ($\{\text{env}\} \times UNIV$) \perp \langle *proof* \rangle

instance

\langle *proof* \rangle

end

13.2 Morphisms to and from the (*sequential, 's, 'v*) *spec* lattice

We can readily convert a (*'s, 'v*) *prog* into a (*'a agent, 's, 'v*) *spec*. More interestingly, on (*'s, 'v*) *prog* we have a Galois connection that embeds specifications into programs. (This connection is termed a *Galois insertion* by Melton et al. (1985) as we also have *prog.s2p.p2s*; Cousot says “Galois retraction”.)

See also §13.4.2 and §13.5.1.

⟨*ML*⟩

lemmas *p2s[iff] = prog.p2s*

⟨*ML*⟩

lemmas *p2s = spec.interference.closed-conv[OF spec.interference.closed.p2s, symmetric, of P for P]*

⟨*ML*⟩

lemmas *p2s-le[spec.idle-le]*

= spec.interference.le-closedE[OF spec.idle.interference.cl-le spec.interference.closed.p2s, of P for P]

lemmas *p2s-minimal[iff] = order.trans[OF spec.idle.minimal-le spec.idle.p2s-le]*

⟨*ML*⟩

lemma *p2s-leI:*

assumes *prog.p2s c ≤ prog.p2s d*

shows *c ≤ d*

⟨*proof*⟩

⟨*ML*⟩

named-theorems *simps* ⟨*simp rules for const*⟨*p2s*⟩⟩

lemmas *bot = bot-prog.rep-eq*

lemmas *top = top-prog.rep-eq*

lemmas *inf = inf-prog.rep-eq*

lemmas *sup = sup-prog.rep-eq*

lemmas *Inf = Inf-prog.rep-eq*

lemmas *Sup = Sup-prog.rep-eq*

lemma *Sup-not-empty:*

assumes *X ≠ {}*

shows *prog.p2s (⊔ X) = ⊔ (prog.p2s ` X)*

⟨*proof*⟩

lemma *SUP-not-empty:*

assumes *X ≠ {}*

shows *prog.p2s (⊔ x∈X. f x) = (⊔ x∈X. prog.p2s (f x))*

⟨*proof*⟩

lemma *monotone:*

shows *mono prog.p2s*

⟨*proof*⟩

lemmas *strengthen[strg] = st-monotone[OF prog.p2s.monotone]*

lemmas *mono = monotoneD[OF prog.p2s.monotone]*

lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF prog.p2s.monotone*, *simplified*, *of orda P for orda P*]

lemma *mcont*: — Morally *galois.complete-lattice.mcont-lower*
shows *mcont Sup* (\leq) *Sup* (\leq) *prog.p2s*
<proof>

lemmas *mcont2mcont*[*cont-intro*] = *mcont2mcont*[*OF prog.p2s.mcont*, *of luba orda P for luba orda P*]

lemmas *Let-distrib* = *Let-distrib*[**where** *f=prog.p2s*]

lemmas [*prog.p2s.simps*] =
prog.p2s.bot
prog.p2s.top
prog.p2s.inf
prog.p2s.sup
prog.p2s.Inf
prog.p2s.Sup-not-empty
spec.interference.cl.p2s
prog.p2s.Let-distrib

lemma *interference-wind-bind*:
shows *spec.rel* ($\{\text{env}\} \times \text{UNIV}$) $\gg=$ ($\lambda::\text{unit. prog.p2s } P$) = *prog.p2s P*
<proof>

<ML>

definition *s2p* :: (*sequential*, '*s*', '*v*') *spec* \Rightarrow ('*s*', '*v*') *prog where* — Morally the upper of a Galois connection
s2p P = $\bigsqcup \{c. \text{prog.p2s } c \leq P\}$

<ML>

lemma *bottom*:
shows *prog.s2p* \perp = \perp
<proof>

lemma *top*:
shows *prog.s2p* \top = \top
<proof>

lemma *monotone*:
shows *mono prog.s2p*
<proof>

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.s2p.monotone*]

lemmas *mono* = *monotoneD*[*OF prog.s2p.monotone*]

lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF prog.s2p.monotone*, *simplified*]

lemma *p2s*:
shows *prog.s2p* (*prog.p2s P*) = *P*
<proof>

lemma *Sup-le*:
shows $\bigsqcup (\text{prog.s2p } 'X) \leq \text{prog.s2p } (\bigsqcup X)$
<proof>

lemma *sup-le*:

shows $\text{prog.s2p } x \sqcup \text{prog.s2p } y \leq \text{prog.s2p } (x \sqcup y)$
 $\langle \text{proof} \rangle$

lemma *Inf*:

shows $\text{prog.s2p } (\sqcap X) = \sqcap (\text{prog.s2p } ` X)$ (**is** $?lhs = ?rhs$)
 $\langle \text{proof} \rangle$

lemma *inf*:

shows $\text{prog.s2p } (x \sqcap y) = \text{prog.s2p } x \sqcap \text{prog.s2p } y$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *galois*: — the Galois connection

shows $\text{prog.p2s } c \leq S$
 $\iff c \leq \text{prog.s2p } S \wedge \text{spec.term.none } (\text{spec.rel } (\{env\} \times UNIV) :: (-, -, \text{unit}) \text{spec}) \leq S$ (**is** $?lhs \iff ?rhs$)
 $\langle \text{proof} \rangle$

lemma *le*:

shows $\text{prog.p2s } (\text{prog.s2p } S) \leq \text{spec.interference.cl } (\{env\} \times UNIV) S$
 $\langle \text{proof} \rangle$

lemma *insertion*:

fixes $S :: (\text{sequential}, 's, 'v) \text{spec}$
assumes $S \in \text{spec.interference.closed } (\{env\} \times UNIV)$
shows $\text{prog.p2s } (\text{prog.s2p } S) = S$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

13.3 Programming language constructs

We lift the combinators directly from the $(\prime a, \prime s, \prime v)$ *spec* lattice (§8), but need to interference-close primitive actions. Control flow is expressed via HOL's *if-then-else* construct and other case combinators where the scrutinee is a pure value. This means that the atomicity of a process is completely determined by occurrences of *prog.action*.

$\langle ML \rangle$

lift-definition $\text{bind} :: (\prime s, \prime v) \text{prog} \Rightarrow (\prime v \Rightarrow (\prime s, \prime w) \text{prog}) \Rightarrow (\prime s, \prime w) \text{prog}$ **is**
 $\text{spec.bind} \langle \text{proof} \rangle$

adhoc-overloading

$\text{Monad-Syntax.bind prog.bind}$

lift-definition $\text{action} :: (\prime v \times \prime s \times \prime s) \text{set} \Rightarrow (\prime s, \prime v) \text{prog}$ **is**

$\lambda F. \text{spec.interference.cl } (\{env\} \times UNIV) (\text{spec.action } (\text{map-prod id } (\text{Pair self}) ` F)) \langle \text{proof} \rangle$

abbreviation $(\text{input}) \text{det-action} :: (\prime s \Rightarrow (\prime v \times \prime s)) \Rightarrow (\prime s, \prime v) \text{prog}$ **where**

$\text{det-action } f \equiv \text{prog.action } \{(v, s, s'). (v, s') = f s\}$

definition $\text{return} :: \prime v \Rightarrow (\prime s, \prime v) \text{prog}$ **where**

$\text{return } v = \text{prog.action } (\{v\} \times \text{Id})$

definition $\text{guard} :: \prime s \text{pred} \Rightarrow (\prime s, \text{unit}) \text{prog}$ **where**

$\text{guard } g \equiv \text{prog.action } (\{()\} \times \text{Diag } g)$

abbreviation $(\text{input}) \text{read} :: (\prime s \Rightarrow \prime v) \Rightarrow (\prime s, \prime v) \text{prog}$ **where**

$\text{read } F \equiv \text{prog.action } \{(F s, s, s) \mid s. \text{True}\}$

abbreviation *(input)* $write :: ('s \Rightarrow 's) \Rightarrow ('s, unit) prog$ **where**
 $write\ F \equiv prog.action\ \{(\(),\ s,\ F\ s)\ |s.\ True\}$

lift-definition $Parallel :: 'a\ set \Rightarrow ('a \Rightarrow ('s, unit) prog) \Rightarrow ('s, unit) prog$ **is** *spec.Parallel*
 $\langle proof \rangle$

lift-definition $parallel :: ('s, unit) prog \Rightarrow ('s, unit) prog \Rightarrow ('s, unit) prog$ **is** *spec.parallel*
 $\langle proof \rangle$

lift-definition $vmap :: ('v \Rightarrow 'w) \Rightarrow ('s, 'v) prog \Rightarrow ('s, 'w) prog$ **is** *spec.vmap*
 $\langle proof \rangle$

adhoc-overloading
 $Parallel\ prog.Parallel$

adhoc-overloading
 $parallel\ prog.parallel$

lemma *return-alt-def*:
shows $prog.return\ v = prog.read\ \langle v \rangle$
 $\langle proof \rangle$

lemma *parallel-alt-def*:
shows $prog.parallel\ P\ Q = prog.Parallel\ UNIV\ (\lambda a::bool.\ if\ a\ then\ P\ else\ Q)$
 $\langle proof \rangle$

lift-definition $rel :: 's\ rel \Rightarrow ('s, 'v) prog$ **is** $\lambda r.\ spec.rel\ (\{env\} \times UNIV \cup \{self\} \times r)$
 $\langle proof \rangle$

lift-definition $steps :: ('s, 'v) prog \Rightarrow 's\ rel$ **is** $\lambda P.\ spec.steps\ P\ \{\{self\}\}$ $\langle proof \rangle$

lift-definition $invmap :: ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('t, 'w) prog \Rightarrow ('s, 'v) prog$ **is**
 $spec.invmap\ id$
 $\langle proof \rangle$

abbreviation $sinvmap :: ('s \Rightarrow 't) \Rightarrow ('t, 'v) prog \Rightarrow ('s, 'v) prog$ **where**
 $sinvmap\ sf \equiv prog.invmap\ sf\ id$

abbreviation $vinvmap :: ('v \Rightarrow 'w) \Rightarrow ('s, 'w) prog \Rightarrow ('s, 'v) prog$ **where**
 $vinvmap\ vf \equiv prog.invmap\ id\ vf$

declare $prog.bind-def[code\ del]$
declare $prog.action-def[code\ del]$
declare $prog.return-def[code\ del]$
declare $prog.Parallel-def[code\ del]$
declare $prog.parallel-def[code\ del]$
declare $prog.vmap-def[code\ del]$
declare $prog.rel-def[code\ del]$
declare $prog.steps-def[code\ del]$
declare $prog.invmap-def[code\ del]$

13.3.1 Laws of programming

$\langle ML \rangle$

lemma *bind[prog.p2s.simps]*:
shows $prog.p2s\ (f \ggg g) = prog.p2s\ f \ggg (\lambda x.\ prog.p2s\ (g\ x))$
 $\langle proof \rangle$

lemmas $action = prog.action.rep-eq$

lemma *return*:

shows $prog.p2s (prog.return v) = spec.interference.cl (\{env\} \times UNIV) (spec.return v)$
 $\langle proof \rangle$

lemma *guard*:

shows $prog.p2s (prog.guard g) = spec.interference.cl (\{env\} \times UNIV) (spec.guard g)$
 $\langle proof \rangle$

lemmas $Parallel[prog.p2s.simps] = prog.Parallel.rep-eq[simplified, of as Ps \text{ for } as Ps, unfolded comp-def]$

lemmas $parallel[prog.p2s.simps] = prog.parallel.rep-eq$

lemmas $invmap[prog.p2s.simps] = prog.invmap.rep-eq$

lemmas $rel[prog.p2s.simps] = prog.rel.rep-eq$

$\langle ML \rangle$

lemma *transfer[transfer-rule]*:

shows $rel-fun (=) cr-prog (\lambda v. spec.interference.cl (\{env\} \times UNIV) (spec.return v)) prog.return$
 $\langle proof \rangle$

lemma *cong*:

fixes $F :: ('v \times 's \times 's) \text{ set}$

assumes $\bigwedge v s s'. (v, s, s') \in F \implies s' = s$

assumes $\bigwedge v s s' s''. v \in fst ' F \implies (v, s, s) \in F$

shows $prog.action F = (\bigsqcup (v, s, s') \in F. prog.return v)$
 $\langle proof \rangle$

lemma *rel-le*:

shows $prog.return v \leq prog.rel r$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *empty*:

shows $prog.action \{\} = \perp$
 $\langle proof \rangle$

lemma *monotone*:

shows $mono (prog.action :: - \Rightarrow ('s, 'v) prog)$
 $\langle proof \rangle$

lemmas $strengthen[strg] = st-monotone[OF prog.action.monotone]$

lemmas $mono = monotoneD[OF prog.action.monotone]$

lemmas $mono2mono[cont-intro, partial-function-mono] = monotone2monotone[OF prog.action.monotone, simplified]$

lemma *Sup*:

shows $prog.action (\bigsqcup Fs) = (\bigsqcup F \in Fs. prog.action F)$
 $\langle proof \rangle$

lemmas $sup = prog.action.Sup[\text{where } Fs = \{F, G\} \text{ for } F G, simplified]$

lemma *Inf-le*:

shows $prog.action (\bigcap Fs) \leq (\bigcap F \in Fs. prog.action F)$
 $\langle proof \rangle$

lemma *inf-le*:

shows $\text{prog.action } (F \sqcap G) \leq \text{prog.action } F \sqcap \text{prog.action } G$
 ⟨proof⟩

lemma *invmap-le*: — a strict refinement

shows $\text{prog.p2s } (\text{prog.action } (\text{map-prod id } (\text{map-prod sf sf}) - ' F))$
 $\leq \text{spec.invmap sf } (\text{prog.p2s } (\text{prog.action } F))$
 ⟨proof⟩

lemma *return-const*:

fixes $F :: 's \text{ rel}$
fixes $V :: 'v \text{ set}$
fixes $W :: 'w \text{ set}$
assumes $V \neq \{\}$
assumes $W \neq \{\}$
shows $\text{prog.action } (V \times F) = \text{prog.action } (W \times F) \gg (\bigsqcup_{v \in V}. \text{prog.return } v)$
 ⟨proof⟩

lemma *rel-le*:

assumes $\bigwedge v s s'. (v, s, s') \in F \implies (s, s') \in r \vee s = s'$
shows $\text{prog.action } F \leq \text{prog.rel } r$
 ⟨proof⟩

lemma *invmap-le*:

shows $\text{prog.action } (\text{map-prod vf } (\text{map-prod sf sf}) - ' F) \leq \text{prog.invmap sf vf } (\text{prog.action } F)$
 ⟨proof⟩

lemma *action-le*:

shows $\text{spec.action } (\text{map-prod id } (\text{Pair self}) - ' F) \leq \text{prog.p2s } (\text{prog.action } F)$
 ⟨proof⟩

⟨ML⟩

lemmas *if-distrL* = *if-distrib*[**where** $f = \lambda x. x \gg g$ **for** $g :: - \Rightarrow (-, -) \text{ prog}$]

lemma *mono*:

assumes $f \leq f'$
assumes $\bigwedge x. g x \leq g' x$
shows $\text{prog.bind } f g \leq \text{prog.bind } f' g'$
 ⟨proof⟩

lemma *strengthen*[*strg*]:

assumes $\text{st-ord } F f f'$
assumes $\bigwedge x. \text{st-ord } F (g x) (g' x)$
shows $\text{st-ord } F (\text{prog.bind } f g) (\text{prog.bind } f' g')$
 ⟨proof⟩

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:

assumes $\text{monotone orda } (\leq) f$
assumes $\bigwedge x. \text{monotone orda } (\leq) (\lambda y. g y x)$
shows $\text{monotone orda } (\leq) (\lambda x. \text{prog.bind } (f x) (g x))$
 ⟨proof⟩

The monad laws hold unconditionally in the $(', 'v)$ *prog* lattice.

lemma *bind*:

shows $f \gg g \gg h = \text{prog.bind } f (\lambda x. g x \gg h)$
 ⟨proof⟩

lemma *return*:

shows $\text{returnL}: (\gg) (\text{prog.return } v) = (\lambda g :: 'v \Rightarrow ('s, 'w) \text{ prog. } g \ v) \text{ (is ?thesis1)}$
and $\text{returnR}: f \gg \text{prog.return} = f \text{ (is ?thesis2)}$
 $\langle \text{proof} \rangle$

lemma *botL*:
shows $\text{prog.bind } \perp = \perp$
 $\langle \text{proof} \rangle$

lemma *botR-le*:
shows $\text{prog.bind } f \ \langle \perp \rangle \leq f \text{ (is ?thesis1)}$
and $\text{prog.bind } f \ \perp \leq f \text{ (is ?thesis2)}$
 $\langle \text{proof} \rangle$

lemma
fixes $f :: (-, -) \text{ prog}$
fixes $f_1 :: (-, -) \text{ prog}$
shows $\text{supL}: (f_1 \sqcup f_2) \gg g = (f_1 \gg g) \sqcup (f_2 \gg g)$
and $\text{supR}: f \gg (\lambda x. g_1 \ x \sqcup g_2 \ x) = (f \gg g_1) \sqcup (f \gg g_2)$
 $\langle \text{proof} \rangle$

lemma *SUPL*:
fixes $X :: - \text{ set}$
fixes $f :: - \Rightarrow (-, -) \text{ prog}$
shows $(\bigsqcup_{x \in X}. f \ x) \gg g = (\bigsqcup_{x \in X}. f \ x \gg g)$
 $\langle \text{proof} \rangle$

lemma *SUPR*:
fixes $X :: - \text{ set}$
fixes $f :: (-, -) \text{ prog}$
shows $f \gg (\lambda v. \bigsqcup_{x \in X}. g \ x \ v) = (\bigsqcup_{x \in X}. f \ \gg \ g \ x) \sqcup (f \ \gg \ \perp)$
 $\langle \text{proof} \rangle$

lemma *SupR*:
fixes $X :: - \text{ set}$
fixes $f :: (-, -) \text{ prog}$
shows $f \gg (\bigsqcup X) = (\bigsqcup_{x \in X}. f \ \gg \ x) \sqcup (f \ \gg \ \perp)$
 $\langle \text{proof} \rangle$

lemma *SUPR-not-empty*:
fixes $f :: (-, -) \text{ prog}$
assumes $X \neq \{\}$
shows $f \gg (\lambda v. \bigsqcup_{x \in X}. g \ x \ v) = (\bigsqcup_{x \in X}. f \ \gg \ g \ x)$
 $\langle \text{proof} \rangle$

lemma *mcont2mcont[cont-intro]*:
assumes $\text{mcont } \text{luba } \text{orda } \text{Sup } (\leq) \ f$
assumes $\bigwedge v. \text{mcont } \text{luba } \text{orda } \text{Sup } (\leq) \ (\lambda x. g \ x \ v)$
shows $\text{mcont } \text{luba } \text{orda } \text{Sup } (\leq) \ (\lambda x. \text{prog.bind } (f \ x) \ (g \ x))$
 $\langle \text{proof} \rangle$

lemma *inf-rel*:
shows $\text{prog.rel } r \ \sqcap \ (f \ \gg \ g) = \text{prog.rel } r \ \sqcap \ f \ \gg \ (\lambda x. \text{prog.rel } r \ \sqcap \ g \ x)$
and $(f \ \gg \ g) \ \sqcap \ \text{prog.rel } r = \text{prog.rel } r \ \sqcap \ f \ \gg \ (\lambda x. \text{prog.rel } r \ \sqcap \ g \ x)$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *bot*:

shows $\text{prog.guard } \perp = \perp$
and $\text{prog.guard } \langle \text{False} \rangle = \perp$
 $\langle \text{proof} \rangle$

lemma top:

shows $\text{prog.guard } (\top :: 'a \text{ pred}) = \text{prog.return } ()$ (**is** *?thesis1*)
and $\text{prog.guard } (\langle \text{True} \rangle :: 'a \text{ pred}) = \text{prog.return } ()$ (**is** *?thesis2*)
 $\langle \text{proof} \rangle$

lemma return-le:

shows $\text{prog.guard } g \leq \text{prog.return } ()$
 $\langle \text{proof} \rangle$

lemma monotone:

shows $\text{mono } (\text{prog.guard} :: 's \text{ pred} \Rightarrow -)$
 $\langle \text{proof} \rangle$

lemmas $\text{strengthen}[strg] = \text{st-monotone}[OF \text{ prog.guard.monotone}]$

lemmas $\text{mono} = \text{monotoneD}[OF \text{ prog.guard.monotone}]$

lemmas $\text{mono2mono}[\text{cont-intro}, \text{partial-function-mono}] = \text{monotone2monotone}[OF \text{ prog.guard.monotone}, \text{simplified}]$

lemma less: — Non-triviality

assumes $g < g'$
shows $\text{prog.guard } g < \text{prog.guard } g'$
 $\langle \text{proof} \rangle$

lemma if:

shows $(\text{if } b \text{ then } t \text{ else } e) = (\text{prog.guard } \langle b \rangle \gg t) \sqcup (\text{prog.guard } \langle \neg b \rangle \gg e)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma bot:

assumes $\bigwedge a. a \in bs \implies Ps a = \perp$
assumes $as \cap bs \neq \{\}$
shows $\text{prog.Parallel } as Ps = \text{prog.Parallel } (as - bs) Ps \gg \perp$
 $\langle \text{proof} \rangle$

lemma mono:

assumes $\bigwedge a. a \in as \implies Ps a \leq Ps' a$
shows $\text{prog.Parallel } as Ps \leq \text{prog.Parallel } as Ps'$
 $\langle \text{proof} \rangle$

lemma strengthen-Parallel[*strg*]:

assumes $\bigwedge a. a \in as \implies \text{st-ord } F (Ps a) (Ps' a)$
shows $\text{st-ord } F (\text{prog.Parallel } as Ps) (\text{prog.Parallel } as Ps')$
 $\langle \text{proof} \rangle$

lemma mono2mono[*cont-intro*, *partial-function-mono*]:

assumes $\bigwedge a. a \in as \implies \text{monotone } \text{orda } (\leq) (F a)$
shows $\text{monotone } \text{orda } (\leq) (\lambda f. \text{prog.Parallel } as (\lambda a. F a f))$
 $\langle \text{proof} \rangle$

lemma cong:

assumes $as = as'$
assumes $\bigwedge a. a \in as' \implies Ps a = Ps' a$
shows $\text{prog.Parallel } as Ps = \text{prog.Parallel } as' Ps'$

$\langle \text{proof} \rangle$

lemma *no-agents*:

shows $\text{prog.Parallel } \{ \} Ps = \text{prog.return } ()$

$\langle \text{proof} \rangle$

lemma *singleton-agents*:

shows $\text{prog.Parallel } \{ a \} Ps = Ps a$

$\langle \text{proof} \rangle$

lemma *rename-UNIV*:

assumes *inj-on f as*

shows $\text{prog.Parallel as } Ps$

$= \text{prog.Parallel UNIV } (\lambda b. \text{if } b \in f \text{ ' as then } Ps \text{ (inv-into as f b) else prog.return } ())$

$\langle \text{proof} \rangle$

lemmas *rename = spec.Parallel.rename[transferred]*

lemma *return*:

assumes $\bigwedge a. a \in bs \implies Ps a = \text{prog.return } ()$

shows $\text{prog.Parallel as } Ps = \text{prog.Parallel } (as - bs) Ps$

$\langle \text{proof} \rangle$

lemma *unwind*:

assumes $a: f \ggg g \leq Ps a$ — The selected process starts with action *f*

assumes $a \in as$

shows $f \ggg (\lambda v. \text{prog.Parallel as } (Ps(a:=g v))) \leq \text{prog.Parallel as } Ps$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemmas *commute = spec.parallel.commute[transferred]*

lemmas *assoc = spec.parallel.assoc[transferred]*

lemmas *mono = spec.parallel.mono[transferred]*

lemma *strengthen[strg]*:

assumes *st-ord F P P'*

assumes *st-ord F Q Q'*

shows *st-ord F (prog.parallel P Q) (prog.parallel P' Q')*

$\langle \text{proof} \rangle$

lemma *mono2mono[cont-intro, partial-function-mono]*:

assumes *monotone orda* (\leq) *F*

assumes *monotone orda* (\leq) *G*

shows *monotone orda* (\leq) $(\lambda f. \text{prog.parallel } (F f) (G f))$

$\langle \text{proof} \rangle$

lemma *bot*:

shows *botL*: $\text{prog.parallel } \perp P = P \ggg \perp$ (**is** *?thesis1*)

and *botR*: $\text{prog.parallel } P \perp = P \ggg \perp$ (**is** *?thesis2*)

$\langle \text{proof} \rangle$

lemma *return*:

shows *returnL*: $\text{prog.return } () \parallel P = P$ (**is** *?thesis1*)

and *returnR*: $P \parallel \text{prog.return } () = P$ (**is** *?thesis2*)

$\langle \text{proof} \rangle$

lemma *Sup-not-empty*:

fixes $X :: (-, \text{unit}) \text{ prog set}$
assumes $X \neq \{\}$
shows *SupL-not-empty*: $\sqcup X \parallel Q = (\sqcup P \in X. P \parallel Q)$ (**is** *?thesis1* Q)
and *SupR-not-empty*: $P \parallel \sqcup X = (\sqcup Q \in X. P \parallel Q)$ (**is** *?thesis2*)
 $\langle \text{proof} \rangle$

lemma *sup*:
fixes $P :: (-, \text{unit}) \text{ prog}$
shows *supL*: $P \sqcup Q \parallel R = (P \parallel R) \sqcup (Q \parallel R)$
and *supR*: $P \parallel Q \sqcup R = (P \parallel Q) \sqcup (P \parallel R)$
 $\langle \text{proof} \rangle$

lemma *mcont2mcont[cont-intro]*:
assumes *mcont luba orda Sup* $(\leq) P$
assumes *mcont luba orda Sup* $(\leq) Q$
shows *mcont luba orda Sup* $(\leq) (\lambda x. \text{prog.parallel } (P \ x) \ (Q \ x))$
 $\langle \text{proof} \rangle$

lemma *unwindL*:
fixes $f :: ('s, 'v) \text{ prog}$
assumes $a: f \ggg g \leq P$ — The selected process starts with action f
shows $f \ggg (\lambda v. g \ v \parallel Q) \leq P \parallel Q$
 $\langle \text{proof} \rangle$

lemma *unwindR*:
fixes $f :: ('s, 'v) \text{ prog}$
assumes $a: f \ggg g \leq Q$ — The selected process starts with action f
shows $f \ggg (\lambda v. P \parallel g \ v) \leq P \parallel Q$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *parallel-le*:
fixes $P :: (-, -) \text{ prog}$
shows $P \ggg Q \leq P \parallel Q$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *bot*:
shows $\text{prog.invmap } sf \ vf \ \perp = (\text{prog.rel } (\text{map-prod } sf \ sf \ -' \text{Id}) :: (-, \text{unit}) \text{ prog}) \ggg \perp$
 $\langle \text{proof} \rangle$

lemma *id*:
shows $\text{prog.invmap } id \ id \ P = P$
and $\text{prog.invmap } (\lambda x. x) \ (\lambda x. x) \ P = P$
 $\langle \text{proof} \rangle$

lemma *comp*:
shows $\text{prog.invmap } sf \ vf \ (\text{prog.invmap } sg \ vg \ P) = \text{prog.invmap } (\lambda s. sg \ (sf \ s)) \ (\lambda s. vg \ (vf \ s)) \ P$ (**is** *?thesis1* P)
and $\text{prog.invmap } sf \ vf \ \circ \ \text{prog.invmap } sg \ vg = \text{prog.invmap } (sg \ \circ \ sf) \ (vg \ \circ \ vf)$ (**is** *?thesis2*)
 $\langle \text{proof} \rangle$

lemma *monotone*:
shows *mono* $(\text{prog.invmap } sf \ vf)$
 $\langle \text{proof} \rangle$

lemmas *strengthen[strg]* = *st-monotone[OF prog.invmap.monotone]*

lemmas *mono* = *monotoneD*[*OF prog.invmap.monotone*]

lemma *mono2mono*[*cont-intro, partial-function-mono*]:
 assumes *monotone orda* (\leq) *t*
 shows *monotone orda* (\leq) ($\lambda x. \text{prog.invmap } sf \text{ } vf \text{ } (t \ x)$)
<proof>

lemma *Sup*:
 fixes *sf* :: 's \Rightarrow 't
 fixes *vf* :: 'v \Rightarrow 'w
 shows *prog.invmap sf vf* ($\sqcup X$) = \sqcup (*prog.invmap sf vf* ' *X*) \sqcup *prog.invmap sf vf* \perp
<proof>

lemma *Sup-not-empty*:
 assumes *X* \neq {}
 shows *prog.invmap sf vf* ($\sqcup X$) = \sqcup (*prog.invmap sf vf* ' *X*)
<proof>

lemma *mcont*:
 shows *mcont Sup* (\leq) *Sup* (\leq) (*prog.invmap sf vf*)
<proof>

lemmas *mcont2mcont*[*cont-intro*] = *mcont2mcont*[*OF prog.invmap.mcont, of luba orda P for luba orda P*]

lemma *bind*:
 shows *prog.invmap sf vf* (*f* \gg *g*) = *prog.sinvmap sf f* \gg ($\lambda v. \text{prog.invmap } sf \text{ } vf \text{ } (g \ v)$)
<proof>

lemma *parallel*:
 shows *prog.invmap sf vf* (*P* \parallel *Q*) = *prog.invmap sf vf P* \parallel *prog.invmap sf vf Q*
<proof>

lemma *invmap-image-vimage-commute*:
 shows *map-prod id* (*map-prod id sf*) - ' *map-prod id* (*Pair self*) ' *F*
 = *map-prod id* (*Pair self*) ' *map-prod id sf* - ' *F*
<proof>

lemma *action*:
 shows *prog.invmap sf vf* (*prog.action F*)
 = *prog.rel* (*map-prod sf sf* - ' *Id*)
 \gg ($\lambda :: \text{unit. prog.action } (\text{map-prod id } (\text{map-prod sf sf}) - ' F)$)
 \gg ($\lambda v. \text{prog.rel } (\text{map-prod sf sf} - ' \text{Id})$)
 \gg ($\lambda :: \text{unit. } \sqcup v' \in vf - ' \{v\}. \text{prog.return } v'$)
<proof>

<ML>

lemma *bot*:
 shows *prog.vmap vf* \perp = \perp
<proof>

lemma *unitL*:
 shows *f* \gg *g* = *prog.vmap* <()> *f* \gg *g*
<proof>

lemma *eq-return*:
 shows *prog.vmap vf P* = *P* \gg *prog.return* \circ *vf* (**is** ?thesis1)
 and *prog.vmap vf P* = *P* \gg ($\lambda v. \text{prog.return } (vf \ v)$) (**is** ?thesis2)

$\langle proof \rangle$

lemma *action*:

shows $prog.vmap\ vf\ (prog.action\ F) = prog.action\ (map\ prod\ vf\ id\ 'F)$

$\langle proof \rangle$

lemma *return*:

shows $prog.vmap\ vf\ (prog.return\ v) = prog.return\ (vf\ v)$

$\langle proof \rangle$

$\langle ML \rangle$

interpretation *kleene*: $kleene\ prog.return\ ()\ \lambda x\ y.\ prog.bind\ x\ \langle y \rangle$

$\langle proof \rangle$

interpretation *rel*: $galois.complete.lattice.class\ prog.steps\ prog.rel$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *empty*:

shows $prog.rel\ \{\} = \bigsqcup\ range\ prog.return$

$\langle proof \rangle$

lemmas *monotone* = $prog.rel.monotone-upper$

lemmas *strengthen*[*strg*] = $st-monotone[OF\ prog.rel.monotone]$

lemmas *mono* = $monotoneD[OF\ prog.rel.monotone]$

lemmas *Inf* = $prog.rel.upper-Inf$

lemmas *inf* = $prog.rel.upper-inf$

lemma *reflcl*:

shows $prog.rel\ (r \cup Id) = (prog.rel\ r :: ('s, 'v)\ prog)\ (\mathbf{is}\ ?thesis1)$

and $prog.rel\ (Id \cup r) = (prog.rel\ r :: ('s, 'v)\ prog)\ (\mathbf{is}\ ?thesis2)$

$\langle proof \rangle$

lemma *minus-Id*:

shows $prog.rel\ (r - Id) = prog.rel\ r$

$\langle proof \rangle$

lemma *Id*:

shows $prog.rel\ Id = \bigsqcup\ range\ prog.return$

$\langle proof \rangle$

lemma *unfoldL*:

fixes $r :: 's\ rel$

assumes $Id \subseteq r$

shows $prog.rel\ r = prog.action\ (\{\()\} \times r) \gg prog.rel\ r$

$\langle proof \rangle$

lemma *wind-bind*: — arbitrary interstitial return type

shows $prog.rel\ r \gg prog.rel\ r = prog.rel\ r$

$\langle proof \rangle$

lemma *wind-bind-leading*: — arbitrary interstitial return type

assumes $r' \subseteq r$

shows $prog.rel\ r' \gg prog.rel\ r = prog.rel\ r$

$\langle proof \rangle$

lemma *wind-bind-trailing*: — arbitrary interstitial return type

assumes $r' \subseteq r$

shows $\text{prog.rel } r \gg \text{prog.rel } r' = \text{prog.rel } r$ (**is** $?lhs = ?rhs$)

<proof>

Interstitial unit, for unfolding

lemmas *unwind-bind* = $\text{prog.rel.wind-bind}$ [**where** $'c=\text{unit}$, *symmetric*]

lemmas *unwind-bind-leading* = $\text{prog.rel.wind-bind-leading}$ [**where** $'c=\text{unit}$, *symmetric*]

lemmas *unwind-bind-trailing* = $\text{prog.rel.wind-bind-trailing}$ [**where** $'c=\text{unit}$, *symmetric*]

lemma *mono-conv*:

shows $\text{prog.rel } r = \text{prog.kleene.star } (\text{prog.action } (\{\{\}\} \times r^=))$ (**is** $?lhs = ?rhs$)

<proof>

<ML>

lemma *inf-rel*:

assumes *refl* r

shows $\text{prog.action } F \sqcap \text{prog.rel } r = \text{prog.action } (F \cap \text{UNIV} \times r)$ (**is** $?thesis1$)

and $\text{prog.rel } r \sqcap \text{prog.action } F = \text{prog.action } (F \cap \text{UNIV} \times r)$ (**is** $?thesis2$)

<proof>

lemma *inf-rel-reflcl*:

shows $\text{prog.action } F \sqcap \text{prog.rel } r = \text{prog.action } (F \cap \text{UNIV} \times r^=)$

and $\text{prog.rel } r \sqcap \text{prog.action } F = \text{prog.action } (F \cap \text{UNIV} \times r^=)$

<proof>

<ML>

lemma *not-bot*:

shows $\text{prog.return } v \neq (\perp :: ('s, 'v) \text{ prog})$

<proof>

<ML>

lemma *return*:

shows $\text{prog.invmap } sf \ vf \ (\text{prog.return } v)$

$= \text{prog.rel } (\text{map-prod } sf \ sf \ -' \ \text{Id}) \gg (\lambda :: \text{unit}. \sqcup v' \in vf \ -' \ \{v\}. \text{prog.return } v')$

<proof>

lemma *split-invmap*:

fixes $P :: ('s, 'v) \text{ prog}$

shows $\text{prog.invmap } sf \ vf \ P = \text{prog.sinvmap } sf \ P \gg (\lambda v. \sqcup v' \in vf \ -' \ \{v\}. \text{prog.return } v')$

<proof>

<ML>

13.4 Refinement for $('s, 'v) \text{ prog}$

We specialize the rules of §12.1 to the $('s, 'v) \text{ prog}$ lattice. Observe that, as preconditions, postconditions and assumes are not interference closed, we apply the *prog.p2s* morphism and work in the more capacious (*sequential*, $'s, 'v$) *spec* lattice. This syntactic noise could be elided with another definition.

13.4.1 Introduction rules

Refinement is a way of showing inequalities and equalities between programs.

<ML>

lemma *leI*:

assumes $\text{prog.p2s } c \leq \{\langle \text{True} \rangle\}, \top \Vdash \text{prog.p2s } d, \{\lambda\cdot. \langle \text{True} \rangle\}$
shows $c \leq d$
<proof>

lemma *eqI*:

assumes $\text{prog.p2s } c \leq \{\langle \text{True} \rangle\}, \top \Vdash \text{prog.p2s } d, \{\lambda\cdot. \langle \text{True} \rangle\}$
assumes $\text{prog.p2s } d \leq \{\langle \text{True} \rangle\}, \top \Vdash \text{prog.p2s } c, \{\lambda\cdot. \langle \text{True} \rangle\}$
shows $c = d$
<proof>

<ML>

13.4.2 Galois considerations

Refinement quadruples $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$ denote points in the (*'s, 'v*) *prog* lattice provided G is suitably interference closed.

<ML>

lemma *galois*:

assumes $\text{spec.term.none } (\text{spec.rel } (\{\text{env}\} \times \text{UNIV}) :: (-, -, \text{unit}) \text{spec}) \leq G$
shows $\text{prog.p2s } c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} \iff c \leq \text{prog.s2p } (\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\})$
<proof>

lemmas *s2p-refinement = iffD1[OF refinement.prog.galois, rotated]*

lemma *p2s-s2p*:

assumes $\text{spec.term.none } (\text{spec.rel } (\{\text{env}\} \times \text{UNIV}) :: (-, -, \text{unit}) \text{spec}) \leq G$
shows $\text{prog.p2s } (\text{prog.s2p } (\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\})) \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
<proof>

<ML>

13.4.3 Rules

<ML>

lemma *bot[iff]*:

shows $\text{prog.p2s } \perp \leq \{\!|P|\!\}, A \Vdash \text{prog.p2s } c', \{\!|Q|\!\}$
<proof>

lemma *sup-conv*:

shows $\text{prog.p2s } (c_1 \sqcup c_2) \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
 $\iff \text{prog.p2s } c_1 \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} \wedge \text{prog.p2s } c_2 \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
<proof>

lemmas *sup = iffD2[OF refinement.prog.sup-conv, unfolded conj-explode]*

lemma *if*:

assumes $i \implies \text{prog.p2s } t \leq \{\!|P|\!\}, A \Vdash \text{prog.p2s } t', \{\!|Q|\!\}$
assumes $\neg i \implies \text{prog.p2s } e \leq \{\!|P'|\!\}, A \Vdash \text{prog.p2s } e', \{\!|Q|\!\}$
shows $\text{prog.p2s } (\text{if } i \text{ then } t \text{ else } e) \leq \{\!|\text{if } i \text{ then } P \text{ else } P'|\!\}, A \Vdash \text{prog.p2s } (\text{if } i \text{ then } t' \text{ else } e'), \{\!|Q|\!\}$
<proof>

lemmas *if' = refinement.prog.if[where P=P and P'=P, simplified] for P*

lemma *case-option*:

assumes $opt = None \implies prog.p2s\ none \leq \{P_n\}, A \Vdash prog.p2s\ none', \{Q\}$
assumes $\bigwedge v. opt = Some\ v \implies prog.p2s\ (some\ v) \leq \{P_s\ v\}, A \Vdash prog.p2s\ (some'\ v), \{Q\}$
shows $prog.p2s\ (case-option\ none\ some\ opt) \leq \{case\ opt\ of\ None \Rightarrow P_n \mid Some\ v \Rightarrow P_s\ v\}, A \Vdash prog.p2s\ (case-option\ none'\ some'\ opt), \{Q\}$
 <proof>

lemma case-sum:

assumes $\bigwedge v. x = Inl\ v \implies prog.p2s\ (left\ v) \leq \{P_l\ v\}, A \Vdash prog.p2s\ (left'\ v), \{Q\}$
assumes $\bigwedge v. x = Inr\ v \implies prog.p2s\ (right\ v) \leq \{P_r\ v\}, A \Vdash prog.p2s\ (right'\ v), \{Q\}$
shows $prog.p2s\ (case-sum\ left\ right\ x) \leq \{case-sum\ P_l\ P_r\ x\}, A \Vdash prog.p2s\ (case-sum\ left'\ right'\ x), \{Q\}$
 <proof>

lemma case-list:

assumes $x = [] \implies prog.p2s\ nil \leq \{P_n\}, A \Vdash prog.p2s\ nil', \{Q\}$
assumes $\bigwedge v\ vs. x = v \# vs \implies prog.p2s\ (cons\ v\ vs) \leq \{P_c\ v\ vs\}, A \Vdash prog.p2s\ (cons'\ v\ vs), \{Q\}$
shows $prog.p2s\ (case-list\ nil\ cons\ x) \leq \{case-list\ P_n\ P_c\ x\}, A \Vdash prog.p2s\ (case-list\ nil'\ cons'\ x), \{Q\}$
 <proof>

lemma action:

fixes $F :: ('v \times 's \times 's)\ set$
assumes $\bigwedge v\ s\ s'. \llbracket P\ s; (v, s, s') \in F; (self, s, s') \in spec.steps\ A \vee s = s' \rrbracket \implies Q\ v\ s'$
assumes $\bigwedge v\ s\ s'. \llbracket P\ s; (v, s, s') \in F \rrbracket \implies (v, s, s') \in F'$
assumes $sP: stable\ (spec.steps\ A\ \{\{env\}\})\ P$
assumes $\bigwedge v\ s\ s'. \llbracket P\ s; (v, s, s') \in F \rrbracket \implies stable\ (spec.steps\ A\ \{\{env\}\})\ (Q\ v)$
shows $prog.p2s\ (prog.action\ F) \leq \{P\}, A \Vdash prog.p2s\ (prog.action\ F'), \{Q\}$
 <proof>

lemma return:

assumes $sQ: stable\ (spec.steps\ A\ \{\{env\}\})\ (Q\ v)$
shows $prog.p2s\ (prog.return\ v) \leq \{Q\ v\}, A \Vdash prog.p2s\ (prog.return\ v), \{Q\}$
 <proof>

lemma invmap-return:

assumes $sQ: stable\ (spec.steps\ A\ \{\{env\}\})\ (Q\ v)$
assumes $vf\ v = v'$
shows $prog.p2s\ (prog.return\ v) \leq \{Q\ v\}, A \Vdash prog.p2s\ (prog.invmap\ sf\ vf\ (prog.return\ v')), \{Q\}$
 <proof>

lemma bind-abstract:

fixes $f :: ('s, 'v)\ prog$
fixes $f' :: ('s, 'v')\ prog$
fixes $g :: 'v \Rightarrow ('s, 'w)\ prog$
fixes $g' :: 'v' \Rightarrow ('s, 'w)\ prog$
fixes $vf :: 'v \Rightarrow 'v'$
assumes $\bigwedge v. prog.p2s\ (g\ v) \leq \{Q'\ (vf\ v)\}, refinement.spec.bind.res\ (spec.pre\ P \sqcap spec.term.all\ A \sqcap prog.p2s\ f')\ A\ (vf\ v) \Vdash prog.p2s\ (g'\ (vf\ v)), \{Q\}$
assumes $prog.p2s\ f \leq \{P\}, spec.term.all\ A \Vdash spec.vinvmmap\ vf\ (prog.p2s\ f'), \{\lambda v. Q'\ (vf\ v)\}$
shows $prog.p2s\ (f \ggg g) \leq \{P\}, A \Vdash prog.p2s\ (f' \ggg g'), \{Q\}$
 <proof>

lemma bind:

assumes $\bigwedge v. prog.p2s\ (g\ v) \leq \{Q'\ v\}, refinement.spec.bind.res\ (spec.pre\ P \sqcap spec.term.all\ A \sqcap prog.p2s\ f')\ A\ v \Vdash prog.p2s\ (g'\ v), \{Q\}$
assumes $prog.p2s\ f \leq \{P\}, spec.term.all\ A \Vdash prog.p2s\ f', \{Q'\}$
shows $prog.p2s\ (f \ggg g) \leq \{P\}, A \Vdash prog.p2s\ (f' \ggg g'), \{Q\}$
 <proof>

lemmas rev-bind = refinement.prog.bind[rotated]

lemma *Parallel*:

fixes $A :: (\text{sequential}, 's, \text{unit}) \text{ spec}$

fixes $Q :: 'a \Rightarrow 's \text{ pred}$

fixes $P_s :: 'a \Rightarrow ('s, \text{unit}) \text{ prog}$

fixes $P_s' :: 'a \Rightarrow ('s, \text{unit}) \text{ prog}$

assumes $\bigwedge a. a \in as \implies \text{prog.p2s } (P_s a) \leq \{\!\{P a\}\!\}$, *refinement.spec.env-hyp* $P A as (\text{prog.p2s} \circ P_s') a \Vdash \text{prog.p2s } (P_s' a)$, $\{\!\{\lambda rv. Q a\}\!\}$

shows $\text{prog.p2s } (\text{prog.Parallel } as P_s) \leq \{\!\{\prod a \in as. P a\}\!\}$, $A \Vdash \text{prog.p2s } (\text{prog.Parallel } as P_s')$, $\{\!\{\lambda rv. \prod a \in as. Q a\}\!\}$

$\langle \text{proof} \rangle$

lemma *parallel*:

assumes $\text{prog.p2s } c_1 \leq \{\!\{P_1\}\!\}$, *refinement.spec.env-hyp* $(\lambda a. \text{if } a \text{ then } P_1 \text{ else } P_2) A \text{ UNIV } (\lambda a. \text{if } a \text{ then } \text{prog.p2s } c_1' \text{ else } \text{prog.p2s } c_2')$ $\text{True} \Vdash \text{prog.p2s } c_1'$, $\{\!\{Q_1\}\!\}$

assumes $\text{prog.p2s } c_2 \leq \{\!\{P_2\}\!\}$, *refinement.spec.env-hyp* $(\lambda a. \text{if } a \text{ then } P_1 \text{ else } P_2) A \text{ UNIV } (\lambda a. \text{if } a \text{ then } \text{prog.p2s } c_1' \text{ else } \text{prog.p2s } c_2')$ $\text{False} \Vdash \text{prog.p2s } c_2'$, $\{\!\{Q_2\}\!\}$

shows $\text{prog.p2s } (\text{prog.parallel } c_1 c_2) \leq \{\!\{P_1 \wedge P_2\}\!\}$, $A \Vdash \text{prog.p2s } (\text{prog.parallel } c_1' c_2')$, $\{\!\{\lambda v. Q_1 v \wedge Q_2 v\}\!\}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

13.5 A relational assume/guarantee program logic for the $('s, 'v)$ prog lattice

Similarly we specialize the assume/guarantee program logic of §12.2 to $('s, 'v)$ prog.

References:

- de Roever, de Boer, Hannemann, Hooman, Lakhnech, Poel, and Zwiers (2001); Xu, de Roever, and He (1997)
- Prensa Nieto (2003, §7)
- Vafeiadis (2008, §3)

13.5.1 Galois considerations

For suitably stable $P, Q, \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$ is interference closed and hence denotes a point in $('s, 'v)$ prog. In other words we can replace programs with their specifications.

$\langle ML \rangle$

lemma *galois*:

shows $\text{prog.p2s } c \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\} \iff c \leq \text{prog.s2p } (\{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\})$

$\langle \text{proof} \rangle$

lemmas $s2p\text{-ag} = \text{iffD1}[OF \text{ ag.prog.galois}]$

lemma *p2s-s2p-ag*:

shows $\text{prog.p2s } (\text{prog.s2p } (\{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\})) \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$

$\langle \text{proof} \rangle$

lemma *p2s-s2p-ag-stable*:

assumes *stable* $A P$

assumes $\bigwedge v. \text{stable } A (Q v)$

shows $\text{prog.p2s } (\text{prog.s2p } (\{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\})) = \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *bot[iff]*:

shows $\text{prog.p2s } \perp \leq \{\!\{P}\!\}, A \vdash G, \{\!\{Q}\!\}$
<proof>

<ML>

lemma *sup-conv*:

shows $\text{prog.p2s } (c_1 \sqcup c_2) \leq \{\!\{P}\!\}, A \vdash G, \{\!\{Q}\!\} \longleftrightarrow \text{prog.p2s } c_1 \leq \{\!\{P}\!\}, A \vdash G, \{\!\{Q}\!\} \wedge \text{prog.p2s } c_2 \leq \{\!\{P}\!\}, A \vdash G, \{\!\{Q}\!\}$
<proof>

lemmas *sup = iffD2[OF ag.prog.sup-conv, unfolded conj-explode]*

lemma *bind*: — Assumptions in weakest-pre order

assumes $\bigwedge v. \text{prog.p2s } (g \ v) \leq \{\!\{Q' \ v\}\!\}, A \vdash G, \{\!\{Q}\!\}$
assumes $\text{prog.p2s } f \leq \{\!\{P}\!\}, A \vdash G, \{\!\{Q'}\!\}$
shows $\text{prog.p2s } (f \ggg g) \leq \{\!\{P}\!\}, A \vdash G, \{\!\{Q}\!\}$
<proof>

lemma *action*: — Conclusion is insufficiently instantiated for use

fixes $F :: ('v \times 's \times 's) \text{ set}$
assumes $Q: \bigwedge v \ s \ s'. \llbracket P \ s; (v, s, s') \in F \rrbracket \Longrightarrow Q \ v \ s'$
assumes $G: \bigwedge v \ s \ s'. \llbracket P \ s; s \neq s'; (v, s, s') \in F \rrbracket \Longrightarrow (s, s') \in G$
assumes $sP: \text{stable } A \ P$
assumes $sQ: \bigwedge s \ s' \ v. \llbracket P \ s; (v, s, s') \in F \rrbracket \Longrightarrow \text{stable } A \ (Q \ v)$
shows $\text{prog.p2s } (\text{prog.action } F) \leq \{\!\{P}\!\}, A \vdash G, \{\!\{Q}\!\}$
<proof>

lemma *guard*:

assumes $\bigwedge s. \llbracket P \ s; g \ s \rrbracket \Longrightarrow Q \ () \ s$
assumes $\text{stable } A \ P$
assumes $\text{stable } A \ (Q \ ())$
shows $\text{prog.p2s } (\text{prog.guard } g) \leq \{\!\{P}\!\}, A \vdash G, \{\!\{Q}\!\}$
<proof>

lemma *Parallel*:

assumes $\bigwedge a. a \in as \Longrightarrow \text{prog.p2s } (Ps \ a) \leq \{\!\{P \ a\}\!\}, A \cup (\bigcup a' \in as - \{a\}. G \ a') \vdash G \ a, \{\!\{\lambda v. Q \ a\}\!\}$
shows $\text{prog.p2s } (\text{prog.Parallel } as \ Ps) \leq \{\!\{\prod a \in as. P \ a\}\!\}, A \vdash \bigcup a \in as. G \ a, \{\!\{\lambda v. \prod a \in as. Q \ a\}\!\}$
<proof>

lemma *parallel*:

assumes $\text{prog.p2s } c_1 \leq \{\!\{P_1}\!\}, A \cup G_2 \vdash G_1, \{\!\{Q_1}\!\}$
assumes $\text{prog.p2s } c_2 \leq \{\!\{P_2}\!\}, A \cup G_1 \vdash G_2, \{\!\{Q_2}\!\}$
shows $\text{prog.p2s } (\text{prog.parallel } c_1 \ c_2) \leq \{\!\{P_1 \wedge P_2}\!\}, A \vdash G_1 \cup G_2, \{\!\{\lambda v. Q_1 \ v \wedge Q_2 \ v\}\!\}$
<proof>

lemma *return*:

assumes $sQ: \text{stable } A \ (Q \ v)$
shows $\text{prog.p2s } (\text{prog.return } v) \leq \{\!\{Q \ v\}\!\}, A \vdash G, \{\!\{Q}\!\}$
<proof>

lemma *if*:

assumes $b \Longrightarrow \text{prog.p2s } c_1 \leq \{\!\{P_1}\!\}, A \vdash G, \{\!\{Q}\!\}$
assumes $\neg b \Longrightarrow \text{prog.p2s } c_2 \leq \{\!\{P_2}\!\}, A \vdash G, \{\!\{Q}\!\}$
shows $\text{prog.p2s } (\text{if } b \text{ then } c_1 \text{ else } c_2) \leq \{\!\{\text{if } b \text{ then } P_1 \text{ else } P_2\}\!\}, A \vdash G, \{\!\{Q}\!\}$
<proof>

lemma *case-option*:

assumes $x = \text{None} \implies \text{prog.p2s } \text{none} \leq \{\{P_n\}, A \vdash G, \{Q\}\}$
assumes $\bigwedge v. x = \text{Some } v \implies \text{prog.p2s } (\text{some } v) \leq \{\{P_s v\}, A \vdash G, \{Q\}\}$
shows $\text{prog.p2s } (\text{case-option none some } x) \leq \{\{\text{case } x \text{ of None} \Rightarrow P_n \mid \text{Some } v \Rightarrow P_s v\}, A \vdash G, \{Q\}\}$
 $\langle \text{proof} \rangle$

lemma case-sum:

assumes $\bigwedge v. x = \text{Inl } v \implies \text{prog.p2s } (\text{left } v) \leq \{\{P_l v\}, A \vdash G, \{Q\}\}$
assumes $\bigwedge v. x = \text{Inr } v \implies \text{prog.p2s } (\text{right } v) \leq \{\{P_r v\}, A \vdash G, \{Q\}\}$
shows $\text{prog.p2s } (\text{case-sum left right } x) \leq \{\{\text{case-sum } P_l P_r x\}, A \vdash G, \{Q\}\}$
 $\langle \text{proof} \rangle$

lemma case-list:

assumes $x = [] \implies \text{prog.p2s } \text{nil} \leq \{\{P_n\}, A \vdash G, \{Q\}\}$
assumes $\bigwedge v \text{ vs. } x = v \# \text{ vs} \implies \text{prog.p2s } (\text{cons } v \text{ vs}) \leq \{\{P_c v \text{ vs}\}, A \vdash G, \{Q\}\}$
shows $\text{prog.p2s } (\text{case-list nil cons } x) \leq \{\{\text{case-list } P_n P_c x\}, A \vdash G, \{Q\}\}$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

13.5.2 A proof of the parallel rule using Abadi and Plotkin's composition principle

Here we show that the key rule for *Parallel* (*ag.spec.Parallel*) can be established using the *spec.ag-circular* rule (§9.2).

The following proof is complicated by the need to discard a lot of contextual information.

notepad

begin

$\langle \text{proof} \rangle$

end

13.6 Specification inhabitation

$\langle ML \rangle$

lemma Sup:

assumes $\text{prog.p2s } P -s, xs \rightarrow P'$
assumes $P \in X$
shows $\text{prog.p2s } (\bigsqcup X) -s, xs \rightarrow P'$
 $\langle \text{proof} \rangle$

lemma supL:

assumes $\text{prog.p2s } P -s, xs \rightarrow P'$
shows $\text{prog.p2s } (P \sqcup Q) -s, xs \rightarrow P'$
 $\langle \text{proof} \rangle$

lemma supR:

assumes $\text{prog.p2s } Q -s, xs \rightarrow Q'$
shows $\text{prog.p2s } (P \sqcup Q) -s, xs \rightarrow Q'$
 $\langle \text{proof} \rangle$

lemma bind:

assumes $\text{prog.p2s } f -s, xs \rightarrow \text{prog.p2s } f'$
shows $\text{prog.p2s } (f \ggg g) -s, xs \rightarrow \text{prog.p2s } (f' \ggg g)$
 $\langle \text{proof} \rangle$

lemma return:

shows $\text{prog.p2s } (\text{prog.return } v) -s, [] \rightarrow \text{spec.return } v$

$\langle proof \rangle$

lemma *action-step*:

fixes $F :: ('v \times 's \times 's) \text{ set}$

assumes $(v, s, s') \in F$

shows $prog.p2s (prog.action F) -s, [(self, s')] \rightarrow prog.p2s (prog.return v)$

$\langle proof \rangle$

lemma *action-stutter*:

fixes $F :: ('v \times 's \times 's) \text{ set}$

assumes $(v, s, s) \in F$

shows $prog.p2s (prog.action F) -s, [] \rightarrow prog.p2s (prog.return v)$

$\langle proof \rangle$

lemma *parallelL*:

assumes $prog.p2s P -s, xs \rightarrow prog.p2s P'$

shows $prog.p2s (P \parallel Q) -s, xs \rightarrow prog.p2s (P' \parallel Q)$

$\langle proof \rangle$

lemma *parallelR*:

assumes $prog.p2s Q -s, xs \rightarrow prog.p2s Q'$

shows $prog.p2s (P \parallel Q) -s, xs \rightarrow prog.p2s (P \parallel Q')$

$\langle proof \rangle$

$\langle ML \rangle$

14 More combinators

Extra combinators:

- *prog.select* shows how we can handle arbitrary choice
- *prog.while* combinator expresses all tail-recursive computations. Its condition is a pure value.

$\langle ML \rangle$

definition *select* :: $'v \text{ set} \Rightarrow ('s, 'v) \text{ prog}$ **where**

select $X = (\bigsqcup x \in X. prog.return x)$

context

notes $[[function-internals]]$

begin

partial-function (*lfp*) *while* :: $('k \Rightarrow ('s, 'k + 'v) \text{ prog}) \Rightarrow 'k \Rightarrow ('s, 'v) \text{ prog}$ **where**

while $c k = c k \gg (\lambda rv. \text{case } rv \text{ of } Inl k' \Rightarrow \text{while } c k' \mid Inr v \Rightarrow prog.return v)$

end

abbreviation *loop* :: $('s, unit) \text{ prog} \Rightarrow ('s, 'w) \text{ prog}$ **where**

loop $P \equiv prog.while (\lambda(). P \gg prog.return (Inl ())) ()$

abbreviation *guardM* :: $bool \Rightarrow ('s, unit) \text{ prog}$ **where**

guardM $b \equiv \text{if } b \text{ then } \perp \text{ else } prog.return ()$

abbreviation *unlessM* :: $bool \Rightarrow ('s, unit) \text{ prog} \Rightarrow ('s, unit) \text{ prog}$ **where**

unlessM $b c \equiv \text{if } b \text{ then } prog.return () \text{ else } c$

abbreviation *whenM* :: $bool \Rightarrow ('s, unit) \text{ prog} \Rightarrow ('s, unit) \text{ prog}$ **where**

$whenM\ b\ c \equiv if\ b\ then\ c\ else\ prog.return\ ()$

definition $app :: ('a \Rightarrow ('s, unit)\ prog) \Rightarrow 'a\ list \Rightarrow ('s, unit)\ prog\ \mathbf{where}$ — Haskell's $mapM$ -
 $app\ f\ xs = foldr\ (\lambda x\ m.\ f\ x \gg m)\ xs\ (prog.return\ ())$

definition $set-app :: ('a \Rightarrow ('s, unit)\ prog) \Rightarrow 'a\ set \Rightarrow ('s, unit)\ prog\ \mathbf{where}$
 $set-app\ f =$
 $prog.\mathit{while}\ (\lambda X.\ if\ X = \{\} \ then\ prog.return\ (Inr\ ()))$
 $\quad else\ prog.select\ X \gg (\lambda x.\ f\ x \gg prog.return\ (Inl\ (X - \{x\})))$

primrec $foldM :: ('b \Rightarrow 'a \Rightarrow ('s, 'b)\ prog) \Rightarrow 'b \Rightarrow 'a\ list \Rightarrow ('s, 'b)\ prog\ \mathbf{where}$
 $foldM\ f\ b\ [] = prog.return\ b$
 $| foldM\ f\ b\ (x \# xs) = do\ \{$
 $\quad b' \leftarrow f\ b\ x;$
 $\quad foldM\ f\ b'\ xs$
 $\}$

primrec $fold-mapM :: ('a \Rightarrow ('s, 'b)\ prog) \Rightarrow 'a\ list \Rightarrow ('s, 'b\ list)\ prog\ \mathbf{where}$
 $fold-mapM\ f\ [] = prog.return\ []$
 $| fold-mapM\ f\ (x \# xs) = do\ \{$
 $\quad y \leftarrow f\ x;$
 $\quad ys \leftarrow fold-mapM\ f\ xs;$
 $\quad prog.return\ (y \# ys)$
 $\}$

$\langle ML \rangle$

lemma $empty$:
shows $prog.select\ \{\} = \perp$
 $\langle proof \rangle$

lemma $singleton$:
shows $prog.select\ \{x\} = prog.return\ x$
 $\langle proof \rangle$

lemma $monotone$:
shows $mono\ prog.select$
 $\langle proof \rangle$

lemmas $strengthen[stg] = st-monotone[OF\ prog.select.monotone]$
lemmas $mono = monotoneD[OF\ prog.select.monotone, of\ P\ Q\ \mathbf{for}\ P\ Q]$
lemmas $mono2mono[cont-intro, partial-function-mono] = monotone2monotone[OF\ prog.select.monotone, simpli-$
 $fied, of\ orda\ P\ \mathbf{for}\ orda\ P]$

lemma Sup :
shows $prog.select\ (\bigcup X) = (\bigsqcup_{x \in X} prog.select\ x)$
 $\langle proof \rangle$

lemma $mcont$:
shows $mcont\ \bigcup\ (\subseteq)\ Sup\ (\leq)\ prog.select$
 $\langle proof \rangle$

lemmas $mcont2mcont[cont-intro] = mcont2mcont[OF\ prog.select.mcont, of\ supa\ orda\ P\ \mathbf{for}\ supa\ orda\ P]$
 $\langle ML \rangle$

lemma $select-le$:
assumes $x \in X$

shows $\text{prog.return } x \leq \text{prog.select } X$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *selectL*:

shows $\text{prog.select } X \ggg g = (\bigsqcup x \in X. g \ x)$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *bot*:

shows $\text{prog.while } \perp = \perp$

$\langle \text{proof} \rangle$

lemma *monotone*: — could hope to prove this with a *strengthen* rule for *lfp.fixp-fun*

shows $\text{mono } (\lambda P. \text{prog.while } P \ s)$

$\langle \text{proof} \rangle$

lemmas $\text{strengthen}[\text{strg}] = \text{st-monotone}[\text{OF } \text{prog.while.monotone}]$

lemmas $\text{mono}' = \text{monotoneD}[\text{OF } \text{prog.while.monotone}, \text{ of } P \ Q \ \text{for } P \ Q]$ — compare with *prog.while.mono*

lemmas $\text{mono2mono}[\text{cont-intro}, \text{ partial-function-mono}] = \text{monotone2monotone}[\text{OF } \text{prog.while.monotone}, \text{ simplified}, \text{ of } \text{orda } P \ \text{for } \text{orda } P]$

lemma *Sup-le*:

shows $(\bigsqcup P \in X. \text{prog.while } P \ s) \leq \text{prog.while } (\bigsqcup X) \ s$

$\langle \text{proof} \rangle$

lemma *Inf-le*:

shows $\text{prog.while } (\bigsqcap X) \ s \leq (\bigsqcap P \in X. \text{prog.while } P \ s)$

$\langle \text{proof} \rangle$

lemma *True-skip-eq-bot*:

shows $\text{prog.while } \langle \text{prog.return } (\text{Inl } x) \rangle \ s = \perp$

$\langle \text{proof} \rangle$

lemma *Inr-eq-return*:

shows $\text{prog.while } \langle \text{prog.return } (\text{Inr } v) \rangle \ s = \text{prog.return } v$

$\langle \text{proof} \rangle$

lemma *kleene-star*:

shows $\text{prog.kleene.star } P$

$= \text{prog.while } (\lambda-. (P \ggg \text{prog.return } (\text{Inl } ())) \sqcup \text{prog.return } (\text{Inr } ())) \ () \ (\text{is } ?lhs = ?rhs)$

$\langle \text{proof} \rangle$

lemma *invmap-le*:

fixes $sf :: 's \Rightarrow 't$

fixes $vf :: 'v \Rightarrow 'w$

shows $\text{prog.while } (\lambda k. \text{prog.invmap } sf \ (\text{map-sum } id \ vf) \ (c \ k)) \ k$

$\leq \text{prog.invmap } sf \ vf \ (\text{prog.while } c \ k) \ (\text{is } ?lhs \ \text{prog.while } k \leq ?rhs \ k)$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *bindL*:

fixes $P :: ('s, \text{unit}) \text{ prog}$

fixes $Q :: ('s, 'w) \text{ prog}$

shows $\text{prog.loop } P \ggg Q = \text{prog.loop } P \ (\text{is } ?lhs = ?rhs)$

$\langle proof \rangle$

lemma *parallel-le*:

shows $prog.loop\ P \leq lfp\ (\lambda R. P \parallel R)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *append*:

shows $prog.foldM\ f\ b\ (xs\ @\ ys) = prog.foldM\ f\ b\ xs \gg (\lambda b'. prog.foldM\ f\ b'\ ys)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *foldM-alt-def*:

shows $prog.foldM\ f\ b\ xs = foldr\ (\lambda x\ m. prog.bind\ m\ (\lambda b. f\ b\ x))\ (rev\ xs)\ (prog.return\ b)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *bot*:

shows $prog.fold-mapM\ \perp = (\lambda xs. case\ xs\ of\ [] \Rightarrow prog.return\ [] \mid - \Rightarrow \perp)$

$\langle proof \rangle$

lemma *append*:

shows $prog.fold-mapM\ f\ (xs\ @\ ys)$

$= prog.fold-mapM\ f\ xs \gg (\lambda xs. prog.fold-mapM\ f\ ys \gg (\lambda ys. prog.return\ (xs\ @\ ys)))$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *bot*:

shows $prog.app\ \perp = (\lambda xs. case\ xs\ of\ [] \Rightarrow prog.return\ () \mid - \Rightarrow \perp)$

and $prog.app\ (\lambda -. \perp) = (\lambda xs. case\ xs\ of\ [] \Rightarrow prog.return\ () \mid - \Rightarrow \perp)$

$\langle proof \rangle$

lemma *Nil*:

shows $prog.app\ f\ [] = prog.return\ ()$

$\langle proof \rangle$

lemma *Cons*:

shows $prog.app\ f\ (x\ \#\ xs) = f\ x \gg prog.app\ f\ xs$

$\langle proof \rangle$

lemmas *simps* =

prog.app.bot

prog.app.Nil

prog.app.Cons

lemma *append*:

shows $prog.app\ f\ (xs\ @\ ys) = prog.app\ f\ xs \gg prog.app\ f\ ys$

$\langle proof \rangle$

lemma *monotone*:

shows $mono\ (\lambda f. prog.app\ f\ xs)$

$\langle proof \rangle$

lemmas *strengthen[stg]* = *st-monotone[OF prog.app.monotone]*

lemmas *mono* = *monotoneD*[*OF prog.app.monotone*]

lemmas *mono2mono*[*cont-intro, partial-function-mono*] = *monotone2monotone*[*OF prog.app.monotone, simplified, of orda P for orda P*]

lemma *Sup-le*:

shows $(\bigsqcup f \in X. \text{prog.app } f \text{ } xs) \leq \text{prog.app } (\bigsqcup X) \text{ } xs$
<proof>

<ML>

lemma *app*:

fixes *sf* :: 's \Rightarrow 't
fixes *vf* :: 'v \Rightarrow unit
shows $\text{prog.invmap } sf \text{ } vf \text{ } (\text{prog.app } f \text{ } xs)$
 $= \text{prog.app } (\lambda x. \text{prog.sinvmap } sf \text{ } (f \text{ } x)) \text{ } xs \gg \text{prog.invmap } sf \text{ } vf \text{ } (\text{prog.return } ())$
<proof>

<ML>

lemma *app-le*:

fixes *sf* :: 's \Rightarrow 't
fixes *vf* :: 'v \Rightarrow unit
shows $\text{prog.app } (\lambda x. \text{prog.sinvmap } sf \text{ } (f \text{ } x)) \text{ } xs \leq \text{prog.sinvmap } sf \text{ } (\text{prog.app } f \text{ } xs)$
<proof>

<ML>

lemma *bot*:

shows $X \neq \{\}$ $\implies \text{prog.set-app } \perp \text{ } X = \perp$
and $X \neq \{\}$ $\implies \text{prog.set-app } (\lambda \cdot. \perp) \text{ } X = \perp$
<proof>

lemma *empty*:

shows $\text{prog.set-app } f \text{ } \{\} = \text{prog.return } ()$
<proof>

lemma *not-empty*:

assumes $X \neq \{\}$
shows $\text{prog.set-app } f \text{ } X = \text{prog.select } X \gg (\lambda x. f \text{ } x \gg \text{prog.set-app } f \text{ } (X - \{x\}))$
<proof>

lemmas *simps* =

prog.set-app.bot
prog.set-app.empty
prog.set-app.not-empty

<ML>

lemma *set-app-le*:

assumes $X = \text{set } xs$
assumes *distinct xs*
shows $\text{prog.app } f \text{ } xs \leq \text{prog.set-app } f \text{ } X$
<proof>

<ML>

lemma *set-app-alt-def*:

assumes *finite X*

shows $\text{prog.set-app } f X = (\bigsqcup xs \in \{ys. \text{set } ys = X \wedge \text{distinct } ys\}. \text{prog.app } f xs)$ (**is** $?lhs = ?rhs$)
 ⟨proof⟩

⟨ML⟩

lemma *select-sp*:

assumes $\bigwedge s x. \llbracket P s; x \in X \rrbracket \Longrightarrow Q x s$

assumes $\bigwedge v. \text{stable } A (P \wedge Q v)$

shows $\text{prog.p2s } (\text{prog.select } X) \leq \llbracket P \rrbracket, A \vdash G, \llbracket \lambda v. P \wedge Q v \rrbracket$

⟨proof⟩

lemma *while*:

fixes $c :: 'k \Rightarrow ('s, 'k + 'v) \text{ prog}$

assumes $c: \bigwedge k. \text{prog.p2s } (c k) \leq \llbracket P k \rrbracket, A \vdash G, \llbracket \text{case-sum } I Q \rrbracket$

assumes $IP: \bigwedge s v. I v s \Longrightarrow P v s$

assumes $sQ: \bigwedge v. \text{stable } A (Q v)$

shows $\text{prog.p2s } (\text{prog.while } c k) \leq \llbracket I k \rrbracket, A \vdash G, \llbracket Q \rrbracket$

⟨proof⟩

lemma *app*:

fixes $xs :: 'a \text{ list}$

fixes $f :: 'a \Rightarrow ('s, \text{unit}) \text{ prog}$

fixes $P :: 'a \text{ list} \Rightarrow 's \text{ pred}$

assumes $\bigwedge x ys zs. xs = ys @ x \# zs \Longrightarrow \text{prog.p2s } (f x) \leq \llbracket P ys \rrbracket, A \vdash G, \llbracket \lambda -. P (ys @ [x]) \rrbracket$

assumes $\bigwedge ys. \text{prefix } ys xs \Longrightarrow \text{stable } A (P ys)$

shows $\text{prog.p2s } (\text{prog.app } f xs) \leq \llbracket P [] \rrbracket, A \vdash G, \llbracket \lambda -. P xs \rrbracket$

⟨proof⟩

lemma *app-set*:

fixes $X :: 'a \text{ set}$

fixes $f :: 'a \Rightarrow ('s, \text{unit}) \text{ prog}$

fixes $P :: 'a \text{ set} \Rightarrow 's \text{ pred}$

assumes $\bigwedge Y x. \llbracket Y \subseteq X; x \in X - Y \rrbracket \Longrightarrow \text{prog.p2s } (f x) \leq \llbracket P Y \rrbracket, A \vdash G, \llbracket \lambda -. P (\text{insert } x Y) \rrbracket$

assumes $\bigwedge Y. Y \subseteq X \Longrightarrow \text{Stability.stable } A (P Y)$

shows $\text{prog.p2s } (\text{prog.set-app } f X) \leq \llbracket P \{\} \rrbracket, A \vdash G, \llbracket \lambda -. P X \rrbracket$

⟨proof⟩

lemma *foldM*:

fixes $xs :: 'a \text{ list}$

fixes $f :: 'b \Rightarrow 'a \Rightarrow ('s, 'b) \text{ prog}$

fixes $I :: 'b \Rightarrow 'a \Rightarrow 's \text{ pred}$

fixes $P :: 'b \Rightarrow 's \text{ pred}$

assumes $f: \bigwedge b x. x \in \text{set } xs \Longrightarrow \text{prog.p2s } (f b x) \leq \llbracket I b x \rrbracket, A \vdash G, \llbracket P \rrbracket$

assumes $P: \bigwedge b x s. \llbracket P b s; x \in \text{set } xs \rrbracket \Longrightarrow I b x s$

assumes $sP: \bigwedge b. \text{stable } A (P b)$

shows $\text{prog.p2s } (\text{prog.foldM } f b xs) \leq \llbracket P b \rrbracket, A \vdash G, \llbracket P \rrbracket$

⟨proof⟩

⟨ML⟩

⟨proof⟩⟨proof⟩⟨proof⟩

15 Structural local state

15.1 *spec.local*

We develop a few combinators for structural local state. The goal is to encapsulate a local state of type $'ls$ in a process $('a \text{ agent}, 'ls \times 's, 'v) \text{ spec}$. Applying spec.smap snd yields a process of type $('a \text{ agent}, 's, 'v) \text{ spec}$. We also constrain environment steps to not affect $'ls$, yielding a plausible data refinement rule (see §15.6.1).

abbreviation (*input*) $localize1 :: ('b \Rightarrow 's \Rightarrow 'a) \Rightarrow 'b \Rightarrow 'ls \times 's \Rightarrow 'a$ **where**
 $localize1 f b s \equiv f b (snd s)$

$\langle ML \rangle$

definition $qrm :: ('a \text{ agent}, 'ls \times 's)$ *steps* **where** — cf *ag.assm*
 $qrm = range \text{ proc} \times UNIV \cup \{env\} \times (Id \times_R UNIV)$

abbreviation (*input*) $interference \equiv spec.rel \text{ spec.local.qrm}$

$\langle ML \rangle$

definition $local :: ('a \text{ agent}, 'ls \times 's, 'v)$ *spec* $\Rightarrow ('a \text{ agent}, 's, 'v)$ *spec* **where**
 $local P = spec.smap \text{ snd} (spec.local.interference \sqcap P)$

$\langle ML \rangle$

lemma *local-le-conv*:

shows $\langle \sigma \rangle \leq spec.local P$
 $\longleftrightarrow (\exists \sigma'. \langle \sigma' \rangle \leq P$
 $\quad \wedge trace.steps \sigma' \subseteq spec.local.qrm$
 $\quad \wedge \langle \sigma \rangle \leq \langle trace.map \text{ id } \text{ snd } \text{ id } \sigma' \rangle)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *local-le[spec.idle-le]*: — Converse does not hold

assumes $spec.idle \leq P$
shows $spec.idle \leq spec.local P$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *refl*:

shows $refl (spec.local.qrm \text{ “ } \{a\})$

$\langle proof \rangle$

lemma *member*:

shows $(\text{proc } a, s, s') \in spec.local.qrm$
and $(env, s, s') \in spec.local.qrm \longleftrightarrow fst s = fst s'$

$\langle proof \rangle$

lemma *inter*:

shows $UNIV \times Id \cap spec.local.qrm = UNIV \times Id$
and $spec.local.qrm \cap UNIV \times Id = UNIV \times Id$
and $spec.local.qrm \cap \{self\} \times Id = \{self\} \times Id$
and $spec.local.qrm \cap \{env\} \times UNIV = \{env\} \times (Id \times_R UNIV)$
and $spec.local.qrm \cap \{env\} \times (UNIV \times_R Id) = \{env\} \times Id$
and $spec.local.qrm \cap A \times (Id \times_R r) = A \times (Id \times_R r)$

$\langle proof \rangle$

lemmas *simps[simp]* =

$spec.local.qrm.refl$
 $spec.local.qrm.member$
 $spec.local.qrm.inter$

$\langle ML \rangle$

lemma *smap-snd*:

shows $\text{spec.smap snd spec.local.interference} = \top$
<proof>

<ML>

lemma *inf-interference*:

shows $\text{spec.local } P = \text{spec.local } (P \sqcap \text{spec.local.interference})$
<proof>

lemma *bot*:

shows $\text{spec.local } \perp = \perp$
<proof>

lemma *top*:

shows $\text{spec.local } \top = \top$
<proof>

lemma *monotone*:

shows *mono spec.local*
<proof>

lemmas *strengthen[strg] = st-monotone[OF spec.local.monotone]*

lemmas *mono = monotoneD[OF spec.local.monotone]*

lemmas *mono2mono[cont-intro, partial-function-mono]*
= monotone2monotone[OF spec.local.monotone, simplified, of orda P for orda P]

lemma *Sup*:

shows $\text{spec.local } (\bigsqcup X) = (\bigsqcup x \in X. \text{spec.local } x)$
<proof>

lemmas *sup = spec.local.Sup[where X={X, Y} for X Y, simplified]*

lemma *mcont2mcont[cont-intro]*:

assumes *mcont luba orda Sup (\leq) P*
shows *mcont luba orda Sup (\leq) ($\lambda x. \text{spec.local } (P x)$)*
<proof>

lemma *idle*:

shows $\text{spec.local spec.idle} = \text{spec.idle}$
<proof>

lemma *action*:

fixes $F :: ('v \times 'a \text{ agent} \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
shows $\text{spec.local } (\text{spec.action } F)$
 $= \text{spec.action } (\text{map-prod id } (\text{map-prod id } (\text{map-prod snd snd})) \text{ '}$
 $(F \cap \text{UNIV} \times \text{spec.local.qrm}))$
<proof>

lemma *return*:

shows $\text{spec.local } (\text{spec.return } v) = \text{spec.return } v$
<proof>

lemma *bind-le*: — Converse does not hold

shows $\text{spec.local } (f \ggg g) \leq \text{spec.local } f \ggg (\lambda v. \text{spec.local } (g v))$
<proof>

lemma *interference*:

shows $spec.local (spec.rel (\{env\} \times UNIV)) = spec.rel (\{env\} \times UNIV)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *local-le*:

shows $spec.map id sf vf (spec.local P) \leq spec.local (spec.map id (map-prod id sf) vf P)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *local*:

shows $spec.vmap vf (spec.local P) = spec.local (spec.vmap vf P)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *smap-snd*:

fixes $P :: ('a, 'ls \times 't, 'w) spec$

fixes $sf :: 's \Rightarrow 't$

fixes $vf :: 'v \Rightarrow 'w$

shows $spec.invmap id sf vf (spec.smap snd P)$

$= spec.smap snd (spec.invmap id (map-prod id sf) vf P)$ (**is** ?lhs = ?rhs)

$\langle proof \rangle$

lemma *local*:

fixes $P :: ('a agent, 'ls \times 't, 'v) spec$

fixes $sf :: 's \Rightarrow 't$

shows $spec.invmap id sf vf (spec.local P) = spec.local (spec.invmap id (map-prod id sf) vf P)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *local*:

shows $spec.term.none (spec.local P) = spec.local (spec.term.none P)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *local*:

shows $spec.term.all (spec.local P) = spec.local (spec.term.all P)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *local*:

assumes $P \in spec.term.closed -$

shows $spec.local P \in spec.term.closed -$

$\langle proof \rangle$

$\langle ML \rangle$

15.2 Local state transformations

We want to reorder, introduce and eliminate actions that affect local state while preserving observable behaviour under *spec.local*.

The closure that arises from *spec.local*, i.e.:

lemma

defines $cl \equiv spec.map\text{-}invmap.cl - - id\ snd\ id$

assumes $spec.local.interference \sqcap P$

$\leq cl (spec.local.interference \sqcap Q)$

shows $spec.local P \leq spec.local Q$

$\langle proof \rangle$

expresses all transformations, but does not decompose over (\gg); in other words we do not have $cl\ f \gg (\lambda v. cl\ (g\ v)) \leq cl\ (f \gg g)$ as the local states that $cl\ f$ terminates with may not satisfy g . (Observe that we do not expect the converse to hold as then all local states would need to be preserved.)

We therefore define a closure that preserves the observable state and the initial and optionally final (if terminating) local states via a projection:

$\langle ML \rangle$

definition $prj :: bool \Rightarrow ('a, 'ls \times 's, 'v)\ trace.t \Rightarrow ('a, 's, 'v)\ trace.t \times 'ls \times 'ls\ option$ **where**

$prj\ T\ \sigma = (\text{h}(trace.map\ id\ snd\ id\ \sigma),$

$\text{fst}\ (trace.init\ \sigma),$

$\text{if}\ T\ \text{then}\ map\ option\ (\text{fst}\ (trace.final\ \sigma))\ (trace.term\ \sigma)\ \text{else}\ None)$

$\langle ML \rangle$

lemma *natural*:

shows $seq\text{-}ctxt.prj\ T\ (\text{h}\sigma) = seq\text{-}ctxt.prj\ T\ \sigma$

$\langle proof \rangle$

lemma *idle*:

shows $seq\text{-}ctxt.prj\ T\ (trace.T\ s\ []\ None) = (trace.T\ (snd\ s)\ []\ None, \text{fst}\ s, None)$

$\langle proof \rangle$

lemmas $simps[simp] =$

$seq\text{-}ctxt.prj.natural$

$\langle ML \rangle$

interpretation $seq\text{-}ctxt: galois.image\text{-}vimage\ seq\text{-}ctxt.prj\ T\ \text{for}\ T\ \langle proof \rangle$

$\langle ML \rangle$

lemma *partial-sel-equivE*:

assumes $seq\text{-}ctxt.equivalent\ T\ \sigma_1\ \sigma_2$

obtains $trace.init\ \sigma_1 = trace.init\ \sigma_2$

and $trace.term\ \sigma_1 = trace.term\ \sigma_2$

and $\llbracket T; \exists v. trace.term\ \sigma_1 = Some\ v \rrbracket \implies trace.final\ \sigma_1 = trace.final\ \sigma_2$

$\langle proof \rangle$

lemma *downwards-existsE*:

assumes $\sigma_1' \leq \sigma_1$

assumes $seq\text{-}ctxt.equivalent\ T\ \sigma_1\ \sigma_2$

obtains σ_2'

where $\sigma_2' \leq \sigma_2$

and $seq\text{-}ctxt.equivalent\ T\ \sigma_1'\ \sigma_2'$

$\langle proof \rangle$

lemma *downwards-existsE2*:

assumes $\sigma_1' \leq \sigma_1$

assumes $seq\text{-}ctxt.equivalent\ T\ \sigma_1'\ \sigma_2'$

obtains σ_2

where $\sigma_2' \leq \sigma_2$

and *seq-ctxt.equivalent* $T \sigma_1 \sigma_2$

<proof>

lemma *map-sf-eq-id*:

assumes *seq-ctxt.equivalent* $\text{True} \sigma_1 \sigma_2$

shows *seq-ctxt.equivalent* $\text{True} (\text{trace.map af id vf } \sigma_1) (\text{trace.map af id vf } \sigma_2)$

<proof>

lemma *mono*:

assumes $T \implies T'$

assumes *seq-ctxt.equivalent* $T' \sigma_1 \sigma_2$

shows *seq-ctxt.equivalent* $T \sigma_1 \sigma_2$

<proof>

lemma *append*:

assumes *seq-ctxt.equivalent* $\text{True} (\text{trace.T } s \text{ xs } (\text{Some } v)) (\text{trace.T } s' \text{ xs}' v')$

assumes *seq-ctxt.equivalent* $T (\text{trace.T } (\text{trace.final}' s \text{ xs}) \text{ ys } w) (\text{trace.T } t' \text{ ys}' w')$

shows *seq-ctxt.equivalent* $T (\text{trace.T } s \text{ (xs @ ys)} w) (\text{trace.T } s' \text{ (xs}' @ \text{ys}') } w')$

<proof>

<ML>

definition $cl :: \text{bool} \Rightarrow ('a, 'ls \times 's, 'v) \text{ spec} \Rightarrow ('a, 'ls \times 's, 'v) \text{ spec}$ **where**

$cl \ T \ P = \bigsqcup (\text{spec.singleton } \{ \sigma_1. \exists \sigma_2. \langle \sigma_2 \rangle \leq P \wedge \text{seq-ctxt.equivalent } T \ \sigma_1 \ \sigma_2 \})$

<ML>

lemma *cl-le-conv[spec.singleton.le-conv]*:

shows $\langle \sigma \rangle \leq \text{spec.seq-ctxt.cl } T \ P \longleftrightarrow (\exists \sigma'. \langle \sigma' \rangle \leq P \wedge \text{seq-ctxt.equivalent } T \ \sigma \ \sigma')$ (**is** ?lhs \longleftrightarrow ?rhs)

<proof>

<ML>

interpretation *seq-ctxt*: *closure-complete-distrib-lattice-distributive-class* *spec.seq-ctxt.cl* T **for** F

<proof>

<ML>

lemma *cl-le-conv[spec.idle-le]*:

shows $\text{spec.idle} \leq \text{spec.seq-ctxt.cl } T \ P \longleftrightarrow \text{spec.idle} \leq P$ (**is** ?lhs \longleftrightarrow ?rhs)

<proof>

<ML>

lemma *bot[simp]*:

shows $\text{spec.seq-ctxt.cl } T \ \perp = \perp$

<proof>

lemma *mono*:

assumes $T' \implies T$

assumes $P \leq P'$

shows $\text{spec.seq-ctxt.cl } T \ P \leq \text{spec.seq-ctxt.cl } T' \ P'$

<proof>

lemma *strengthen[strg]*:

assumes *st-ord* $(\neg F) \ T \ T'$

assumes *st-ord* $F \ P \ P'$

shows *st-ord* $F \ (\text{spec.seq-ctxt.cl } T \ P) \ (\text{spec.seq-ctxt.cl } T' \ P')$

$\langle proof \rangle$

lemma *Sup*:

shows $spec.seq-ctxt.cl\ T\ (\bigsqcup X) = \bigsqcup (spec.seq-ctxt.cl\ T\ 'X)$

$\langle proof \rangle$

lemmas $sup = spec.seq-ctxt.cl.Sup[\text{where } X=\{P, Q\} \text{ for } P\ Q, \text{ simplified}]$

lemma *singleton*:

shows $spec.seq-ctxt.cl\ T\ \langle \sigma \rangle = \bigsqcup (spec.singleton\ ' \{\sigma'.\ seq-ctxt.equivalent\ T\ \sigma\ \sigma'\})$ (is ?lhs = ?rhs)

$\langle proof \rangle$

lemma *idle*: — not *simp* friendly

shows $spec.seq-ctxt.cl\ T\ (spec.idle :: ('a, 'ls \times 's, 'v)\ spec)$

$= spec.term.none\ (spec.rel\ (UNIV \times (UNIV \times_R\ Id)) :: ('a, 'ls \times 's, 'w)\ spec)$ (is ?lhs = ?rhs)

$\langle proof \rangle$

lemma *invmap-le*:

shows $spec.seq-ctxt.cl\ True\ (spec.invmap\ af\ id\ vf\ P) \leq spec.invmap\ af\ id\ vf\ (spec.seq-ctxt.cl\ True\ P)$

$\langle proof \rangle$

lemma *map-le*:

shows $spec.map\ af\ id\ vf\ (spec.seq-ctxt.cl\ True\ P) \leq spec.seq-ctxt.cl\ True\ (spec.map\ af\ id\ vf\ P)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *cl*:

shows $spec.term.none\ (spec.seq-ctxt.cl\ T\ P) = spec.seq-ctxt.cl\ T\ (spec.term.none\ P)$

$\langle proof \rangle$

lemma *cl-True-False*:

shows $spec.seq-ctxt.cl\ True\ (spec.term.none\ f) = spec.seq-ctxt.cl\ False\ (spec.term.none\ f)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *cl-le*:

shows $spec.seq-ctxt.cl\ T\ (spec.term.all\ P) \leq spec.term.all\ (spec.seq-ctxt.cl\ T\ P)$

$\langle proof \rangle$

lemma *cl-False*:

shows $spec.seq-ctxt.cl\ False\ (spec.term.all\ P) = spec.term.all\ (spec.seq-ctxt.cl\ False\ P)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *cl-le*:

shows $spec.seq-ctxt.cl\ True\ f \ggg (\lambda v. spec.seq-ctxt.cl\ T\ (g\ v)) \leq spec.seq-ctxt.cl\ T\ (f \ggg g)$

$\langle proof \rangle$

lemma *clL-le*:

shows $spec.seq-ctxt.cl\ True\ f \ggg g \leq spec.seq-ctxt.cl\ T\ (f \ggg g)$

$\langle proof \rangle$

lemma *clR-le*:

shows $f \ggg (\lambda v. spec.seq-ctxt.cl\ T\ (g\ v)) \leq spec.seq-ctxt.cl\ T\ (f \ggg g)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *cl-local-le*: — the RHS is the closure that arises from *spec.local*, ignoring the constraint

shows *spec.seq-ctxt.cl* $T P \leq \text{spec.map-invmap.cl} - - \text{id snd id } P$

$\langle \text{proof} \rangle$

lemma *cl-local*:

shows *spec.local* (*spec.seq-ctxt.cl* T (*spec.local.interference* $\sqcap P$))

= *spec.local* P (**is** *?lhs* = *?rhs*)

$\langle \text{proof} \rangle$

lemma *cl-imp-local-le*:

assumes *spec.local.interference* $\sqcap P$

$\leq \text{spec.seq-ctxt.cl } \text{False} (\text{spec.local.interference} \sqcap Q)$

shows *spec.local* $P \leq \text{spec.local } Q$

$\langle \text{proof} \rangle$

lemma *cl-inf-pre*:

shows *spec.pre* $P \sqcap \text{spec.seq-ctxt.cl } T c = \text{spec.seq-ctxt.cl } T (\text{spec.pre } P \sqcap c)$

$\langle \text{proof} \rangle$

lemma *cl-pre-le-conv*:

shows *spec.seq-ctxt.cl* $T c \leq \text{spec.pre } P \longleftrightarrow c \leq \text{spec.pre } P$ (**is** *?lhs* \longleftrightarrow *?rhs*)

$\langle \text{proof} \rangle$

lemma *cl-inf-post*:

shows *spec.post* $Q \sqcap \text{spec.seq-ctxt.cl } \text{True } c = \text{spec.seq-ctxt.cl } \text{True} (\text{spec.post } Q \sqcap c)$

$\langle \text{proof} \rangle$

lemma *cl-post-le-conv*:

shows *spec.seq-ctxt.cl* $\text{True } c \leq \text{spec.post } Q \longleftrightarrow c \leq \text{spec.post } Q$ (**is** *?lhs* \longleftrightarrow *?rhs*)

$\langle \text{proof} \rangle$

$\langle ML \rangle$

15.2.1 Permuting local actions

We can reorder operations on the local state as these are not observable.

Firstly: an initial action F that does not change the observable state can be swapped with an arbitrary action G .

$\langle ML \rangle$

lemma *cl-action-permuteL-le*:

fixes $F :: ('v \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

fixes $G :: 'v \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

fixes $G' :: ('v' \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

fixes $F' :: 'v' \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

— F does not change $'s$, can be partial

assumes $F: \bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in F \rrbracket \implies \text{snd } s' = \text{snd } s$

— The final state and return value are independent of the order of actions. F' does not change $'s$, cannot be partial

assumes $FGG'F': \bigwedge v w a a' s s' t. \llbracket P s; (v, a', s, t) \in F; (w, a, t, s') \in G v \rrbracket$

$\implies \exists v' a'' a''' s'' t'. (v', a'', s, t') \in G' \wedge (w, a''', t', s') \in F' v'$

$\wedge \text{snd } s' = \text{snd } t' \wedge (\text{snd } s \neq \text{snd } t' \longrightarrow a'' = a) \wedge (T \longrightarrow \text{fst } s'' = \text{fst } s') \wedge \text{snd } s'' = \text{snd } t'$

shows (*spec.action* $F \ggg (\lambda v. \text{spec.action } (G v))$) $\sqcap \text{spec.pre } P$

$\leq \text{spec.seq-ctxt.cl } T (\text{spec.action } G' \ggg (\lambda v. \text{spec.action } (F' v)))$ (**is** $- \leq$ *?rhs*)

$\langle \text{proof} \rangle$

Secondly: an initial action G that does change the observable state can be swapped with an arbitrary action F that does not observably change the state.

lemma *cl-action-permuteR-le*:

fixes $G :: ('v \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

fixes $F :: 'v \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

fixes $F' :: ('v' \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

fixes $G' :: 'v' \Rightarrow ('w' \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

— F does not stall if G makes an observable state change

assumes $G: \bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in G; \text{snd } s' \neq \text{snd } s \rrbracket$

$\implies \exists v' w a'' t s''. (v', a'', s, t) \in F' \wedge (w, a, t, s'') \in G \wedge \text{snd } t = \text{snd } s \wedge \text{snd } s'' = \text{snd } s'$

— The final state and return value are independent of the order of actions

assumes $GFF'G': \bigwedge v w a a' s s' t. \llbracket P s; (v, a, s, t) \in G; (w, a', t, s') \in F v \rrbracket$

$\implies \text{snd } s' = \text{snd } t \wedge (\exists v' a'' a''' s'' t'. (v', a'', s, t') \in F' \wedge (w, a''', t', s'') \in G' v'$

$\wedge \text{snd } t' = \text{snd } s \wedge (T \longrightarrow \text{fst } s'' = \text{fst } s') \wedge \text{snd } s'' = \text{snd } s' \wedge (\text{snd}$

$s'' \neq \text{snd } t' \longrightarrow a''' = a)$

shows $(\text{spec.action } G \gg (\lambda v. \text{spec.action } (F v))) \sqcap \text{spec.pre } P$

$\leq \text{spec.seq-ctxt.cl } T (\text{spec.action } F' \gg (\lambda v. \text{spec.action } (G' v)))$

<proof>

lemma *cl-action-bind-action-pre-post*:

fixes $F' :: ('v \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

fixes $G' :: 'v \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

fixes $Q :: 'w \Rightarrow ('ls \times 's) \text{ pred}$

assumes $\bigwedge v w a a' s s' s''. \llbracket P s; (v, a, s, s') \in F; (w, a', s', s'') \in G v \rrbracket \implies Q w s''$

shows $\text{spec.pre } P \sqcap \text{spec.seq-ctxt.cl } \text{True} (\text{spec.action } F' \gg (\lambda v. \text{spec.action } (G v))) \leq \text{spec.post } Q$

<proof>

lemma *cl-rev-kleene-star-action-permute-le*:

fixes $F G :: (\text{unit} \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

— F does not stall if G changes the observable state

assumes $G: \bigwedge a s s'. \llbracket (((), a, s, s') \in G; \text{snd } s' \neq \text{snd } s) \rrbracket$

$\implies \exists w a'' t s''. (((), a'', s, t) \in F \wedge (((), a, t, s'') \in G \wedge \text{snd } t = \text{snd } s \wedge \text{snd } s'' = \text{snd } s')$

— The final state is independent of order of actions, F does not change $'s$, can be partial

assumes $GFFG: \bigwedge a a' s s' t. \llbracket (((), a, s, t) \in G; (((), a', t, s') \in F) \rrbracket$

$\implies \text{snd } s' = \text{snd } t \wedge (\exists a'' a''' t'. (((), a'', s, t') \in F \wedge (((), a''', t', s') \in G$

$\wedge \text{snd } t' = \text{snd } s \wedge (\text{snd } s' \neq \text{snd } t' \longrightarrow a''' = a)$

shows $\text{spec.kleene.rev-star } (\text{spec.action } G) \gg (\lambda::\text{unit. spec.action } F)$

$\leq \text{spec.seq-ctxt.cl } \text{True} (\text{spec.action } F \gg \text{spec.kleene.rev-star } (\text{spec.action } G)) \text{ (is ?lhs spec.kleene.rev-star } \leq$

?rhs)

<proof>

lemma *cl-local-action-interference-permute-le*: — local actions permute with interference

fixes $F :: (\text{unit} \times 'a \text{ agent} \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

fixes $r :: 's \text{ rel}$

— F does not block

assumes $\bigwedge s ls. \exists v a ls'. (v, a, (ls, s), (ls', s)) \in F$

— F is insensitive to and does not modify the shared state

assumes $\bigwedge v a s s' s'' ls ls'. (v, a, (ls, s), (ls', s')) \in F$

$\implies s' = s \wedge (v, a, (ls, s''), (ls', s'')) \in F$

shows $\text{spec.rel } (A \times (\text{Id} \times_R r)) \gg (\lambda::\text{unit. spec.action } F)$

$\leq \text{spec.seq-ctxt.cl } \text{True} (\text{spec.action } F \gg \text{spec.rel } (A \times (\text{Id} \times_R r)))$

<proof>

lemma *cl-action-mumble-trailing-le*:

assumes $\bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in F \rrbracket$

$\implies \exists a' ls'. (v, a', s, (ls', \text{snd } s')) \in F'$

$\wedge (\text{snd } s' \neq \text{snd } s \longrightarrow a' = a) \wedge (T \longrightarrow ls' = \text{fst } s')$

shows $\text{spec.action } F \sqcap \text{spec.pre } P \leq \text{spec.seq-ctxt.cl } T \text{ (spec.action } F')$
 ⟨proof⟩

lemma *cl-action-mumbleL-le*:

assumes $\bigwedge w a s s'. \llbracket P s; (w, a, s, s') \in G \rrbracket$
 $\implies \exists v a' a'' t s''. (v, a', s, t) \in F' \wedge (w, a'', t, s') \in G' v$
 $\quad \wedge \text{snd } t = \text{snd } s \wedge (T \longrightarrow \text{fst } s'' = \text{fst } s')$
 $\quad \wedge \text{snd } s'' = \text{snd } s' \wedge (\text{snd } s'' \neq \text{snd } t \longrightarrow a'' = a)$

shows $\text{spec.action } G \sqcap \text{spec.pre } P \leq \text{spec.seq-ctxt.cl } T \text{ (spec.action } F' \ggg (\lambda v. \text{spec.action } (G' v)))$
 ⟨proof⟩

lemma *cl-action-mumbleR-le*:

assumes $\bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in G \rrbracket$
 $\implies \exists w a' a'' t. (w, a', s, t) \in G' \wedge (v, a'', t, s') \in F' w$
 $\quad \wedge \text{snd } t = \text{snd } s' \wedge (\text{snd } t \neq \text{snd } s \longrightarrow a' = a)$

shows $\text{spec.action } G \sqcap \text{spec.pre } P \leq \text{spec.seq-ctxt.cl } T \text{ (spec.action } G' \ggg (\lambda v. \text{spec.action } (F' v)))$
 ⟨proof⟩

lemma *cl-action-mumble-expandL-le*:

assumes $\bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in F \rrbracket \implies \text{snd } s' = \text{snd } s$
assumes $\bigwedge v w a a' s s' s''. \llbracket P s; (v, a, s, s') \in F; (w, a', s', s'') \in G v \rrbracket$
 $\implies \exists s'''. (w, a', s, s''') \in G' \wedge \text{snd } s''' = \text{snd } s'' \wedge (T \longrightarrow \text{fst } s''' = \text{fst } s'')$

shows $(\text{spec.action } F \ggg (\lambda v. \text{spec.action } (G v))) \sqcap \text{spec.pre } P \leq \text{spec.seq-ctxt.cl } T \text{ (spec.action } G')$
 ⟨proof⟩

lemma *cl-action-mumble-expandR-le*:

assumes $\bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in G; \text{snd } s' \neq \text{snd } s \rrbracket \implies \exists v' s''. (v', a, s, s'') \in G' \wedge \text{snd } s'' = \text{snd } s'$
assumes $\bigwedge v w a a' s s' t. \llbracket P s; (v, a, s, t) \in G; (w, a', t, s') \in F v \rrbracket$
 $\implies \text{snd } s' = \text{snd } t \wedge (\exists a'' s''. (w, a'', s, s'') \in G' \wedge \text{snd } s'' = \text{snd } s' \wedge (T \longrightarrow \text{fst } s'' = \text{fst } s') \wedge$
 $(\text{snd } s'' \neq \text{snd } s \longrightarrow a'' = a))$

shows $(\text{spec.action } G \ggg (\lambda v. \text{spec.action } (F v))) \sqcap \text{spec.pre } P \leq \text{spec.seq-ctxt.cl } T \text{ (spec.action } G')$
 ⟨proof⟩

⟨ML⟩

lemma *init-write-interference-permute-le*:

fixes $P :: ('a \text{ agent}, 'ls \times 's, 'v) \text{ spec}$
shows $\text{spec.local } (\text{spec.rel } (\{\text{env}\} \times \text{UNIV}) \ggg (\lambda :: \text{unit}. \text{spec.write } (\text{proc } a) (\text{map-prod } \langle \text{ls} \rangle \text{ id} \ggg P))$
 $\leq \text{spec.local } (\text{spec.write } (\text{proc } a) (\text{map-prod } \langle \text{ls} \rangle \text{ id} \ggg (\text{spec.rel } (\{\text{env}\} \times \text{UNIV}) \ggg (\lambda :: \text{unit}. P)))$

⟨proof⟩

lemma *init-write-interference2-permute-le*:

fixes $P :: ('a \text{ agent}, 'ls \times 's, 'v) \text{ spec}$
shows $\text{spec.local } (\text{spec.rel } (A \times (\text{Id} \times_R r)) \ggg (\lambda :: \text{unit}. \text{spec.write } (\text{proc } a) (\text{map-prod } \langle \text{ls} \rangle \text{ id} \ggg P))$
 $\leq \text{spec.local } (\text{spec.write } (\text{proc } a) (\text{map-prod } \langle \text{ls} \rangle \text{ id} \ggg (\text{spec.rel } (A \times (\text{Id} \times_R r)) \ggg (\lambda :: \text{unit}. P)))$

⟨proof⟩

lemma *trailing-local-act*:

fixes $F :: 'v \Rightarrow ('w \times 'a \text{ agent} \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
shows $\text{spec.local } (P \ggg (\lambda v. \text{spec.action } (F v)))$

$= \text{spec.local } (P \ggg (\lambda v. \text{spec.action } \{(w, a, (ls, s), (ls, s')) \mid w a ls s ls' s'. (w, a, (ls, s), (ls', s')) \in F v \wedge (a$
 $= \text{env} \longrightarrow ls' = ls)\})) \text{ (is ?lhs = ?rhs)}$

⟨proof⟩

⟨ML⟩

15.3 *spec.localize*

We can transform a process into one with the same observable behavior that ignores a local state. For compositionality we allow the *env* steps to change the local state but not the *self* steps.

$\langle ML \rangle$

definition *localize* :: 'ls rel \Rightarrow ('a agent, 's, 'v) spec \Rightarrow ('a agent, 'ls \times 's, 'v) spec **where**
localize r P = spec.rel ({env} \times (r \times_R UNIV) \cup range proc \times (Id \times_R UNIV)) \sqcap spec.sinvmap snd P

$\langle ML \rangle$

lemma *localize-le*:

assumes spec.idle \leq P

shows spec.idle \leq spec.localize r P

$\langle proof \rangle$

$\langle ML \rangle$

lemma *localize*:

shows spec.term.none (spec.localize r P) = spec.localize r (spec.term.none P)

$\langle proof \rangle$

$\langle ML \rangle$

lemma *localize*:

shows spec.term.all (spec.localize r P) = spec.localize r (spec.term.all P)

$\langle proof \rangle$

$\langle ML \rangle$

lemma *localize*:

assumes P \in spec.term.closed -

shows spec.localize r P \in spec.term.closed -

$\langle proof \rangle$

$\langle ML \rangle$

lemma *singleton*:

fixes σ :: ('a agent, 's, 'v) trace.t

shows spec.localize Id $\langle \sigma \rangle$ = (\bigsqcup ls::'ls. $\langle trace.map id (Pair ls) id \sigma \rangle$) (is ?lhs = ?rhs)

$\langle proof \rangle$

lemma *bot*:

shows spec.localize r \perp = \perp

$\langle proof \rangle$

lemma *top*:

shows spec.localize r \top = spec.rel ({env} \times (r \times_R UNIV) \cup range proc \times (Id \times_R UNIV))

$\langle proof \rangle$

lemma *Sup*:

shows spec.localize r (\bigsqcup X) = (\bigsqcup x \in X. spec.localize r x)

$\langle proof \rangle$

lemmas sup = spec.localize.Sup[**where** X={X, Y} **for** X Y, *simplified*]

lemma *mono*:

assumes r \subseteq r'

assumes $P \leq P'$
shows $\text{spec.localize } r \ P \leq \text{spec.localize } r' \ P'$
 $\langle \text{proof} \rangle$

lemma *strengthen*[*strg*]:
assumes $\text{st-ord } F \ r \ r'$
assumes $\text{st-ord } F \ P \ P'$
shows $\text{st-ord } F \ (\text{spec.localize } r \ P) \ (\text{spec.localize } r' \ P')$
 $\langle \text{proof} \rangle$

lemma *mono2mono*[*cont-intro, partial-function-mono*]:
assumes $\text{monotone } \text{orda } (\leq) \ r$
assumes $\text{monotone } \text{orda } (\leq) \ P$
shows $\text{monotone } \text{orda } (\leq) \ (\lambda x. \text{spec.localize } (r \ x) \ (P \ x))$
 $\langle \text{proof} \rangle$

lemma *mcont2mcont*[*cont-intro*]:
assumes $\text{mcont } \text{luba } \text{orda } \text{Sup } (\leq) \ P$
shows $\text{mcont } \text{luba } \text{orda } \text{Sup } (\leq) \ (\lambda x. \text{spec.localize } r \ (P \ x))$
 $\langle \text{proof} \rangle$

lemma *bind*:
shows $\text{spec.localize } r \ (f \ggg g) = \text{spec.localize } r \ f \ggg (\lambda v. \text{spec.localize } r \ (g \ v))$
 $\langle \text{proof} \rangle$

lemma *action*:
fixes $F :: ('v \times 'a \ \text{agent} \times 's \times 's) \ \text{set}$
shows $\text{spec.localize } r \ (\text{spec.action } F)$
 $= \text{spec.rel } (\{\text{env}\} \times (r \times_R \text{Id}))$
 $\ggg (\lambda :: \text{unit}. \text{spec.action } ((\text{map-prod id } (\text{map-prod id } (\text{map-prod snd } \text{snd})) - ' F)$
 $\quad \cap \text{UNIV} \times (\{\text{env}\} \times (r \times_R \text{UNIV}) \cup \text{range proc} \times (\text{Id} \times_R \text{UNIV}) \cup \text{UNIV} \times \text{Id}))$
 $\ggg (\lambda v. \text{spec.rel } (\{\text{env}\} \times (r \times_R \text{Id})) \ggg (\lambda :: \text{unit}. \text{spec.return } v)))$
 $\langle \text{proof} \rangle$

lemma *return*:
shows $(\text{spec.localize } r \ (\text{spec.return } v) :: ('a \ \text{agent}, 'ls \times 's, 'v) \ \text{spec})$
 $= \text{spec.rel } (\{\text{env}\} \times (r \times_R \text{Id})) \ggg (\lambda :: \text{unit}. \text{spec.return } v)$
 $\langle \text{proof} \rangle$

lemma *rel*:
shows $\text{spec.localize } r \ (\text{spec.rel } s)$
 $= \text{spec.rel } ((\{\text{env}\} \times (r \times_R \text{UNIV}) \cup \text{range proc} \times (\text{Id} \times_R \text{UNIV}))$
 $\quad \cap \text{map-prod id } (\text{map-prod snd } \text{snd}) - ' (s \cup \text{UNIV} \times \text{Id}))$
 $\langle \text{proof} \rangle$

lemma *rel-le*:
shows $\text{spec.localize } \text{Id } P \leq \text{spec.rel } (\text{UNIV} \times (\text{Id} \times_R \text{UNIV}))$
 $\langle \text{proof} \rangle$

lemma *parallel*:
shows $\text{spec.localize } \text{UNIV } (P \parallel Q) = \text{spec.localize } \text{UNIV } P \parallel \text{spec.localize } \text{UNIV } Q$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *localize-le*:
assumes $\text{Id} \subseteq r$
shows $\text{spec.action } (\text{map-prod id } (\text{map-prod id } (\text{map-prod snd } \text{snd})) - ' F \cap \text{UNIV} \times \text{UNIV} \times (\text{Id} \times_R \text{UNIV}))$

$\leq \text{spec.localize } r \text{ (spec.action } F)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma localize:

assumes $P \in \text{spec.interference.closed } (\{\text{env}\} \times \text{UNIV})$

shows $\text{spec.localize UNIV } P \in \text{spec.interference.closed } (\{\text{env}\} \times \text{UNIV})$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma localize:

assumes $\text{Id} \subseteq r$

shows $\text{spec.local } (\text{spec.localize } r P) = P$ (**is** $?lhs = ?rhs$)

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma smap-sndL:

assumes $\text{UNIV} \times (\text{Id} \times_R \text{UNIV}) \subseteq r$

shows $\text{spec.smap snd } f \ggg g = \text{spec.smap snd } (f \ggg (\lambda v. \text{spec.rel } r \sqcap \text{spec.sinvmap snd } (g v)))$ (**is** $?lhs = ?rhs$)

$\langle \text{proof} \rangle$

lemma smap-sndR:

assumes $\text{UNIV} \times (\text{Id} \times_R \text{UNIV}) \subseteq r$

shows $f \ggg (\lambda v. \text{spec.smap snd } (g v)) = \text{spec.smap snd } (\text{spec.rel } r \sqcap \text{spec.sinvmap snd } f \ggg g)$ (**is** $?lhs = ?rhs$)

$\langle \text{proof} \rangle$

lemma localL:

shows $\text{spec.local } f \ggg g = \text{spec.local } (f \ggg (\lambda v. \text{spec.localize Id } (g v)))$

$\langle \text{proof} \rangle$

lemma localR:

shows $f \ggg (\lambda v. \text{spec.local } (g v)) = \text{spec.local } (\text{spec.localize Id } f \ggg g)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma cl-le:

shows $\text{spec.local } (\text{spec.cam.cl } (\{\text{env}\} \times (s \times_R r)) P) \leq \text{spec.cam.cl } (\{\text{env}\} \times r) (\text{spec.local } P)$

$\langle \text{proof} \rangle$

lemma cl:

assumes $\text{Id} \subseteq r_l$

shows $\text{spec.local } (\text{spec.cam.cl } (\{\text{env}\} \times (r_l \times_R r)) P)$

$= \text{spec.cam.cl } (\{\text{env}\} \times r) (\text{spec.local } P)$ (**is** $?lhs = ?rhs$)

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma local:

assumes $\text{Id} \subseteq s$

assumes $P \in \text{spec.cam.closed } (\{\text{env}\} \times (s \times_R r))$

shows $\text{spec.local } P \in \text{spec.cam.closed } (\{\text{env}\} \times r)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *cl-le*:

shows $\text{spec.local } (\text{spec.interference.cl } (\{\text{env}\} \times (s \times_R r)) P)$
 $\leq \text{spec.interference.cl } (\{\text{env}\} \times r) (\text{spec.local } P)$

$\langle \text{proof} \rangle$

lemma *cl*:

assumes $Id \subseteq s$

shows $\text{spec.local } (\text{spec.interference.cl } (\{\text{env}\} \times (s \times_R r)) P)$
 $= \text{spec.interference.cl } (\{\text{env}\} \times r) (\text{spec.local } P)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *local*:

assumes $P \in \text{spec.interference.closed } (\{\text{env}\} \times (Id \times_R r))$

shows $\text{spec.local } P \in \text{spec.interference.closed } (\{\text{env}\} \times r)$

$\langle \text{proof} \rangle$

lemma *local-UNIV*:

assumes $P \in \text{spec.interference.closed } (\{\text{env}\} \times UNIV)$

shows $\text{spec.local } P \in \text{spec.interference.closed } (\{\text{env}\} \times UNIV)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

15.4 *spec.local_init*

$\langle ML \rangle$

definition *local-init* :: $'a \Rightarrow 'ls \Rightarrow ('a \text{ agent}, 'ls \times 's, 'v) \text{ spec} \Rightarrow ('a \text{ agent}, 's, 'v) \text{ spec}$ **where**

$\text{local-init } a \text{ } ls \text{ } P = \text{spec.local } (\text{spec.write } (\text{proc } a) (\text{map-prod } \langle ls \rangle \text{ id}) \gg P)$

$\langle ML \rangle$

lemma *local-init-le-conv*:

shows $\langle \sigma \rangle \leq \text{spec.local-init } a \text{ } ls \text{ } P$

$\longleftrightarrow \langle \sigma \rangle \leq \text{spec.idle} \vee (\exists \sigma'. \langle \sigma' \rangle \leq P$

$\wedge \text{trace.steps } \sigma' \subseteq \text{spec.local.qrm}$

$\wedge \langle \sigma \rangle \leq \langle \text{trace.map id snd id } \sigma' \rangle$

$\wedge \text{fst } (\text{trace.init } \sigma') = ls) \text{ (is ?lhs } \longleftrightarrow \text{ ?rhs)}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *local-init-le[spec.idle-le]*:

shows $\text{spec.idle} \leq \text{spec.local-init } a \text{ } ls \text{ } P$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *Sup*:

shows $\text{spec.local-init } a \text{ } ls \text{ } (\bigsqcup X) = (\bigsqcup x \in X. \text{spec.local-init } a \text{ } ls \text{ } x) \sqcup \text{spec.idle}$

$\langle \text{proof} \rangle$

lemma *Sup-not-empty*:

assumes $X \neq \{\}$

shows $\text{spec.local-init } a \text{ } ls \text{ } (\bigsqcup X) = (\bigsqcup x \in X. \text{spec.local-init } a \text{ } ls \text{ } x)$

$\langle proof \rangle$

lemmas $sup = spec.local-init.Sup-not-empty[\mathbf{where} X=\{X, Y\} \mathbf{for} X Y, simplified]$

lemma *bot*:

shows $spec.local-init a ls \perp = spec.idle$

$\langle proof \rangle$

lemma *top*:

shows $spec.local-init a ls \top = (\top :: ('a agent, 's, 'v) spec)$

$\langle proof \rangle$

lemma *monotone*:

shows $mono (spec.local-init a ls :: ('a agent, 'ls \times 's, 'v) spec \Rightarrow -)$

$\langle proof \rangle$

lemmas $strengthen[strg] = st-monotone[OF spec.local-init.monotone]$

lemmas $mono = monotoneD[OF spec.local-init.monotone]$

lemma *mono2mono[cont-intro, partial-function-mono]*:

assumes $monotone orda (\leq) P$

shows $monotone orda (\leq) (\lambda x. spec.local-init a ls (P x))$

$\langle proof \rangle$

lemma *mcont2mcont[cont-intro]*:

assumes $mcont luba orda Sup (\leq) P$

shows $mcont luba orda Sup (\leq) (\lambda x. spec.local-init a ls (P x))$

$\langle proof \rangle$

lemma *action*:

fixes $F :: ('v \times 'a agent \times ('ls \times 's) \times ('ls \times 's)) set$

shows $spec.local-init a ls (spec.action F)$

$= spec.action \{(v, a, s, s') \mid v a ls' s s'. (v, a, (ls, s), (ls', s')) \in F \wedge (a = env \longrightarrow ls' = ls)\}$ (**is** ?lhs = ?rhs)

$\langle proof \rangle$

lemma *return*:

shows $spec.local-init a ls (spec.return v) = spec.return v$

$\langle proof \rangle$

lemma *localize-le*:

assumes $spec.idle \leq P$

shows $spec.local-init a ls (spec.localize r P) \leq P$

$\langle proof \rangle$

lemma *localize*:

assumes $spec.idle \leq P$

assumes $Id \subseteq r$

shows $spec.local-init a ls (spec.localize r P) = P$ (**is** ?lhs = ?rhs)

$\langle proof \rangle$

lemma *inf-interference*:

shows $spec.local-init a ls P = spec.local-init a ls (P \sqcap spec.local.interference)$

$\langle proof \rangle$

lemma *eq-local*:

assumes $spec.idle \leq P$

shows $(\bigsqcup ls. spec.local-init a ls P) = spec.local P$

$\langle proof \rangle$

lemma *ag-le*:

shows $\text{spec.local-init } a \text{ } ls \ (\{\!\{P\}\!\}, \text{Id} \times_R A \vdash \text{UNIV} \times_R G, \{\!\{\lambda v \ s. Q \ v \ (\text{snd } s)\}\!\})$
 $\leq \{\!\{\lambda s. P \ (ls, s)\}\!\}, A \vdash G, \{\!\{Q\}\!\}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *local-initL*:

shows $\text{spec.local-init } a \text{ } ls \ f \ggg g = \text{spec.local-init } a \text{ } ls \ (f \ggg (\lambda v. \text{spec.localize Id } (g \ v)))$

$\langle \text{proof} \rangle$

lemma *local-initR*:

shows $f \ggg (\lambda v. \text{spec.local-init } a \text{ } ls \ (g \ v)) = \text{spec.local-init } a \text{ } ls \ (\text{spec.localize Id } f \ggg g)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *local-init*:

fixes $P :: ('a \ \text{agent}, 'ls \times 't, 'v) \ \text{spec}$

fixes $sf :: 's \Rightarrow 't$

shows $\text{spec.sinvmap } sf \ (\text{spec.local-init } a \text{ } ls \ P)$

$= \text{spec.local-init } a \text{ } ls \ (\text{spec.rel } (\text{UNIV} \times (\text{Id} \times_R \text{map-prod } sf \ sf - ' \text{Id}))$

$\ggg (\lambda :: \text{unit}. \text{spec.sinvmap } (\text{map-prod id } sf) \ P)) \ (\text{is } ?lhs = ?rhs)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *local-init*:

shows $\text{spec.vmap } vf \ (\text{spec.local-init } a \text{ } ls \ P) = \text{spec.local-init } a \text{ } ls \ (\text{spec.vmap } vf \ P)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *local-init*:

shows $\text{spec.vinvmap } vf \ (\text{spec.local-init } a \text{ } ls \ P) = \text{spec.local-init } a \text{ } ls \ (\text{spec.vinvmap } vf \ P)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *local-init*:

shows $\text{spec.term.none } (\text{spec.local-init } a \text{ } ls \ P) = \text{spec.local-init } a \text{ } ls \ (\text{spec.term.none } P)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *local-init*:

shows $\text{spec.term.all } (\text{spec.local-init } a \text{ } ls \ P)$

$= \text{spec.local-init } a \text{ } ls \ (\text{spec.term.all } P) \sqcup \sqcup \text{range spec.return}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *local-init*:

assumes $P \in \text{spec.interference.closed } (\{\!\{env\}\!\} \times (\text{Id} \times_R r))$

shows $\text{spec.local-init } a \text{ } ls \ P \in \text{spec.interference.closed } (\{\!\{env\}\!\} \times r)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

15.5 Hoist to ('s, 'v) prog

$\langle ML \rangle$

lift-definition $local :: ('ls \times 's, 'v) prog \Rightarrow ('s, 'v) prog$ **is** $spec.local$
 $\langle proof \rangle$

definition $local-init :: 'ls \Rightarrow ('ls \times 's, 'v) prog \Rightarrow ('s, 'v) prog$ **where**
 $local-init\ ls\ P = prog.local\ (prog.write\ (map-prod\ \langle ls \rangle\ id) \gg P)$
— equivalent to lifting $spec.local-init$; see $prog.p2s.local-init$

lift-definition $localize :: ('s, 'v) prog \Rightarrow ('ls \times 's, 'v) prog$ **is** $spec.localize\ UNIV$
 $\langle proof \rangle$

$\langle ML \rangle$

lemmas $local[prog.p2s.simps] = prog.local.rep-eq$

lemma $local-init[prog.p2s.simps]$:
shows $prog.p2s\ (prog.local-init\ ls\ P) = spec.local-init\ ()\ ls\ (prog.p2s\ P)$ (**is** $?lhs = ?rhs$)
 $\langle proof \rangle$

$\langle ML \rangle$

lemma Sup :
shows $prog.local\ (\bigsqcup X) = (\bigsqcup_{x \in X} prog.local\ x)$
 $\langle proof \rangle$

lemmas $sup = prog.local.Sup$ [**where** $X = \{X, Y\}$ **for** $X\ Y$, *simplified*]

lemma bot :
shows $prog.local\ \perp = \perp$
 $\langle proof \rangle$

lemma top :
shows $prog.local\ \top = \top$
 $\langle proof \rangle$

lemma $monotone$:
shows $mono\ prog.local$
 $\langle proof \rangle$

lemmas $strengthen[strg] = st-monotone[OF\ prog.local.monotone]$
lemmas $mono = monotoneD[OF\ prog.local.monotone]$
lemmas $mono2mono[cont-intro, partial-function-mono]$
 $= monotone2monotone[OF\ prog.local.monotone, simplified, of\ orda\ P\ \mathbf{for}\ orda\ P]$

lemma $mcont2mcont[cont-intro]$:
assumes $mcont\ luba\ orda\ Sup\ (\leq)\ P$
shows $mcont\ luba\ orda\ Sup\ (\leq)\ (\lambda x. prog.local\ (P\ x))$
 $\langle proof \rangle$

lemma $bind-botR$:
shows $prog.local\ (P \gg \perp) = prog.local\ P \gg \perp$
 $\langle proof \rangle$

lemma *action*:

shows $prog.local (prog.action F) = prog.action (map-prod id (map-prod snd snd) ' F)$
<proof>

lemma *return*:

shows $prog.local (prog.return v) = prog.return v$
<proof>

<ML>

lemma *transfer[transfer-rule]*:

shows $rel-fun (=) (rel-fun cr-prog cr-prog) (spec.local-init ()) prog.local-init$
<proof>

lemma *Sup*:

shows $prog.local-init ls (\sqcup X) = (\sqcup x \in X. prog.local-init ls x)$
<proof>

lemmas $sup = prog.local-init.Sup$ [where $X = \{X, Y\}$ for $X Y$, *simplified*]

lemma *bot[simp]*:

shows $prog.local-init ls \perp = \perp$
<proof>

lemma *top*:

shows $prog.local-init ls \top = \top$
<proof>

lemma *monotone*:

shows $mono (prog.local-init ls)$
<proof>

lemmas $strengthen[strg] = st-monotone[OF prog.local-init.monotone]$

lemmas $mono = monotoneD[OF prog.local-init.monotone]$

lemma *mono2mono[cont-intro, partial-function-mono]*:

assumes $monotone\ orda (\leq) P$
shows $monotone\ orda (\leq) (\lambda x. prog.local-init ls (P x))$
<proof>

lemma *mcont2mcont[cont-intro]*:

assumes $mcont\ luba\ orda\ Sup (\leq) P$
shows $mcont\ luba\ orda\ Sup (\leq) (\lambda x. prog.local-init ls (P x))$
<proof>

lemma *bind-botR*:

shows $prog.local-init ls (P \gg= \perp) = prog.local-init ls P \gg= \perp$
<proof>

lemma *return*:

shows $prog.local-init ls (prog.return v) = prog.return v$ (**is** ?lhs = ?rhs)
<proof>

lemma *eq-local*:

shows $(\sqcup ls. prog.local-init ls P) = prog.local P$
<proof>

<ML>

lemma *localize-alt-def*:

shows $\text{prog.localize } P = \text{prog.rel } (Id \times_R UNIV) \sqcap \text{prog.sinvmap } \text{snd } P$
<proof>

<ML>

lemma *Sup*:

shows $\text{prog.localize } (\bigsqcup X) = (\bigsqcup x \in X. \text{prog.localize } x)$
<proof>

lemmas $\text{sup} = \text{prog.localize.Sup}[\text{where } X = \{X, Y\} \text{ for } X \ Y, \text{ simplified}]$

lemma *bot*:

shows $\text{prog.localize } \perp = \perp$
<proof>

lemma *top*:

shows $\text{prog.localize } \top = \text{prog.rel } (Id \times_R UNIV)$
<proof>

lemma *monotone*:

shows $\text{mono } \text{prog.localize}$
<proof>

lemmas $\text{strengthen}[strg] = \text{st-monotone}[OF \ \text{prog.localize.monotone}]$

lemmas $\text{mono} = \text{monotoneD}[OF \ \text{prog.localize.monotone}]$

lemmas $\text{mono2mono}[\text{cont-intro}, \text{partial-function-mono}]$
 $= \text{monotone2monotone}[OF \ \text{prog.localize.monotone}, \text{simplified}, \text{of } \text{orda } P \ \text{for } \text{orda } P]$

lemma *mcont2mcont*[*cont-intro*]:

assumes $\text{mcont } \text{luba } \text{orda } \text{Sup } (\leq) \ P$
shows $\text{mcont } \text{luba } \text{orda } \text{Sup } (\leq) \ (\lambda x. \text{prog.localize } (P \ x))$
<proof>

lemmas $\text{p2s}[\text{prog.p2s.simps}] = \text{prog.localize.rep-eq}$

lemma *bind*:

shows $\text{prog.localize } (f \ggg g) = \text{prog.localize } f \ggg (\lambda v. \text{prog.localize } (g \ v))$
<proof>

lemma *parallel*:

shows $\text{prog.localize } (P \parallel Q) = \text{prog.localize } P \parallel \text{prog.localize } Q$
<proof>

lemma *rel*:

fixes $r :: 's \ \text{rel}$
shows $\text{prog.localize } (\text{prog.rel } r) = \text{prog.rel } (Id \times_R r)$
<proof>

lemma *action*:

shows $\text{prog.localize } (\text{prog.action } F)$
 $= \text{prog.action } (\text{map-prod } \text{id } (\text{map-prod } \text{snd } \text{snd}) - ' F \cap UNIV \times (Id \times_R UNIV))$
<proof>

<ML>

lemma *localize*:

fixes $P :: ('s, 'v) \text{ prog}$
shows $\text{prog.local} (\text{prog.localize } P :: ('ls \times 's, 'v) \text{ prog}) = P$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

15.6 Refinement rules

$\langle ML \rangle$

We use *localizeA* to hoist assumes similarly to *spec.localize*.

definition $\text{localizeA} :: (\text{sequential}, 's, 'v) \text{ spec} \Rightarrow (\text{sequential}, 'ls \times 's, 'v) \text{ spec}$ **where**
 $\text{localizeA } P = \text{spec.local.interference} \sqcap \text{spec.sinvmap } \text{snd } P$

$\langle ML \rangle$

lemma *bot*:

shows $\text{spec.localizeA } \perp = \perp$
 $\langle \text{proof} \rangle$

lemma *top*:

shows $\text{spec.localizeA } \top = \text{spec.local.interference}$
 $\langle \text{proof} \rangle$

lemma *ag-assm*:

shows $\text{spec.localizeA} (\text{ag.assm } A) = \text{ag.assm} (Id \times_R A)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *localI*: — Introduce local state

fixes $A :: (\text{sequential}, 's, 'v) \text{ spec}$
fixes $c :: (\text{sequential}, 'ls \times 's, 'v) \text{ spec}$
fixes $c' :: (\text{sequential}, 's, 'v) \text{ spec}$
fixes $P :: 's \text{ pred}$
fixes $Q :: 'v \Rightarrow 's \text{ pred}$
assumes $c \leq \{\!\{ \lambda s. P (\text{snd } s) \}\!\}$, $\text{spec.localizeA } A \Vdash \text{spec.sinvmap } \text{snd } c'$, $\{\!\{ \lambda v s. Q v (\text{snd } s) \}\!\}$
shows $\text{spec.local } c \leq \{\!\{ P \}\!\}$, $A \Vdash c'$, $\{\!\{ Q \}\!\}$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *local-seq-ctxt-cl*:

fixes $A :: (\text{sequential}, 's, 'v) \text{ spec}$
fixes $P :: 's \text{ pred}$
fixes $Q :: 'v \Rightarrow 's \text{ pred}$
fixes $c :: (\text{sequential}, 'ls \times 's, 'v) \text{ spec}$
fixes $c' :: (\text{sequential}, 'ls \times 's, 'v) \text{ spec}$
assumes $\text{spec.local.interference} \sqcap c$
 $\leq \{\!\{ \lambda s. P (\text{snd } s) \}\!\}$, $\text{spec.localizeA } A \Vdash \text{spec.seq-ctxt.cl } \text{False} (\text{spec.local.interference} \sqcap c')$, $\{\!\{ \lambda v s. Q v (\text{snd } s) \}\!\}$
shows $\text{spec.local } c \leq \{\!\{ P \}\!\}$, $A \Vdash \text{spec.local } c'$, $\{\!\{ Q \}\!\}$
 $\langle \text{proof} \rangle$

lemma *cl-bind*:

fixes $f :: ('a \text{ agent}, 'ls \times 's, 'v) \text{ spec}$
fixes $g :: 'v \Rightarrow ('a \text{ agent}, 'ls \times 's, 'w) \text{ spec}$
assumes $g: \bigwedge v. g v \leq \{\!\{ Q' v \}\!\}$, $\text{refinement.spec.bind.res} (\text{spec.pre } P \sqcap \text{spec.term.all } A \sqcap \text{spec.seq-ctxt.cl } \text{True } f')$
 $A v \Vdash \text{spec.seq-ctxt.cl } T (g' v)$, $\{\!\{ Q \}\!\}$

assumes $f: f \leq \{\!\{P\}\!\}$, $\text{spec.term.all } A \Vdash \text{spec.seq-ctxt.cl True } f', \{\!\{Q'\}\!\}$
shows $f \ggg g \leq \{\!\{P\}\!\}$, $A \Vdash \text{spec.seq-ctxt.cl } T (f' \ggg g'), \{\!\{Q\}\!\}$
 $\langle \text{proof} \rangle$

lemma *cl-action-permuteL*:

fixes $F :: ('v \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
fixes $G :: 'v \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
fixes $G' :: ('v' \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
fixes $F' :: 'v' \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
fixes $Q :: 'w \Rightarrow ('ls \times 's) \text{ pred}$
assumes $F: \bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in F \rrbracket \Longrightarrow \text{snd } s' = \text{snd } s$
assumes $FGG'F': \bigwedge v w a a' s s' t. \llbracket P s; (v, a', s, t) \in F; (w, a, t, s') \in G v \rrbracket$
 $\Longrightarrow \exists v' a'' a''' t'. (v', a'', s, t') \in G' \wedge (w, a''', t', s') \in F' v'$
 $\wedge \text{snd } s' = \text{snd } t' \wedge (\text{snd } s \neq \text{snd } t' \longrightarrow a'' = a)$
assumes $Q: \bigwedge v w a a' s s' s''. \llbracket P s; (v, a, s, s') \in G'; (w, a', s', s'') \in F' v \rrbracket \Longrightarrow Q w s''$
shows $\text{spec.action } F \ggg (\lambda v. \text{spec.action } (G v)) \leq \{\!\{P\}\!\}$, $A \Vdash \text{spec.seq-ctxt.cl } T (\text{spec.action } G' \ggg (\lambda v. \text{spec.action } (F' v))), \{\!\{Q\}\!\}$
 $\langle \text{proof} \rangle$

lemma *cl-action-permuteR*:

fixes $G :: ('v \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
fixes $F :: 'v \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
fixes $F' :: ('v' \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
fixes $G' :: 'v' \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
assumes $G: \bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in G; \text{snd } s' \neq \text{snd } s \rrbracket$
 $\Longrightarrow \exists v' w a'' t s''. (v', a'', s, t) \in F' \wedge (w, a, t, s'') \in G' v' \wedge \text{snd } t = \text{snd } s \wedge \text{snd } s'' = \text{snd } s'$
assumes $GFF'G': \bigwedge v w a a' s s' t. \llbracket P s; (v, a, s, t) \in G; (w, a', t, s') \in F v \rrbracket$
 $\Longrightarrow \text{snd } s' = \text{snd } t \wedge (\exists v' a'' a''' t'. (v', a'', s, t') \in F' \wedge (w, a''', t', s') \in G' v'$
 $\wedge \text{snd } t' = \text{snd } s \wedge (\text{snd } s' \neq \text{snd } t' \longrightarrow a''' = a))$
assumes $Q: \bigwedge v w a a' s s' s''. \llbracket P s; (v, a, s, s') \in F'; (w, a', s', s'') \in G' v \rrbracket \Longrightarrow Q w s''$
shows $\text{spec.action } G \ggg (\lambda v. \text{spec.action } (F v)) \leq \{\!\{P\}\!\}$, $A \Vdash \text{spec.seq-ctxt.cl } T (\text{spec.action } F' \ggg (\lambda v. \text{spec.action } (G' v))), \{\!\{Q\}\!\}$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *localI*: — Introduce local state

fixes $A :: (\text{sequential}, 's, 'v) \text{ spec}$
fixes $c :: ('ls \times 's, 'v) \text{ prog}$
fixes $c' :: (\text{sequential}, 's, 'v) \text{ spec}$
fixes $P :: 's \text{ pred}$
fixes $Q :: 'v \Rightarrow 's \text{ pred}$
assumes $\text{prog.p2s } c \leq \{\!\{\lambda s. P (\text{snd } s)\}\!\}$, $\text{spec.localizeA } A \Vdash \text{spec.sinvmap snd } c', \{\!\{\lambda v s. Q v (\text{snd } s)\}\!\}$
shows $\text{prog.p2s } (\text{prog.local } c) \leq \{\!\{P\}\!\}$, $A \Vdash c', \{\!\{Q\}\!\}$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

15.6.1 Data refinement

In this setting a (concrete) specification c is a *data refinement* of (abstract) specification c' if:

- the observable state changes coincide
- concrete local states are mapped to abstract local states by sf which then coincide

Observations:

- pre/post are in terms of the concrete local states

- sf can be used to lift these to the abstract local states
- we do not require c or c' to disallow the environment from changing the local state
- essentially a Skolemization of Lamport’s existentials (Lamport 1994, §8)

References:

- de Roever and Engelhardt (1998, Chapter 14 “Refinement Methods due to Abadi and Lamport and to Lynch”)
- in general c will need to be augmented with auxiliary variables

$\langle ML \rangle$

lemma data:

fixes $A :: (sequential, 's, 'v) spec$

fixes $c :: (sequential, 'cls \times 's, 'v) spec$

fixes $c' :: (sequential, 'als \times 's, 'v) spec$

fixes $sf :: 'cls \Rightarrow 'als$

assumes $c \leq \{\!\{ \lambda s. P (snd s) \}\!\}, spec.localizeA A \Vdash spec.sinvmap (map-prod sf id) c', \{\!\{ \lambda v s. Q v (snd s) \}\!\}$

shows $spec.local c \leq \{\!\{ P \}\!\}, A \Vdash spec.local c', \{\!\{ Q \}\!\}$

$\langle proof \rangle$

$\langle ML \rangle$

lemma data:

fixes $A :: (sequential, 's, 'v) spec$

fixes $c :: ('cls \times 's, 'v) prog$

fixes $c' :: ('als \times 's, 'v) prog$

fixes $sf :: 'cls \Rightarrow 'als$

assumes $prog.p2s c \leq \{\!\{ \lambda s. P (snd s) \}\!\}, spec.localizeA A \Vdash spec.sinvmap (map-prod sf id) (prog.p2s c'), \{\!\{ \lambda v s. Q v (snd s) \}\!\}$

shows $prog.p2s (prog.local c) \leq \{\!\{ P \}\!\}, A \Vdash prog.p2s (prog.local c'), \{\!\{ Q \}\!\}$

$\langle proof \rangle$

$\langle ML \rangle$

15.7 Assume/guarantee

$\langle ML \rangle$

lemma local:

fixes $A G :: 's rel$

fixes $P :: 's pred$

fixes $Q :: 'v \Rightarrow 's pred$

fixes $c :: (sequential, 'ls \times 's, 'v) spec$

assumes $c \leq \{\!\{ \lambda s. P (snd s) \}\!\}, Id \times_R A \vdash UNIV \times_R G, \{\!\{ \lambda v s. Q v (snd s) \}\!\}$

shows $spec.local c \leq \{\!\{ P \}\!\}, A \vdash G, \{\!\{ Q \}\!\}$

$\langle proof \rangle$

lemma localize-lift:

fixes $A G :: 's rel$

fixes $P :: 's \Rightarrow bool$

fixes $Q :: 'v \Rightarrow 's \Rightarrow bool$

fixes $c :: (sequential, 's, 'v) spec$

notes $inf.bounded-iff[simp del]$

assumes $c: c \leq \{\!\{ P \}\!\}, A \vdash G, \{\!\{ Q \}\!\}$

shows $spec.localize UNIV c \leq \{\!\{ \lambda s. P (snd s) \}\!\}, UNIV \times_R A \vdash Id \times_R G, \{\!\{ \lambda v s: 'ls \times 's. Q v (snd s) \}\!\}$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *local*:

fixes $A\ G :: 's\ rel$

fixes $P :: 's\ pred$

fixes $Q :: 'v \Rightarrow 's\ pred$

fixes $c :: ('ls \times 's, 'v)\ prog$

assumes $prog.p2s\ c \leq \{\!\{ \lambda s. P\ (snd\ s) \}\!\}, Id \times_R A \vdash UNIV \times_R G, \{\!\{ \lambda v\ s. Q\ v\ (snd\ s) \}\!\}$

shows $prog.p2s\ (prog.local\ c) \leq \{\!\{ P \}\!\}, A \vdash G, \{\!\{ Q \}\!\}$

$\langle proof \rangle$

lemma *localize-lift*:

fixes $A\ G :: 's\ rel$

fixes $P :: 's \Rightarrow bool$

fixes $Q :: 'v \Rightarrow 's \Rightarrow bool$

fixes $c :: ('s, 'v)\ prog$

assumes $prog.p2s\ c \leq \{\!\{ P \}\!\}, A \vdash G, \{\!\{ Q \}\!\}$

shows $prog.p2s\ (prog.localize\ c) \leq \{\!\{ \lambda s. P\ (snd\ s) \}\!\}, UNIV \times_R A \vdash Id \times_R G, \{\!\{ \lambda v\ s. Q\ v\ (snd\ s) \}\!\}$

$\langle proof \rangle$

$\langle ML \rangle$

15.8 Specification inhabitation

$\langle ML \rangle$

lemma *localize*:

assumes $P\ -s, xs \rightarrow P'$

assumes $Id \subseteq r$

shows $spec.localize\ r\ P\ -(ls, s), map\ (map-prod\ id\ (Pair\ ls))\ xs \rightarrow spec.localize\ r\ P'$

$\langle proof \rangle$

lemma *local*:

assumes $P\ -(ls, s), xs \rightarrow spec.return\ v$

assumes $trace.steps'\ (ls, s)\ xs \subseteq spec.local.qrm$

shows $spec.local\ P\ -s, map\ (map-prod\ id\ snd)\ xs \rightarrow spec.return\ v$

$\langle proof \rangle$

lemma *local-init*:

assumes $P\ -(ls, s), xs \rightarrow P'$

assumes $trace.steps'\ (ls, s)\ xs \subseteq spec.local.qrm$

shows $spec.local-init\ a\ ls\ P\ -s, map\ (map-prod\ id\ snd)\ xs \rightarrow spec.local-init\ a\ (fst\ (trace.final'\ (ls, s)\ xs))\ P'$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *localize*:

assumes $prog.p2s\ P\ -s, xs \rightarrow prog.p2s\ P'$

shows $prog.p2s\ (prog.localize\ P)\ -(ls, s), map\ (map-prod\ id\ (Pair\ ls))\ xs \rightarrow prog.p2s\ (prog.localize\ P')$

$\langle proof \rangle$

lemma *local*:

assumes $prog.p2s\ P\ -(ls, s), xs \rightarrow spec.return\ v$

assumes $trace.steps'\ (ls, s)\ xs \subseteq spec.local.qrm$

shows $prog.p2s\ (prog.local\ P)\ -s, map\ (map-prod\ id\ snd)\ xs \rightarrow spec.return\ v$

$\langle proof \rangle$

lemma *local-init*:

assumes $\text{prog.p2s } P \text{ } \neg(ls, s), xs \rightarrow \text{prog.p2s } P'$
assumes $\text{trace.steps}' (ls, s) xs \subseteq \text{spec.local.qrm}$
shows $\text{prog.p2s } (\text{prog.local-init } ls P) \text{ } \neg s, \text{map } (\text{map-prod id snd}) xs \rightarrow \text{prog.p2s } (\text{prog.local-init } (\text{fst } (\text{trace.final}' (ls, s) xs)) P')$
<proof>

<ML>

16 A Temporal Logic of Safety (TLS)

We model systems with finite and infinite sequences of states, closed under stuttering following Lamport (1994). This theory relates the safety logic of §8 to the powerset (quotiented by stuttering) representing properties of these sequences (see §16.6). Most of this story is standard but the addition of finite sequences does have some impact.

References:

- historical motivations for future-time linear temporal logic (LTL): Manna and Pnueli (1991); Owicki and Lamport (1982).
- a discussion on the merits of proving liveness: <https://cs.nyu.edu/acsys/beyond-safety/liveness.htm>

Observations:

- Lamport (and Abadi et al) treat infinite stuttering as termination
 - Lamport (2000, p189): “we can represent a terminating execution of any system by an infinite behavior that ends with a sequence of nothing but stuttering steps. We have no need of finite behaviors (finite sequences of states), so we consider only infinite ones.”
 - this conflates divergence with termination
 - we separate those concepts here so we can support sequential composition
- the traditional account of liveness properties breaks down (see §24)

16.1 Stuttering

An infinitary version of *trace.natural'*.

Observations:

- we need to normalize the agent labels for sequences that infinitely stutter

Source materials:

- \$ISABELLE_HOME/src/HOL/Corec_Examples/LFilter.thy.
- \$AFP/Coinductive/Coinductive_List.thy
- \$AFP/Coinductive/TLList.thy
- \$AFP/TLA/Sequence.thy.

definition *trailing* :: $'c \Rightarrow ('a, 'b) \text{tlist} \Rightarrow ('c, 'b) \text{tlist}$ **where**
trailing $s xs = (\text{if } \text{tfinite } xs \text{ then } TNil \text{ (terminal } xs) \text{ else } \text{trepeat } s)$

corecursive *collapse* :: $'s \Rightarrow ('a \times 's, 'v) \text{tlist} \Rightarrow ('a \times 's, 'v) \text{tlist}$ **where**
collapse $s xs = (\text{if } \text{snd } ' \text{tset } xs \subseteq \{s\} \text{ then } \text{trailing } (\text{undefined}, s) xs$

else if $\text{snd } (\text{thd } xs) = s$ then $\text{collapse } s$ ($\text{ttl } xs$)
 else $\text{TCons } (\text{thd } xs) (\text{collapse } (\text{snd } (\text{thd } xs)) (\text{ttl } xs))$)

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *trailing*:

shows $\text{tmap } sf \ vf$ ($\text{trailing } s \ xs$) = $\text{trailing } (sf \ s)$ ($\text{tmap } sf \ vf \ xs$)

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *trailing*:

shows $\text{tlength } (\text{trailing } s \ xs) \leq \text{tlength } xs$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *simps[simp]*:

shows *TNil*: $\text{trailing } s$ (*TNil* b) = *TNil* b

and *TCons*: $\text{trailing } s$ (*TCons* $x \ xs$) = $\text{trailing } s \ xs$

and *ttl*: $\text{ttl } (\text{trailing } s \ xs) = \text{trailing } s \ xs$

and *idempotent*: $\text{trailing } s$ ($\text{trailing } s \ xs$) = $\text{trailing } s \ xs$

and *tset-finite*: $\text{tset } (\text{trailing } s \ xs) = (\text{if } \text{tfinite } xs \ \text{then } \{\} \ \text{else } \{s\})$

and *trepeat*: $\text{trailing } s$ ($\text{trepeat } s$) = $\text{trepeat } s$

$\langle \text{proof} \rangle$

lemma *eq-TNil-conv*:

shows $\text{trailing } s \ xs = \text{TNil } b \iff \text{tfinite } xs \wedge \text{terminal } xs = b$

and $\text{TNil } b = \text{trailing } s \ xs \iff \text{tfinite } xs \wedge \text{terminal } xs = b$

and *is-TNil* ($\text{trailing } s \ xs$) $\iff \text{tfinite } xs$

$\langle \text{proof} \rangle$

lemma *eq-TCons-conv*:

shows $\text{trailing } s \ xs = \text{TCons } y \ ys \iff \neg \text{tfinite } xs \wedge \text{TCons } y \ ys = \text{trepeat } s$

and $\text{TCons } y \ ys = \text{trailing } s \ xs \iff \neg \text{tfinite } xs \wedge \text{TCons } y \ ys = \text{trepeat } s$

$\langle \text{proof} \rangle$

lemma *tmap*:

shows $\text{trailing } s$ ($\text{tmap } sf \ vf \ xs$) = $\text{tmap } id \ vf$ ($\text{trailing } s \ xs$)

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *unique*:

assumes $\bigwedge s \ xs. f \ s \ xs = (\text{if } \text{snd } ' \ \text{tset } xs \subseteq \{s\} \ \text{then } \text{trailing } (\text{undefined}, s) \ xs$

else if $\text{snd } (\text{thd } xs) = s$ then $f \ s$ ($\text{ttl } xs$)

else $\text{TCons } (\text{thd } xs) (f \ (\text{snd } (\text{thd } xs)) (\text{ttl } xs))$)

shows $f = \text{collapse}$

$\langle \text{proof} \rangle$

lemma *collapse*:

shows $\text{collapse } s$ ($\text{collapse } s \ xs$) = $\text{collapse } s \ xs$

$\langle \text{proof} \rangle$

lemma *simps[simp]*:

shows *TNil*: $\text{collapse } s$ (*TNil* b) = *TNil* b

and *TCons*: $\text{collapse } s$ (*TCons* $x \ xs$) = ($\text{if } \text{snd } x = s$ then $\text{collapse } s \ xs$ else $\text{TCons } x$ ($\text{collapse } (\text{snd } x) \ xs$))

and trailing: $\text{collapse } s \text{ (trailing (undefined, } s) \text{ } xs) = \text{trailing (undefined, } s) \text{ } xs$
 ⟨proof⟩

lemma tshift-stuttering:

assumes $\text{snd ' set } xs \subseteq \{s\}$

shows $\text{collapse } s \text{ (tshift } xs \text{ } ys) = \text{collapse } s \text{ } ys$

⟨proof⟩

lemma infinite-trailing:

assumes $\neg \text{tfinite } xs$

assumes $\text{snd ' tset } xs \subseteq \{s'\}$

shows $\text{collapse } s \text{ } xs = (\text{if } s = s' \text{ then } \text{trepeat (undefined, } s') \text{ else } \text{TCons (thd } xs) \text{ (trepeat (undefined, } s')))$

⟨proof⟩

lemma eq-TNil-conv:

shows $\text{collapse } s \text{ } xs = \text{TNil } b \longleftrightarrow \text{tfinite } xs \wedge \text{snd ' tset } xs \subseteq \{s\} \wedge \text{terminal } xs = b$ (**is** $?lhs \longleftrightarrow ?rhs$)

and $\text{TNil } b = \text{collapse } s \text{ } xs \longleftrightarrow \text{tfinite } xs \wedge \text{snd ' tset } xs \subseteq \{s\} \wedge \text{terminal } xs = b$ (**is** $?thesis1$)

⟨proof⟩

lemma is-TNil-conv:

shows $\text{is-TNil (collapse } s \text{ } xs) \longleftrightarrow \text{tfinite } xs \wedge \text{snd ' tset } xs \subseteq \{s\}$ (**is** $?thesis2$)

⟨proof⟩

lemma eq-TConsE:

assumes $\text{collapse } s \text{ } xs = \text{TCons } y \text{ } ys$

obtains

$(\text{trailing-stuttering}) \neg \text{tfinite } xs$

and $\text{snd ' tset } xs = \{s\}$

and $\text{TCons } y \text{ } ys = \text{trepeat (undefined, } s)$

| $(\text{step}) \text{ us } \text{ys}' \text{ where } xs = \text{tshift us (TCons } y \text{ } \text{ys}'$

and $\text{snd ' set } us \subseteq \{s\}$

and $\text{snd } y \neq s$

and $\text{collapse (snd } y) \text{ } \text{ys}' = \text{ys}$

⟨proof⟩

lemma eq-TCons-conv:

shows $\text{collapse } s \text{ } xs = \text{TCons } y \text{ } ys$

$\longleftrightarrow (\neg \text{tfinite } xs \wedge \text{snd ' tset } xs = \{s\} \wedge \text{TCons } y \text{ } ys = \text{trepeat (undefined, } s))$

$\vee (\exists xs' \text{ } \text{ys}'. xs = \text{tshift } xs' \text{ (TCons } y \text{ } \text{ys}') \wedge \text{snd ' set } xs' \subseteq \{s\} \wedge \text{snd } y \neq s \wedge \text{collapse (snd } y) \text{ } \text{ys}' = \text{ys})$ (**is** $?lhs \longleftrightarrow ?rhs$)

and $\text{TCons } y \text{ } ys = \text{collapse } s \text{ } xs$

$\longleftrightarrow (\neg \text{tfinite } xs \wedge \text{snd ' tset } xs = \{s\} \wedge \text{TCons } y \text{ } ys = \text{trepeat (undefined, } s))$

$\vee (\exists xs' \text{ } \text{ys}'. xs = \text{tshift } xs' \text{ (TCons } y \text{ } \text{ys}') \wedge \text{snd ' set } xs' \subseteq \{s\} \wedge \text{snd } y \neq s \wedge \text{collapse (snd } y) \text{ } \text{ys}' = \text{ys})$ (**is** $?thesis1$)

⟨proof⟩

lemma tfinite:

shows $\text{tfinite (collapse } s \text{ } xs) \longleftrightarrow \text{tfinite } xs$ (**is** $?lhs \longleftrightarrow ?rhs$)

⟨proof⟩

lemma tfinite-conv:

assumes $\text{collapse } s \text{ } xs = \text{collapse } s' \text{ } \text{xs}'$

shows $\text{tfinite } xs \longleftrightarrow \text{tfinite } \text{xs}'$

⟨proof⟩

lemma terminal:

shows $\text{terminal (collapse } s \text{ } xs) = \text{terminal } xs$

⟨proof⟩

lemma *tlength*:

shows $tlength (collapse\ s\ xs) \leq tlength\ xs$
<proof>

lemma *tset-memberD*:

assumes $(a, s') \in tset (collapse\ s\ xs)$
shows $s' \in snd\ 'tset\ xs$
<proof>

lemma *tset-memberD2*:

assumes $(a, s') \in tset\ xs$
shows $s = s' \vee s' \in snd\ 'tset\ (collapse\ s\ xs)$
<proof>

lemma *tshift*:

shows $collapse\ s\ (tshift\ xs\ ys) = tshift\ (trace.natural'\ s\ xs)\ (collapse\ (trace.final'\ s\ xs)\ ys)$
<proof>

lemma *trepeat*:

shows $collapse\ s\ (trepeat\ (a, s)) = trepeat\ (undefined, s)$
<proof>

lemma *eq-trepeat-conv*:

shows $trepeat\ (undefined, s) = collapse\ s\ xs \longleftrightarrow \neg tfinite\ xs \wedge snd\ 'tset\ xs = \{s\}$ (**is** *?thesis1*)
and $collapse\ s\ xs = trepeat\ (undefined, s) \longleftrightarrow \neg tfinite\ xs \wedge snd\ 'tset\ xs = \{s\}$ (**is** *?thesis2*)
<proof>

lemma *trePLICATE*:

shows $collapse\ s\ (trePLICATE\ i\ (a, s)\ v) = TNil\ v$
<proof>

lemma *eq-tshift-conv*:

shows $collapse\ s\ xs = tshift\ ys\ zs$
 $\longleftrightarrow (\exists xs'\ xs''\ ys'.\ tshift\ xs'\ xs'' = xs \wedge trace.natural'\ s\ xs' @ ys' = ys$
 $\wedge ((\neg tfinite\ xs'' \wedge snd\ 'tset\ xs'' = \{trace.final'\ s\ xs'\}) \wedge tshift\ ys'\ zs = trepeat\ (undefined, trace.final'\ s$
 $xs'))$
 $\vee (ys' = [] \wedge collapse\ (trace.final'\ s\ xs')\ xs'' = zs))$ (**is** *?lhs* \longleftrightarrow *?rhs*)
and $tshift\ ys\ zs = collapse\ s\ xs$
 $\longleftrightarrow (\exists xs'\ xs''\ ys'.\ tshift\ xs'\ xs'' = xs \wedge trace.natural'\ s\ xs' @ ys' = ys$
 $\wedge ((\neg tfinite\ xs'' \wedge snd\ 'tset\ xs'' = \{trace.final'\ s\ xs'\}) \wedge tshift\ ys'\ zs = trepeat\ (undefined, trace.final'\ s$
 $xs'))$
 $\vee (ys' = [] \wedge collapse\ (trace.final'\ s\ xs')\ xs'' = zs))$ (**is** *?thesis1*)
<proof>

lemma *eq-collapse-ttake-dropn-conv*:

shows $collapse\ s\ xs = collapse\ s\ ys$
 $\longleftrightarrow (\exists j.\ trace.natural'\ s\ (fst\ (ttake\ i\ xs)) = trace.natural'\ s\ (fst\ (ttake\ j\ ys))$
 $\wedge snd\ (ttake\ i\ xs) = snd\ (ttake\ j\ ys)$
 $\wedge collapse\ (trace.final'\ s\ (fst\ (ttake\ i\ xs)))\ (tdropn\ i\ xs)$
 $= collapse\ (trace.final'\ s\ (fst\ (ttake\ i\ xs)))\ (tdropn\ j\ ys))$ (**is** *?lhs* \longleftrightarrow $(\exists j.\ ?rhs\ i\ j\ s\ xs\ ys)$)
<proof>

lemmas *eq-collapse-ttake-dropnE* = *exE*[*OF* *iffD1*[*OF* *collapse.eq-collapse-ttake-dropn-conv*]]

lemma *tshift-tdropn*:

assumes $trace.natural'\ s\ (fst\ (ttake\ i\ xs)) = trace.natural'\ s\ ys$
shows $collapse\ s\ (tshift\ ys\ (tdropn\ i\ xs)) = collapse\ s\ xs$

$\langle proof \rangle$

lemma *map-collapse*:

shows $collapse\ (sf\ s)\ (tmap\ (map\text{-}prod\ af\ sf)\ vf\ (collapse\ s\ xs))$
 $=\ collapse\ (sf\ s)\ (tmap\ (map\text{-}prod\ af\ sf)\ vf\ xs)\ (\mathbf{is}\ ?lhs\ s\ xs = ?rhs\ s\ xs)$

$\langle proof \rangle$

$\langle ML \rangle$

definition *natural* $:: ('a, 's, 'v)\ behavior.t \Rightarrow ('a, 's, 'v)\ behavior.t\ (\Downarrow_T)$ **where**

$\Downarrow_T \omega = behavior.B\ (behavior.init\ \omega)\ (collapse\ (behavior.init\ \omega)\ (behavior.rest\ \omega))$

$\langle ML \rangle$

lemma *collapse[simp]*:

shows $behavior.sset\ (behavior.B\ s\ (collapse\ s\ xs)) = behavior.sset\ (behavior.B\ s\ xs)$

$\langle proof \rangle$

lemma *natural[simp]*:

shows $behavior.sset\ (\Downarrow_T \omega) = behavior.sset\ \omega$

$\langle proof \rangle$

lemma *continue*:

shows $behavior.sset\ (\sigma @_{-B} xs) = trace.sset\ \sigma \cup (case\ trace.term\ \sigma\ of\ None \Rightarrow snd\ 'tset\ xs\ |\ Some\ - \Rightarrow \{\})$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *sel[simp]*:

shows $behavior.init\ (\Downarrow_T \omega) = behavior.init\ \omega$

and $behavior.rest\ (\Downarrow_T \omega) = collapse\ (behavior.init\ \omega)\ (behavior.rest\ \omega)$

$\langle proof \rangle$

lemma *TNil*:

shows $\Downarrow_T (behavior.B\ s\ (TNil\ v)) = behavior.B\ s\ (TNil\ v)$

$\langle proof \rangle$

lemma *tfinite*:

shows $tfinite\ (behavior.rest\ (\Downarrow_T \omega)) \longleftrightarrow tfinite\ (behavior.rest\ \omega)$

$\langle proof \rangle$

lemma *continue*:

shows $\Downarrow_T (\sigma @_{-B} xs) = \Downarrow_T \sigma @_{-B} (collapse\ (trace.final\ \sigma)\ xs)$

$\langle proof \rangle$

lemma *tshift*:

shows $\Downarrow_T (behavior.B\ s\ (tshift\ as\ xs)) = behavior.B\ s\ (collapse\ s\ (tshift\ as\ xs))$

$\langle proof \rangle$

lemma *trepeat*:

shows $\Downarrow_T (behavior.B\ s\ (trepeat\ (a, s))) = behavior.B\ s\ (trepeat\ (undefined, s))$

$\langle proof \rangle$

lemma *trePLICATE*:

shows $\Downarrow_T (behavior.B\ s\ (trePLICATE\ i\ (a, s)\ v)) = behavior.B\ s\ (TNil\ v)$

$\langle proof \rangle$

lemma *map-natural*:

shows $\Downarrow_T(\text{behavior.map af sf vf } (\Downarrow_T \omega)) = \Downarrow_T(\text{behavior.map af sf vf } \omega)$
 ⟨proof⟩

lemma idle:

assumes $\text{behavior.sset } \omega \subseteq \{\text{behavior.init } \omega\}$

shows $\Downarrow_T \omega = \text{behavior.B } (\text{behavior.init } \omega) (\text{trailing } (\text{undefined}, \text{behavior.init } \omega) (\text{behavior.rest } \omega))$
 ⟨proof⟩

⟨ML⟩

interpretation stuttering: *galois.image-vimage-idempotent* \Downarrow_T

⟨proof⟩

⟨ML⟩

abbreviation $\text{syn} :: ('a, 's, 'v) \text{behavior.t} \Rightarrow ('a, 's, 'v) \text{behavior.t} \Rightarrow \text{bool}$ (**infix** \simeq_T 50) **where**

$\omega_1 \simeq_T \omega_2 \equiv \text{behavior.stuttering.equivalent } \omega_1 \omega_2$

lemma map:

assumes $\omega_1 \simeq_T \omega_2$

shows $\text{behavior.map af sf vf } \omega_1 \simeq_T \text{behavior.map af sf vf } \omega_2$
 ⟨proof⟩

lemma takeE:

assumes $\omega_1 \simeq_T \omega_2$

obtains j **where** $\text{behavior.take } i \omega_1 \simeq_S \text{behavior.take } j \omega_2$
 ⟨proof⟩

lemma idle-dropn:

assumes $\text{behavior.dropn } i \omega = \text{Some } \omega'$

assumes $\text{behavior.sset } \omega \subseteq \{\text{behavior.init } \omega\}$

shows $\omega \simeq_T \omega'$
 ⟨proof⟩

⟨ML⟩

lemma takeE:

fixes $\sigma :: ('a, 's, 'v) \text{trace.t}$

assumes $\text{behavior.take } i \omega \simeq_S \sigma$

obtains $\omega' j$ **where** $\omega \simeq_T \omega'$ **and** $\sigma = \text{behavior.take } j \omega'$
 ⟨proof⟩

lemmas $\text{rev-takeE} = \text{trace.stuttering.equiv.behavior.takeE}[OF \text{sym}]$

⟨ML⟩

lemma takeE:

fixes $\omega :: ('a, 's, 'v) \text{behavior.t}$

obtains j **where** $\Downarrow(\text{behavior.take } i \omega) = \text{behavior.take } j (\Downarrow_T \omega)$
 ⟨proof⟩

⟨ML⟩

16.2 The $('a, 's, 'v)$ tls lattice

This is our version of Lamport's TLA lattice which we treat in a "semantic" way similarly to [Abadi and Merz \(1996\)](#).

Observations:

- there is a somewhat natural partial ordering on the *tls* lattice induced by the connection with the *spec* lattice (see §16.6 and §24) which we do not use

typedef (*'a*, *'s*, *'v*) *tls* = *behavior.stuttering.closed* :: (*'a*, *'s*, *'v*) *behavior.t set set*
morphisms *unTLS TLS*
<proof>

setup-lifting *type-definition-tls*

instantiation *tls* :: (*type, type, type*) *complete-boolean-algebra*
begin

lift-definition *bot-tls* :: (*'a*, *'s*, *'v*) *tls* **is** *empty* *<proof>*
lift-definition *top-tls* :: (*'a*, *'s*, *'v*) *tls* **is** *UNIV* *<proof>*
lift-definition *sup-tls* :: (*'a*, *'s*, *'v*) *tls* \Rightarrow (*'a*, *'s*, *'v*) *tls* \Rightarrow (*'a*, *'s*, *'v*) *tls* **is** *sup* *<proof>*
lift-definition *inf-tls* :: (*'a*, *'s*, *'v*) *tls* \Rightarrow (*'a*, *'s*, *'v*) *tls* \Rightarrow (*'a*, *'s*, *'v*) *tls* **is** *inf* *<proof>*
lift-definition *less-eq-tls* :: (*'a*, *'s*, *'v*) *tls* \Rightarrow (*'a*, *'s*, *'v*) *tls* \Rightarrow *bool* **is** *less-eq* *<proof>*
lift-definition *less-tls* :: (*'a*, *'s*, *'v*) *tls* \Rightarrow (*'a*, *'s*, *'v*) *tls* \Rightarrow *bool* **is** *less* *<proof>*
lift-definition *Inf-tls* :: (*'a*, *'s*, *'v*) *tls set* \Rightarrow (*'a*, *'s*, *'v*) *tls* **is** *Inf* *<proof>*
lift-definition *Sup-tls* :: (*'a*, *'s*, *'v*) *tls set* \Rightarrow (*'a*, *'s*, *'v*) *tls* **is** $\lambda X. Sup X \sqcup behavior.stuttering.cl \{ \}$ *<proof>*
lift-definition *minus-tls* :: (*'a*, *'s*, *'v*) *tls* \Rightarrow (*'a*, *'s*, *'v*) *tls* \Rightarrow (*'a*, *'s*, *'v*) *tls* **is** *minus* *<proof>*
lift-definition *uminus-tls* :: (*'a*, *'s*, *'v*) *tls* \Rightarrow (*'a*, *'s*, *'v*) *tls* **is** *uminus* *<proof>*

instance
<proof>

end

declare

SUPE[**where** *'a*=(*'a*, *'s*, *'v*) *tls*, *intro!*]
SupE[**where** *'a*=(*'a*, *'s*, *'v*) *tls*, *intro!*]
Sup-le-iff[**where** *'a*=(*'a*, *'s*, *'v*) *tls*, *simp*]
SupI[**where** *'a*=(*'a*, *'s*, *'v*) *tls*, *intro*]
SUPI[**where** *'a*=(*'a*, *'s*, *'v*) *tls*, *intro*]
rev-SUPI[**where** *'a*=(*'a*, *'s*, *'v*) *tls*, *intro?*]
INFE[**where** *'a*=(*'a*, *'s*, *'v*) *tls*, *intro*]

<ML>

lemma *boolean-implication-transfer*[*transfer-rule*]:

shows *rel-fun* (*pcr-tls* (=) (=) (=)) (*rel-fun* (*pcr-tls* (=) (=) (=)) (*pcr-tls* (=) (=) (=))) (\longrightarrow_B) (\longrightarrow_B)
<proof>

lemma *bot-not-top*:

shows $\perp \neq (\top :: ('a, 's, 'v) \text{tls})$
<proof>

<ML>

16.3 Irreducible elements

<ML>

definition *singleton* :: (*'a*, *'s*, *'v*) *behavior.t* \Rightarrow (*'a*, *'s*, *'v*) *behavior.t set* **where**
singleton $\omega = behavior.stuttering.cl \{ \omega \}$

lemma *singleton-le-conv*:

shows *raw.singleton* $\sigma_1 \leq raw.singleton \sigma_2 \iff \Downarrow_T \sigma_1 = \Downarrow_T \sigma_2$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lift-definition *singleton* :: ('a, 's, 'v) behavior.t \Rightarrow ('a, 's, 'v) tls ($\langle - \rangle_T [0]$) **is** raw.singleton

$\langle \text{proof} \rangle$

abbreviation *singleton-behavior-syn* :: 's \Rightarrow ('a \times 's, 'v) tllist \Rightarrow ('a, 's, 'v) tls ($\langle -, - \rangle_T [0, 0]$) **where**
 $\langle s, xs \rangle_T \equiv \langle \text{behavior}.B\ s\ xs \rangle_T$

$\langle ML \rangle$

lemma *Sup-prime*:

shows *Sup-prime* $\langle \omega \rangle_T$

$\langle \text{proof} \rangle$

lemma *nchotomy*:

shows $\exists X \in \text{behavior.stuttering.closed}. x = \bigsqcup (\text{tls.singleton } ' X)$

$\langle \text{proof} \rangle$

lemmas *exhaust* = *bexE[OF tls.singleton.nchotomy]*

lemma *collapse[simp]*:

shows $\bigsqcup (\text{tls.singleton } ' \{\omega. \langle \omega \rangle_T \leq P\}) = P$

$\langle \text{proof} \rangle$

lemmas *not-bot* = *Sup-prime-not-bot[OF tls.singleton.Sup-prime]* — Non-triviality

$\langle ML \rangle$

lemma *singleton-le-ext-conv*:

shows $P \leq Q \iff (\forall \omega. \langle \omega \rangle_T \leq P \longrightarrow \langle \omega \rangle_T \leq Q)$ (**is** ?lhs \iff ?rhs)

$\langle \text{proof} \rangle$

lemmas *singleton-le-conv* = *raw.singleton-le-conv[transferred]*

lemmas *singleton-le-extI* = *iffD2[OF tls.singleton-le-ext-conv, rule-format]*

lemma *singleton-eq-conv[simp]*:

shows $\langle \omega \rangle_T = \langle \omega' \rangle_T \iff \omega \simeq_T \omega'$

$\langle \text{proof} \rangle$

lemma *singleton-cong*:

assumes $\omega \simeq_T \omega'$

shows $\langle \omega \rangle_T = \langle \omega' \rangle_T$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

named-theorems *le-conv* \langle *simplification rules for* $\langle \langle \sigma \rangle_T \leq \text{const } \dots \rangle$ \rangle

lemma *boolean-implication-le-conv[tls.singleton.le-conv]*:

shows $\langle \sigma \rangle_T \leq P \longrightarrow_B Q \iff (\langle \sigma \rangle_T \leq P \longrightarrow \langle \sigma \rangle_T \leq Q)$

$\langle \text{proof} \rangle$

lemmas *antisym* = *antisym[OF tls.singleton-le-extI tls.singleton-le-extI]*

lemmas *top* = *tls.singleton.collapse[of \top , simplified, symmetric]*

lemma *simps*[*simp*]:

shows $\langle \Downarrow_T \omega \rangle_T = \langle \omega \rangle_T$

and $\langle s, xs \rangle_T \leq \langle s, \text{collapse } s \ xs \rangle_T$

and $\text{snd } \text{'set } ys \subseteq \{s\} \implies \langle s, \text{tshift } ys \ xs \rangle_T = \langle s, xs \rangle_T$

and $\langle s, \text{TCons } (a, s) \ xs \rangle_T = \langle s, xs \rangle_T$

<proof>

lemmas *Sup-irreducible* = *iffD1*[*OF heyting.Sup-prime-Sup-irreducible-iff tls.singleton.Sup-prime*]

lemmas *sup-irreducible* = *Sup-irreducible-on-imp-sup-irreducible-on*[*OF tls.singleton.Sup-irreducible, simplified*]

lemmas *Sup-leE*[*elim*] = *Sup-prime-onE*[*OF tls.singleton.Sup-prime, simplified*]

lemmas *sup-le-conv*[*simp*] = *sup-irreducible-le-conv*[*OF tls.singleton.sup-irreducible*]

lemmas *Sup-le-conv*[*simp*] = *Sup-prime-on-conv*[*OF tls.singleton.Sup-prime, simplified*]

lemmas *compact-point* = *Sup-prime-is-compact*[*OF tls.singleton.Sup-prime*]

lemmas *compact*[*cont-intro*] = *compact-points-are-ccpo-compact*[*OF tls.singleton.compact-point*]

<ML>

16.4 The idle process

The idle process contains no transitions and does not terminate.

<ML>

definition *idle* :: ('a, 's, 'v) *behavior.t set where*

idle = $(\bigcup s. \text{raw.singleton } (\text{behavior.B } s \ (\text{trepeat } (\text{undefined}, s))))$

lemma *idle-alt-def*:

shows $\text{raw.idle} = \{\omega. \neg \text{tfinite } (\text{behavior.rest } \omega) \wedge \text{behavior.sset } \omega \subseteq \{\text{behavior.init } \omega\}\}$ (**is** ?*lhs* = ?*rhs*)

<proof>

<ML>

lemma *not-tfinite*:

assumes $\omega \in \text{raw.idle}$

shows $\neg \text{tfinite } (\text{behavior.rest } \omega)$

<proof>

<ML>

lemma *idle*[*iff*]:

shows $\text{raw.idle} \in \text{behavior.stuttering.closed}$

<proof>

<ML>

lift-definition *idle* :: ('a, 's, 'v) *tls is raw.idle* *<proof>*

lemma *idle-alt-def*:

shows $\text{tls.idle} = (\bigsqcup s. \langle s, \text{trepeat } (\text{undefined}, s) \rangle_T)$

<proof>

<ML>

lemma *idle-le-conv*[*tls.singleton.le-conv*]:

shows $\langle \omega \rangle_T \leq \text{tls.idle} \iff \neg \text{tfinite } (\text{behavior.rest } \omega) \wedge \text{behavior.sset } \omega \subseteq \{\text{behavior.init } \omega\}$

<proof>

<ML>

lemma *minimal-le*:

shows $\langle s, \text{repeat } (\text{undefined}, s) \rangle_T \leq \text{tls.idle}$
<proof>

<ML>

16.5 Temporal Logic for ('a, 's, 'v) tls

The following is a straightforward shallow embedding of the now-traditional anchored semantics of LTL [Manna and Pnueli \(1988\)](#).

References:

- [\\$AFP/TLA/Liveness.thy](#)
- [\\$ISABELLE_HOME/src/HOL/TLA/TLA.thy](#)
- https://en.wikipedia.org/wiki/Linear_temporal_logic
- [Kröger and Merz \(2008\)](#)
- [Warford, Vega, and Staley \(2020\)](#)

Observations:

- as we lack next/X/⊙ (due to stuttering closure) we do not have induction for \mathcal{U} (until)
- [Lamport \(1994\)](#) omitted the LTL “until” operator from TLA as he considered it too hard to use
- As [De Giacomo and Vardi \(2013\)](#) observe, things get non-standard on finite traces
 - see §24 for an example
 - [Maier \(2004\)](#) provides an alternative account

<ML>

definition *state-prop* :: ('a, 's, 'v) behavior.t set **where**
state-prop $P = \{\omega. P (\text{behavior.init } \omega)\}$

definition

until :: ('a, 's, 'v) behavior.t set \Rightarrow ('a, 's, 'v) behavior.t set \Rightarrow ('a, 's, 'v) behavior.t set

where

until $P Q = \{\omega. \exists i. \exists \omega' \in Q. \text{behavior.dropn } i \omega = \text{Some } \omega' \wedge (\forall j < i. \text{the } (\text{behavior.dropn } j \omega) \in P)\}$

definition

eventually :: ('a, 's, 'v) behavior.t set \Rightarrow ('a, 's, 'v) behavior.t set

where

eventually $P = \text{raw.until UNIV } P$

definition

always :: ('a, 's, 'v) behavior.t set \Rightarrow ('a, 's, 'v) behavior.t set

where

always $P = \neg \text{raw.eventually } (\neg P)$

abbreviation (*input*) *unless* $P Q \equiv \text{raw.until } P Q \cup \text{raw.always } P$

definition *terminated* :: ('a, 's, 'v) behavior.t set **where**

terminated $= \{\omega. \text{tfinite } (\text{behavior.rest } \omega) \wedge \text{behavior.sset } \omega \subseteq \{\text{behavior.init } \omega\}\}$

lemma *untilI*:

assumes $\text{behavior.dropn } i \omega = \text{Some } \omega'$

assumes $\omega' \in Q$
assumes $\bigwedge j. j < i \implies \text{the } (\text{behavior.dropn } j \ \omega) \in P$
shows $\omega \in \text{raw.until } P \ Q$
 <proof>

lemma eventually-alt-def:
shows $\text{raw.eventually } P = \{\omega . \exists \omega' \in P. \exists i. \text{behavior.dropn } i \ \omega = \text{Some } \omega'\}$
 <proof>

lemma always-alt-def:
shows $\text{raw.always } P = \{\omega . \forall i \ \omega'. \text{behavior.dropn } i \ \omega = \text{Some } \omega' \implies \omega' \in P\}$
 <proof>

lemma alwaysI:
assumes $\bigwedge i \ \omega'. \text{behavior.dropn } i \ \omega = \text{Some } \omega' \implies \omega' \in P$
shows $\omega \in \text{raw.always } P$
 <proof>

lemma alwaysD:
assumes $\omega \in \text{raw.always } P$
assumes $\text{behavior.dropn } i \ \omega = \text{Some } \omega'$
shows $\omega' \in P$
 <proof>

<ML>

lemma monotone:
shows $\text{mono raw.state-prop}$
 <proof>

lemma_simps:
shows
 $\text{raw.state-prop } \langle \text{False} \rangle = \{\}$
 $\text{raw.state-prop } \perp = \{\}$
 $\text{raw.state-prop } \langle \text{True} \rangle = \text{UNIV}$
 $\text{raw.state-prop } \top = \text{UNIV}$
 $-\ \text{raw.state-prop } P = \text{raw.state-prop } (- P)$
 $\text{raw.state-prop } P \cup \text{raw.state-prop } Q = \text{raw.state-prop } (P \sqcup Q)$
 $\text{raw.state-prop } P \cap \text{raw.state-prop } Q = \text{raw.state-prop } (P \sqcap Q)$
 $(\text{raw.state-prop } P \longrightarrow_B \text{raw.state-prop } Q) = \text{raw.state-prop } (P \longrightarrow_B Q)$
 <proof>

lemma Inf:
shows $\text{raw.state-prop } (\bigcap X) = \bigcap (\text{raw.state-prop } ` X)$
 <proof>

lemma Sup:
shows $\text{raw.state-prop } (\bigcup X) = \bigcup (\text{raw.state-prop } ` X)$
 <proof>

<ML>

lemma inf-always-le:
fixes $P :: ('a, 's, 'v) \text{behavior.t set}$
assumes $P \in \text{behavior.stuttering.closed}$
shows $\text{raw.terminated } \cap P \subseteq \text{raw.always } P$
 <proof>

$\langle ML \rangle$

lemma *base*:

shows $\omega \in Q \implies \omega \in \text{raw.until } P \ Q$

and $Q \subseteq \text{raw.until } P \ Q$

$\langle \text{proof} \rangle$

lemma *step*:

assumes $\omega \in P$

assumes $\text{behavior.tl } \omega = \text{Some } \omega'$

assumes $\omega' \in \text{raw.until } P \ Q$

shows $\omega \in \text{raw.until } P \ Q$

$\langle \text{proof} \rangle$

lemmas *intro*[*intro*] =

raw.until.base

raw.until.step

lemma *induct*[*case-names base step, consumes 1, induct set: raw.until*]:

assumes $\omega \in \text{raw.until } P \ Q$

assumes *base*: $\bigwedge \omega. \omega \in Q \implies R \ \omega$

assumes *step*: $\bigwedge \omega \ \omega'. [\omega \in P; \text{behavior.tl } \omega = \text{Some } \omega'; \omega' \in \text{raw.until } P \ Q; R \ \omega'] \implies R \ \omega$

shows $R \ \omega$

$\langle \text{proof} \rangle$

lemma *mono*:

assumes $P \subseteq P'$

assumes $Q \subseteq Q'$

shows $\text{raw.until } P \ Q \subseteq \text{raw.until } P' \ Q'$

$\langle \text{proof} \rangle$

lemma *botL*:

shows $\text{raw.until } \{\} \ Q = Q$

$\langle \text{proof} \rangle$

lemma *botR*:

shows $\text{raw.until } P \ \{\} = \{\}$

$\langle \text{proof} \rangle$

lemma *untilR*:

shows $\text{raw.until } P \ (\text{raw.until } P \ Q) = \text{raw.until } P \ Q$ (**is** *?lhs = ?rhs*)

$\langle \text{proof} \rangle$

lemma *InfL-not-empty*:

assumes $X \neq \{\}$

shows $\text{raw.until } (\bigcap X) \ Q = (\bigcap_{x \in X}. \text{raw.until } x \ Q)$ (**is** *?lhs = ?rhs*)

$\langle \text{proof} \rangle$

lemma *SupR*:

shows $\text{raw.until } P \ (\bigcup X) = \bigcup (\text{raw.until } P \ ` \ X)$

$\langle \text{proof} \rangle$

lemma *weakenL*:

shows $\text{raw.until } UNIV \ P = \text{raw.until } (\neg \ P) \ P$ (**is** *?lhs = ?rhs*)

$\langle \text{proof} \rangle$

lemma *implication-ordering-le*: — Warford et al. (2020, (16))

shows $\text{raw.until } P \ Q \cap \text{raw.until } (\neg \ Q) \ R \subseteq \text{raw.until } P \ R$

$\langle proof \rangle$

lemma *infR-ordering-le*: — Warford et al. (2020, (18))

shows $raw.until\ P\ (Q \cap R) \subseteq raw.until\ (raw.until\ P\ Q)\ R$ (**is** $?lhs \subseteq ?rhs$)

$\langle proof \rangle$

lemma *untilL*:

shows $raw.until\ (raw.until\ P\ Q)\ Q \subseteq raw.until\ P\ Q$ (**is** $?lhs \subseteq ?rhs$)

$\langle proof \rangle$

lemma *alwaysR-le*:

shows $raw.until\ P\ (raw.always\ Q) \subseteq raw.always\ (raw.until\ P\ Q)$ (**is** $?lhs \subseteq ?rhs$)

$\langle proof \rangle$

$\langle ML \rangle$

lemma *neg*:

shows $\neg\ (raw.until\ P\ Q \cup raw.always\ P) = raw.until\ (\neg\ Q)\ (\neg\ P \cap \neg\ Q)$ (**is** $?lhs = ?rhs$)

$\langle proof \rangle$

$\langle ML \rangle$

lemma *terminated*:

shows $raw.eventually\ raw.terminated = \{\omega. tfinite\ (behavior.rest\ \omega)\}$ (**is** $?lhs = ?rhs$)

$\langle proof \rangle$

$\langle ML \rangle$

lemma *state-prop[intro]*:

shows $raw.state-prop\ P \in behavior.stuttering.closed$

$\langle proof \rangle$

lemma *terminated[intro]*:

shows $raw.terminated \in behavior.stuttering.closed$

$\langle proof \rangle$

lemma *until[intro]*:

assumes $P \in behavior.stuttering.closed$

assumes $Q \in behavior.stuttering.closed$

shows $raw.until\ P\ Q \in behavior.stuttering.closed$

$\langle proof \rangle$

lemma *eventually[intro]*:

assumes $P \in behavior.stuttering.closed$

shows $raw.eventually\ P \in behavior.stuttering.closed$

$\langle proof \rangle$

lemma *always[intro]*:

assumes $P \in behavior.stuttering.closed$

shows $raw.always\ P \in behavior.stuttering.closed$

$\langle proof \rangle$

$\langle ML \rangle$

definition *valid* :: $('a, 's, 'v) tls \Rightarrow bool$ **where**

$valid\ P \iff P = \top$

lift-definition *state-prop* :: $'s\ pred \Rightarrow ('a, 's, 'v) tls$ **is** $raw.state-prop$ $\langle proof \rangle$

lift-definition *terminated* :: ('a, 's, 'v) tls **is** raw.terminated ⟨proof⟩
lift-definition *until* :: ('a, 's, 'v) tls ⇒ ('a, 's, 'v) tls ⇒ ('a, 's, 'v) tls **is** raw.until ⟨proof⟩

definition *eventually* :: ('a, 's, 'v) tls ⇒ ('a, 's, 'v) tls **where**
eventually P = tls.until ⊤ P

definition *always* :: ('a, 's, 'v) tls ⇒ ('a, 's, 'v) tls **where**
always P = ¬tls.eventually (¬P)

definition *release* :: ('a, 's, 'v) tls ⇒ ('a, 's, 'v) tls ⇒ ('a, 's, 'v) tls **where**
release P Q = ¬(tls.until (¬P) (¬Q))

definition *unless* :: ('a, 's, 'v) tls ⇒ ('a, 's, 'v) tls ⇒ ('a, 's, 'v) tls **where**
unless P Q = tls.until P Q ⊓ tls.always P

abbreviation (*input*) *always-imp-syn* :: ('a, 's, 'v) tls ⇒ ('a, 's, 'v) tls ⇒ ('a, 's, 'v) tls **where**
always-imp-syn P Q ≡ tls.always (P →_B Q)

abbreviation (*input*) *leads-to* :: ('a, 's, 'v) tls ⇒ ('a, 's, 'v) tls ⇒ ('a, 's, 'v) tls **where**
leads-to P Q ≡ tls.always-imp-syn P (tls.eventually Q)

bundle *notation*

begin

notation *tls.valid* (|= - [30] 30)
notation *tls.state-prop* (⟨-⟩ [0])
notation *tls.until* (**infix** U 85)
notation *tls.eventually* (◇- [87] 87)
notation *tls.always* (□- [87] 87)
notation *tls.release* (**infixr** R 85)
notation *tls.unless* (**infixr** W 85)
notation *tls.always-imp-syn* (**infixr** →_□ 75)
notation *tls.leads-to* (**infixr** ∼ 75)

end

bundle *no-notation*

begin

no-notation *tls.valid* (|= - [30] 30)
no-notation *tls.state-prop* (⟨-⟩ [0])
no-notation *tls.until* (**infixr** U 85)
no-notation *tls.eventually* (◇- [87] 87)
no-notation *tls.always* (□- [87] 87)
no-notation *tls.release* (**infixr** R 85)
no-notation *tls.unless* (**infixr** W 85)
no-notation *tls.always-imp-syn* (**infixr** →_□ 75)
no-notation *tls.leads-to* (**infixr** ∼ 75)

end

unbundle *tls.notation*

lemma *validI*:

assumes ⊤ ≤ P

shows |= P

⟨proof⟩

$\langle ML \rangle$

lemma *trans*[*trans*]:

assumes $\models P$

assumes $P \leq Q$

shows $\models Q$

$\langle proof \rangle$

lemma *mp*:

assumes $\models P \longrightarrow_B Q$

assumes $\models P$

shows $\models Q$

$\langle proof \rangle$

lemmas *rev-mp* = *tls.valid.mp*[*rotated*]

$\langle ML \rangle$

lemma *uminus-le-conv*[*tls.singleton.le-conv*]:

shows $\langle \omega \rangle_T \leq -P \longleftrightarrow \neg \langle \omega \rangle_T \leq P$

$\langle proof \rangle$

lemma *state-prop-le-conv*[*tls.singleton.le-conv*]:

shows $\langle \omega \rangle_T \leq \text{tls.state-prop } P \longleftrightarrow P \text{ (behavior.init } \omega)$

$\langle proof \rangle$

lemma *terminated-le-conv*[*tls.singleton.le-conv*]:

shows $\langle \omega \rangle_T \leq \text{tls.terminated} \longleftrightarrow \text{tfinite (behavior.rest } \omega) \wedge \text{behavior.sset } \omega \subseteq \{\text{behavior.init } \omega\}$

$\langle proof \rangle$

lemma *until-le-conv*[*tls.singleton.le-conv*]:

fixes $P :: ('a, 's, 'v) \text{tls}$

shows $\langle \omega \rangle_T \leq P \mathcal{U} Q \longleftrightarrow (\exists i \omega'. \text{behavior.dropn } i \omega = \text{Some } \omega'$

$\wedge \langle \omega' \rangle_T \leq Q$

$\wedge (\forall j < i. \langle \text{the (behavior.dropn } j \omega) \rangle_T \leq P)) \text{ (is ?lhs } \longleftrightarrow \text{ ?rhs)}$

$\langle proof \rangle$

lemma *eventually-le-conv*[*tls.singleton.le-conv*]:

shows $\langle \omega \rangle_T \leq \diamond P \longleftrightarrow (\exists i \omega'. \text{behavior.dropn } i \omega = \text{Some } \omega' \wedge \langle \omega' \rangle_T \leq P)$

$\langle proof \rangle$

lemma *always-le-conv*[*tls.singleton.le-conv*]:

shows $\langle \omega \rangle_T \leq \text{tls.always } P \longleftrightarrow (\forall i \omega'. \text{behavior.dropn } i \omega = \text{Some } \omega' \longrightarrow \langle \omega' \rangle_T \leq P)$

$\langle proof \rangle$

$\langle ML \rangle$

interpretation *until*: *closure-complete-lattice-distributive-class* *tls.until* *P* **for** *P*

$\langle proof \rangle$

$\langle ML \rangle$

lemmas *botL* = *raw.until.botL*[*transferred*]

lemmas *botR* = *raw.until.botR*[*transferred*]

lemmas *topR* = *tls.until.cl-top*

lemmas *expansiveR* = *tls.until.expansive*[*of* *P* *Q* **for** *P* *Q*]

lemmas *weakenL* = *raw.until.weakenL*[*transferred*]

lemmas *mono* = *raw.until.mono*[*transferred*]

lemma *strengthen*[*strg*]:

assumes *st-ord* *F P P'*

assumes *st-ord* *F Q Q'*

shows *st-ord* *F (P U Q) (P' U Q')*

<proof>

lemma *SupL-le*:

shows $(\bigsqcup_{x \in X}. x \mathcal{U} R) \leq (\bigsqcup X) \mathcal{U} R$

<proof>

lemma *supL-le*:

shows $P \mathcal{U} R \sqcup Q \mathcal{U} R \leq (P \sqcup Q) \mathcal{U} R$

<proof>

lemma *SupR*:

shows $P \mathcal{U} (\bigsqcup X) = \bigsqcup ((\mathcal{U} P) \cdot X)$

<proof>

lemmas *supR* = *tls.until.cl-sup*

lemmas *InfL-not-empty* = *raw.until.InfL-not-empty*[*transferred*]

lemmas *infL* = *tls.until.InfL-not-empty*[**where** $X = \{P, Q\}$ **for** $P Q$, *simplified*, **of** $P Q R$ **for** $P Q R$]

lemmas *InfR-le* = *tls.until.cl-Inf-le*

lemmas *infR-le* = *tls.until.cl-inf-le*[*of* $P Q R$ **for** $P Q R$]

lemma *implication-ordering-le*: — Warford et al. (2020, (16))

shows $P \mathcal{U} Q \sqcap (-Q) \mathcal{U} R \leq P \mathcal{U} R$

<proof>

lemma *supL-ordering-le*: — Warford et al. (2020, (17))

shows $P \mathcal{U} (Q \mathcal{U} R) \leq (P \sqcup Q) \mathcal{U} R$ (**is** *?lhs* \leq *?rhs*)

<proof>

lemma *infR-ordering-le*: — Warford et al. (2020, (18))

shows $P \mathcal{U} (Q \sqcap R) \leq (P \mathcal{U} Q) \mathcal{U} R$

<proof>

lemma *boolean-implication-distrib-le*: — Warford et al. (2020, (19))

shows $(P \longrightarrow_B Q) \mathcal{U} R \leq (P \mathcal{U} R) \longrightarrow_B (Q \mathcal{U} R)$

<proof>

lemma *excluded-middleR*: — Warford et al. (2020, (23))

shows $\models P \mathcal{U} Q \sqcup P \mathcal{U} (-Q)$

<proof>

lemmas *untilR* = *tls.until.idempotent*(1)[*of* $P Q$ **for** $P Q$]

lemma *untilL*:

shows $(P \mathcal{U} Q) \mathcal{U} Q = P \mathcal{U} Q$ (**is** *?lhs* = *?rhs*)

<proof>

lemma *absorb*:

shows $P \mathcal{U} P = P$

<proof>

lemma *absorb-supL*: — Warford et al. (2020, (23))

shows $P \sqcup P \mathcal{U} Q = P \sqcup Q$

<proof>

lemma *absorb-supR*: — Warford et al. (2020, (23))

shows $Q \sqcup P \mathcal{U} Q = P \mathcal{U} Q$

<proof>

lemma *eventually-le*:

shows $P \mathcal{U} Q \leq \diamond Q$

<proof>

lemma *absorb-eventually*:

shows *inf-eventually-absorbR*: $P \mathcal{U} Q \sqcap \diamond Q = P \mathcal{U} Q$ — Warford et al. (2020, (39))

and *sup-eventually-absorbR*: $P \mathcal{U} Q \sqcup \diamond Q = \diamond Q$ — Warford et al. (2020, (40))

and *eventually-absorbR*: $P \mathcal{U} \diamond Q = \diamond Q$ — Warford et al. (2020, (41))

<proof>

lemma *sup-le*: — Warford et al. (2020, (28))

shows $P \mathcal{U} Q \leq P \sqcup Q$

<proof>

lemma *ordering*: — Warford et al. (2020, (251))

shows $(-P) \mathcal{U} Q \sqcup (-Q) \mathcal{U} P = \diamond(P \sqcup Q)$ (**is** *?lhs = ?rhs*)

<proof>

lemmas *simps =*

tls.until.expansiveR

tls.until.botL

tls.until.botR

tls.until.absorb

tls.until.absorb-supL

tls.until.absorb-supR

tls.until.untilL

tls.until.untilR

<ML>

interpretation *eventually*: *closure-complete-lattice-distributive-class* *tls.eventually*

<proof>

lemma *eventually-alt-def*:

shows $\diamond P = (-P) \mathcal{U} P$

<proof>

<ML>

lemma *transfer[transfer-rule]*:

shows *rel-fun* (*pcr-tls* (=) (=) (=)) (*pcr-tls* (=) (=) (=)) *raw.eventually* *tls.eventually*

<proof>

lemma *bot*:

shows $\diamond \perp = \perp$

<proof>

lemma *bot-conv*:

shows $\diamond P = \perp \longleftrightarrow P = \perp$

<proof>

lemmas $top = tls.eventually.cl-top$

lemmas $monotone = tls.eventually.monotone-cl$

lemmas $mono = tls.eventually.mono-cl$

lemmas $Sup = tls.eventually.cl-Sup[simplified\ tls.eventually.bot,\ simplified]$

lemmas $sup = tls.eventually.Sup[where\ X=\{P,\ Q\}\ for\ P\ Q,\ simplified]$

lemmas $Inf-le = tls.eventually.cl-Inf-le$

lemmas $inf-le = tls.eventually.cl-inf-le$

lemma $neg:$

shows $-\diamond P = \square(-P)$

$\langle proof \rangle$

lemma $boolean-implication-le:$

shows $\diamond P \longrightarrow_B \diamond Q \leq \diamond(P \longrightarrow_B Q)$

$\langle proof \rangle$

lemmas $simps =$

$tls.eventually.bot$

$tls.eventually.top$

$tls.eventually.expansive$

$tls.eventually-def[symmetric]$

lemma $terminated:$

shows $\diamond tls.terminated = \bigsqcup (tls.singleton\ \{\omega.\ tfinite\ (behavior.rest\ \omega)\})$

$\langle proof \rangle$

lemma $always-imp-le:$

shows $P \longrightarrow_{\square} Q \leq \diamond P \longrightarrow_B \diamond Q$

$\langle proof \rangle$

lemma $until:$

shows $\diamond(P\ \mathcal{U}\ Q) = \diamond Q$

$\langle proof \rangle$

$\langle ML \rangle$

lemma $always-alt-def:$

shows $\square P = P\ \mathcal{W}\ \perp$

$\langle proof \rangle$

$\langle ML \rangle$

lemma $transfer[transfer-rule]:$

shows $rel_fun\ (pcr-tls\ (=)\ (=)\ (=))\ (pcr-tls\ (=)\ (=)\ (=))\ raw.always\ tls.always$

$\langle proof \rangle$

$tls.always$ is an interior operator

lemma $idempotent[simp]:$

shows $\square\square P = \square P$

$\langle proof \rangle$

lemma $contractive:$

shows $\square P \leq P$

$\langle proof \rangle$

lemma *monotone*^[iff]:
shows *mono tls.always*
 ⟨*proof*⟩

lemmas *strengthen*^[strg] = *st-monotone*^[OF tls.always.monotone]
lemmas *mono*^[trans] = *monoD*^[OF tls.always.monotone]

lemma *bot*:
shows $\Box \perp = \perp$
 ⟨*proof*⟩

lemma *top*:
shows $\Box \top = \top$
 ⟨*proof*⟩

lemma *top-conv*:
shows $\Box P = \top \longleftrightarrow P = \top$
 ⟨*proof*⟩

lemma *Sup-le*:
shows $\bigsqcup (tls.always \text{ ' } X) \leq \Box (\bigsqcup X)$
 ⟨*proof*⟩

lemma *sup-le*:
shows $\Box P \sqcup \Box Q \leq \Box (P \sqcup Q)$
 ⟨*proof*⟩

lemma *Inf*:
shows $\Box (\prod X) = \prod (tls.always \text{ ' } X)$
 ⟨*proof*⟩

lemma *inf*:
shows $\Box (P \sqcap Q) = \Box P \sqcap \Box Q$
 ⟨*proof*⟩

lemma *neg*:
shows $\neg \Box P = \Diamond (\neg P)$
 ⟨*proof*⟩

lemma *always-necessitation*:
assumes $\models P$
shows $\models \Box P$
 ⟨*proof*⟩

lemma *valid-conv*:
shows $\models \Box P \longleftrightarrow \models P$
 ⟨*proof*⟩

lemma *always-imp-le*:
shows $P \longrightarrow_{\Box} Q \leq \Box P \longrightarrow_B \Box Q$
 ⟨*proof*⟩

lemma *eventually-le*:
shows $\Box P \leq \Diamond P$
 ⟨*proof*⟩

lemma *not-until-le*: — Warford et al. (2020, (81))

shows $\Box P \leq -(Q \mathcal{U} (-P))$
<proof>

lemmas *simps* =
 tls.always.bot
 tls.always.top
 tls.always.contractive
 tls.always-alt-def[symmetric]

<ML>

lemma *until-alwaysR-le*: — Warford et al. (2020, (140))
shows $P \mathcal{U} \Box Q \leq \Box(P \mathcal{U} Q)$
<proof>

lemma *until-alwaysR*: — Warford et al. (2020, (141))
shows $P \mathcal{U} \Box P = \Box P$
<proof>

lemma *eventually-always-always-eventually-le*: — Warford et al. (2020, (145))
shows $\Diamond \Box P \leq \Box \Diamond P$
<proof>

lemma *always-inf-eventually-eventually-le*:
shows $\Box P \sqcap \Diamond Q \leq \Diamond(P \sqcap Q)$
<proof>

lemma *always-always-imp*: — Kröger and Merz (2008, §2.2: T33 frame)
shows $\models \Box P \longrightarrow_B \Box Q \longrightarrow_B \Box(P \sqcap Q)$
<proof>

lemma *always-eventually-imp*: — Kröger and Merz (2008, §2.2: T34 frame)
shows $\models \Box P \longrightarrow_B \Diamond Q \longrightarrow_B \Diamond(P \sqcap Q)$
<proof>

lemma *always-imp-always-generalization*: — Kröger and Merz (2008, §2.2: T35)
shows $\Box P \longrightarrow_{\Box} Q \leq \Box P \longrightarrow_B \Box Q$
<proof>

lemma *always-imp-eventually-generalization*: — Kröger and Merz (2008, §2.2: T36)
shows $P \longrightarrow_{\Box} \Diamond Q \leq \Diamond P \longrightarrow_B \Diamond Q$
<proof>

The following show that there is no point nesting *tls.always* and *tls.eventually* more than two deep.

lemma *always-eventually-always-absorption*: — Kröger and Merz (2008, §2.2: T37)
shows $\Diamond \Box \Diamond P = \Box \Diamond P$
<proof>

lemma *eventually-always-eventually-absorption*: — Kröger and Merz (2008, §2.2: T38)
shows $\Box \Diamond \Box P = \Diamond \Box P$
<proof>

lemma *always-imp-always-eventually-le*: — Warford et al. (2020, (157))
shows $P \longrightarrow_{\Box} Q \leq \Box \Diamond P \longrightarrow_B \Box \Diamond Q$
<proof>

lemma *always-imp-eventually-always-le*: — Warford et al. (2020, (158))
shows $P \longrightarrow_{\Box} Q \leq \Diamond \Box P \longrightarrow_B \Diamond \Box Q$

$\langle proof \rangle$

lemma *always-eventually-inf-le*:

shows $\Box \Diamond (P \sqcap Q) \leq \Box \Diamond P \sqcap \Box \Diamond Q$ — Warford et al. (2020, (159))

$\langle proof \rangle$

lemma *eventually-always-sup-le*:

shows $\Diamond \Box P \sqcap \Diamond \Box Q \leq \Diamond \Box (P \sqcup Q)$ — Warford et al. (2020, (160))

$\langle proof \rangle$

lemma *always-eventually-sup*: — Warford et al. (2020, (161))

fixes $P :: ('a, 's, 'v) \text{ tls}$

shows $\Box \Diamond (P \sqcup Q) = \Box \Diamond P \sqcup \Box \Diamond Q$ (**is** $?lhs = ?rhs$)

$\langle proof \rangle$

lemma *eventually-always-inf*: — Warford et al. (2020, (162))

shows $\Diamond \Box (P \sqcap Q) = \Diamond \Box P \sqcap \Diamond \Box Q$

$\langle proof \rangle$

lemma *eventual-latching*: — Warford et al. (2020, (163))

shows $\Diamond \Box (P \longrightarrow_B \Box Q) = \Diamond \Box (\neg P) \sqcup \Diamond \Box Q$ (**is** $?lhs = ?rhs$)

$\langle proof \rangle$

$\langle ML \rangle$

lemma *transfer[transfer-rule]*:

shows $rel\text{-fun } (pcr\text{-tls } (=) (=) (=)) (rel\text{-fun } (pcr\text{-tls } (=) (=) (=)) (pcr\text{-tls } (=) (=) (=)))$
 $(\lambda P Q. raw.until P Q \cup raw.always P)$
 $tls.unless$

$\langle proof \rangle$

lemma *neg*: — Warford et al. (2020, (170))

shows $\neg(P \mathcal{W} Q) = (\neg Q) \mathcal{U} (\neg P \sqcap \neg Q)$

$\langle proof \rangle$

lemma *alwaysR-le*: — Warford et al. (2020, (177))

shows $P \mathcal{W} \Box Q \leq \Box (P \mathcal{W} Q)$

$\langle proof \rangle$

lemma *sup-le*: — Warford et al. (2020, (206))

shows $P \mathcal{W} Q \leq P \sqcup Q$

$\langle proof \rangle$

lemma *ordering*: — Warford et al. (2020, (252))

shows $\models (\neg P) \mathcal{W} Q \sqcup (\neg Q) \mathcal{W} P$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *eq-unless-inf-eventually*:

shows $P \mathcal{U} Q = (P \mathcal{W} Q) \sqcap \Diamond Q$

$\langle proof \rangle$

lemma *always-strengthen-le*: — Warford et al. (2020, (83))

shows $\Box P \sqcap (Q \mathcal{U} R) \leq (P \sqcap Q) \mathcal{U} (P \sqcap R)$

$\langle proof \rangle$

lemma *until-weakI*:

shows $\Box P \sqcap \Diamond Q \leq P \mathcal{U} Q$ (is ?lhs \leq ?rhs) — Warford et al. (2020, (84))
<proof>

lemma *always-impL*: — Warford et al. (2020, (86))
shows $P \longrightarrow_{\Box} P' \sqcap P \mathcal{U} Q \leq P' \mathcal{U} Q$ (is ?thesis1)
and $P \mathcal{U} Q \sqcap P \longrightarrow_{\Box} P' \leq P' \mathcal{U} Q$ (is ?thesis2)
<proof>

lemma *always-impR*: — Warford et al. (2020, (85))
shows $Q \longrightarrow_{\Box} Q' \sqcap P \mathcal{U} Q \leq P \mathcal{U} Q'$ (is ?thesis1)
and $P \mathcal{U} Q \sqcap Q \longrightarrow_{\Box} Q' \leq P \mathcal{U} Q'$ (is ?thesis2)
<proof>

lemma *neg*: — Warford et al. (2020, (173))
shows $-(P \mathcal{U} Q) = (-Q) \mathcal{W} (-P \sqcap -Q)$
<proof>

<ML>

lemmas *monotone* = *raw.state-prop.monotone*[*transferred*]
lemmas *strengthen*[*strg*] = *st-monotone*[*OF tls.state-prop.monotone*]
lemmas *mono* = *monoD*[*OF tls.state-prop.monotone*]

lemma *Sup*:
shows $\text{tls.state-prop } (\bigsqcup X) = \bigsqcup (\text{tls.state-prop } ` X)$
<proof>

lemma *Inf*:
shows $\text{tls.state-prop } (\bigsqcap X) = \bigsqcap (\text{tls.state-prop } ` X)$
<proof>

lemmas *simps* = *raw.state-prop.simps*[*transferred*]
<ML>

lemma *not-bot*:
shows $\text{tls.terminated} \neq \perp$
<proof>

lemma *not-top*:
shows $\text{tls.terminated} \neq \top$
<proof>

lemma *inf-always*:
shows $\text{tls.terminated} \sqcap \Box P = \text{tls.terminated} \sqcap P$
<proof>

lemma *always-le-conv*:
shows $\text{tls.terminated} \leq \Box P \longleftrightarrow \text{tls.terminated} \leq P$
<proof>

lemma *inf-eventually*:
shows $\text{tls.terminated} \sqcap \Diamond P = \text{tls.terminated} \sqcap P$ (is ?lhs = ?rhs)
<proof>

lemma *eventually-le-conv*:
shows $\text{tls.terminated} \leq \text{tls.eventually } P \longleftrightarrow \text{tls.terminated} \leq P$
<proof>

lemma *eq-always-terminated*:

shows $tls.terminated = \Box tls.terminated$

$\langle proof \rangle$

$\langle ML \rangle$

16.5.1 Leads-to and leads-to-via

So-called *response* properties are of the form $P \longrightarrow_{\Box} \Diamond Q$ (pronounced “ P leads to Q ”, written $P \rightsquigarrow Q$) (Manna and Pnueli 1991). This connective is similar to the “ensures” modality of Chandy and Misra (1989, §3.4.4).

Jackson (1998) used the more general “ P leads to Q via I ” form $P \longrightarrow_{\Box} I \mathcal{U} Q$ to establish liveness properties in a sequential setting.

lemma *leads-to-refl*:

shows $\models P \rightsquigarrow P$

$\langle proof \rangle$

lemma *leads-to-mono*:

assumes $P' \leq P$

assumes $Q \leq Q'$

shows $P \rightsquigarrow Q \leq P' \rightsquigarrow Q'$

$\langle proof \rangle$

lemma *leads-to-supL*:

shows $(P \rightsquigarrow R) \sqcap (Q \rightsquigarrow R) \leq (P \sqcup Q) \rightsquigarrow R$

$\langle proof \rangle$

lemma *always-imp-leads-to*:

shows $P \longrightarrow_{\Box} Q \leq P \rightsquigarrow Q$

$\langle proof \rangle$

lemma *leads-to-eventually*:

shows $\Diamond P \sqcap (P \rightsquigarrow Q) \leq \Diamond Q$

$\langle proof \rangle$

lemma *leads-to-leads-to-via*:

shows $P \longrightarrow_{\Box} Q \mathcal{U} R \leq P \rightsquigarrow R$

$\langle proof \rangle$

lemma *leads-to-trans*:

shows $P \rightsquigarrow Q \sqcap Q \rightsquigarrow R \leq P \rightsquigarrow R$ (is ?lhs \leq ?rhs)

$\langle proof \rangle$

lemma *leads-to-via-weakenR*:

shows $Q \longrightarrow_{\Box} Q' \sqcap P \longrightarrow_{\Box} I \mathcal{U} Q \leq P \longrightarrow_{\Box} I \mathcal{U} Q'$

$\langle proof \rangle$

lemma *leads-to-via-supL*: — useful for case distinctions

shows $P \longrightarrow_{\Box} I \mathcal{U} Q \sqcap P' \longrightarrow_{\Box} I' \mathcal{U} Q \leq P \sqcup P' \longrightarrow_{\Box} (I \sqcup I') \mathcal{U} Q$

$\langle proof \rangle$

lemma *leads-to-via-trans*:

shows $(P \longrightarrow_{\Box} I \mathcal{U} Q) \sqcap (Q \longrightarrow_{\Box} I' \mathcal{U} R) \leq P \longrightarrow_{\Box} (I \sqcup I') \mathcal{U} R$ (is ?lhs \leq ?rhs)

$\langle proof \rangle$

lemma *leads-to-via-disj*: — more like a chaining rule

shows $(P \longrightarrow_{\Box} I \mathcal{U} Q) \sqcap (Q \longrightarrow_{\Box} I' \mathcal{U} R) \leq (P \sqcup Q) \longrightarrow_{\Box} (I \sqcup I') \mathcal{U} R$

$\langle proof \rangle$

16.5.2 Fairness

A few renderings of weak fairness. [van Glabbeek and Höfner \(2019\)](#) call this “response to insistence” as a generalisation of weak fairness.

definition *weakly-fair* :: $(\prime a, \prime s, \prime v) tls \Rightarrow (\prime a, \prime s, \prime v) tls \Rightarrow (\prime a, \prime s, \prime v) tls$ **where**
weakly-fair enabled taken = $\Box enabled \longrightarrow_{\Box} \Diamond taken$

lemma *weakly-fair-def2*:

shows $tls.weakly-fair\ enabled\ taken = \Box(-(\Box(enabled \sqcap -taken)))$

$\langle proof \rangle$

lemma *weakly-fair-def3*:

shows $tls.weakly-fair\ enabled\ taken = \Diamond\Box enabled \longrightarrow_B \Box\Diamond taken$

$\langle proof \rangle$

lemma *weakly-fair-def4*:

shows $tls.weakly-fair\ enabled\ taken = \Box\Diamond(enabled \longrightarrow_B taken)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *mono*:

assumes $P' \leq P$

assumes $Q \leq Q'$

shows $tls.weakly-fair\ P\ Q \leq tls.weakly-fair\ P'\ Q'$

$\langle proof \rangle$

lemma *strengthen[stg]*:

assumes $st-ord\ (\neg F)\ P\ P'$

assumes $st-ord\ F\ Q\ Q'$

shows $st-ord\ F\ (tls.weakly-fair\ P\ Q)\ (tls.weakly-fair\ P'\ Q')$

$\langle proof \rangle$

lemma *weakly-fair-triv*:

shows $\Box\Diamond(\neg enabled) \leq tls.weakly-fair\ enabled\ taken$

$\langle proof \rangle$

lemma *mp*:

shows $tls.weakly-fair\ enabled\ taken \sqcap \Box enabled \leq \Diamond taken$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *weakly-fair*:

shows $\Box(tls.weakly-fair\ enabled\ taken) = tls.weakly-fair\ enabled\ taken$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *weakly-fair*:

shows $\Diamond(tls.weakly-fair\ enabled\ taken) = tls.weakly-fair\ enabled\ taken$

$\langle proof \rangle$

$\langle ML \rangle$

Similarly for strong fairness. [van Glabbeek and Höfner \(2019\)](#) call this “response to persistence” as a generalisation

of strong fairness.

definition *strongly-fair* :: $(\prime a, \prime s, \prime v) \text{ tls} \Rightarrow (\prime a, \prime s, \prime v) \text{ tls} \Rightarrow (\prime a, \prime s, \prime v) \text{ tls}$ **where**
strongly-fair enabled taken = $\Box \Diamond \text{enabled} \longrightarrow_{\Box} \Diamond \text{taken}$

lemma *strongly-fair-def2*:

shows $\text{tls.strongly-fair enabled taken} = \Box (\neg \Box (\Diamond \text{enabled} \sqcap \neg \text{taken}))$
 $\langle \text{proof} \rangle$

lemma *strongly-fair-def3*:

shows $\text{tls.strongly-fair enabled taken} = \Box \Diamond \text{enabled} \longrightarrow_B \Box \Diamond \text{taken}$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *mono*:

assumes $P' \leq P$
assumes $Q \leq Q'$
shows $\text{tls.strongly-fair } P \ Q \leq \text{tls.strongly-fair } P' \ Q'$
 $\langle \text{proof} \rangle$

lemma *strengthen[strg]*:

assumes $\text{st-ord } (\neg F) \ P \ P'$
assumes $\text{st-ord } F \ Q \ Q'$
shows $\text{st-ord } F \ (\text{tls.strongly-fair } P \ Q) \ (\text{tls.strongly-fair } P' \ Q')$
 $\langle \text{proof} \rangle$

lemma *supL*: — does not hold for *tls.weakly-fair*

shows $\text{tls.strongly-fair } (\text{enabled1} \sqcup \text{enabled2}) \ \text{taken}$
 $= (\text{tls.strongly-fair enabled1 taken} \sqcap \text{tls.strongly-fair enabled2 taken})$
 $\langle \text{proof} \rangle$

lemma *weakly-fair-le*:

shows $\text{tls.strongly-fair enabled taken} \leq \text{tls.weakly-fair enabled taken}$
 $\langle \text{proof} \rangle$

lemma *always-enabled-weakly-fair-strongly-fair*:

shows $\Box \text{enabled} \leq \text{tls.weakly-fair enabled taken} \longleftrightarrow_B \text{tls.strongly-fair enabled taken}$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *strongly-fair*:

shows $\Box (\text{tls.strongly-fair enabled taken}) = \text{tls.strongly-fair enabled taken}$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *strongly-fair*:

shows $\Diamond (\text{tls.strongly-fair enabled taken}) = \text{tls.strongly-fair enabled taken}$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

16.6 Safety Properties

We now carve the safety properties out of the $(\prime a, \prime s, \prime v) \text{ tls}$ lattice.

References:

- [Alpern and Schneider \(1985\)](#); [Alpern, Demers, and Schneider \(1986\)](#); [Schneider \(1987, §2\)](#)
 - observes that Lamport’s earlier definitions do not work without stuttering
 - provides the now standard definition that works with and without stuttering
- [Abadi and Lamport \(1991, §2.2\)](#): topological definitions and intuitions
- [Sistla \(1994, §2.2\)](#)

We go a different way: we establish a Galois connection with $(\prime a, \prime s, \prime v)$ *spec*.

Observations:

- our safety closure for $(\prime a, \prime s, \prime v)$ *tls* introduces infinite sequences to stand for the prefixes in $(\prime a, \prime s, \prime v)$ *spec*
 - i.e., the non-termination of trace σ ($\text{trace.term } \sigma = \text{None}$) is represented by a behavior ending with $\text{trace.final } \sigma$ infinitely stuttered
 - [Abadi and Lamport \(1991, §2.1\)](#) consider these behaviors to represent terminating processes

$\langle ML \rangle$

definition *to-spec* :: $(\prime a, \prime s, \prime v)$ *behavior.t set* \Rightarrow $(\prime a, \prime s, \prime v)$ *trace.t set* **where**
to-spec $T = \{\text{behavior.take } i \ \omega \mid \omega \text{ i. } \omega \in T\}$

definition *from-spec* :: $(\prime a, \prime s, \prime v)$ *trace.t set* \Rightarrow $(\prime a, \prime s, \prime v)$ *behavior.t set* **where**
from-spec $S = \{\omega . \forall i. \text{behavior.take } i \ \omega \in S\}$

interpretation *safety*: *galois.powerset raw.to-spec raw.from-spec*
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *empty*:

shows *raw.from-spec* $\{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *singleton*:

shows *raw.from-spec* (*Safety-Logic.raw.singleton* σ)
 $= \bigcup (\text{raw.singleton } \omega \text{ . } \forall i. \text{behavior.take } i \ \omega \in \text{Safety-Logic.raw.singleton } \sigma)$ (**is** *?lhs = ?rhs*)
 $\langle \text{proof} \rangle$

lemma *sup*:

assumes $P \in \text{raw.spec.closed}$
assumes $Q \in \text{raw.spec.closed}$
shows *raw.from-spec* $(P \cup Q) = \text{raw.from-spec } P \cup \text{raw.from-spec } Q$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *singleton*:

shows *raw.to-spec* (*TLS.raw.singleton* ω)
 $= (\bigcup i. \text{Safety-Logic.raw.singleton } (\text{behavior.take } i \ \omega))$ (**is** *?lhs = ?rhs*)
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *cl-altI*:

assumes $\bigwedge i. \exists \omega' \in P. \text{behavior.take } i \ \omega = \text{behavior.take } i \ \omega'$
shows $\omega \in \text{raw.safety.cl } P$

<proof>

lemma *cl-altE*:

assumes $\omega \in \text{raw.safety.cl } P$

obtains ω' **where** $\omega' \in P$ **and** $\text{behavior.take } i \ \omega = \text{behavior.take } i \ \omega'$

<proof>

lemma *cl-alt-def*: — [Alpern et al. \(1986\)](#): the classical definition: ω belongs to the safety closure of P if every prefix of ω can be extended to a behavior in P

shows $\text{raw.safety.cl } P = \{\omega. \forall i. \exists \beta. \text{behavior.take } i \ \omega @_{-B} \beta \in P\}$ (**is** *?lhs = ?rhs*)

<proof>

lemma *closed-alt-def*: — If ω is not in P then some prefix of ω has irretrievably gone wrong

shows $\text{raw.safety.closed} = \{P. \forall \omega. \omega \notin P \longrightarrow (\exists i. \forall \beta. \text{behavior.take } i \ \omega @_{-B} \beta \notin P)\}$

<proof>

lemma *closed-alt-def2*: — Contraposition gives the customary prefix-closure definition

shows $\text{raw.safety.closed} = \{P. \forall \omega. (\forall i. \exists \beta. \text{behavior.take } i \ \omega @_{-B} \beta \in P) \longrightarrow \omega \in P\}$

<proof>

lemma *closedI2*:

assumes $\bigwedge \omega. (\bigwedge i. \exists \beta. \text{behavior.take } i \ \omega @_{-B} \beta \in P) \implies \omega \in P$

shows $P \in \text{raw.safety.closed}$

<proof>

lemma *closedE2*:

assumes $P \in \text{raw.safety.closed}$

assumes $\bigwedge i. \omega \notin P \implies \exists \beta. \text{behavior.take } i \ \omega @_{-B} \beta \in P$

shows $\omega \in P$

<proof>

<ML>

lemma *state-prop*:

shows $\text{raw.safety.cl } (\text{raw.state-prop } P) = \text{raw.state-prop } P$

<proof>

lemma *terminated-iff*:

assumes $\omega \in \text{raw.terminated}$

shows $\omega \in \text{raw.safety.cl } P \longleftrightarrow \omega \in P$ (**is** *?lhs \longleftrightarrow ?rhs*)

<proof>

lemma *terminated*:

shows $\text{raw.safety.cl } \text{raw.terminated} = \text{raw.idle} \cup \text{raw.terminated}$ (**is** *?lhs = ?rhs*)

<proof>

lemma *le-terminated-bot*:

assumes $P \in \text{behavior.stuttering.closed}$

assumes $\text{raw.safety.cl } P \subseteq \text{raw.terminated}$

shows $P = \{\}$

<proof>

lemma *always-le*:

shows $\text{raw.safety.cl } (\text{raw.always } P) \subseteq \text{raw.always } (\text{raw.safety.cl } P)$

<proof>

lemma *eventually*:

assumes $P \neq \perp$

shows $raw.safety.cl (raw.eventually P)$
 $= -raw.eventually raw.terminated \cup raw.eventually P$ (**is** $?lhs = ?rhs$)
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *always-eventually*:

assumes $P \in raw.safety.closed$
assumes $\forall i. \exists j \geq i. \exists \beta. behavior.take\ j\ \omega\ @_{-B}\ \beta \in P$
shows $\omega \in P$
 $\langle proof \rangle$

lemma *sup*:

assumes $P \in raw.safety.closed$
assumes $Q \in raw.safety.closed$
shows $P \cup Q \in raw.safety.closed$
 $\langle proof \rangle$

lemma *unless*: — [Sistla \(1994, §3.1\)](#) – minimality is irrelevant

assumes $P \in raw.safety.closed$
assumes $Q \in raw.safety.closed$
shows $raw.unless\ P\ Q \in raw.safety.closed$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *to-spec*:

shows $range\ raw.to-spec \subseteq downwards.closed$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *to-spec*:

shows $raw.to-spec\ ' behavior.stuttering.closed \subseteq trace.stuttering.closed$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *to-spec*:

shows $raw.to-spec\ ' behavior.stuttering.closed \subseteq raw.spec.closed$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *from-spec*:

shows $raw.from-spec\ ' trace.stuttering.closed$
 $\subseteq (behavior.stuttering.closed :: ('a, 's, 'v)\ behavior.t\ set\ set)$
 $\langle proof \rangle$

lemma *safety-cl*:

assumes $P \in behavior.stuttering.closed$
shows $raw.safety.cl\ P \in behavior.stuttering.closed$
 $\langle proof \rangle$

$\langle ML \rangle$

lift-definition $to-spec :: ('a, 's, 'v)\ tls \Rightarrow ('a, 's, 'v)\ spec$ **is** $raw.to-spec$

$\langle proof \rangle$

lift-definition *from-spec* :: ('a, 's, 'v) spec \Rightarrow ('a, 's, 'v) tls **is** raw.from-spec
<proof>

interpretation *safety*: galois.complete-lattice-class tls.to-spec tls.from-spec
<proof>

<ML>

lemma *singleton*:

notes spec.singleton.transfer[transfer-rule]

shows tls.from-spec (spec.singleton σ)

= \sqcup (tls.singleton ' { ω . $\forall i$. behavior.take i $\omega \in$ Safety-Logic.raw.singleton σ })

<proof>

lemmas bot = raw.from-spec.empty[transferred]

lemma *sup*:

shows tls.from-spec ($P \sqcup Q$) = tls.from-spec $P \sqcup$ tls.from-spec Q

<proof>

lemmas Inf = tls.safety.upper-Inf

lemmas inf = tls.safety.upper-inf

<ML>

lemma *singleton*:

notes spec.singleton.transfer[transfer-rule]

shows tls.to-spec (tls.singleton ω) = (\sqcup i . spec.singleton (behavior.take i ω))

<proof>

lemmas bot = tls.safety.lower-bot

lemmas Sup = tls.safety.lower-Sup

lemmas sup = tls.safety.lower-sup

<ML>

lemma *transfer*[transfer-rule]:

shows rel-fun (pcr-tls (=) (=) (=)) (pcr-tls (=) (=) (=)) raw.safety.cl tls.safety.cl

<proof>

lemma *bot*[iff]:

shows tls.safety.cl $\perp = \perp$

<proof>

lemma *sup*:

shows tls.safety.cl ($P \sqcup Q$) = tls.safety.cl $P \sqcup$ tls.safety.cl Q

<proof>

lemmas state-prop = raw.safety.cl.state-prop[transferred]

lemmas always-le = raw.safety.cl.always-le[transferred]

lemma *eventually*: — all the infinite traces and any finite ones that satisfy $\diamond P$

assumes $P \neq \perp$

shows tls.safety.cl ($\diamond P$) = $-\diamond$ tls.terminated $\sqcup \diamond P$

<proof>

lemma *terminated-iff*:

assumes $\langle \omega \rangle_T \leq \text{tls.terminated}$

shows $\langle \omega \rangle_T \leq \text{tls.safety.cl } P \longleftrightarrow \langle \omega \rangle_T \leq P$ (**is** $?lhs \longleftrightarrow ?rhs$)

$\langle \text{proof} \rangle$

lemma *terminated*:

shows $\text{tls.safety.cl } \text{tls.terminated} = \text{tls.idle} \sqcup \text{tls.terminated}$

$\langle \text{proof} \rangle$

lemma *not-terminated*:

shows $\text{tls.safety.cl } (\neg \text{tls.terminated}) = \neg \text{tls.terminated}$ (**is** $?lhs = ?rhs$)

$\langle \text{proof} \rangle$

lemma *le-terminated-conv*:

shows $\text{tls.safety.cl } P \leq \text{tls.terminated} \longleftrightarrow P = \perp$ (**is** $?lhs \longleftrightarrow ?rhs$)

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *transfer[transfer-rule]*:

shows $\text{rel-set } (\text{pcr-tls } (=) (=) (=))$

$(\text{behavior.stuttering.closed} \cap \text{raw.safety.closed})$

tls.safety.closed (**is** $\text{rel-set} - ?lhs ?rhs$)

$\langle \text{proof} \rangle$

lemma *bot*:

shows $\perp \in \text{tls.safety.closed}$

$\langle \text{proof} \rangle$

lemma *sup*:

assumes $P \in \text{tls.safety.closed}$

assumes $Q \in \text{tls.safety.closed}$

shows $P \sqcup Q \in \text{tls.safety.closed}$

$\langle \text{proof} \rangle$

lemmas $\text{inf} = \text{tls.safety.closed-inf}$

lemma *boolean-implication*:

assumes $\neg P \in \text{tls.safety.closed}$

assumes $Q \in \text{tls.safety.closed}$

shows $P \longrightarrow_B Q \in \text{tls.safety.closed}$

$\langle \text{proof} \rangle$

lemma *state-prop*:

shows $\text{tls.state-prop } P \in \text{tls.safety.closed}$

$\langle \text{proof} \rangle$

lemma *not-terminated*:

shows $\neg \text{tls.terminated} \in \text{tls.safety.closed}$

$\langle \text{proof} \rangle$

lemma *unless*:

assumes $P \in \text{tls.safety.closed}$

assumes $Q \in \text{tls.safety.closed}$

shows $\text{tls.unless } P Q \in \text{tls.safety.closed}$

$\langle \text{proof} \rangle$

lemma *always*:

assumes $P \in \text{tls.safety.closed}$
shows $\text{tls.always } P \in \text{tls.safety.closed}$
 ⟨proof⟩

⟨ML⟩

lemma *until-unless-le*:

assumes $P \in \text{tls.safety.closed}$
assumes $Q \in \text{tls.safety.closed}$
shows $\text{tls.safety.cl } (\text{tls.until } P \ Q) \leq \text{tls.unless } P \ Q$
 ⟨proof⟩

⟨ML⟩

lemma *to-spec-le-conv*[*tls.singleton.le-conv*]:

notes *spec.singleton.transfer*[*transfer-rule*]
shows $\langle \sigma \rangle \leq \text{tls.to-spec } P \longleftrightarrow (\exists \omega \ i. \langle \omega \rangle_T \leq P \wedge \sigma = \text{behavior.take } i \ \omega)$
 ⟨proof⟩

lemma *from-spec-le-conv*[*tls.singleton.le-conv*]:

notes *spec.singleton.transfer*[*transfer-rule*]
shows $\langle \omega \rangle_T \leq \text{tls.from-spec } P \longleftrightarrow (\forall i. \langle \text{behavior.take } i \ \omega \rangle \leq P)$
 ⟨proof⟩

lemma *safety-cl-le-conv*[*tls.singleton.le-conv*]:

shows $\langle \omega \rangle_T \leq \text{tls.safety.cl } P \longleftrightarrow (\forall i. \exists \omega'. \langle \omega' \rangle_T \leq P \wedge \text{behavior.take } i \ \omega = \text{behavior.take } i \ \omega')$
 ⟨proof⟩

⟨ML⟩

16.7 Maps

⟨ML⟩

definition *map* :: $('a \Rightarrow 'b) \Rightarrow ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('a, 's, 'v) \text{tls} \Rightarrow ('b, 't, 'w) \text{tls}$ **where**
 $\text{map } \text{af } \text{sf } \text{vf } P = \bigsqcup (\text{tls.singleton } \text{'behavior.map } \text{af } \text{sf } \text{vf } \text{' } \{ \sigma. \langle \sigma \rangle_T \leq P \})$

definition *invmap* :: $('a \Rightarrow 'b) \Rightarrow ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('b, 't, 'w) \text{tls} \Rightarrow ('a, 's, 'v) \text{tls}$ **where**
 $\text{invmap } \text{af } \text{sf } \text{vf } P = \bigsqcup (\text{tls.singleton } \text{'behavior.map } \text{af } \text{sf } \text{vf } \text{' } \{ \sigma. \langle \sigma \rangle_T \leq P \})$

abbreviation *amap* :: $('a \Rightarrow 'b) \Rightarrow ('a, 's, 'v) \text{tls} \Rightarrow ('b, 's, 'v) \text{tls}$ **where**

$\text{amap } \text{af} \equiv \text{tls.map } \text{af } \text{id } \text{id}$

abbreviation *ainvmap* :: $('a \Rightarrow 'b) \Rightarrow ('b, 's, 'v) \text{tls} \Rightarrow ('a, 's, 'v) \text{tls}$ **where**

$\text{ainvmap } \text{af} \equiv \text{tls.invmap } \text{af } \text{id } \text{id}$

abbreviation *smap* :: $('s \Rightarrow 't) \Rightarrow ('a, 's, 'v) \text{tls} \Rightarrow ('a, 't, 'v) \text{tls}$ **where**

$\text{smap } \text{sf} \equiv \text{tls.map } \text{id } \text{sf } \text{id}$

abbreviation *sinvmap* :: $('s \Rightarrow 't) \Rightarrow ('a, 't, 'v) \text{tls} \Rightarrow ('a, 's, 'v) \text{tls}$ **where**

$\text{sinvmap } \text{sf} \equiv \text{tls.invmap } \text{id } \text{sf } \text{id}$

abbreviation *vmap* :: $('v \Rightarrow 'w) \Rightarrow ('a, 's, 'v) \text{tls} \Rightarrow ('a, 's, 'w) \text{tls}$ **where** — aka *liftM*

$\text{vmap } \text{vf} \equiv \text{tls.map } \text{id } \text{id } \text{vf}$

abbreviation *vinvmap* :: $('v \Rightarrow 'w) \Rightarrow ('a, 's, 'w) \text{tls} \Rightarrow ('a, 's, 'v) \text{tls}$ **where**

$\text{vinvmap } \text{vf} \equiv \text{tls.invmap } \text{id } \text{id } \text{vf}$

interpretation *map-invmap*: *galois.complete-lattice-distributive-class*

$\text{tls.map } \text{af } \text{sf } \text{vf}$

$\text{tls.invmap } \text{af } \text{sf } \text{vf}$ **for** $\text{af } \text{sf } \text{vf}$

⟨proof⟩

$\langle ML \rangle$

lemma *map-le-conv*[*tls.singleton.le-conv*]:

shows $\langle \omega \rangle_T \leq \text{tls.map af sf vf } P \longleftrightarrow (\exists \omega'. \langle \omega' \rangle_T \leq P \wedge \langle \omega \rangle_T \leq \langle \text{behavior.map af sf vf } \omega' \rangle_T)$
<proof>

lemma *invmap-le-conv*[*tls.singleton.le-conv*]:

shows $\langle \omega \rangle_T \leq \text{tls.invmap af sf vf } P \longleftrightarrow \langle \text{behavior.map af sf vf } \omega \rangle_T \leq P$
<proof>

$\langle ML \rangle$

lemmas *bot* = *tls.map-invmap.lower-bot*

lemmas *monotone* = *tls.map-invmap.monotone-lower*

lemmas *mono* = *monotoneD[OF tls.map.monotone]*

lemmas *Sup* = *tls.map-invmap.lower-Sup*

lemmas *sup* = *tls.map-invmap.lower-sup*

lemmas *Inf-le* = *tls.map-invmap.lower-Inf-le* — Converse does not hold

lemmas *inf-le* = *tls.map-invmap.lower-inf-le* — Converse does not hold

lemmas *invmap-le* = *tls.map-invmap.lower-upper-contractive*

lemma *singleton*:

shows $\text{tls.map af sf vf } \langle \omega \rangle_T = \langle \text{behavior.map af sf vf } \omega \rangle_T$
<proof>

lemma *top*:

assumes *surj af*

assumes *surj sf*

assumes *surj vf*

shows $\text{tls.map af sf vf } \top = \top$

<proof>

lemma *id*:

shows $\text{tls.map id id id } P = P$

and $\text{tls.map } (\lambda x. x) (\lambda x. x) (\lambda x. x) P = P$

<proof>

lemma *comp*:

shows $\text{tls.map af sf vf } \circ \text{tls.map ag sg vg} = \text{tls.map } (af \circ ag) (sf \circ sg) (vf \circ vg)$ (**is** *?lhs = ?rhs*)

and $\text{tls.map af sf vf } (\text{tls.map ag sg vg } P) = \text{tls.map } (\lambda a. af (ag a)) (\lambda s. sf (sg s)) (\lambda v. vf (vg v)) P$ (**is** *?thesis1*)

<proof>

lemmas *map* = *tls.map.comp*

$\langle ML \rangle$

lemmas *bot* = *tls.map-invmap.upper-bot*

lemmas *top* = *tls.map-invmap.upper-top*

lemmas *monotone* = *tls.map-invmap.monotone-upper*

lemmas *mono* = *monotoneD[OF tls.invmap.monotone]*

lemmas *Sup* = *tls.map-invmap.upper-Sup*

lemmas *sup* = *tls.map-invmap.upper-sup*

lemmas $Inf = tls.map-invmap.upper-Inf$

lemmas $inf = tls.map-invmap.upper-inf$

lemma *singleton*:

shows $tls.invmap\ af\ sf\ vf\ \langle\omega\rangle_T = \sqcup (tls.singleton\ \{\omega'.\ \langle behavior.map\ af\ sf\ vf\ \omega'\rangle_T \leq \langle\omega\rangle_T\})$
<proof>

lemma *id*:

shows $tls.invmap\ id\ id\ id\ P = P$
and $tls.invmap\ (\lambda x. x)\ (\lambda x. x)\ (\lambda x. x)\ P = P$
<proof>

lemma *comp*:

shows $tls.invmap\ af\ sf\ vf\ (tls.invmap\ ag\ sg\ vg\ P) = tls.invmap\ (\lambda x. ag\ (af\ x))\ (\lambda s. sg\ (sf\ s))\ (\lambda v. vg\ (vf\ v))\ P$
(is ?lhs P = ?rhs P)
and $tls.invmap\ af\ sf\ vf\ \circ\ tls.invmap\ ag\ sg\ vg = tls.invmap\ (ag\ \circ\ af)\ (sg\ \circ\ sf)\ (vg\ \circ\ vf)$ **(is ?thesis1)**
<proof>

lemmas $invmap = tls.invmap.comp$

<ML>

lemma *map*:

shows $tls.to-spec\ (tls.map\ af\ sf\ vf\ P) = spec.map\ af\ sf\ vf\ (tls.to-spec\ P)$
<proof>

<ML>

16.8 Abadi's axioms for TLA

The axioms for “propositional” TLA due to Abadi (1990) hold in this model. These are complete for *tls.always* and *tls.eventually*.

Observations:

- Abadi says that the temporal system is D aka S4.3Dum; see Goldblatt (1992, §8)
 - the only interesting axiom here is 5: the discrete-time Dummett axiom
- “propositional” means that actions are treated separately; we omit this part as we don't have actions ala TLA

<ML>

lemma *Ax1*:

shows $\models \Box(P \longrightarrow_B Q) \longrightarrow_B \Box P \longrightarrow_B \Box Q$
<proof>

lemma *Ax2*:

shows $\models \Box P \longrightarrow_B P$
<proof>

lemma *Ax3*:

shows $\models \Box P \longrightarrow_B \Box\Box P$
<proof>

lemma *Ax4*:

— “a classical way to express that time is linear – that any two instants in the future are ordered” Warford et al. (2020, (254) Lemmon formula)

shows $\models \Box(\Box P \longrightarrow_B Q) \sqcup \Box(\Box Q \longrightarrow_B P)$
 $\langle proof \rangle$

lemma Ax5:

— “expresses the discreteness of time” See also Warford et al. (2020, §4.1 “the Dummett formula”): for them “next” encodes discreteness

fixes $P :: ('a, 's, 'v) tls$
shows $\models \Box(\Box(P \longrightarrow_B \Box P) \longrightarrow_B P) \longrightarrow_B \Diamond \Box P \longrightarrow_B P$ (**is** $\models ?goal$)
 $\langle proof \rangle$

lemma Ax6:

assumes $\models P$
shows $\models \Box P$
 $\langle proof \rangle$

lemma Ax8:

assumes $\models P$
assumes $\models P \longrightarrow_B Q$
shows $\models Q$
 $\langle proof \rangle$

$\langle ML \rangle$

16.9 Tweak syntax

unbundle *tls.no-notation*

no-notation *tls.singleton* $(\langle \!-\! \rangle_T)$

$\langle ML \rangle$

bundle *extra-notation*

begin

notation *tls.singleton* $(\langle \!-\! \rangle_T [0])$

notation *tls.from-spec* $(\langle \!-\! \rangle [0])$

end

bundle *no-extra-notation*

begin

no-notation *tls.singleton* $(\langle \!-\! \rangle_T [0])$

no-notation *tls.from-spec* $(\langle \!-\! \rangle [0])$

end

$\langle ML \rangle$

17 Atomic sections

By restricting the environment to stuttering steps we can consider arbitrary processes to be atomic, i.e., free of interference.

$\langle ML \rangle$

definition *atomic* $:: 'a \Rightarrow ('a, 's, 'v) spec \Rightarrow ('a, 's, 'v) spec$ **where**

$atomic\ a\ P = P \sqcap spec.rel\ (\{a\} \times UNIV)$

$\langle ML \rangle$

lemma *atomic-le-conv[spec.idle-le]*:

shows $spec.idle \leq spec.atomic\ a\ P \longleftrightarrow spec.idle \leq P$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *atomic*:

shows $spec.term.none\ (spec.atomic\ a\ P) = spec.atomic\ a\ (spec.term.none\ P)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *atomic*:

shows $spec.term.all\ (spec.atomic\ a\ P) = spec.atomic\ a\ (spec.term.all\ P)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *bot[simp]*:

shows $spec.atomic\ a\ \perp = \perp$

$\langle proof \rangle$

lemma *top[simp]*:

shows $spec.atomic\ a\ \top = spec.rel\ (\{a\} \times UNIV)$

$\langle proof \rangle$

lemma *contractive*:

shows $spec.atomic\ a\ P \leq P$

$\langle proof \rangle$

lemma *idempotent[simp]*:

shows $spec.atomic\ a\ (spec.atomic\ a\ P) = spec.atomic\ a\ P$

$\langle proof \rangle$

lemma *monotone*:

shows $mono\ (spec.atomic\ a)$

$\langle proof \rangle$

lemmas *strengthen[strg] = st-monotone[OF spec.atomic.monotone]*

lemmas *mono = monotoneD[OF spec.atomic.monotone]*

lemmas *mono2mono[cont-intro, partial-function-mono]*

$= monotone2monotone[OF spec.atomic.monotone, simplified, of\ orda\ P\ \mathbf{for}\ orda\ P]$

lemma *Sup*:

shows $spec.atomic\ a\ (\bigsqcup X) = \bigsqcup (spec.atomic\ a\ ` X)$

$\langle proof \rangle$

lemmas *sup = spec.atomic.Sup[where X={P, Q} for P Q, simplified]*

lemma *mcont2mcont[cont-intro]*:

assumes $mcont\ luba\ orda\ Sup\ (\leq)\ P$

shows $mcont\ luba\ orda\ Sup\ (\leq)\ (\lambda x. spec.atomic\ a\ (P\ x))$

$\langle proof \rangle$

lemma *Inf-not-empty*:

assumes $X \neq \{\}$

shows $\text{spec.atomic } a (\bigsqcap X) = \bigsqcap (\text{spec.atomic } a \text{ ' } X)$

$\langle \text{proof} \rangle$

lemmas $\text{inf} = \text{spec.atomic.Inf-not-empty}[\text{where } X=\{P, Q\} \text{ for } P Q, \text{ simplified}]$

lemma *idle*:

shows $\text{spec.atomic } a \text{ spec.idle} = \text{spec.idle}$

$\langle \text{proof} \rangle$

lemma *action*:

shows $\text{spec.atomic } a (\text{spec.action } F) = \text{spec.action } (F \cap \text{UNIV} \times (\{a\} \times \text{UNIV} \cup \text{UNIV} \times \text{Id}))$

$\langle \text{proof} \rangle$

lemma *return*:

shows $\text{spec.atomic } a (\text{spec.return } v) = \text{spec.return } v$

$\langle \text{proof} \rangle$

lemma *bind*:

shows $\text{spec.atomic } a (f \ggg g) = \text{spec.atomic } a f \ggg (\lambda v. \text{spec.atomic } a (g v))$

$\langle \text{proof} \rangle$

lemma *map-le*:

fixes $af :: 'a \Rightarrow 'b$

shows $\text{spec.map } af \text{ sf } vf (\text{spec.atomic } a P) \leq \text{spec.atomic } (af a) (\text{spec.map } af \text{ sf } vf P)$

$\langle \text{proof} \rangle$

lemma *invmap*:

fixes $af :: 'a \Rightarrow 'b$

shows $\text{spec.atomic } a (\text{spec.invmap } af \text{ sf } vf P) \leq \text{spec.invmap } af \text{ sf } vf (\text{spec.atomic } (af a) P)$

$\langle \text{proof} \rangle$

lemma *rel*:

shows $\text{spec.atomic } a (\text{spec.rel } r) = \text{spec.rel } (r \cap \{a\} \times \text{UNIV})$

$\langle \text{proof} \rangle$

lemma *interference*:

shows $\text{spec.atomic } (\text{proc } a) (\text{spec.rel } (\{\text{env}\} \times \text{UNIV})) = \text{spec.rel } \{\}$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *cl*:

shows $\text{spec.atomic } (\text{proc } a) (\text{spec.cam.cl } (\{\text{env}\} \times \text{UNIV}) P) = \text{spec.atomic } (\text{proc } a) P$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *cl*:

shows $\text{spec.atomic } (\text{proc } a) (\text{spec.interference.cl } (\{\text{env}\} \times \text{UNIV}) P) = \text{spec.return } () \ggg \text{spec.atomic } (\text{proc } a)$

P

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lift-definition $\text{atomic} :: ('s, 'v) \text{ prog} \Rightarrow ('s, 'v) \text{ prog}$ **is**

$\lambda P. \text{spec.interference.cl } (\{\text{env}\} \times \text{UNIV}) (\text{spec.atomic self } P)$ $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *bot[simp]*:
 shows *prog.atomic* $\perp = \perp$
 $\langle proof \rangle$

lemma *contractive*:
 shows *prog.atomic* $P \leq P$
 $\langle proof \rangle$

lemma *idempotent[simp]*:
 shows *prog.atomic* (*prog.atomic* P) = *prog.atomic* P
 $\langle proof \rangle$

lemma *monotone*:
 shows *mono prog.atomic*
 $\langle proof \rangle$

lemmas *strengthen[strg] = st-monotone[OF prog.atomic.monotone]*

lemmas *mono = monotoneD[OF prog.atomic.monotone]*

lemmas *mono2mono[cont-intro, partial-function-mono] = monotone2monotone[OF prog.atomic.monotone, simplified, of orda P for orda P]*

lemma *Sup*:
 shows *prog.atomic* ($\bigsqcup X$) = \bigsqcup (*prog.atomic* ‘ X)
 $\langle proof \rangle$

lemmas *sup = prog.atomic.Sup[where X={P, Q} for P Q, simplified]*

lemma *mcont*:
 shows *mcont Sup* (\leq) *Sup* (\leq) *prog.atomic*
 $\langle proof \rangle$

lemmas *mcont2mcont[cont-intro] = mcont2mcont[OF prog.atomic.mcont, of luba orda P for luba orda P]*

lemma *Inf-le*:
 shows *prog.atomic* ($\bigsqcap X$) $\leq \bigsqcap$ (*prog.atomic* ‘ X)
 $\langle proof \rangle$

lemmas *inf-le = prog.atomic.Inf-le[where X={P, Q} for P Q, simplified]*

lemma *action*:
 shows *prog.atomic* (*prog.action* F) = *prog.action* F
 $\langle proof \rangle$

lemma *return*:
 shows *prog.atomic* (*prog.return* v) = *prog.return* v
 $\langle proof \rangle$

lemma *bind-le*:
 shows *prog.atomic* ($f \ggg g$) \leq *prog.atomic* $f \ggg$ ($\lambda v. \text{prog.atomic } (g v)$)
 $\langle proof \rangle$

$\langle ML \rangle$

lemmas *atomic = prog.atomic.rep-eq*

$\langle ML \rangle$

17.1 Inhabitation

$\langle ML \rangle$

lemma *atomic*:

assumes $P -s, xs \rightarrow P'$

assumes $trace.steps' s xs \subseteq \{a\} \times UNIV$

shows $spec.atomic a P -s, xs \rightarrow spec.atomic a P'$

$\langle proof \rangle$

lemma *atomic-term*:

assumes $P -s, xs \rightarrow spec.return v$

assumes $trace.steps' s xs \subseteq \{a\} \times UNIV$

shows $spec.atomic a P -s, xs \rightarrow spec.return v$

$\langle proof \rangle$

lemma *atomic-diverge*:

assumes $P -s, xs \rightarrow \perp$

assumes $trace.steps' s xs \subseteq \{a\} \times UNIV$

shows $spec.atomic a P -s, xs \rightarrow \perp$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *atomic-term*:

assumes $prog.p2s P -s, xs \rightarrow spec.return v$

assumes $trace.steps' s xs \subseteq \{self\} \times UNIV$

shows $prog.p2s (prog.atomic P) -s, xs \rightarrow spec.return v$

$\langle proof \rangle$

lemma *atomic-diverge*:

assumes $prog.p2s P -s, xs \rightarrow \perp$

assumes $trace.steps' s xs \subseteq \{self\} \times UNIV$

shows $prog.p2s (prog.atomic P) -s, xs \rightarrow \perp$

$\langle proof \rangle$

$\langle ML \rangle$

17.2 Assume/guarantee

$\langle ML \rangle$

lemma *atomic*:

assumes $prog.p2s c \leq \{P\}, Id \vdash G, \{Q\}$

assumes $P: stable A P$

assumes $Q: \bigwedge v. stable A (Q v)$

shows $prog.p2s (prog.atomic c) \leq \{P\}, A \vdash G, \{Q\}$

$\langle proof \rangle$

$\langle ML \rangle$

18 Exceptions

A sketch of how we might handle exceptions in this framework.

$\langle ML \rangle$

type-synonym $('s, 'x, 'v) \text{ exn} = ('s, 'x + 'v) \text{ prog}$

definition $\text{action} :: ('v \times 's \times 's) \text{ set} \Rightarrow ('s, 'x, 'v) \text{ raw.exn}$ **where**
 $\text{action} = \text{prog.action} \circ \text{image} (\text{map-prod Inr id})$

definition $\text{return} :: 'v \Rightarrow ('s, 'x, 'v) \text{ raw.exn}$ **where**
 $\text{return} = \text{prog.return} \circ \text{Inr}$

definition $\text{throw} :: 'x \Rightarrow ('s, 'x, 'v) \text{ raw.exn}$ **where**
 $\text{throw} = \text{prog.return} \circ \text{Inl}$

definition $\text{catch} :: ('s, 'x, 'v) \text{ raw.exn} \Rightarrow ('x \Rightarrow ('s, 'x, 'v) \text{ raw.exn}) \Rightarrow ('s, 'x, 'v) \text{ raw.exn}$ **where**
 $\text{catch } f \text{ handler} = f \gg\equiv \text{case-sum handler raw.return}$

definition $\text{bind} :: ('s, 'x, 'v) \text{ raw.exn} \Rightarrow ('v \Rightarrow ('s, 'x, 'v) \text{ raw.exn}) \Rightarrow ('s, 'x, 'v) \text{ raw.exn}$ **where**
 $\text{bind } f \text{ g} = f \gg\equiv \text{case-sum raw.throw g}$

definition $\text{parallel} :: ('s, 'x, \text{unit}) \text{ raw.exn} \Rightarrow ('s, 'x, \text{unit}) \text{ raw.exn} \Rightarrow ('s, 'x, \text{unit}) \text{ raw.exn}$ **where**
 $\text{parallel } P \text{ Q} = (P \gg\equiv \text{case-sum } \perp \text{ prog.return} \parallel Q \gg\equiv \text{case-sum } \perp \text{ prog.return}) \gg\equiv \text{raw.return}$

$\langle ML \rangle$

lemma bind :

shows $\text{raw.bind} (\text{raw.bind } f \text{ g}) \text{ h} = \text{raw.bind } f (\lambda x. \text{raw.bind} (g \text{ x}) \text{ h})$
 $\langle \text{proof} \rangle$

lemma return :

shows $\text{returnL}: \text{raw.bind} (\text{raw.return } v) = (\lambda g. g \text{ v})$
and $\text{returnR}: \text{raw.bind } f \text{ raw.return} = f$
 $\langle \text{proof} \rangle$

lemma throwL :

shows $\text{raw.bind} (\text{raw.throw } x) = (\lambda g. \text{raw.throw } x)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma catch :

shows $\text{raw.catch} (\text{raw.catch } f \text{ handler}_1) \text{ handler}_2 = \text{raw.catch } f (\lambda x. \text{raw.catch} (\text{handler}_1 \text{ x}) \text{ handler}_2)$
 $\langle \text{proof} \rangle$

lemma returnL :

shows $\text{raw.catch} (\text{raw.return } v) = (\lambda \text{handler}. \text{raw.return } v)$
 $\langle \text{proof} \rangle$

lemma throw :

shows $\text{throwL}: \text{raw.catch} (\text{raw.throw } x) = (\lambda g. g \text{ x})$
and $\text{throwR}: \text{raw.catch } f \text{ raw.throw} = f$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma commute :

shows $\text{raw.parallel } P \text{ Q} = \text{raw.parallel } Q \text{ P}$
 $\langle \text{proof} \rangle$

lemma assoc :

shows $\text{raw.parallel } P (\text{raw.parallel } Q \text{ R}) = \text{raw.parallel} (\text{raw.parallel } P \text{ Q}) \text{ R}$

⟨proof⟩

lemma *return*:

shows *raw.parallel* (*raw.return* ()) *P* = *raw.catch* *P* ($\lambda x. \perp$) (**is** *?thesis1*)

and *raw.parallel* *P* (*raw.return* ()) = *raw.catch* *P* ($\lambda x. \perp$) (**is** *?thesis2*)

⟨proof⟩

lemma *throw*:

shows *raw.parallel* (*raw.throw* *x*) *P* = *raw.bind* (*raw.catch* *P* ($\lambda x. \perp$)) ($\lambda x. \perp$) (**is** *?thesis1*)

and *raw.parallel* *P* (*raw.throw* *x*) = *raw.bind* (*raw.catch* *P* ($\lambda x. \perp$)) ($\lambda x. \perp$) (**is** *?thesis2*)

⟨proof⟩

⟨ML⟩

typedef (*'s*, *'x*, *'v*) *exn* = *UNIV* :: (*'s*, *'x*, *'v*) *raw.exn* *set*

⟨proof⟩

setup-lifting *type-definition-exn*

instantiation *exn* :: (*type*, *type*, *type*) *complete-distrib-lattice*

begin

lift-definition *bot-exn* :: (*'s*, *'x*, *'v*) *exn* **is** \perp ⟨proof⟩

lift-definition *top-exn* :: (*'s*, *'x*, *'v*) *exn* **is** \top ⟨proof⟩

lift-definition *sup-exn* :: (*'s*, *'x*, *'v*) *exn* \Rightarrow (*'s*, *'x*, *'v*) *exn* \Rightarrow (*'s*, *'x*, *'v*) *exn* **is** *sup* ⟨proof⟩

lift-definition *inf-exn* :: (*'s*, *'x*, *'v*) *exn* \Rightarrow (*'s*, *'x*, *'v*) *exn* \Rightarrow (*'s*, *'x*, *'v*) *exn* **is** *inf* ⟨proof⟩

lift-definition *less-eq-exn* :: (*'s*, *'x*, *'v*) *exn* \Rightarrow (*'s*, *'x*, *'v*) *exn* \Rightarrow *bool* **is** *less-eq* ⟨proof⟩

lift-definition *less-exn* :: (*'s*, *'x*, *'v*) *exn* \Rightarrow (*'s*, *'x*, *'v*) *exn* \Rightarrow *bool* **is** *less* ⟨proof⟩

lift-definition *Inf-exn* :: (*'s*, *'x*, *'v*) *exn* *set* \Rightarrow (*'s*, *'x*, *'v*) *exn* **is** *Inf* ⟨proof⟩

lift-definition *Sup-exn* :: (*'s*, *'x*, *'v*) *exn* *set* \Rightarrow (*'s*, *'x*, *'v*) *exn* **is** *Sup* ⟨proof⟩

instance ⟨proof⟩

end

⟨ML⟩

lift-definition *action* :: (*'v* \times *'s* \times *'s*) *set* \Rightarrow (*'s*, *'x*, *'v*) *exn* **is** *raw.action* ⟨proof⟩

lift-definition *return* :: *'v* \Rightarrow (*'s*, *'x*, *'v*) *exn* **is** *raw.return* ⟨proof⟩

lift-definition *throw* :: *'x* \Rightarrow (*'s*, *'x*, *'v*) *exn* **is** *raw.throw* ⟨proof⟩

lift-definition *catch* :: (*'s*, *'x*, *'v*) *exn* \Rightarrow (*'x* \Rightarrow (*'s*, *'x*, *'v*) *exn*) \Rightarrow (*'s*, *'x*, *'v*) *exn* **is** *raw.catch* ⟨proof⟩

lift-definition *bind* :: (*'s*, *'x*, *'v*) *exn* \Rightarrow (*'v* \Rightarrow (*'s*, *'x*, *'v*) *exn*) \Rightarrow (*'s*, *'x*, *'v*) *exn* **is** *raw.bind* ⟨proof⟩

lift-definition *parallel* :: (*'s*, *'x*, *unit*) *exn* \Rightarrow (*'s*, *'x*, *unit*) *exn* \Rightarrow (*'s*, *'x*, *unit*) *exn* **is** *raw.parallel* ⟨proof⟩

adhoc-overloading

Monad-Syntax.bind *exn.bind*

adhoc-overloading

parallel *exn.parallel*

⟨ML⟩

lemma *bind*:

shows $f \ggg g \ggg h = \text{exn.bind } f (\lambda x. g x \ggg h)$

⟨proof⟩

lemma *return*:

shows *returnL*: (\ggg) (*exn.return* *v*) = ($\lambda g. g v$) (**is** *?thesis1*)

and *returnR*: $f \ggg \text{exn.return } = f$ (**is** *?thesis2*)

$\langle proof \rangle$

lemma *throwL*:

shows $(\gg=) (exn.throw\ x) = (\lambda g. exn.throw\ x)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *catch*:

shows $exn.catch\ (exn.catch\ f\ handler_1)\ handler_2 = exn.catch\ f\ (\lambda x. exn.catch\ (handler_1\ x)\ handler_2)$

$\langle proof \rangle$

lemma *returnL*:

shows $exn.catch\ (exn.return\ v) = (\lambda handler. exn.return\ v)$

$\langle proof \rangle$

lemma *throw*:

shows *throwL*: $exn.catch\ (exn.throw\ x) = (\lambda g. g\ x)$

and *throwR*: $exn.catch\ f\ exn.throw = f$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *commute*:

shows $exn.parallel\ P\ Q = exn.parallel\ Q\ P$

$\langle proof \rangle$

lemma *assoc*:

shows $exn.parallel\ P\ (exn.parallel\ Q\ R) = exn.parallel\ (exn.parallel\ P\ Q)\ R$

$\langle proof \rangle$

lemma *return*:

shows *returnL*: $exn.return\ () \parallel P = exn.catch\ P\ \perp$

and *returnR*: $P \parallel exn.return\ () = exn.catch\ P\ \perp$

$\langle proof \rangle$

lemma *throw*:

shows *throwL*: $exn.throw\ x \parallel P = exn.catch\ P\ \perp \gg= \perp$

and *throwR*: $P \parallel exn.throw\ x = exn.catch\ P\ \perp \gg= \perp$

$\langle proof \rangle$

$\langle ML \rangle$

19 Assume/Guarantee rule sets

The rules in *ConcurrentHOL.Refinement* are deficient in various ways:

- redundant stability requirements
- interleaving of program decomposition with stability goals
- insufficiently instantiated

The following are some experimental rules aimed at practical assume/guarantee reasoning.

19.1 Implicit stabilisation

We can define a relation *ceilr* P to be the largest (weakest assumption) for which P is stable. This always yields a preorder (i.e., it is reflexive and transitive). Later we use this to inline stability side conditions into

assume/guarantee rules (§19.1.1).

This relation is not very pleasant to work with: it is not monotonic and does not have many useful algebraic properties. However it suffices to defer the checking of assumes (see §19.1.1).

This is a cognate of the *strongest guarantee* used by de Roever et al. (2001, Definition 8.31) in their completeness proof for the rely-guarantee method.

definition *ceilr* :: 'a pred \Rightarrow 'a rel **where**
ceilr $P = \bigsqcup \{r. \text{stable } r \ P\}$

lemma *ceilr-alt-def*:

shows *ceilr* $P = \{(s, s'). P \ s \longrightarrow P \ s'\}$
<proof>

lemma *ceilrE[elim]*:

assumes $(x, y) \in \text{ceilr } P$
assumes $P \ x$
shows $P \ y$
<proof>

<ML>

named-theorems *simps* *<simp rules for const <ceilr>>*

lemma *bot[ceilr.simps]*:

shows *ceilr* $\perp = \text{UNIV}$
<proof>

lemma *top[ceilr.simps]*:

shows *ceilr* $\top = \text{UNIV}$
<proof>

lemma *const[ceilr.simps]*:

shows *ceilr* $\langle c \rangle = \text{UNIV}$
and *ceilr* $(P \wedge \langle c \rangle) = (\text{if } c \text{ then } \text{ceilr } P \text{ else } \text{UNIV})$
and *ceilr* $(\langle c \rangle \wedge P) = (\text{if } c \text{ then } \text{ceilr } P \text{ else } \text{UNIV})$
and *ceilr* $(P \wedge \langle c \rangle \wedge P') = (\text{if } c \text{ then } \text{ceilr } (P \wedge P') \text{ else } \text{UNIV})$
<proof>

lemma *Id-le*:

shows $\text{Id} \subseteq \text{ceilr } P$
<proof>

lemmas *refl[iff] = ceilr.Id-le[folded refl-alt-def]*

lemma *trans[iff]*:

shows *trans* $(\text{ceilr } P)$
<proof>

lemma *stable[stable.intro]*:

shows *stable* $(\text{ceilr } P) \ P$
<proof>

lemma *largest[stable.intro]*:

assumes *stable* $r \ P$
shows $r \subseteq \text{ceilr } P$
<proof>

lemma *disj-subseteq*: — Converse does not hold

shows $\text{ceilr } (P \vee Q) \subseteq \text{ceilr } P \cup \text{ceilr } Q$
 $\langle \text{proof} \rangle$

lemma *Ex-subseteq*: — Converse does not hold
shows $\text{ceilr } (\exists x. P x) \subseteq (\bigcup x. \text{ceilr } (P x))$
 $\langle \text{proof} \rangle$

lemma *conj-subseteq*: — Converse does not hold
shows $\text{ceilr } P \cap \text{ceilr } Q \subseteq \text{ceilr } (P \wedge Q)$
 $\langle \text{proof} \rangle$

lemma *All-subseteq*: — Converse does not hold
shows $(\bigcap x. \text{ceilr } (P x)) \subseteq \text{ceilr } (\forall x. P x)$
 $\langle \text{proof} \rangle$

lemma *const-implies[ceilr.simps]*:
shows $\text{ceilr } (\langle P \rangle \longrightarrow Q) = (\text{if } P \text{ then } \text{ceilr } Q \text{ else } \text{UNIV})$
 $\langle \text{proof} \rangle$

lemma *Id-proj-on*:
shows $(\bigcap c. \text{ceilr } (\langle c \rangle = f)) = \text{Id}_f$
and $(\bigcap c. \text{ceilr } (f = \langle c \rangle)) = \text{Id}_f$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *Inter-ceilr*:
shows $\text{stable } (\bigcap v. \text{ceilr } (Q v)) (Q v)$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

We can internalize the stability conditions; see §19.1.1 for further discussion.

$\langle \text{ML} \rangle$

lemma *p2s-s2p-ag-ceilr*:
shows $\text{prog.p2s } (\text{prog.s2p } (\{P\}, \text{ceilr } P \cap (\bigcap v. \text{ceilr } (Q v)) \vdash G, \{Q\}))$
 $= \{P\}, \text{ceilr } P \cap (\bigcap v. \text{ceilr } (Q v)) \vdash G, \{Q\}$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

19.1.1 Assume/guarantee rules using implicit stability

We use *ceilr* to incorporate stability side conditions directly into the assume/guarantee rules. In other words, instead of working with arbitrary relations, we work with the largest (most general) *assume* that makes the relevant predicates *stable*.

In practice this allows us to defer all stability obligations to the end of a proof, which may be in any convenient context (typically a function). This approach could be considered a semantic version of how [Zakowski, Cachera, Demange, Petri, Pichardie, Jagannathan, and Vitek \(2019\)](#) split sequential and assume/guarantee reasoning. See [Vafeiadis \(2008, §4\)](#) for a discussion on when to check stability.

We defer the *guarantee* proofs by incorporating them into preconditions. This also allows control flow context to be accumulated.

These are backchaining (“weakest precondition”) rules: the guarantee and post condition need to be instantiated and the rules instantiate assume and pre condition schematics.

Note that the rule for (\gg) duplicates stability goals.

See §22 for an example of using these rules.

$\langle ML \rangle$

named-theorems *intro* \langle safe backchaining intro rules \rangle

lemma *init*:

assumes $c \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$

assumes $\bigwedge s. P' s \implies P s$

assumes $A' \subseteq A$ — these rules use *ceilr* which always yields a reflexive relation (*ceilr.refl*)

shows $c \leq \{\!\{P'\}\!\}, A' \vdash G, \{\!\{Q\}\!\}$

\langle proof \rangle

lemmas *mono* = *ag.mono*

lemmas *gen-asm* = *ag.gen-asm*

lemmas *pre* = *ag.pre*

lemmas *pre-pre* = *ag.pre-pre*

lemmas *pre-post* = *ag.pre-post*

lemmas *pre-ag* = *ag.pre-ag*

lemmas *pre-a* = *ag.pre-a*

lemmas *pre-g* = *ag.pre-g*

lemmas *post-imp* = *ag.post-imp*

lemmas *conj-lift* = *ag.conj-lift*

lemmas *disj-lift* = *ag.disj-lift*

lemmas *all-lift* = *ag.all-lift*

lemmas *augment-a* = *ag.augment-a*

lemmas *augment-post* = *ag.augment-post*

lemmas *augment-post-imp* = *ag.augment-post-imp*

lemmas *stable-augment-base* = *ag.stable-augment-base*

lemmas *stable-augment* = *ag.stable-augment*

lemmas *stable-augment-post* = *ag.stable-augment-post*

lemmas *stable-augment-frame* = *ag.stable-augment-frame*

lemma *bind*[*iag.intro*]:

assumes $\bigwedge v. \text{prog.p2s } (g v) \leq \{\!\{Q' v\}\!\}, A_2 v \vdash G, \{\!\{Q\}\!\}$

assumes $\text{prog.p2s } f \leq \{\!\{P\}\!\}, A_1 \vdash G, \{\!\{Q'\}\!\}$

shows $\text{prog.p2s } (f \ggg g) \leq \{\!\{P\}\!\}, A_1 \cap (\bigcap v. A_2 v) \vdash G, \{\!\{Q\}\!\}$

\langle proof \rangle

lemmas *rev-bind* = *iag.bind*[*rotated*]

lemma *read*[*iag.intro*]:

shows $\text{prog.p2s } (\text{prog.read } F) \leq \{\!\{\lambda s. Q (F s) s\}\!\}, \text{ceilr } (\lambda s. Q (F s) s) \cap (\bigcap s. \text{ceilr } (Q (F s))) \vdash G, \{\!\{Q\}\!\}$

\langle proof \rangle

lemma *return*[*iag.intro*]:

shows $\text{prog.p2s } (\text{prog.return } v) \leq \{\!\{Q v\}\!\}, \text{ceilr } (Q v) \vdash G, \{\!\{Q\}\!\}$

\langle proof \rangle

lemma *write*[*iag.intro*]: — this is where *guarantee* obligations arise

shows $\text{prog.p2s } (\text{prog.write } F)$

$\leq \{\!\{\lambda s. Q () (F s) \wedge (s, F s) \in G\}\!\}, \text{ceilr } (\lambda s. Q () (F s) \wedge (s, F s) \in G) \cap \text{ceilr } (Q ()) \vdash G, \{\!\{Q\}\!\}$

\langle proof \rangle

lemma parallel: — not in the *iag* format; instantiate the first two assumptions

assumes *prog.p2s* $c_1 \leq \{P_1\}$, $A_1 \vdash G_1$, $\{Q_1\}$

assumes *prog.p2s* $c_2 \leq \{P_2\}$, $A_2 \vdash G_2$, $\{Q_2\}$

assumes $\bigwedge s. \llbracket Q_1 () s; Q_2 () s \rrbracket \implies Q () s$

assumes $G_2 \subseteq A_1$

assumes $G_1 \subseteq A_2$

assumes $G_1 \cup G_2 \subseteq G$

shows *prog.p2s* (*prog.parallel* $c_1 c_2$) $\leq \{P_1 \wedge P_2\}$, $A_1 \cap A_2 \vdash G$, $\{Q\}$

<proof>

lemmas local = ag.prog.local — not in the *iag* format

lemma if[iag.intro]:

assumes $b \implies \text{prog.p2s } c_1 \leq \{P_1\}$, $A_1 \vdash G$, $\{Q\}$

assumes $\neg b \implies \text{prog.p2s } c_2 \leq \{P_2\}$, $A_2 \vdash G$, $\{Q\}$

shows *prog.p2s* (*if* b *then* c_1 *else* c_2) $\leq \{\text{if } b \text{ then } P_1 \text{ else } P_2\}$, $A_1 \cap A_2 \vdash G$, $\{Q\}$

<proof>

lemma case-option[iag.intro]:

assumes $x = \text{None} \implies \text{prog.p2s } \text{none} \leq \{P_n\}$, $A_n \vdash G$, $\{Q\}$

assumes $\bigwedge v. x = \text{Some } v \implies \text{prog.p2s } (\text{some } v) \leq \{P_s v\}$, $A_s v \vdash G$, $\{Q\}$

shows *prog.p2s* (*case-option* none $\text{some } x$) $\leq \{\text{case } x \text{ of } \text{None} \Rightarrow P_n \mid \text{Some } v \Rightarrow P_s v\}$, *case-option* $A_n A_s x \vdash G$, $\{Q\}$

<proof>

lemma case-sum[iag.intro]:

assumes $\bigwedge v. x = \text{Inl } v \implies \text{prog.p2s } (\text{left } v) \leq \{P_l v\}$, $A_l v \vdash G$, $\{Q\}$

assumes $\bigwedge v. x = \text{Inr } v \implies \text{prog.p2s } (\text{right } v) \leq \{P_r v\}$, $A_r v \vdash G$, $\{Q\}$

shows *prog.p2s* (*case-sum* left $\text{right } x$) $\leq \{\text{case-sum } P_l P_r x\}$, *case-sum* $A_l A_r x \vdash G$, $\{Q\}$

<proof>

lemma case-list[iag.intro]:

assumes $x = [] \implies \text{prog.p2s } \text{nil} \leq \{P_n\}$, $A_n \vdash G$, $\{Q\}$

assumes $\bigwedge v \text{ vs}. x = v \# \text{vs} \implies \text{prog.p2s } (\text{cons } v \text{ vs}) \leq \{P_c v \text{ vs}\}$, $A_c v \text{ vs} \vdash G$, $\{Q\}$

shows *prog.p2s* (*case-list* nil $\text{cons } x$) $\leq \{\text{case-list } P_n P_c x\}$, *case-list* $A_n A_c x \vdash G$, $\{Q\}$

<proof>

lemma while:

fixes $c :: 'k \Rightarrow ('s, 'k + 'v) \text{ prog}$

assumes $c: \bigwedge k. \text{prog.p2s } (c k) \leq \{P k\}$, $A \vdash G$, $\{\text{case-sum } I Q\}$

shows *prog.p2s* (*prog.while* $c k$) $\leq \{\bigwedge v s. I v s \longrightarrow P v s \wedge I k\}$, $A \cap (\bigcap v. \text{ceilr } (Q v)) \vdash G$, $\{Q\}$

<proof>

lemmas whenM = iag.if[where $c_1=c$ and $A_1=A$ and $P_1=P$, OF - iag.return[where $v=()$] for $A c P$

<ML>

19.2 Refinement with relational assumes

Two sets of refinement rules:

- relational assumes
- relational assumes and *prog.sinvmap* (inverse state abstraction)

<ML>

lemma bind:

assumes $\bigwedge v. \text{prog.p2s } (g \ v) \leq \{\{Q' \ v\}\}$, $\text{ag.assm } A \vdash \text{prog.p2s } (g' \ v), \{\{Q\}\}$
assumes $\text{prog.p2s } f \leq \{\{P\}\}$, $\text{ag.assm } A \vdash \text{prog.p2s } f', \{\{Q'\}\}$
shows $\text{prog.p2s } (f \ggg g) \leq \{\{P\}\}$, $\text{ag.assm } A \vdash \text{prog.p2s } (f' \ggg g'), \{\{Q\}\}$
 <proof>

lemmas $\text{rev-bind} = \text{rar.prog.bind}[\text{rotated}]$

lemma *action*:

fixes $F :: ('v \times 's \times 's) \text{ set}$
fixes $F' :: ('v \times 's \times 's) \text{ set}$
assumes $Q: \bigwedge v \ s \ s'. \llbracket P \ s; (v, s, s') \in F \rrbracket \implies Q \ v \ s'$
assumes $F': \bigwedge v \ s \ s'. \llbracket P \ s; (v, s, s') \in F \rrbracket \implies (v, s, s') \in F'$
assumes $sP: \text{stable } A \ P$
assumes $sQ: \bigwedge v \ s \ s'. \llbracket P \ s; (v, s, s') \in F \rrbracket \implies \text{stable } A \ (Q \ v)$
shows $\text{prog.p2s } (\text{prog.action } F) \leq \{\{P\}\}$, $\text{ag.assm } A \vdash \text{prog.p2s } (\text{prog.action } F'), \{\{Q\}\}$
 <proof>

lemma *return*:

assumes $sQ: \text{stable } A \ (Q \ v)$
shows $\text{prog.p2s } (\text{prog.return } v) \leq \{\{Q \ v\}\}$, $\text{ag.assm } A \vdash \text{prog.p2s } (\text{prog.return } v), \{\{Q\}\}$
 <proof>

lemma *parallel-refinement*:

assumes $c_1: \text{prog.p2s } c_1 \leq \{\{P_1\}\}$, $\text{ag.assm } (A \cup G_2) \vdash \text{prog.p2s } (c_1' \sqcap \text{prog.rel } G_1), \{\{Q_1\}\}$
assumes $c_2: \text{prog.p2s } c_2 \leq \{\{P_2\}\}$, $\text{ag.assm } (A \cup G_1) \vdash \text{prog.p2s } (c_2' \sqcap \text{prog.rel } G_2), \{\{Q_2\}\}$
shows $\text{prog.p2s } (c_1 \parallel c_2) \leq \{\{P_1 \wedge P_2\}\}$, $\text{ag.assm } A \vdash \text{prog.p2s } (c_1' \sqcap \text{prog.rel } G_1 \parallel c_2' \sqcap \text{prog.rel } G_2), \{\{\lambda v. Q_1 \ v \wedge Q_2 \ v\}\}$
 <proof>

lemma *parallel*:

assumes $\text{prog.p2s } c_1 \leq \{\{P_1\}\}$, $\text{ag.assm } (A \cup G_2) \vdash \text{prog.p2s } c_1', \{\{Q_1\}\}$
assumes $\text{prog.p2s } c_1 \leq \{\{P_1\}\}$, $A \cup G_2 \vdash G_1, \{\{\top\}\}$
assumes $\text{prog.p2s } c_2 \leq \{\{P_2\}\}$, $\text{ag.assm } (A \cup G_1) \vdash \text{prog.p2s } c_2', \{\{Q_2\}\}$
assumes $\text{prog.p2s } c_2 \leq \{\{P_2\}\}$, $A \cup G_1 \vdash G_2, \{\{\top\}\}$
shows $\text{prog.p2s } (c_1 \parallel c_2) \leq \{\{P_1 \wedge P_2\}\}$, $\text{ag.assm } A \vdash \text{prog.p2s } (c_1' \parallel c_2'), \{\{\lambda v. Q_1 \ v \wedge Q_2 \ v\}\}$
 <proof>

lemma *while*:

fixes $c :: 'k \Rightarrow ('s, 'k + 'v) \text{ prog}$
fixes $c' :: 'k \Rightarrow ('s, 'k + 'v) \text{ prog}$
assumes $c: \bigwedge k. \text{prog.p2s } (c \ k) \leq \{\{P \ k\}\}$, $\text{ag.assm } A \vdash \text{prog.p2s } (c' \ k), \{\{\text{case-sum } I \ Q\}\}$
assumes $IP: \bigwedge s \ v. I \ v \ s \implies P \ v \ s$
assumes $sQ: \bigwedge v. \text{stable } A \ (Q \ v)$
shows $\text{prog.p2s } (\text{prog.while } c \ k) \leq \{\{I \ k\}\}$, $\text{ag.assm } A \vdash \text{prog.p2s } (\text{prog.while } c' \ k), \{\{Q\}\}$
 <proof>

lemma *app*:

fixes $xs :: 'a \text{ list}$
fixes $f :: 'a \Rightarrow ('s, \text{unit}) \text{ prog}$
fixes $P :: 'a \text{ list} \Rightarrow 's \text{ pred}$
assumes $\bigwedge x \ ys \ zs. xs = ys @ x \# zs \implies \text{prog.p2s } (f \ x) \leq \{\{P \ ys\}\}$, $\text{ag.assm } A \vdash \text{prog.p2s } (f' \ x), \{\{\lambda-. P \ (ys @ [x])\}\}$
assumes $\bigwedge ys. \text{prefix } ys \ xs \implies \text{stable } A \ (P \ ys)$
shows $\text{prog.p2s } (\text{prog.app } f \ xs) \leq \{\{P \ []\}\}$, $\text{ag.assm } A \vdash \text{prog.p2s } (\text{prog.app } f' \ xs), \{\{\lambda-. P \ xs\}\}$
 <proof>

lemmas $\text{if} = \text{refinement.prog.if}[\text{where } A = \text{ag.assm } A \ \text{for } A]$

lemmas $\text{case-option} = \text{refinement.prog.case-option}[\text{where } A = \text{ag.assm } A \ \text{for } A]$

$\langle ML \rangle$

abbreviation (*input*) $absfn\ sf\ c \equiv prog.p2s\ (prog.sinvmap\ sf\ c)$

lemma *bind*:

assumes $\bigwedge v. prog.p2s\ (g\ v) \leq \{\!\{Q'\ v\}\!\}$, $ag.assm\ A \vdash rair.prog.absfn\ sf\ (g'\ v), \{\!\{Q\}\!\}$
assumes $prog.p2s\ f \leq \{\!\{P\}\!\}$, $ag.assm\ A \vdash rair.prog.absfn\ sf\ f', \{\!\{Q'\}\!\}$
shows $prog.p2s\ (f \ggg g) \leq \{\!\{P\}\!\}$, $ag.assm\ A \vdash rair.prog.absfn\ sf\ (f' \ggg g'), \{\!\{Q\}\!\}$
 $\langle proof \rangle$

lemmas *rev-bind* = $rair.prog.bind[rotated]$

lemma *action*:

fixes $F :: ('v \times 's \times 's)\ set$
fixes $F' :: ('v \times 't \times 't)\ set$
fixes $sf :: 's \Rightarrow 't$
assumes $Q: \bigwedge v\ s\ s'. \llbracket P\ s; (v, s, s') \in F \rrbracket \Longrightarrow Q\ v\ s'$
assumes $F': \bigwedge v\ s\ s'. \llbracket P\ s; (v, s, s') \in F \rrbracket \Longrightarrow (v, sf\ s, sf\ s') \in F'$
assumes $sP: stable\ A\ P$
assumes $sQ: \bigwedge v\ s\ s'. \llbracket P\ s; (v, s, s') \in F \rrbracket \Longrightarrow stable\ A\ (Q\ v)$
shows $prog.p2s\ (prog.action\ F) \leq \{\!\{P\}\!\}$, $ag.assm\ A \vdash rair.prog.absfn\ sf\ (prog.action\ F'), \{\!\{Q\}\!\}$
 $\langle proof \rangle$

lemma *return*:

assumes $sQ: stable\ A\ (Q\ v)$
shows $prog.p2s\ (prog.return\ v) \leq \{\!\{Q\ v\}\!\}$, $ag.assm\ A \vdash rair.prog.absfn\ sf\ (prog.return\ v), \{\!\{Q\}\!\}$
 $\langle proof \rangle$

lemma *parallel*:

fixes $sf :: 's \Rightarrow 't$
assumes $prog.p2s\ c_1 \leq \{\!\{P_1\}\!\}$, $ag.assm\ (A \cup G_2) \vdash rair.prog.absfn\ sf\ c_1', \{\!\{Q_1\}\!\}$
assumes $prog.p2s\ c_1 \leq \{\!\{P_1\}\!\}$, $A \cup G_2 \vdash G_1, \{\!\{\top\}\!\}$
assumes $prog.p2s\ c_2 \leq \{\!\{P_2\}\!\}$, $ag.assm\ (A \cup G_1) \vdash rair.prog.absfn\ sf\ c_2', \{\!\{Q_2\}\!\}$
assumes $prog.p2s\ c_2 \leq \{\!\{P_2\}\!\}$, $A \cup G_1 \vdash G_2, \{\!\{\top\}\!\}$
shows $prog.p2s\ (c_1 \parallel c_2) \leq \{\!\{P_1 \wedge P_2\}\!\}$, $ag.assm\ A \vdash rair.prog.absfn\ sf\ (c_1' \parallel c_2'), \{\!\{\lambda v. Q_1\ v \wedge Q_2\ v\}\!\}$
 $\langle proof \rangle$

lemma *while*:

fixes $c :: 'k \Rightarrow ('s, 'k + 'v)\ prog$
fixes $c' :: 'k \Rightarrow ('t, 'k + 'v)\ prog$
fixes $sf :: 's \Rightarrow 't$
assumes $c: \bigwedge k. prog.p2s\ (c\ k) \leq \{\!\{P\ k\}\!\}$, $ag.assm\ A \vdash rair.prog.absfn\ sf\ (c'\ k), \{\!\{case-sum\ I\ Q\}\!\}$
assumes $IP: \bigwedge s\ v. I\ v\ s \Longrightarrow P\ v\ s$
assumes $sQ: \bigwedge v. stable\ A\ (Q\ v)$
shows $prog.p2s\ (prog.while\ c\ k) \leq \{\!\{I\ k\}\!\}$, $ag.assm\ A \vdash rair.prog.absfn\ sf\ (prog.while\ c'\ k), \{\!\{Q\}\!\}$
 $\langle proof \rangle$

lemma *app*:

fixes $xs :: 'a\ list$
fixes $f :: 'a \Rightarrow ('s, unit)\ prog$
fixes $P :: 'a\ list \Rightarrow 's\ pred$
assumes $\bigwedge x\ ys\ zs. xs = ys @ x \# zs \Longrightarrow prog.p2s\ (f\ x) \leq \{\!\{P\ ys\}\!\}$, $ag.assm\ A \vdash rair.prog.absfn\ sf\ (f'\ x), \{\!\{\lambda-. P\ (ys @ [x])\}\!\}$
assumes $\bigwedge ys. prefix\ ys\ xs \Longrightarrow stable\ A\ (P\ ys)$
shows $prog.p2s\ (prog.app\ f\ xs) \leq \{\!\{P\ []\}\!\}$, $ag.assm\ A \vdash rair.prog.absfn\ sf\ (prog.app\ f'\ xs), \{\!\{\lambda-. P\ xs\}\!\}$
 $\langle proof \rangle$

lemma *if*:

assumes $i \implies \text{prog.p2s } t \leq \{P\}$, $\text{ag.assm } A \Vdash \text{rair.prog.absfn sf } t'$, $\{Q\}$

assumes $\neg i \implies \text{prog.p2s } e \leq \{P'\}$, $\text{ag.assm } A \Vdash \text{rair.prog.absfn sf } e'$, $\{Q\}$

shows $\text{prog.p2s } (\text{if } i \text{ then } t \text{ else } e) \leq \{\text{if } i \text{ then } P \text{ else } P'\}$, $\text{ag.assm } A \Vdash \text{rair.prog.absfn sf } (\text{if } i \text{ then } t' \text{ else } e')$, $\{Q\}$

$\langle \text{proof} \rangle$

lemma *case-option*:

assumes $\text{opt} = \text{None} \implies \text{prog.p2s } \text{none} \leq \{P_n\}$, $\text{ag.assm } A \Vdash \text{rair.prog.absfn sf } \text{none}'$, $\{Q\}$

assumes $\bigwedge v. \text{opt} = \text{Some } v \implies \text{prog.p2s } (\text{some } v) \leq \{P_s v\}$, $\text{ag.assm } A \Vdash \text{rair.prog.absfn sf } (\text{some}' v)$, $\{Q\}$

shows $\text{prog.p2s } (\text{case-option } \text{none } \text{some } \text{opt}) \leq \{\text{case } \text{opt} \text{ of } \text{None} \Rightarrow P_n \mid \text{Some } v \Rightarrow P_s v\}$, $\text{ag.assm } A \Vdash \text{rair.prog.absfn sf } (\text{case-option } \text{none}' \text{ some}' \text{opt})$, $\{Q\}$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

20 Wickerson, Dodds and Parkinson: explicit stabilisation

Notes on [Wickerson, Dodds, and Parkinson \(2010\)](#) (all references here are to the technical report):

- motivation: techniques for eliding redundant stability conditions
 - the standard rules check the interstitial assertion in c ; d twice
- they claim in §7 to supersede the “mid stability” of [Vafeiadis \(2008, §4.1\)](#) (wssa, sswa)
- Appendix D:
 - not a complete set of rules
 - ATOMR-S does not self-compose: consider c ; d – the interstitial assertion is either a floor or ceiling
 - * every step therefore requires a use of weakening/monotonicity

The basis of their approach is to make assertions a function of a relation (a *rely*). By considering a set of relations, a single rely-guarantee specification can satisfy several call sites. Separately they tweak the RGSep rules of [Vafeiadis \(2008\)](#).

The definitions are formally motivated as follows (§3):

Our operators can also be defined using Dijkstra’s predicate transformer semantics: $\lfloor p \rfloor R$ is the weakest precondition of R^* given postcondition p , while $\lceil p \rceil R$ is the strongest postcondition of R^* given precondition p .

The following adapts their definitions and proofs to our setting.

$\langle \text{ML} \rangle$

definition *floor* :: $'a \text{ rel} \Rightarrow 'a \text{ pred} \Rightarrow 'a \text{ pred}$ **where** — An interior operator, or a closure in the dual lattice
 $\text{floor } r P s \iff (\forall s'. (s, s') \in r^* \longrightarrow P s')$

definition *ceiling* :: $'a \text{ rel} \Rightarrow 'a \text{ pred} \Rightarrow 'a \text{ pred}$ **where** — A closure operator
 $\text{ceiling } r P s \iff (\exists s'. (s', s) \in r^* \wedge P s')$

$\langle \text{ML} \rangle$

lemma *empty-rel[simp]*:

shows $\text{wdp.floor } \{\} P = P$

$\langle \text{proof} \rangle$

lemma *reflcl*:

shows $\text{wdp.floor } (r^=) = \text{wdp.floor } r$
 $\langle \text{proof} \rangle$

lemma *const*:

shows $\text{wdp.floor } r \langle c \rangle = \langle c \rangle$
 $\langle \text{proof} \rangle$

lemma *contractive*:

shows $\text{wdp.floor } r P \leq P$
 $\langle \text{proof} \rangle$

lemma *idempotent*:

shows $\text{wdp.floor } r (\text{wdp.floor } r P) = \text{wdp.floor } r P$
 $\langle \text{proof} \rangle$

lemma *mono*:

assumes $r' \subseteq r$
assumes $P \leq P'$
shows $\text{wdp.floor } r P \leq \text{wdp.floor } r' P'$
 $\langle \text{proof} \rangle$

lemma *strengthen[stg]*:

assumes $\text{st-ord } (\neg F) r r'$
assumes $\text{st-ord } F P P'$
shows $\text{st-ord } F (\text{wdp.floor } r P) (\text{wdp.floor } r' P')$
 $\langle \text{proof} \rangle$

lemma *weakest*:

assumes $Q \leq P$
assumes $\text{stable } r Q$
shows $Q \leq \text{wdp.floor } r P$
 $\langle \text{proof} \rangle$

lemma *Chernoff*:

assumes $P \leq Q$
shows $(\text{wdp.floor } r P \wedge Q) \leq \text{wdp.floor } r Q$
 $\langle \text{proof} \rangle$

lemma *floor1*:

assumes $r \subseteq r'$
shows $\text{wdp.floor } r' (\text{wdp.floor } r P) = \text{wdp.floor } r' P$
 $\langle \text{proof} \rangle$

lemma *floor2*:

assumes $r \subseteq r'$
shows $\text{wdp.floor } r (\text{wdp.floor } r' P) = \text{wdp.floor } r' P$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

interpretation *ceiling: closure-complete-lattice-distributive-class* $\text{wdp.ceiling } r$ **for** r

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *empty-rel[simp]*:

shows $\text{wdp.ceiling } \{ \} P = P$
 $\langle \text{proof} \rangle$

lemma reflcl:

shows $\text{wdp.ceiling } (r^-) = \text{wdp.ceiling } r$
 $\langle \text{proof} \rangle$

lemma const:

shows $\text{wdp.ceiling } r \langle c \rangle = \langle c \rangle$
 $\langle \text{proof} \rangle$

lemma mono:

assumes $r \subseteq r'$
assumes $P \leq P'$
shows $\text{wdp.ceiling } r P \leq \text{wdp.ceiling } r' P'$
 $\langle \text{proof} \rangle$

lemma strengthen[*strg*]:

assumes $\text{st-ord } F r r'$
assumes $\text{st-ord } F P P'$
shows $\text{st-ord } F (\text{wdp.ceiling } r P) (\text{wdp.ceiling } r' P')$
 $\langle \text{proof} \rangle$

lemma strongest:

assumes $P \leq Q$
assumes $\text{stable } r Q$
shows $\text{wdp.ceiling } r P \leq Q$
 $\langle \text{proof} \rangle$

lemma ceiling1:

assumes $r \subseteq r'$
shows $\text{wdp.ceiling } r' (\text{wdp.ceiling } r P) = \text{wdp.ceiling } r' P$
 $\langle \text{proof} \rangle$

lemma ceiling2:

assumes $r \subseteq r'$
shows $\text{wdp.ceiling } r (\text{wdp.ceiling } r' P) = \text{wdp.ceiling } r' P$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma floor:

shows $\text{stable } r (\text{wdp.floor } r P)$
 $\langle \text{proof} \rangle$

lemma ceiling:

shows $\text{stable } r (\text{wdp.ceiling } r P)$
 $\langle \text{proof} \rangle$

lemma floor-conv:

assumes $\text{stable } r P$
shows $P = \text{wdp.floor } r P$
 $\langle \text{proof} \rangle$

lemma ceiling-conv:

assumes $\text{stable } r P$
shows $P = \text{wdp.ceiling } r P$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *floor-alt-def*: — Wickerson et al. (2010, §3)
shows $\text{wdp.floor } r P = \bigsqcup \{Q. Q \leq P \wedge \text{stable } r Q\}$
 $\langle \text{proof} \rangle$

lemma *ceiling-alt-def*: — Wickerson et al. (2010, §3)
shows $\text{wdp.ceiling } r P = \bigsqcap \{Q. P \leq Q \wedge \text{stable } r Q\}$
 $\langle \text{proof} \rangle$

lemma *duality-floor-ceiling*:
shows $\text{wdp.ceiling } r (\neg P) = (\neg \text{wdp.floor } (r^{-1}) P)$
 $\langle \text{proof} \rangle$

lemma *ceiling-floor*:
assumes $r \subseteq r'$
shows $\text{wdp.ceiling } r (\text{wdp.floor } r' P) = \text{wdp.floor } r' P$
 $\langle \text{proof} \rangle$

lemma *floor-ceiling*:
assumes $r \subseteq r'$
shows $\text{wdp.floor } r (\text{wdp.ceiling } r' P) = \text{wdp.ceiling } r' P$
 $\langle \text{proof} \rangle$

lemma *floor-ceilr*:
shows $\text{wdp.floor } (\text{ceilr } P) P = P$
 $\langle \text{proof} \rangle$

lemma *ceiling-ceilr*:
shows $\text{wdp.ceiling } (\text{ceilr } P) P = P$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

20.1 Assume/Guarantee rules

§3.2 traditional assume/guarantee rules $\langle ML \rangle$

lemma *action*: — arbitrary A
fixes $F :: ('v \times 's \times 's)$ set
assumes $Q: \bigwedge v s s'. \llbracket P s; (v, s, s') \in F \rrbracket \Longrightarrow Q v s'$
assumes $G: \bigwedge v s s'. \llbracket P s; s \neq s'; (v, s, s') \in F \rrbracket \Longrightarrow (s, s') \in G$
shows $\text{prog.p2s } (\text{prog.action } F) \leq \{\!\{ \text{wdp.floor } A P \}\!\}, A \vdash G, \{\!\{ \lambda v. \text{wdp.ceiling } A (Q v) \}\!\}$
 $\langle \text{proof} \rangle$

lemmas $\text{mono} = \text{ag.mono}$
lemmas $\text{bind} = \text{ag.prog.bind}$

etc. – the other rules are stock

$\langle ML \rangle$

§4, Appendix C parametric specifications **definition** $\text{pag} :: ('s \text{ rel} \Rightarrow 's \text{ pred}) \Rightarrow 's \text{ rel set} \Rightarrow 's \text{ rel} \Rightarrow ('s \text{ rel} \Rightarrow 'v \Rightarrow 's \text{ pred}) \Rightarrow (\text{sequential}, 's, 'v) \text{ spec } (\{\!\{-\}\!\}, - / \vdash_P -, \{\!\{-\}\!\} [0,0,0,0] 100)$ **where**
 $\{\!\{P\}\!\}, As \vdash_P G, \{\!\{Q\}\!\} = (\bigsqcap A \in As. \{\!\{P A\}\!\}, A \vdash G, \{\!\{Q A\}\!\})$

$\langle ML \rangle$

lemma *empty*:
shows $\{\!\{P\}\!\}, \{\!\}\vdash_P G, \{\!\{Q\}\!\} = \top$

$\langle proof \rangle$

lemma singleton:

shows $\{\!\{P\}\!\}, \{A\} \vdash_P G, \{\!\{Q\}\!\} = \{\!\{P A\}\!\}, A \vdash G, \{\!\{Q A\}\!\}$

$\langle proof \rangle$

lemma mono: — strengthening of the WEAKEN rule in Figure 4, needed for the example

assumes $\bigwedge A. A \in As' \implies P' A \leq P A$

assumes $As' \leq As$

assumes $G \leq G'$

assumes $\bigwedge A. A \in As' \implies Q A \leq Q' A$

shows $\{\!\{P\}\!\}, As \vdash_P G, \{\!\{Q\}\!\} \leq \{\!\{P'\}\!\}, As' \vdash_P G', \{\!\{Q'\}\!\}$

$\langle proof \rangle$

lemma action: — allow assertions to depend on assume A , needed for the example

fixes $F :: ('v \times 's \times 's)$ set

assumes $Q: \bigwedge A v s s'. \llbracket A \in As; P A s; (v, s, s') \in F \rrbracket \implies Q A v s'$

assumes $G: \bigwedge A v s s'. \llbracket A \in As; P A s; s \neq s'; (v, s, s') \in F \rrbracket \implies (s, s') \in G$

shows $prog.p2s (prog.action F) \leq \{\!\{\lambda A. wdp.floor A (P A)\}\!\}, As \vdash_P G, \{\!\{\lambda A v. wdp.ceiling A (Q A v)\}\!\}$

$\langle proof \rangle$

lemmas $sup = ag.prog.sup$

lemma bind:

assumes $\bigwedge v. prog.p2s (g v) \leq \{\!\{\lambda A. Q' A v\}\!\}, As \vdash_P G, \{\!\{Q\}\!\}$

assumes $prog.p2s f \leq \{\!\{P\}\!\}, As \vdash_P G, \{\!\{Q'\}\!\}$

shows $prog.p2s (f \gg g) \leq \{\!\{P\}\!\}, As \vdash_P G, \{\!\{Q\}\!\}$

$\langle proof \rangle$

lemma parallel:

assumes $prog.p2s c_1 \leq \{\!\{P_1\}\!\}, (\cup) G_2 \text{ ' } A \vdash_P G_1, \{\!\{Q_1\}\!\}$

assumes $prog.p2s c_2 \leq \{\!\{P_2\}\!\}, (\cup) G_1 \text{ ' } A \vdash_P G_2, \{\!\{Q_2\}\!\}$

shows $prog.p2s (prog.parallel c_1 c_2)$

$\leq \{\!\{\lambda R. P_1 (R \cup G_2) \wedge P_2 (R \cup G_1)\}\!\}, A \vdash_P G_1 \cup G_2, \{\!\{\lambda R v. Q_1 (R \cup G_2) v \wedge Q_2 (R \cup G_1) v\}\!\}$

$\langle proof \rangle$

etc. – the other rules follow similarly

$\langle ML \rangle$

20.2 Examples

There is not always a single (traditional) most general assume/guarantee specification (§2.1).

type-synonym $state = int$ — just x

abbreviation $(input) incr \equiv prog.write ((+) 1)$ — atomic increment

abbreviation $(input) increases :: int rel$ **where** $increases \equiv \{(x, x'). x \leq x'\}$

lemma ag-incr1: — the precondition is stable as the rely is very strong

shows $prog.p2s incr \leq \{\!(=) c\!\}, \{\!\}\vdash increases, \{\!\{(=) (c + 1)\}\!\}$

$\langle proof \rangle$

lemma ag-incr2: — note the weaker precondition due to the larger assume

shows $prog.p2s incr \leq \{\!(\leq) c\!\}, increases \vdash increases, \{\!\{(\leq) (c + 1)\}\!\}$

$\langle proof \rangle$

lemma ag-incr1-par-incr1:

shows $prog.p2s (incr \parallel incr) \leq \{\!\{\lambda x. c \leq x\}\!\}, increases \vdash increases, \{\!\{\lambda x. c + 1 \leq x\}\!\}$

$\langle proof \rangle$

Using explicit stabilisation we can squash the two specifications for *incr* into a single one (§4).

lemma — postcondition cannot be simplified for arbitrary A

shows $\text{prog.p2s } \text{incr} \leq \{\!\{ \text{wdp.ceiling } A \text{ } ((=) \ c) \!\!\}, A \vdash \text{increases}, \{\!\{ \langle \text{wdp.ceiling } A \ (\lambda s. \text{wdp.ceiling } A \ ((=) \ c) \ (s - 1)) \rangle \!\!\}$

$\langle \text{proof} \rangle$

abbreviation (*input*) *comm-xpp* :: *int rel set where*

$\text{comm-xpp} \equiv \{A. \forall p \ s. \text{wdp.ceiling } A \ p \ (s - 1) = \text{wdp.ceiling } A \ (\lambda s. \ p \ (s - 1)) \ s\}$

lemma *pag-incr*: — postcondition can be simplified wrt *comm-xpp*

shows $\text{prog.p2s } \text{incr} \leq \{\!\{ \lambda A. \text{wdp.ceiling } A \ ((=) \ c) \!\!\}, \text{comm-xpp} \vdash_P \text{increases}, \{\!\{ \lambda A. \langle \text{wdp.ceiling } A \ ((=) \ (c + 1)) \rangle \!\!\}$

$\langle \text{proof} \rangle$

lemma

shows $\text{prog.p2s } \text{incr} \leq \{\!\{ (=) \ c \!\!\}, \{\!\{ \} \!\!\} \vdash \text{increases}, \{\!\{ (=) \ (c + 1) \!\!\}$

$\langle \text{proof} \rangle$

lemma

shows $\text{prog.p2s } \text{incr} \leq \{\!\{ (\leq) \ c \!\!\}, \text{increases} \vdash \text{increases}, \{\!\{ (\leq) \ (c + 1) \!\!\}$

$\langle \text{proof} \rangle$

21 Example: inhabitation

The following is a simple example of showing that a specification is inhabited.

lemma

shows $\langle 0::\text{nat}, [(self, 1), (self, 2)], \text{Some } () \rangle$
 $\leq \text{prog.p2s } (\text{prog.while } \langle \text{prog.write } ((+) \ 1) \gg (\text{prog.return } (\text{Inl } ())) \sqcup \text{prog.return } (\text{Inr } ())) \langle () \rangle$

$\langle \text{proof} \rangle$

$\langle \text{proof} \rangle$

22 Example: findP

We demonstrate assume/guarantee reasoning by showing the safety of *findP*, a classic exercise in concurrency verification. It has been treated by at least:

- [Karp and Miller \(1969, Example 5.1\)](#)
- [Rosen \(1976, §3\)](#)
- [Owicki and Gries \(1976, §4 Example 2\)](#)
- [Jones \(1983, §2.4\)](#)
- [Xu et al. \(1994, §3.1\)](#)
- [Brookes \(1996, p161\)](#) (no proof)
- [de Roeвер et al. \(2001, Examples 3.57 and 8.26\)](#) (atomic guarded commands)
- [Dingel \(2002, §6.2\)](#) (refinement)
- [Prensa Nieto \(2003, §10\)](#) (mechanized, arbitrary number of threads)
- [Apt, de Boer, and Olderog \(2009, §7.4, §8.6\)](#)
- [Hayes and Jones \(2017, §4\)](#) (refinement)

We take the task to be of finding the first element of a given array A that satisfies a given predicate *pred*, if it exists, or yielding *length A* if it does not. This search is performed with two threads: one searching the even indices and the other the odd. There is the possibility of a thread terminating early if it notices that the other thread has found a better candidate than it could.

We generalise previous treatments by allowing the predicate to be specified modularly and to be a function of the state. It is required to be pure, i.e., it cannot change the observable/shared state, though it could have its own local state.

Our search loops are defined recursively; one could just as easily use *prog.while*. We use a list and not an array for simplicity – at this level of abstraction there is no difference – and a mix of variables, where the monadic ones are purely local and the state-based are shared between the threads. The lens allows the array to be a value or reside in the (observable/shared) state.

type-synonym $'s \text{ state} = (\text{nat} \times \text{nat}) \times 's$

abbreviation $\text{foundE} :: \text{nat} \Rightarrow 's \text{ state}$ **where** $\text{foundE} \equiv \text{fst}_L ;_L \text{fst}_L$

abbreviation $\text{foundO} :: \text{nat} \Rightarrow 's \text{ state}$ **where** $\text{foundO} \equiv \text{snd}_L ;_L \text{fst}_L$

context

fixes $\text{pred} :: 'a \Rightarrow ('s, \text{bool}) \text{ prog}$

fixes $\text{predPre} :: 's \text{ pred}$

fixes $\text{predP} :: 'a \Rightarrow 's \text{ pred}$

fixes $A :: 's \text{ rel}$

fixes $\text{array} :: 'a \text{ list} \Rightarrow 's$

— A guarantee of *Id* indicates that $\text{pred } a$ is observationally pure.

assumes $\text{iag-pred}: \bigwedge a. \text{prog.p2s} (\text{pred } a) \leq \{\{\text{predPre} \wedge \langle a \rangle \in \text{SET } \text{getarray}\}\}, A^\# \cap \text{Id}_{\text{getarray}} \cap \text{ceilr } \text{predPre} \cap \text{Id}_{\text{predP } a} \vdash \text{Id}, \{\{\lambda rv. \langle rv \rangle = \text{predP } a\}\}$

begin

abbreviation $\text{array}' :: 'a \text{ list} \Rightarrow 's \text{ state}$ **where** $\text{array}' \equiv \text{array} ;_L \text{snd}_L$

partial-function (*lfp*) $\text{findP-loop-evens} :: \text{nat} \Rightarrow ('s \text{ state}, \text{unit}) \text{ prog}$ **where**

$\text{findP-loop-evens } i =$

$\text{do } \{ fO \leftarrow \text{prog.read } \text{get}_{\text{foundO}}$
 $;\text{ prog.whenM } (i < fO)$

$(\text{do } \{ v \leftarrow \text{prog.read } (\lambda s. \text{get}_{\text{array}'} s ! i)$
 $;\text{ b} \leftarrow \text{prog.localize } (\text{pred } v)$

$;\text{ if } b \text{ then } \text{prog.write } (\lambda s. \text{put}_{\text{foundE}} s i) \text{ else } \text{findP-loop-evens } (i + 2)$
 $\})$

$\}$

partial-function (*lfp*) $\text{findP-loop-odds} :: \text{nat} \Rightarrow ('s \text{ state}, \text{unit}) \text{ prog}$ **where**

$\text{findP-loop-odds } i =$

$\text{do } \{ fE \leftarrow \text{prog.read } \text{get}_{\text{foundE}}$
 $;\text{ prog.whenM } (i < fE)$

$(\text{do } \{ v \leftarrow \text{prog.read } (\lambda s. \text{get}_{\text{array}'} s ! i)$
 $;\text{ b} \leftarrow \text{prog.localize } (\text{pred } v)$

$;\text{ if } b \text{ then } \text{prog.write } (\lambda s. \text{put}_{\text{foundO}} s i) \text{ else } \text{findP-loop-odds } (i + 2)$
 $\})$

$\}$

definition $\text{findP} :: ('s, \text{nat}) \text{ prog}$ **where**

$\text{findP} = \text{prog.local } ($

$\text{do } \{ N \leftarrow \text{prog.read } (\text{SIZE } \text{get}_{\text{array}'})$

$;\text{ prog.write } (\lambda s. \text{put}_{\text{foundE}} s N)$

$;\text{ prog.write } (\lambda s. \text{put}_{\text{foundO}} s N)$

$;\text{ (findP-loop-evens } 0 \parallel \text{findP-loop-odds } 1)$

$;\text{ fE} \leftarrow \text{prog.read } (\text{get}_{\text{foundE}})$

$;\text{ fO} \leftarrow \text{prog.read } (\text{get}_{\text{foundO}})$

$;\text{ prog.return } (\text{min } fE \text{ fO})$

$\})$

Relies and guarantees **abbreviation** $(\text{input}) A' :: 's \text{ rel}$ **where** $A' \equiv A^\# \cap \text{ceilr } \text{predPre} \cap (\bigcap a. \text{Id}_{\text{predP } a})$

definition AE :: 's state rel where

$$AE = UNIV \times_R A' \cap Id_{get_array'} \cap Id_{get_foundE} \cap \leq_{get_foundO}$$

definition GE :: 's state rel where

$$GE = Id_{snd} \cap Id_{get_foundO} \cap \leq_{get_foundE}$$

definition AO :: 's state rel where

$$AO = UNIV \times_R A' \cap Id_{get_array'} \cap Id_{get_foundO} \cap \leq_{get_foundE}$$

definition GO :: 's state rel where

$$GO = Id_{snd} \cap Id_{get_foundE} \cap \leq_{get_foundO}$$

lemma AG -refl-trans:

shows

refl AE

refl AO

trans $A \implies$ trans AE

trans $A \implies$ trans AO

refl GE

refl GO

trans GE

trans GO

\langle proof \rangle

lemma AG -containment:

shows $GO \subseteq AE$

and $GE \subseteq AO$

\langle proof \rangle

lemma G -containment:

shows $GE \cup GO \subseteq UNIV \times_R Id$

\langle proof \rangle

Safety proofs lemma ag -findP-loop-evens:

shows prog.p2s (findP-loop-evens i)

$\leq \{\{ \langle even\ i \rangle \wedge (\lambda s. predPre (snd\ s)) \wedge get_foundE = SIZE\ get_array' \wedge get_foundO \leq SIZE\ get_array' \}, AE \vdash GE,$

$\{\{ \lambda -. (get_foundE < SIZE\ get_array' \longrightarrow localize1\ predP\ \$\$ get_array' ! get_foundE)$

$\wedge (\forall j. \langle i \leq j \wedge even\ j \rangle \wedge \langle j \rangle < pred_min\ get_foundE\ get_foundO \longrightarrow \neg localize1\ predP\ \$\$ get_array'$

$\!\ \langle j \rangle \}\}$

\langle proof \rangle

lemma ag -findP-loop-odds:

shows prog.p2s (findP-loop-odds i)

$\leq \{\{ \langle odd\ i \rangle \wedge (\lambda s. predPre (snd\ s)) \wedge get_foundO = SIZE\ get_array' \wedge get_foundE \leq SIZE\ get_array' \}, AO \vdash GO,$

$\{\{ \lambda -. (get_foundO < SIZE\ get_array' \longrightarrow localize1\ predP\ \$\$ get_array' ! get_foundO)$

$\wedge (\forall j. \langle i \leq j \wedge odd\ j \rangle \wedge \langle j \rangle < pred_min\ get_foundE\ get_foundO \longrightarrow \neg localize1\ predP\ \$\$ get_array'$

$\!\ \langle j \rangle \}\}$

\langle proof \rangle

theorem ag -findP:

shows prog.p2s findP

$\leq \{\{ predPre \}, A' \cap Id_{get_array}$

$\vdash Id, \{\{ \lambda v\ s. v = (LEAST\ i. i < SIZE\ get_array\ s \longrightarrow predP (get_array\ s ! i)\ s) \}\}$

\langle proof \rangle

end

We conclude by showing how we can instantiate the above with a *coprime* predicate.

$\langle ML \rangle$

type-synonym $'s \text{ state} = (\text{nat} \times \text{nat}) \times 's$

abbreviation $x :: \text{nat} \implies 's \text{ gcd.state}$ **where** $x \equiv \text{fst}_L ;_L \text{fst}_L$

abbreviation $y :: \text{nat} \implies 's \text{ gcd.state}$ **where** $y \equiv \text{snd}_L ;_L \text{fst}_L$

definition $\text{seq} :: \text{nat} \Rightarrow \text{nat} \Rightarrow ('s, \text{nat}) \text{ prog}$ **where**

```

seq a b =
  prog.local (
    do { prog.write ( $\lambda s. \text{put}_{\text{gcd}.x} s a$ )
      ; prog.write ( $\lambda s. \text{put}_{\text{gcd}.y} s b$ )
      ; prog.while ( $\lambda -.$ 
        do {  $xv \leftarrow \text{prog.read } \text{get}_{\text{gcd}.x}$ 
          ;  $yv \leftarrow \text{prog.read } \text{get}_{\text{gcd}.y}$ 
          ; if  $xv = yv$ 
            then prog.return (Inr ())
            else (do { (if  $xv < yv$ 
                      then prog.write ( $\lambda s. \text{put}_{\text{gcd}.y} s (yv - xv)$ )
                      else prog.write ( $\lambda s. \text{put}_{\text{gcd}.x} s (xv - yv)$ ))
                ; prog.return (Inl ()) } )
          }) ()
      ; prog.read  $\text{get}_{\text{gcd}.x}$ 
    })

```

$\langle ML \rangle$

lemma seq :

shows $\text{prog.p2s} (\text{gcd.seq } a \ b) \leq \{\langle \text{True} \rangle\}, \text{UNIV} \vdash \text{Id}, \{\lambda v. \langle v = \text{gcd } a \ b \rangle\}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

definition $\text{findP-coprime} :: (\text{nat} \times \text{nat list}, \text{nat}) \text{ prog}$ **where**

$\text{findP-coprime} = \text{findP} (\lambda a. \text{prog.read } \text{get}_{\text{fst}_L} \gg \text{gcd.seq } a \gg (\lambda c. \text{prog.return } (c = 1))) \text{snd}_L$

lemma $\text{ag-findP-coprime}'$:

shows $\text{prog.p2s } \text{findP-coprime}$

$\leq \{\langle \text{True} \rangle\}, \text{Id}$

$\vdash \text{Id}, \{\lambda rv \ s. rv = (\text{LEAST } i. i < \text{length } (\text{get}_{\text{snd}_L} s) \longrightarrow \text{coprime } (\text{get}_{\text{fst}_L} s) (\text{get}_{\text{snd}_L} s ! i))\}$

$\langle \text{proof} \rangle$

lemma ag-findP-coprime : — Shuffle the parameter to the precondition, exploiting purity.

shows $\text{prog.p2s } \text{findP-coprime}$

$\leq \{\langle a \rangle = \text{get}_{\text{fst}_L}\}, \text{Id}$

$\vdash \text{Id}, \{\lambda rv \ s. rv = (\text{LEAST } i. i < \text{length } (\text{get}_{\text{snd}_L} s) \longrightarrow \text{coprime } a (\text{get}_{\text{snd}_L} s ! i))\}$

$\langle \text{proof} \rangle$

23 Example: data refinement (search)

We show a very simple example of data refinement: implementing sets with functional queues for breadth-first search (BFS). The objective here is to transfer a simple correctness property proven on the abstract level to the concrete level.

Observations:

- there is no concurrency in the BFS: this is just about data refinement

- however arbitrary interference is allowed
- the abstract level does not require the implementation of the pending set to make progress
- the concrete level does not require a representation invariant
- depth optimality is not shown

References:

- queue ADT: `$ISABELLE_HOME/src/Doc/Codegen/Introduction.thy`
- BFS verification:
 - J. C. Filliâtre http://toccata.lri.fr/gallery/vstte12_bfs.en.html
 - `$AFP/Refine_Monadic/examples/Breadth_First_Search.thy`
 - our model is quite different

$\langle ML \rangle$

```
record ('a, 's) interface =
  init :: ('s, unit) prog
  enq  :: 'a  $\Rightarrow$  ('s, unit) prog
  deq  :: ('s, 'a option) prog
```

```
type-synonym 'a abstract = 'a set
```

```
definition abstract :: ('a, 'a pending.abstract  $\times$  's) pending.interface where
  abstract =
    (| pending.interface.init = prog.write (map-prod <{}> id)
      , pending.interface.enq =  $\lambda x$ . prog.write (map-prod (insert x) id)
      , pending.interface.deq = prog.action ({(None, s, s) | s. fst s = {}}
                                              $\cup$  {(Some x, (insert x X, s), (X, s)) | X s x. True})
    |)
```

```
type-synonym 'a concrete = 'a list  $\times$  'a list — a queue
```

```
fun cdeq-update :: 'a pending.concrete  $\times$  's  $\Rightarrow$  'a option  $\times$  'a pending.concrete  $\times$  's where
  cdeq-update (([], []), s) = (None, (([], []), s))
| cdeq-update ((xs, []), s) = cdeq-update (([], rev xs), s)
| cdeq-update ((xs, y # ys), s) = (Some y, ((xs, ys), s))
```

```
definition concrete :: ('a, 'a pending.concrete  $\times$  's) pending.interface where
  concrete =
    (| pending.interface.init = prog.write (map-prod <([], [])> id)
      , pending.interface.enq =  $\lambda x$ . prog.write (map-prod (map-prod ((#) x) id) id)
      , pending.interface.deq = prog.det-action pending.cdeq-update
    |)
```

```
abbreviation absfn' :: 'a pending.concrete  $\Rightarrow$  'a list where — queue as a list
  absfn' s  $\equiv$  snd s @ rev (fst s)
```

```
definition absfn :: 'a pending.concrete  $\Rightarrow$  'a pending.abstract where
  absfn s = set (pending.absfn' s)
```

$\langle ML \rangle$

lemma *init*:

fixes $Q :: \text{unit} \Rightarrow 'a \text{ pending.abstract} \times 's \Rightarrow \text{bool}$
fixes $A :: 's \text{ rel}$
assumes $\text{stable} (Id \times_R A) (Q ())$
shows $\text{prog.p2s} (\text{pending.init pending.abstract}) \leq \{\!\{ \lambda s. Q () (\{\}, \text{snd } s) \}\!\}, Id \times_R A \vdash UNIV \times_R Id, \{\!\{ Q \}\!\}$
 $\langle \text{proof} \rangle$

lemma *enq*:

fixes $x :: 'a$
fixes $Q :: \text{unit} \Rightarrow 'a \text{ pending.abstract} \times 's \Rightarrow \text{bool}$
fixes $A :: 's \text{ rel}$
assumes $\text{stable} (Id \times_R A) (Q ())$
shows $\text{prog.p2s} (\text{pending.enq pending.abstract } x) \leq \{\!\{ \lambda s. Q () (\text{insert } x (\text{fst } s), \text{snd } s) \}\!\}, Id \times_R A \vdash UNIV \times_R Id, \{\!\{ Q \}\!\}$
 $\langle \text{proof} \rangle$

lemma *deq*:

fixes $Q :: 'a \text{ option} \Rightarrow 'a \text{ pending.abstract} \times 's \Rightarrow \text{bool}$
fixes $A :: 's \text{ rel}$
assumes $\bigwedge v. \text{stable} (Id \times_R A) (Q v)$
shows $\text{prog.p2s} (\text{pending.deq pending.abstract}) \leq \{\!\{ \lambda s. \text{if } \text{fst } s = \{\} \text{ then } Q \text{ None } s \text{ else } (\forall x X. \text{fst } s = \text{insert } x X \longrightarrow Q (\text{Some } x) (X, \text{snd } s)) \}\!\}, Id \times_R A \vdash UNIV \times_R Id, \{\!\{ Q \}\!\}$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

record $('a, 's) \text{ interface} =$

$\text{init} :: ('s, \text{unit}) \text{ prog}$
 $\text{ins} :: 'a \Rightarrow ('s, \text{unit}) \text{ prog}$
 $\text{mem} :: 'a \Rightarrow ('s, \text{bool}) \text{ prog}$

type-synonym $'a \text{ abstract} = 'a \text{ list}$ — model finite sets

definition $\text{abstract} :: ('a, 's \times 'a \text{ set.abstract} \times 't) \text{ set.interface}$ **where**

$\text{abstract} =$
 $\{\!\{ \text{set.interface.init} = \text{prog.write} (\text{map-prod } id (\text{map-prod } \langle [] \rangle id))$
 $, \text{set.interface.ins} = \lambda x. \text{prog.write} (\text{map-prod } id (\text{map-prod } (\langle \# \rangle x) id))$
 $, \text{set.interface.mem} = \lambda x. \text{prog.read} (\lambda s. x \in \text{set} (\text{fst} (\text{snd } s))) \}\!\}$
 $\}\!\}$

$\langle ML \rangle$

lemma *init*:

fixes $Q :: \text{unit} \Rightarrow 's \times 'a \text{ set.abstract} \times 't \Rightarrow \text{bool}$
fixes $A :: 's \text{ rel}$
assumes $\text{stable} (A \times_R Id \times_R B) (Q ())$
shows $\text{prog.p2s} (\text{set.init set.abstract}) \leq \{\!\{ \lambda s. Q () (\text{fst } s, [], \text{snd} (\text{snd } s)) \}\!\}, A \times_R Id \times_R B \vdash Id \times_R UNIV \times_R Id, \{\!\{ Q \}\!\}$
 $\langle \text{proof} \rangle$

lemma *ins*:

fixes $x :: 'a$
fixes $Q :: \text{unit} \Rightarrow 's \times 'a \text{ set.abstract} \times 't \Rightarrow \text{bool}$
fixes $A :: 's \text{ rel}$
assumes $\text{stable} (A \times_R Id \times_R B) (Q ())$
shows $\text{prog.p2s} (\text{set.ins set.abstract } x) \leq \{\!\{ \lambda s. Q () (\text{fst } s, x \# \text{fst} (\text{snd } s), \text{snd} (\text{snd } s)) \}\!\}, A \times_R Id \times_R B \vdash Id \times_R UNIV \times_R Id, \{\!\{ Q \}\!\}$
 $\langle \text{proof} \rangle$

lemma *mem*:

fixes $Q :: \text{bool} \Rightarrow 's \times 'a \text{ set.abstract} \times 't \Rightarrow \text{bool}$
assumes $\bigwedge v. \text{stable } (A \times_R \text{Id} \times_R B) (Q v)$
shows $\text{prog.p2s } (\text{set.mem } \text{set.abstract } x) \leq \llbracket \lambda s. Q (x \in \text{set } (\text{fst } (\text{snd } s))) \rrbracket, A \times_R \text{Id} \times_R B \vdash \text{Id} \times_R \text{UNIV} \times_R \text{Id}, \llbracket Q \rrbracket$
<proof>

<ML>

context

fixes $\text{pending} :: ('a, 'p \times 'a \text{ set.abstract} \times 's) \text{ pending.interface}$
fixes $f :: 'a \Rightarrow 'a \text{ list}$
begin

definition $\text{loop} :: 'a \text{ pred} \Rightarrow ('p \times 'a \text{ set.abstract} \times 's, 'a \text{ option}) \text{ prog where}$

$\text{loop } p =$
 $\text{prog.while } (\lambda -.$
 $\text{do } \{ \text{aopt} \leftarrow \text{pending.deq } \text{pending}$
 $\text{; case } \text{aopt of}$
 $\text{None} \Rightarrow \text{prog.return } (\text{Inr } \text{None})$
 $| \text{Some } x \Rightarrow$
 $\text{if } p x$
 $\text{then } \text{prog.return } (\text{Inr } (\text{Some } x))$
 $\text{else do } \{ \text{prog.app } (\lambda y. \text{do } \{ b \leftarrow \text{set.mem } \text{set.abstract } y;$
 $\text{prog.unlessM } b (\text{do } \{ \text{set.ins } \text{set.abstract } y$
 $\text{; pending.enq } \text{pending } y \}) \}$
 (f x)
 $\text{; prog.return } (\text{Inl } ())$
 $\}$
 $\}) ()$

definition $\text{main} :: 'a \text{ pred} \Rightarrow 'a \Rightarrow ('p \times 'a \text{ set.abstract} \times 's, 'a \text{ option}) \text{ prog where}$

$\text{main } p x =$
 $\text{do } \{$
 $\text{set.init } \text{set.abstract}$
 $\text{; pending.init } \text{pending}$
 $\text{; set.ins } \text{set.abstract } x$
 $\text{; pending.enq } \text{pending } x$
 $\text{; loop } p$
 $\}$

definition $\text{search} :: 'a \text{ pred} \Rightarrow 'a \Rightarrow ('s, 'a \text{ option}) \text{ prog where}$

$\text{search } p x = \text{prog.local } (\text{prog.local } (\text{main } p x))$

end

abbreviation $(\text{input}) \text{ aloop} \equiv \text{loop } \text{pending.abstract}$

abbreviation $(\text{input}) \text{ amain} \equiv \text{main } \text{pending.abstract}$

abbreviation $(\text{input}) \text{ asearch} \equiv \text{search } \text{pending.abstract}$

abbreviation $(\text{input}) \text{ bfs} \equiv \text{search } \text{pending.concrete}$

lemma

shows $\text{pending-g: } \text{UNIV} \times_R \text{Id} \subseteq \text{UNIV} \times_R \text{UNIV} \times_R \text{Id}$
and $\text{set-g: } \text{Id} \times_R \text{UNIV} \times_R \text{Id} \subseteq \text{UNIV} \times_R \text{UNIV} \times_R \text{Id}$
<proof>

context

fixes $f :: 'a \Rightarrow 'a \text{ list}$

fixes $P :: 'a \text{ pred}$

fixes $x_0 :: 'a$

begin

abbreviation (*input*) $step :: 'a \text{ rel}$ **where**

$step \equiv \{(x, y). y \in \text{set } (f x)\}$

abbreviation (*input*) $path :: 'a \text{ rel}$ **where**

$path \equiv step^*$

definition $aloop-invP :: 'a \text{ pending.abstract} \Rightarrow 'a \text{ set.abstract} \Rightarrow \text{bool}$ **where**

$aloop-invP \ q \ v \longleftrightarrow$

$q \subseteq \text{set } v$

$\wedge \text{set } v \subseteq \text{path} \text{ `` } \{x_0\}$

$\wedge \text{set } v \cap \text{Collect } P \subseteq q$

$\wedge x_0 \in \text{set } v$

definition $vclosureP :: 'a \Rightarrow 'a \text{ pending.abstract} \Rightarrow 'a \text{ set.abstract} \Rightarrow \text{bool}$ **where**

$vclosureP \ x \ q \ v \longleftrightarrow (x \in \text{set } v - q \longrightarrow \text{step} \text{ `` } \{x\} \subseteq \text{set } v)$

definition $search-postP :: 'a \text{ option} \Rightarrow \text{bool}$ **where**

$search-postP \ rv = (\text{case } rv \text{ of}$

$\text{None} \Rightarrow \text{finite } (\text{path} \text{ `` } \{x_0\}) \wedge (\text{path} \text{ `` } \{x_0\} \cap \text{Collect } P = \{\})$

$| \text{Some } y \Rightarrow (x_0, y) \in \text{path} \wedge P \ y)$

abbreviation $aloop-inv \ s \equiv aloop-invP \ (fst \ s) \ (fst \ (snd \ s))$

abbreviation $vclosure \ x \ s \equiv vclosureP \ x \ (fst \ s) \ (fst \ (snd \ s))$

abbreviation $search-post \ rv \equiv \langle search-postP \ rv \rangle$

lemma $vclosureP\text{-closed}$:

assumes $\text{set } v \subseteq \text{path} \text{ `` } \{x_0\}$

assumes $\forall y. vclosureP \ y \ \{\} \ v$

assumes $x_0 \in \text{set } v$

shows $\text{path} \text{ `` } \{x_0\} = \text{set } v$

$\langle \text{proof} \rangle$

lemma $vclosureP\text{-app}$:

assumes $\forall y. x \neq y \longrightarrow \text{local.vclosureP} \ y \ q \ v$

assumes $\text{set } (f \ x) \subseteq \text{set } v$

shows $vclosureP \ y \ q \ v$

$\langle \text{proof} \rangle$

lemma $vclosureP\text{-init}$:

shows $vclosureP \ x \ \{x_0\} \ [x_0]$

$\langle \text{proof} \rangle$

lemma $vclosureP\text{-step}$:

assumes $\forall z. x \neq z \longrightarrow vclosureP \ z \ q \ v$

assumes $x \neq z$

shows $vclosureP \ z \ (\text{insert } y \ q) \ (y \ \# \ v)$

$\langle \text{proof} \rangle$

lemma $vclosureP\text{-dequeue}$:

assumes $\forall z. vclosureP \ z \ (\text{insert } x \ q) \ v$

assumes $x \neq z$

shows $vclosureP \ z \ q \ v$

$\langle \text{proof} \rangle$

lemma *aloop-invPD*:

assumes *aloop-invP* $q\ v$

assumes $x \in q$

shows $(x_0, x) \in \text{path}$

$\langle \text{proof} \rangle$

lemma *aloop-invP-init*:

shows *aloop-invP* $\{x_0\}\ [x_0]$

$\langle \text{proof} \rangle$

lemma *aloop-invP-step*:

assumes *aloop-invP* $q\ v$

assumes $(x_0, x) \in \text{path}$

assumes $y \in \text{set } (f\ x) - \text{set } v$

shows *aloop-invP* $(\text{insert } y\ q)\ (y \# v)$

$\langle \text{proof} \rangle$

lemma *aloop-invP-dequeue*:

assumes *aloop-invP* $(\text{insert } x\ q)\ v$

assumes $\neg P\ x$

shows *aloop-invP* $q\ v$

$\langle \text{proof} \rangle$

lemma *search-postcond-None*:

assumes *aloop-invP* $\{\}\ v$

assumes $\forall y. \text{vclosureP } y\ \{\}\ v$

shows *search-postP* *None*

$\langle \text{proof} \rangle$

lemma *search-postcond-Some*:

assumes *aloop-invP* $q\ v$

assumes $x \in q$

assumes $P\ x$

shows *search-postP* $(\text{Some } x)$

$\langle \text{proof} \rangle$

lemmas *stable-simps* =

prod.sel

split-def

sum.simps

lemma *ag-aloop*:

shows *prog.p2s* $(\text{aloop } f\ P) \leq \{\{\text{aloop-inv} \wedge (\forall x. \text{vclosure } x)\}, \text{Id} \times_R \text{Id} \times_R \text{UNIV} \vdash \text{UNIV} \times_R \text{UNIV} \times_R \text{Id}, \{\{\text{search-post}\}\}$

$\langle \text{proof} \rangle$

lemma *ag-amain*:

shows *prog.p2s* $(\text{amain } f\ P\ x_0) \leq \{\{\langle \text{True} \rangle\}, \text{Id} \times_R \text{Id} \times_R \text{UNIV} \vdash \text{UNIV} \times_R \text{UNIV} \times_R \text{Id}, \{\{\text{search-post}\}\}$

$\langle \text{proof} \rangle$

lemma *ag-asearch*:

shows *prog.p2s* $(\text{asearch } f\ P\ x_0 :: ('s, 'a\ \text{option})\ \text{prog}) \leq \{\{\langle \text{True} \rangle\}, \text{UNIV} \vdash \text{Id}, \{\{\text{search-post}\}\}$

$\langle \text{proof} \rangle$

Refinement abbreviation $A \equiv \text{ag.assm } (\text{Id} \times_R \text{Id} \times_R \text{UNIV})$

abbreviation $\text{absfn } c \equiv \text{prog.sinvmap } (\text{map-prod } \text{pending.absfn } \text{id})\ c$

abbreviation $\text{p2s-absfn } c \equiv \text{prog.p2s } (\text{absfn } c)$

— visited set: reflexive

lemma *ref-set-init*:

shows $\text{prog.p2s } (\text{set.init set.abstract}) \leq \{\!\{ \lambda s. \text{True} \}\!\}, A \Vdash \text{p2s-absfn } (\text{set.init set.abstract}), \{\!\{ \lambda v s. \text{True} \}\!\}$
<proof>

lemma *ref-set-ins*:

shows $\text{prog.p2s } (\text{set.ins set.abstract } x) \leq \{\!\{ \lambda s. \text{True} \}\!\}, A \Vdash \text{p2s-absfn } (\text{set.ins set.abstract } x), \{\!\{ \lambda v s. \text{True} \}\!\}$
<proof>

lemma *ref-set-mem*:

shows $\text{prog.p2s } (\text{set.mem set.abstract } x) \leq \{\!\{ \lambda s. \text{True} \}\!\}, A \Vdash \text{p2s-absfn } (\text{set.mem set.abstract } x), \{\!\{ \lambda v s. \text{True} \}\!\}$
<proof>

lemma *ref-queue-init*:

shows $\text{prog.p2s } (\text{pending.init pending.concrete}) \leq \{\!\{ \lambda s. \text{True} \}\!\}, A \Vdash \text{p2s-absfn } (\text{pending.init pending.abstract}), \{\!\{ \lambda v s. \text{True} \}\!\}$
<proof>

lemma *ref-queue-enq*:

shows $\text{prog.p2s } (\text{pending.enq pending.concrete } x) \leq \{\!\{ \lambda s. \text{True} \}\!\}, A \Vdash \text{p2s-absfn } (\text{pending.enq pending.abstract } x), \{\!\{ \lambda v s. \text{True} \}\!\}$
<proof>

lemma *ref-queue-deq*:

shows $\text{prog.p2s } (\text{pending.deq pending.concrete}) \leq \{\!\{ \lambda s. \text{True} \}\!\}, A \Vdash \text{p2s-absfn } (\text{pending.deq pending.abstract}), \{\!\{ \lambda v s. \text{True} \}\!\}$
<proof>

lemma *ref-bfs-loop*:

shows $\text{prog.p2s } (\text{loop pending.concrete } f P) \leq \{\!\{ \lambda s. \text{True} \}\!\}, A \Vdash \text{p2s-absfn } (\text{loop pending.abstract } f P), \{\!\{ \lambda v s. \text{True} \}\!\}$
<proof>

lemma *ref-bfs-main*:

shows $\text{prog.p2s } (\text{main pending.concrete } f P x) \leq \{\!\{ \langle \text{True} \rangle \}\!\}, A \Vdash \text{p2s-absfn } (\text{amain } f P x), \{\!\{ \lambda v s. \text{True} \}\!\}$
<proof>

theorem *ref-bfs*:

shows $\text{bfs } f P x \leq \text{asearch } f P x$
<proof>

theorem *bfs-post-le*:

shows $\text{prog.p2s } (\text{bfs } f P x_0) \leq \text{spec.post } (\text{search-post})$
<proof>

end

24 Observations about safety closure

We demonstrate that *Sup* does not distribute in $(\prime a, \prime s, \prime v)$ *tl*s as it does in $(\prime a, \prime s, \prime v)$ *spec*: specifically a *Sup* of a set of safety properties in the former need not be a safety property, whereas in the latter it is (see §8.2).

corec $\text{bnats} :: \text{nat} \Rightarrow (\prime a \times \text{nat}, \prime v)$ *tl*list **where**

$\text{bnats } i = \text{TCons } (\text{undefined}, i) (\text{bnats } (\text{Suc } i))$

definition $\text{bnat} :: (\prime a, \text{nat}, \prime v)$ *behavior.t* **where**

$\text{bnat} = \text{behavior.B } 0 (\text{bnats } 1)$

definition $\text{tnats} :: \text{nat} \Rightarrow \text{nat} \Rightarrow (\prime a \times \text{nat})$ *list* **where**

$\text{tnats } i j = \text{map } (\text{Pair } \text{undefined}) (\text{upt } i j)$

definition $tnat :: nat \Rightarrow ('a, nat, 'v) trace.t$ **where**
 $tnat\ i = trace.T\ 0\ (tnats\ (Suc\ 0)\ (Suc\ i))\ None$

lemma $tnat-simps[simp]$:

shows $tnats\ i\ i = []$

and $trace.init\ (tnat\ i) = 0$

and $trace.rest\ (tnat\ i) = tnats\ 1\ (Suc\ i)$

and $length\ (tnats\ i\ j) = j - i$

$\langle proof \rangle$

lemma $take-tnats$:

shows $take\ i\ (tnats\ j\ k) = tnats\ j\ (\min\ (i + j)\ k)$

$\langle proof \rangle$

lemma $take-tnat$:

shows $trace.take\ i\ (tnat\ j) = tnat\ (\min\ i\ j)$

$\langle proof \rangle$

lemma $mono-tnat$:

shows $mono\ tnat$

$\langle proof \rangle$

lemma $final'-tnats$:

shows $trace.final'\ i\ (tnats\ j\ k) = (if\ j < k\ then\ k - 1\ else\ i)$

$\langle proof \rangle$

lemma $sset-tnat$:

shows $trace.sset\ (tnat\ i) = \{j. j \leq i\}$

$\langle proof \rangle$

lemma $natural'-tnats$:

shows $trace.natural'\ i\ (tnats\ (Suc\ i)\ (Suc\ j)) = tnats\ (Suc\ i)\ (Suc\ j)$

$\langle proof \rangle$

lemma $natural-tnat$:

shows $\natural(tnat\ i) = tnat\ i$

$\langle proof \rangle$

lemma $ttake-bnats$:

shows $ttake\ i\ (bnats\ j) = (tnats\ j\ (i + j), None)$

$\langle proof \rangle$

lemma $take-bnat-tnat$:

shows $behavior.take\ i\ bnat = tnat\ i$

$\langle proof \rangle$

unbundle $tls.extra-notation$

definition $bnat-approx :: (unit, nat, unit) spec\ set$ **where**

$bnat-approx = \{\langle behavior.take\ i\ bnat \rangle \mid i. True\}$

lemma $bnat-approx-alt-def$:

shows $bnat-approx = \{\langle tnat\ i \rangle \mid i. True\}$

$\langle proof \rangle$

lemma $not-tls-from-spec-Sup-distrib$: — $tls.from-spec$ is not Sup -distributive

shows $\neg tls.from-spec\ (\bigsqcup\ bnat-approx) \leq \bigsqcup (tls.from-spec\ ' bnat-approx)$ (**is** $\neg ?lhs \leq ?rhs$)

<proof>

definition $bnat' :: (unit, nat, unit) \text{ tls set where}$

$bnat' = \text{tls.from-spec ' bnat-approx}$

lemma *not-tls-safety-cl-Sup-distrib*: — tls.safety.cl is not *Sup*-distributive

shows $\neg \text{tls.safety.cl} (\bigsqcup bnat') \leq \bigsqcup (\text{tls.safety.cl ' bnat'})$

<proof>

definition $cl-bnat' :: (unit, nat, unit) \text{ tls set where}$

$cl-bnat' = \text{tls.safety.cl ' bnat'}$

lemma *not-tls-safety-closed-Sup*:

shows $cl-bnat' \subseteq \text{tls.safety.closed}$

and $\bigsqcup cl-bnat' \notin \text{tls.safety.closed}$

<proof>

Negation does not preserve tls.safety.closed notepad

begin

<proof>

end

24.1 Liveness

Famously arbitrary properties on infinite sequences can be decomposed into *safety* and *liveness* properties. The latter have been identified with the topologically dense sets.

References:

- [Alpern and Schneider \(1985\)](#); [Schneider \(1987\)](#): topological account
- [Kindler \(1994\)](#): overview
- [Lynch \(1996, §8.5.3\)](#)
- [Manolios and Trefler \(2003\)](#): lattice-theoretic account
- [Maier \(2004\)](#): an intuitionistic semantics for LTL (including next/X/⊙) over finite and infinite sequences

<ML>

lemma *dense-alt-def*: — [Lynch \(1996, §8.5.3 Liveness Property\)](#)

shows $(\text{raw.safety.dense} :: ('a, 's, 'v) \text{ behavior.t set set})$

$= \{P. \forall \sigma. \exists xsv. \sigma @_{-B} xsv \in P\}$ (**is** $?lhs = ?rhs$)

<proof>

<ML>

definition $live :: ('a, 's, 'v) \text{ tls set where}$

$live = \text{tls.safety.dense}$

<ML>

lemma *not-bot*:

shows $\perp \notin \text{tls.live}$

<proof>

lemma *top*:

shows $\top \in \text{tls.live}$
 $\langle \text{proof} \rangle$

lemma *live-le*:

assumes $P \in \text{tls.live}$

assumes $P \leq Q$

shows $Q \in \text{tls.live}$

$\langle \text{proof} \rangle$

lemma *inf-safety-eq-top*: — Lynch (1996, Theorem 8.8)

shows $\text{tls.live} \sqcap \text{tls.safety.closed} = \{\top\}$

$\langle \text{proof} \rangle$

lemma *terminating*:

shows $\text{tls.eventually } \text{tls.terminated} \in \text{tls.live}$

$\langle \text{proof} \rangle$

However this definition of liveness does not endorse traditional *response* properties.

corec *alternating* $:: \text{bool} \Rightarrow ('a \times \text{bool}, 'b) \text{ tlist}$ **where**

alternating $b = \text{TCons}(\text{undefined}, b) (\text{alternating } (\neg b))$

abbreviation $(\text{input } A \ b \equiv \text{behavior.B } b) (\text{tls.live.alternating } (\neg b))$

lemma *dropn-alternating*:

shows $\text{behavior.dropn } i (\text{tls.live.A } b) = \text{Some} (\text{tls.live.A } (\text{if even } i \text{ then } b \text{ else } \neg b))$

$\langle \text{proof} \rangle$

notepad

begin

$\langle \text{proof} \rangle$

end

$\langle \text{ML} \rangle$

The famous decomposition definition *Safe* $:: ('a, 's, 'v) \text{ tls} \Rightarrow ('a, 's, 'v) \text{ tls}$ **where**

Safe $P = \text{tls.safety.cl } P$

definition *Live* $:: ('a, 's, 'v) \text{ tls} \Rightarrow ('a, 's, 'v) \text{ tls}$ **where**

Live $P = P \sqcap \neg \text{tls.safety.cl } P$

lemma *decomp*:

shows $P = \text{tls.Safe } P \sqcap \text{tls.Live } P$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *Safe*:

shows $\text{tls.Safe } P \in \text{tls.safety.closed}$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *Live*:

shows $\text{tls.Live } P \in \text{tls.live}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

24.2 A Haskell-like *Ix* class

We allow arbitrary indexing schemes for user-facing arrays via the *Ix* class, which essentially represents a bijection between a subset of an arbitrary type and an initial segment of the naturals.

Source materials:

- Haskell 2010 report: <https://www.haskell.org/onlinereport/haskell2010/haskellch19.html>
- GHC implementation: <https://hackage.haskell.org/package/base-4.16.0.0/docs/src/GHC.Ix.html>
- Haskell pure arrays (just for colour): <https://www.haskell.org/onlinereport/haskell2010/haskellch14.html>
- SML 2D arrays: <https://smlfamily.github.io/Basis/array2.html>

Observations:

- follow Haskell convention here: include the bounds
- could alternatively use an array of one-dimensional arrays but those are not necessarily rectangular
- we can't use *enum* as that requires the whole type to be enumerable

Limitations:

- the basic design assumes laziness; we don't ever want to build the list of indices
 - can be improved either by tweaking the code generator setup or changing the constants here
- array indices typically have partial predecessor and successor operations and are totally ordered on their domain
- no guarantee the *interval* is correct (does not prevent off-by-one errors in instances)

```
class Ix =
  fixes interval :: 'a × 'a ⇒ 'a list
  fixes index :: 'a × 'a ⇒ 'a ⇒ nat
  assumes index:  $i \in \text{set } (\text{interval } b) \implies \text{interval } b ! \text{index } b \ i = i$ 
  assumes interval:  $\text{map } (\text{index } b) (\text{interval } b) = [0..<\text{length } (\text{interval } b)]$ 
```

```
lemma index-length:
  assumes  $i \in \text{set } (\text{interval } b)$ 
  shows  $\text{index } b \ i < \text{length } (\text{interval } b)$ 
 $\langle \text{proof} \rangle$ 
```

```
lemma distinct-interval:
  shows  $\text{distinct } (\text{interval } b)$ 
 $\langle \text{proof} \rangle$ 
```

```
lemma inj-on-index:
  shows  $\text{inj-on } (\text{index } b) (\text{set } (\text{interval } b))$ 
 $\langle \text{proof} \rangle$ 
```

```
lemma index-eq-conv:
  assumes  $i \in \text{set } (\text{interval } b)$ 
  assumes  $j \in \text{set } (\text{interval } b)$ 
  shows  $\text{index } b \ i = \text{index } b \ j \longleftrightarrow i = j$ 
 $\langle \text{proof} \rangle$ 
```


lemma *index-inv-into*:

assumes $i < \text{length } (\text{interval } b)$

shows $\text{inv-into } (\text{set } (\text{interval } b)) (\text{index } b) i \in \text{set } (\text{interval } b)$

$\langle \text{proof} \rangle$

lemma *linear-order-on*:

shows $\text{linear-order-on } (\text{set } (\text{interval } b)) \{(i, j). \{i, j\} \subseteq \text{set } (\text{interval } b) \wedge \text{index } b \ i \leq \text{index } b \ j\}$

$\langle \text{proof} \rangle$

lemma *interval-map*:

shows $\text{map } (\lambda i. f (\text{interval } b \ ! \ i)) [0..<\text{length } (\text{interval } b)] = \text{map } f (\text{interval } b)$

$\langle \text{proof} \rangle$

lemma *index-forE*:

assumes $i < \text{length } (\text{interval } b)$

obtains j **where** $j \in \text{set } (\text{interval } b)$ **and** $\text{index } b \ j = i$

$\langle \text{proof} \rangle$

instantiation $\text{unit} :: Ix$

begin

definition $\text{interval-unit} = (\lambda(x::\text{unit}, y::\text{unit}). [()])$

definition $\text{index-unit} = (\lambda(x::\text{unit}, y::\text{unit}) \ -::\text{unit}. 0::\text{nat})$

instance $\langle \text{proof} \rangle$

end

instantiation $\text{nat} :: Ix$

begin

definition $\text{interval-nat} = (\lambda(l, u::\text{nat}). [l..<\text{Suc } u])$ — bounds are inclusive

definition $\text{index-nat} = (\lambda(l, u::\text{nat}) \ i::\text{nat}. i - l)$

lemma *upt-minus*:

shows $\text{map } (\lambda i. i - l) [l..<u] = [0..<u - l]$

$\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

instantiation $\text{int} :: Ix$

begin

definition $\text{interval-int} = (\lambda(l, u::\text{int}). [l..u])$ — bounds are inclusive

definition $\text{index-int} = (\lambda(l, u::\text{int}) \ i::\text{int}. \text{nat } (i - l))$

lemma *upto-minus*:

shows $\text{map } (\lambda i. \text{nat } (i - l)) [l..u] = [0..<\text{nat } (u - l + 1)]$

$\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

type-synonym $(i, j) \text{ two-dim} = (i \times j) \times (i \times j)$

instantiation $prod :: (Ix, Ix) Ix$

begin

definition $interval-prod = (\lambda((l, l'), (u, u')). List.product (interval (l, u)) (interval (l', u')))$

definition $index-prod = (\lambda((l, l'), (u, u')) (i, i'). index (l, u) i * length (interval (l', u')) + index (l', u') i')$

abbreviation $(input) fst-bounds :: ('a \times 'b) \times ('a \times 'b) \Rightarrow ('a \times 'a)$ **where**

$fst-bounds\ b \equiv (fst\ (fst\ b),\ fst\ (snd\ b))$

abbreviation $(input) snd-bounds :: ('a \times 'b) \times ('a \times 'b) \Rightarrow ('b \times 'b)$ **where**

$snd-bounds\ b \equiv (snd\ (fst\ b),\ snd\ (snd\ b))$

lemma $inj-on-index-prod:$

shows $inj-on\ (index\ ((l, l'), (u, u')))\ (set\ (interval\ ((l, l'), (u, u'))))$

$\langle proof \rangle$

instance

$\langle proof \rangle$

end

$\langle ML \rangle$

lemma $interval-conv:$

shows $(x, y) \in set\ (interval\ b) \longleftrightarrow x \in set\ (interval\ (fst-bounds\ b)) \wedge y \in set\ (interval\ (snd-bounds\ b))$

$\langle proof \rangle$

$\langle ML \rangle$

type-synonym $'i\ square = ('i, 'i)\ two-dim$

definition $square :: 'i::Ix\ Ix.square \Rightarrow bool$ **where**

$square = (\lambda((l, l'), (u, u')). Ix.interval\ (l, u) = Ix.interval\ (l', u'))$

$\langle ML \rangle$

lemma $conv:$

assumes $Ix.square\ b$

shows $i \in set\ (Ix.interval\ (fst-bounds\ b)) \longleftrightarrow i \in set\ (Ix.interval\ (snd-bounds\ b))$

$\langle proof \rangle$

$\langle ML \rangle$

hide-const **(open)** $interval\ index$

25 A polymorphic heap

We model a heap as a partial map from opaque addresses to structured objects.

- we use this extra structure to handle buffered writes (see §27)
- allocation is non-deterministic and partial
- supports explicit deallocation

Limitations:

- does not support polymorphic sum types such as $'a + 'b$ and $'a\ option$ or products or lists

- the class of representable types is too small to represent processes

Source materials:

- `$ISABELLE_HOME/src/HOL/Imperative_HOL/Heap.thy`
 - that model of heaps includes a *lim* field to support deterministic allocation
 - it uses distinct heaps for arrays and references

$\langle ML \rangle$

type-synonym *addr* = *nat* — untyped heap addresses

datatype *rep* — the concrete representation of heap values
= *Addr nat heap.addr* — metadata paired with an address
| *Val nat*

datatype *write* = *Write heap.addr nat heap.rep*

type-synonym *t* = *heap.addr* \rightarrow *heap.rep list* — partial map from addresses to structured encoded values

abbreviation *empty* :: *heap.t* **where**
empty \equiv *Map.empty*

primrec *apply-write* :: *heap.write* \Rightarrow *heap.t* \Rightarrow *heap.t* **where**
apply-write (*heap.Write addr i x*) *s* = *s(addr* \mapsto (*the (s addr)*)[*i := x*])

class *rep* = — the class of representable types
assumes *ex-inj*: \exists *to-heap-rep* :: '*a* \Rightarrow *heap.rep*. *inj to-heap-rep*

$\langle ML \rangle$

lemma *countable-classI*[*intro!*]:
shows *OFCLASS*('a::countable, *heap.rep-class*)
 $\langle proof \rangle$

definition *to* :: '*a*::*heap.rep* \Rightarrow *heap.rep* **where**
to = (*SOME f*. *inj f*)

definition *from* :: *heap.rep* \Rightarrow '*a*::*heap.rep* **where**
from = *inv (heap.rep.to :: 'a \Rightarrow heap.rep)*

lemmas *inj-to*[*simp*] = *someI-ex*[*OF heap.ex-inj, folded heap.rep.to-def*]

lemma *inj-on-to*[*simp, intro*]: *inj-on heap.rep.to S*
 $\langle proof \rangle$

lemma *surj-from*[*simp*]: *surj heap.rep.from*
 $\langle proof \rangle$

lemma *to-split*[*simp*]: *heap.rep.to x = heap.rep.to y* \longleftrightarrow *x = y*
 $\langle proof \rangle$

lemma *from-to*[*simp*]:
shows *heap.rep.from (heap.rep.to x) = x*
 $\langle proof \rangle$

instance *unit* :: *heap.rep* ⟨*proof*⟩

instance *bool* :: *heap.rep* ⟨*proof*⟩

instance *nat* :: *heap.rep* ⟨*proof*⟩

instance *int* :: *heap.rep* ⟨*proof*⟩

instance *char* :: *heap.rep* ⟨*proof*⟩

instance *String.literal* :: *heap.rep* ⟨*proof*⟩

instance *typerep* :: *heap.rep* ⟨*proof*⟩

⟨*ML*⟩

User-facing heap types typically carry more information than an (untyped) address, such as (phantom) typing and a representation invariant that guarantees the soundness of the encoding (for the given value at the given address only). We abstract over that here and provide some generic operations.

Notes:

- intuitively *addr-of* should be surjective but we do not enforce this
- we use sets here but these are not very flexible: all refs must have the same type
 - this means some intuitive facts involving *UNIV* cannot be stated

class *addr-of* =
 fixes *addr-of* :: 'a ⇒ *heap.addr*
 fixes *rep-val-inv* :: 'a ⇒ *heap.rep list pred*

definition *obj-at* :: *heap.rep list pred* ⇒ *heap.addr* ⇒ *heap.t pred* **where**
 obj-at P r s = (case *s r* of *None* ⇒ *False* | *Some v* ⇒ *P v*)

abbreviation (*input*) *present* :: 'a::*heap.addr-of* ⇒ *heap.t pred* **where**
 present r ≡ *heap.obj-at* ⟨*True*⟩ (*heap.addr-of r*)

abbreviation (*input*) *rep-inv* :: 'a::*heap.addr-of* ⇒ *heap.t pred* **where**
 rep-inv r ≡ *heap.obj-at* (*heap.rep-val-inv r*) (*heap.addr-of r*)

abbreviation (*input*) *rep-inv-set* :: 'a::*heap.addr-of* ⇒ *heap.t set* **where**
 rep-inv-set r ≡ *Collect* (*heap.rep-inv r*)

— allows arbitrary transitions provided the *rep-inv* of *r* is respected

abbreviation (*input*) *rep-inv-rel* :: 'a::*heap.addr-of* ⇒ *heap.t rel* **where**
 rep-inv-rel r ≡ *heap.rep-inv-set r* × *heap.rep-inv-set r*

— totality models the idea that all dereferences are “valid” but only some are reasonable

definition *get* :: 'a::*heap.addr-of* ⇒ *heap.t* ⇒ 'v::*heap.rep list* **where**
 get r s = *map heap.rep.from* (*the* (*s* (*heap.addr-of r*)))

definition *set* :: 'a::*heap.addr-of* ⇒ 'v::*heap.rep list* ⇒ *heap.t* ⇒ *heap.t* **where**
 set r v s = *s*(*heap.addr-of r* ↦ *map heap.rep.to v*)

definition *dealloc* :: 'a::*heap.addr-of* ⇒ *heap.t* ⇒ *heap.t* **where**
 dealloc r s = *s* |' {*heap.addr-of r*}

— allows no changes to *rs*, asserts the *rep-inv* of *rs* is valid, arbitrary changes to *−rs*

definition *Id-on* :: 'a::heap.addr-of set \Rightarrow heap.t rel (heap.Id.) **where**

$$\text{heap.Id}_{rs} = (\bigcap r \in rs. \text{heap.rep-inv-rel } r \cap \text{Id}_{\lambda s. s} (\text{heap.addr-of } r))$$

— allows arbitrary changes to *rs* provided the *rep-inv* of *rs* is respected. requires addresses in $-\text{heap.addr-of } 'rs$ to be unchanged

definition *modifies* :: 'a::heap.addr-of set \Rightarrow heap.t rel (heap.modifies.) **where**

$$\text{heap.modifies}_{rs} = (\bigcap r \in rs. \text{heap.rep-inv-rel } r) \cap \{(s, s'). \forall r \in -\text{heap.addr-of } 'rs. s \ r = s' \ r\}$$

$\langle ML \rangle$

lemma *cong*:

assumes $s (\text{heap.addr-of } r) = s' (\text{heap.addr-of } r')$

shows $\text{heap.get } r \ s = \text{heap.get } r' \ s'$

$\langle \text{proof} \rangle$

lemma *Id-on-proj-cong*:

assumes $(s, s') \in \text{heap.Id}_{\{r\}}$

shows $\text{heap.get } r \ s = \text{heap.get } r \ s'$

$\langle \text{proof} \rangle$

lemma *fun-upd*:

shows $\text{heap.get } r (\text{fun-upd } s \ a \ (\text{Some } w))$

$= (\text{if } \text{heap.addr-of } r = a \ \text{then } \text{map } \text{heap.rep.from } w \ \text{else } \text{heap.get } r \ s)$

$\langle \text{proof} \rangle$

lemma *set-eq*:

shows $\text{heap.get } r (\text{heap.set } r \ v \ s) = v$

$\langle \text{proof} \rangle$

lemma *set-neq*:

assumes $\text{heap.addr-of } r \neq \text{heap.addr-of } r'$

shows $\text{heap.get } r (\text{heap.set } r' \ v \ s) = \text{heap.get } r \ s$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *cong*:

assumes $\text{heap.addr-of } r = \text{heap.addr-of } r'$

assumes $v = v'$

assumes $\bigwedge r'. r' \neq \text{heap.addr-of } r \Longrightarrow s \ r' = s' \ r'$

shows $\text{heap.set } r \ v \ s = \text{heap.set } r' \ v' \ s'$

$\langle \text{proof} \rangle$

lemma *empty*:

shows $\text{heap.set } r \ v (\text{heap.empty}) = [\text{heap.addr-of } r \mapsto \text{map } \text{heap.rep.to } v]$

$\langle \text{proof} \rangle$

lemma *fun-upd*:

shows $\text{heap.set } r \ v (\text{fun-upd } s \ a \ w) = (\text{fun-upd } s \ a \ w)(\text{heap.addr-of } r \mapsto \text{map } \text{heap.rep.to } v)$

$\langle \text{proof} \rangle$

lemma *same*:

shows $\text{heap.set } r \ v (\text{heap.set } r \ w \ s) = \text{heap.set } r \ v \ s$

$\langle \text{proof} \rangle$

lemma *twist*:

assumes $\text{heap.addr-of } r \neq \text{heap.addr-of } r'$

shows $\text{heap.set } r \ v (\text{heap.set } r' \ w \ s) = \text{heap.set } r' \ w (\text{heap.set } r \ v \ s)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *cong[cong]*:

fixes $P :: heap.rep\ list\ pred$

assumes $\bigwedge v. s\ r = Some\ v \implies P\ v = P'\ v$

assumes $r = r'$

assumes $s\ r = s'\ r'$

shows $heap.obj-at\ P\ r\ s \longleftrightarrow heap.obj-at\ P'\ r'\ s'$

$\langle proof \rangle$

lemma *split*:

shows $Q\ (heap.obj-at\ P\ r\ s) \longleftrightarrow (s\ r = None \longrightarrow Q\ False) \wedge (\forall v. s\ r = Some\ v \longrightarrow Q\ (P\ v))$

$\langle proof \rangle$

lemma *split-asm*:

shows $Q\ (heap.obj-at\ P\ r\ s) \longleftrightarrow \neg ((s\ r = None \wedge \neg Q\ False) \vee (\exists v. s\ r = Some\ v \wedge \neg Q\ (P\ v)))$

$\langle proof \rangle$

lemmas *splits = heap.obj-at.split heap.obj-at.split-asm*

lemma *empty*:

shows $\neg heap.obj-at\ P\ r\ heap.empty$

$\langle proof \rangle$

lemma *set*:

shows $heap.obj-at\ P\ r\ (heap.set\ r'\ v\ s)$

$\longleftrightarrow (r = heap.addr-of\ r' \wedge P\ (map\ heap.rep.to\ v)) \vee (r \neq heap.addr-of\ r' \wedge heap.obj-at\ P\ r\ s)$

$\langle proof \rangle$

lemma *fun-upd*:

shows $heap.obj-at\ P\ r\ (fun-upd\ s\ a\ (Some\ w)) = (if\ r = a\ then\ P\ w\ else\ heap.obj-at\ P\ r\ s)$

$\langle proof \rangle$

$\langle ML \rangle$

lemmas *simps = — objective: reduce manifest heaps*

heap.get.set-eq

heap.get.fun-upd

heap.set.empty

heap.set.same

heap.set.fun-upd

heap.obj-at.empty

heap.obj-at.fun-upd

$\langle ML \rangle$

lemma *empty[simp]*:

shows $heap.Id_{\{\}} = UNIV$

$\langle proof \rangle$

lemma *sup*:

shows $heap.Id_{X \cup Y} = heap.Id_X \cap heap.Id_Y$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *empty[simp]*:
shows $\text{heap.modifies}\{\} = \text{Id}$
 ⟨proof⟩

lemma *rep-inv-rel-le*:
shows $\text{heap.modifies}_{rs} \subseteq (\bigcap r \in rs. \text{heap.rep-inv-rel } r)$
 ⟨proof⟩

lemma *rep-inv*:
assumes $(s, s') \in \text{heap.modifies}\{a\}$
shows $\text{heap.rep-inv } a \ s$
and $\text{heap.rep-inv } a \ s'$
 ⟨proof⟩

lemma *Id-conv*:
shows $(s, s) \in \text{heap.modifies}_{rs} \longleftrightarrow (\forall r \in rs. (s, s) \in \text{heap.rep-inv-rel } r)$
 ⟨proof⟩

lemma *eqI*:
assumes $(s, s') \in \text{heap.modifies}_{rs}$
assumes $\bigwedge r. \llbracket r \in rs; \text{heap.rep-inv } r \ s; \text{heap.rep-inv } r \ s' \rrbracket \implies s (\text{heap.addr-of } r) = s' (\text{heap.addr-of } r)$
shows $s = s'$
 ⟨proof⟩

⟨ML⟩

lemma *Id-on-frame-cong*:
assumes $\bigwedge s \ s'. (\bigwedge r. r \in rs \implies \text{heap.rep-inv } r \ s \wedge \text{heap.rep-inv } r \ s' \wedge s (\text{heap.addr-of } r) = s' (\text{heap.addr-of } r))$
 $\implies P \ s \longleftrightarrow P' \ s'$
shows $\text{stable heap.Id}_{rs} \ P \longleftrightarrow \text{stable heap.Id}_{rs} \ P'$
 ⟨proof⟩

lemma *Id-on-frameI*:
assumes $\bigwedge s \ s'. (\bigwedge r. r \in rs \implies \text{heap.rep-inv } r \ s \wedge \text{heap.rep-inv } r \ s' \wedge s (\text{heap.addr-of } r) = s' (\text{heap.addr-of } r))$
 $\implies P \ s \longleftrightarrow P \ s'$
shows $\text{stable heap.Id}_{rs} \ P$
 ⟨proof⟩

lemma *Id-on-rep-invI[stable.intro]*:
assumes $r \in rs$
shows $\text{stable heap.Id}_{rs} (\text{heap.rep-inv } r)$
 ⟨proof⟩

⟨ML⟩

25.1 References

datatype $'a \ \text{ref} = \text{Ref} (\text{addr-of}: \text{heap.addr})$

instantiation $\text{ref} :: (\text{heap.rep}) \ \text{heap.addr-of}$
begin

definition $\text{addr-of-ref} :: 'a \ \text{ref} \Rightarrow \text{heap.addr}$ **where**
 $\text{addr-of-ref} = \text{ref.addr-of}$

definition $\text{rep-val-inv-ref} :: 'a \ \text{ref} \Rightarrow \text{heap.rep list pred}$ **where**
 $\text{rep-val-inv-ref } r \ vs \longleftrightarrow (\text{case } vs \ \text{of } [v] \Rightarrow \text{heap.rep.to } (\text{heap.rep.from } v :: 'a) = v \mid _ \Rightarrow \text{False})$

instance $\langle proof \rangle$

end

instance $ref :: (heap.rep) heap.rep$

$\langle proof \rangle$

$\langle ML \rangle$

definition $get :: 'a::heap.rep ref \Rightarrow heap.t \Rightarrow 'a$ **where**

$get\ r\ s = hd\ (heap.get\ r\ s)$

definition $set :: 'a::heap.rep ref \Rightarrow 'a \Rightarrow heap.t \Rightarrow heap.t$ **where**

$set\ r\ v\ s = heap.set\ r\ [v]\ s$

definition $alloc :: 'a \Rightarrow heap.t \Rightarrow ('a::heap.rep ref \times heap.t) set$ **where**

$alloc\ v\ s = \{(r, Ref.set\ r\ v\ s) \mid r. \neg heap.present\ r\ s\}$

lemma $addr-of:$

shows $heap.addr-of\ (Ref\ r) = r$

$\langle proof \rangle$

$\langle ML \rangle$

lemma $fun-upd:$

shows $Ref.get\ r\ (fun-upd\ s\ a\ (Some\ [w]))$

$= (if\ heap.addr-of\ r = a\ then\ heap.rep.from\ w\ else\ Ref.get\ r\ s)$

$\langle proof \rangle$

lemma $set-eq:$

shows $Ref.get\ r\ (Ref.set\ r\ v\ s) = v$

$\langle proof \rangle$

lemma $set-neq:$

fixes $r :: 'a::heap.rep ref$

fixes $r' :: 'b::heap.rep ref$

assumes $addr-of\ r \neq addr-of\ r'$

shows $Ref.get\ r\ (Ref.set\ r'\ v\ s) = Ref.get\ r\ s$

$\langle proof \rangle$

$\langle ML \rangle$

lemma $empty:$

shows $Ref.set\ r\ v\ (heap.empty) = [heap.addr-of\ r \mapsto [heap.rep.to\ v]]$

$\langle proof \rangle$

lemma $fun-upd:$

shows $Ref.set\ r\ v\ (fun-upd\ s\ a\ w) = (fun-upd\ s\ a\ w)(heap.addr-of\ r \mapsto [heap.rep.to\ v])$

$\langle proof \rangle$

lemma $same:$

shows $Ref.set\ r\ v\ (Ref.set\ r\ w\ s) = Ref.set\ r\ v\ s$

$\langle proof \rangle$

lemma $obj-at-conv:$

fixes $a :: heap.addr$

fixes $r :: 'a::heap.rep ref$

fixes $v :: 'a$

fixes $P :: \text{heap.rep list pred}$

shows $\text{heap.obj-at } P \ a \ (\text{Ref.set } r \ v \ s) \longleftrightarrow (a = \text{heap.addr-of } r \wedge P [\text{heap.rep.to } v]) \vee (a \neq \text{heap.addr-of } r \wedge \text{heap.obj-at } P \ a \ s)$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemmas $\text{simps}[\text{simp}] =$

Ref.addr-of

Ref.get.set-eq

Ref.get.set-neq

Ref.get.fun-upd

Ref.set.same

Ref.set.empty

Ref.set.fun-upd

$\text{Ref.set.obj-at-conv}$

$\langle \text{ML} \rangle$

25.2 Arrays

25.2.1 Code generation constants: one-dimensional arrays

We ask that targets of the code generator provide implementations of one-dimensional arrays and the associated operations.

Notes:

- user-facing arrays make use of Ix
- due to the lack of bounds there is no rep-val-inv

datatype $'a \ \text{one-dim-array} = \text{Array} \ (\text{addr-of}: \text{heap.addr})$

instantiation $\text{one-dim-array} :: (\text{type}) \ \text{heap.addr-of}$
begin

definition $\text{addr-of-one-dim-array} :: 'a \ \text{one-dim-array} \Rightarrow \text{heap.addr}$ **where**
 $\text{addr-of-one-dim-array} = \text{addr-of}$

definition $\text{rep-val-inv-one-dim-array} :: 'a \ \text{one-dim-array} \Rightarrow \text{heap.rep list pred}$ **where**
 $[\text{simp}]: \text{rep-val-inv-one-dim-array } a \ vs \longleftrightarrow \text{True}$

instance $\langle \text{proof} \rangle$

end

$\langle \text{ML} \rangle$

definition $\text{get} :: 'a :: \text{heap.rep one-dim-array} \Rightarrow \text{nat} \Rightarrow \text{heap.t} \Rightarrow 'a$ **where**
 $\text{get } a \ i \ s = \text{heap.get } a \ s \ ! \ i$

definition $\text{set} :: 'a :: \text{heap.rep one-dim-array} \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow \text{heap.t} \Rightarrow \text{heap.t}$ **where**
 $\text{set } a \ i \ v \ s = \text{heap.set } a \ ((\text{heap.get } a \ s)[i:=v]) \ s$

definition $\text{alloc} :: 'a \ \text{list} \Rightarrow \text{heap.t} \Rightarrow ('a :: \text{heap.rep one-dim-array} \times \text{heap.t}) \ \text{set}$ **where**
 $\text{alloc } av \ s = \{(a, \text{heap.set } a \ av \ s) \mid a. \neg \text{heap.present } a \ s\}$

definition $\text{list-for} :: 'a :: \text{heap.rep one-dim-array} \Rightarrow \text{heap.t} \Rightarrow 'a \ \text{list}$ **where**
 $\text{list-for } a = \text{heap.get } a$

$\langle ML \rangle$

lemma *weak-cong*:

assumes $i = i'$

assumes $a = a'$

assumes $s \text{ (heap.addr-of } a) = s' \text{ (heap.addr-of } a')$

shows $ODArray.get\ a\ i\ s = ODArray.get\ a'\ i'\ s'$

$\langle proof \rangle$

lemma *weak-Id-on-proj-cong*:

assumes $i = i'$

assumes $a = a'$

assumes $(s, s') \in \text{heap.Id}_{\{a'\}}$

shows $ODArray.get\ a\ i\ s = ODArray.get\ a'\ i'\ s'$

$\langle proof \rangle$

lemma *set-eq*:

assumes $i < \text{length (the (s (heap.addr-of a)))}$

shows $ODArray.get\ a\ i\ (ODArray.set\ a\ i\ v\ s) = v$

$\langle proof \rangle$

lemma *set-neq*:

assumes $i \neq j$

shows $ODArray.get\ a\ i\ (ODArray.set\ a\ j\ v\ s) = ODArray.get\ a\ i\ s$

$\langle proof \rangle$

$\langle ML \rangle$

25.2.2 User-facing arrays

datatype $('i, 'a) \text{ array} = \text{Array (bounds: } ('i \times 'i)) \text{ (arr: } 'a \text{ one-dim-array)}$

hide-const (**open**) *bounds arr*

instantiation $\text{array} :: (Ix, \text{heap.rep}) \text{ heap.addr-of}$

begin

definition $\text{addr-of-array} :: ('a, 'b) \text{ array} \Rightarrow \text{heap.addr}$ **where**

$\text{addr-of-array} = \text{addr-of} \circ \text{array.arr}$

definition $\text{rep-val-inv-array} :: ('a, 'b) \text{ array} \Rightarrow \text{heap.rep list pred}$ **where**

$\text{rep-val-inv-array}\ a\ vs \longleftrightarrow$

$\text{length}\ vs = \text{length}\ (Ix.\text{interval}\ (\text{array.bounds}\ a))$

$\wedge (\forall v \in \text{set}\ vs. \text{heap.rep.to}\ (\text{heap.rep.from}\ v :: 'b) = v)$

instance $\langle proof \rangle$

end

instance $\text{array} :: (\text{countable}, \text{type}) \text{ heap.rep}$

$\langle proof \rangle$

$\langle ML \rangle$

abbreviation (*input*) $\text{square} :: ('i :: Ix \times 'i, 'a) \text{ array} \Rightarrow \text{bool}$ **where**

$\text{square}\ a \equiv Ix.\text{square}\ (\text{array.bounds}\ a)$

abbreviation *(input)* $index :: ('i::Ix, 'a) array \Rightarrow 'i \Rightarrow nat$ **where**
 $index\ a \equiv Ix.index\ (array.bounds\ a)$

abbreviation *(input)* $interval :: ('i::Ix, 'a) array \Rightarrow 'i\ list$ **where**
 $interval\ a \equiv Ix.interval\ (array.bounds\ a)$

definition $get :: ('i::Ix, 'a::heap.rep) array \Rightarrow 'i \Rightarrow heap.t \Rightarrow 'a$ **where**
 $get\ a\ i = OArray.get\ (array.arr\ a)\ (Array.index\ a\ i)$

definition $set :: ('i::Ix, 'a::heap.rep) array \Rightarrow 'i \Rightarrow 'a \Rightarrow heap.t \Rightarrow heap.t$ **where**
 $set\ a\ i\ v = OArray.set\ (array.arr\ a)\ (Array.index\ a\ i)\ v$

definition $list-for :: ('i::Ix, 'a::heap.rep) array \Rightarrow heap.t \Rightarrow 'a\ list$ **where**
 $list-for\ a = OArray.list-for\ (array.arr\ a)$

— can coerce any indexing regime into any other provided the contents fit

definition $coerce :: ('i::Ix, 'a::heap.rep) array \Rightarrow ('j \times 'j) \Rightarrow ('j::Ix, 'a::heap.rep) array\ option$ **where**
 $coerce\ a\ b = (if\ length\ (Array.interval\ a) = length\ (Ix.interval\ b)$
 $\quad then\ Some\ (Array\ b\ (array.arr\ a))$
 $\quad else\ None)$

definition $Id-on :: ('i::Ix, 'a::heap.rep) array \Rightarrow 'i\ set \Rightarrow heap.t\ rel\ (Array.Id-, -)$ **where**
 $Array.Id_{a, is} = heap.rep-inv-rel\ a \cap \{(s, s') . \forall i \in is. Array.get\ a\ i\ s = Array.get\ a\ i\ s'\}$

definition $modifies :: ('i::Ix, 'a::heap.rep) array \Rightarrow 'i\ set \Rightarrow heap.t\ rel\ (Array.modifies-, -)$ **where**
 $Array.modifies_{a, is}$
 $= heap.modifies_{\{a\}} \cap \{(s, s') . \forall i \in set\ (Array.interval\ a) - is. Array.get\ a\ i\ s = Array.get\ a\ i\ s'\}$

lemma $simps[simp]$:

shows $heap.addr-of\ (array.arr\ a) = heap.addr-of\ a$
and $heap.addr-of \circ array.arr = heap.addr-of$

$\langle proof \rangle$

$\langle ML \rangle$

lemma $set-eq$:

assumes $heap.rep-inv\ a\ s$
assumes $i \in set\ (Array.interval\ a)$
shows $Array.get\ a\ i\ (Array.set\ a\ i\ v\ s) = v$

$\langle proof \rangle$

lemma $set-neq$:

assumes $i \in set\ (Array.interval\ a)$
assumes $j \in set\ (Array.interval\ a)$
assumes $i \neq j$
shows $Array.get\ a\ j\ (Array.set\ a\ i\ v\ s) = Array.get\ a\ j\ s$

$\langle proof \rangle$

lemma $Id-on-proj-cong$:

assumes $a = a'$
assumes $i = i'$
assumes $(s, s') \in Array.Id_{a', \{i'\}}$
assumes $i' \in set\ (Array.interval\ a)$
shows $Array.get\ a\ i\ s = Array.get\ a'\ i'\ s'$

$\langle proof \rangle$

lemma $weak-cong$:

assumes $a = a'$

assumes $i = i'$
assumes $s (\text{heap.addr-of } a) = s' (\text{heap.addr-of } a')$
shows $\text{Array.get } a \ i \ s = \text{Array.get } a' \ i' \ s'$
 $\langle \text{proof} \rangle$

lemma *weak-Id-on-proj-cong*:

assumes $i = i'$
assumes $a = a'$
assumes $(s, s') \in \text{heap.Id}_{\{a\}}$
shows $\text{Array.get } a \ i \ s = \text{Array.get } a' \ i' \ s'$
 $\langle \text{proof} \rangle$

lemma *ext*:

assumes $\text{heap.rep-inv } a \ s$
assumes $\text{heap.rep-inv } a \ s'$
assumes $\forall i \in \text{set } (\text{Ix-class.interval } (\text{array.bounds } a)). \text{Array.get } a \ i \ s = \text{Array.get } a \ i \ s'$
shows $s (\text{heap.addr-of } a) = s' (\text{heap.addr-of } a)$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *cong-deref*:

assumes $a = a'$
assumes $i = i'$
assumes $v = v'$
assumes $s \ r = s' \ r'$
assumes $r = r'$
shows $\text{Array.set } a \ i \ v \ s \ r = \text{Array.set } a' \ i' \ v' \ s' \ r'$
 $\langle \text{proof} \rangle$

lemma *same*:

shows $\text{Array.set } a \ i \ v \ (\text{Array.set } a \ i \ v' \ s) = \text{Array.set } a \ i \ v \ s$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *ex-bij-betw*:

fixes $a :: ('i::\text{Ix}, 'a::\text{heap.rep}) \text{array}$
fixes $b :: 'j::\text{Ix} \times 'j$
assumes $\text{Array.coerce } a \ b = \text{Some } a'$
obtains f **where** $\text{map } f \ (\text{Array.interval } a) = \text{Ix.interval } b$
 $\langle \text{proof} \rangle$

lemma *ex-bij-betw2*:

fixes $a :: ('i::\text{Ix}, 'a::\text{heap.rep}) \text{array}$
fixes $b :: 'j::\text{Ix} \times 'j$
assumes $\text{Array.coerce } a \ b = \text{Some } a'$
obtains f **where** $\text{map } f \ (\text{Ix.interval } b) = \text{Array.interval } a$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *set*:

assumes $\text{heap.rep-inv } a \ s$
shows $\text{heap.rep-inv } a \ (\text{Array.set } a \ i \ v \ s)$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *heap-modifies-le*:

shows $Array.modifies_{a, is} \subseteq heap.modifies_{\{a\}}$
<proof>

lemma *heap-rep-inv-rel-le*:

shows $Array.modifies_{a, is} \subseteq heap.rep-inv-rel\ a$
<proof>

lemma *empty*:

shows $Array.modifies_{a, \{\}} = Id \cap heap.rep-inv-rel\ a$ (**is** ?lhs = ?rhs)
<proof>

lemma *mono*:

assumes $is \subseteq js$
shows $Array.modifies_{a, is} \subseteq Array.modifies_{a, js}$
<proof>

lemma *INTER*:

shows $Array.modifies_{a, \bigcap_{x \in X}. f\ x} = (\bigcap_{x \in X}. Array.modifies_{a, f\ x}) \cap heap.modifies_{\{a\}}$
<proof>

lemma *Inter*:

shows $Array.modifies_{a, \bigcap X} = (\bigcap_{x \in X}. Array.modifies_{a, x}) \cap heap.modifies_{\{a\}}$
<proof>

lemma *inter*:

shows $Array.modifies_{a, is} \cap Array.modifies_{a, js} = Array.modifies_{a, is \cap js}$
<proof>

lemma *UNION-subseteq*:

shows $(\bigcup_{x \in X}. Array.modifies_{a, I\ x}) \subseteq Array.modifies_{a, (\bigcup_{x \in X}. I\ x)}$
<proof>

lemma *union-subseteq*:

shows $Array.modifies_{a, is} \cup Array.modifies_{a, js} \subseteq Array.modifies_{a, is \cup js}$
<proof>

lemma *Diag-subseteq*:

assumes $\bigwedge s. P\ s \implies heap.rep-inv\ a\ s$
shows $Diag\ P \subseteq Array.modifies_{a, is}$
<proof>

lemma *get*:

assumes $(s, s') \in Array.modifies_{a, is}$
assumes $i \in set\ (Array.interval\ a) - is$
shows $Array.get\ a\ i\ s' = Array.get\ a\ i\ s$
<proof>

lemma *set*:

assumes $heap.rep-inv\ a\ s$
shows $(s, Array.set\ a\ i\ v\ s) \in heap.modifies_{\{a\}}$
<proof>

lemma *Array-set*:

assumes $heap.rep-inv\ a\ s$
assumes $i \in set\ (Array.interval\ a) \cap is$
shows $(s, Array.set\ a\ i\ v\ s) \in Array.modifies_{a, is}$

$\langle \text{proof} \rangle$

lemma *Array-set-conv*:

assumes $i \in \text{set } (\text{Array.interval } a) \cap \text{is}$

shows $(s, \text{Array.set } a \ i \ v \ s) \in \text{Array.modifies}_{a, \text{is}} \longleftrightarrow \text{heap.rep-inv } a \ s \ (\text{is } ?\text{lhs} \longleftrightarrow ?\text{rhs})$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemmas *simps'* =

Array.rep-inv.set

Array.get.set-eq

$\langle \text{ML} \rangle$

lemma *Id-on-le*:

shows $\text{heap.Id}_{\{a\}} \subseteq \text{Array.Id}_{a, \text{is}}$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *empty*:

shows $\text{Array.Id}_{a, \{\}} = \text{heap.rep-inv-rel } a$

$\langle \text{proof} \rangle$

lemma *mono*:

assumes $\text{is} \subseteq \text{js}$

shows $\text{Array.Id}_{a, \text{js}} \subseteq \text{Array.Id}_{a, \text{is}}$

$\langle \text{proof} \rangle$

lemma *insert*:

shows $\text{Array.Id}_{a, \text{insert } i \ \text{is}} = \text{Array.Id}_{a, \{i\}} \cap \text{Array.Id}_{a, \text{is}}$

$\langle \text{proof} \rangle$

lemma *union*:

shows $\text{Array.Id}_{a, \text{is} \cup \text{js}} = \text{Array.Id}_{a, \text{is}} \cap \text{Array.Id}_{a, \text{js}}$

$\langle \text{proof} \rangle$

lemma *rep-inv-rel*:

shows $\text{Array.Id}_{a, \text{is}} \subseteq \text{heap.rep-inv-rel } a$

$\langle \text{proof} \rangle$

lemma *eq-heap-Id-on*:

assumes $\text{set } (\text{Array.interval } a) \subseteq \text{is}$

shows $\text{Array.Id}_{a, \text{is}} = \text{heap.Id}_{\{a\}}$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

25.2.3 Stability

$\langle \text{ML} \rangle$

lemma *get[stable.intro]*:

assumes $a \in \text{as}$

shows $\text{stable heap.Id}_{\text{as}} (\lambda s. P (\text{Array.get } a \ i \ s))$

$\langle \text{proof} \rangle$

lemma *get-chain*: — difficult to apply
assumes $\bigwedge v. \text{stable heap.Id}_{as} (P v)$
assumes $a \in as$
shows $\text{stable heap.Id}_{as} (\lambda s. P (\text{Array.get } a \ i \ s) \ s)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *get[stable.intro]*:
assumes $i \in is$
shows $\text{stable Array.Id}_{a, is} (\lambda s. P (\text{Array.get } a \ i \ s))$
 $\langle \text{proof} \rangle$

lemma *get-chain*: — difficult to apply
assumes $\bigwedge v. \text{stable Array.Id}_{a, is} (P v)$
assumes $i \in is$
shows $\text{stable Array.Id}_{a, is} (\lambda s. P (\text{Array.get } a \ i \ s) \ s)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *rep-inv[stable.intro]*:
shows $\text{stable Array.Id}_{a, is} (\text{heap.rep-inv } a)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *rep-inv[stable.intro]*:
shows $\text{stable Array.modifies}_{a, is} (\text{heap.rep-inv } a)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *get[stable.intro]*:
assumes $i \in \text{set } (\text{Array.interval } a) - is$
shows $\text{stable Array.modifies}_{a, is} (\lambda s. P (\text{Array.get } a \ i \ s))$
 $\langle \text{proof} \rangle$

lemma *get-chain*: — difficult to apply
assumes $\bigwedge v. \text{stable Array.modifies}_{a, is} (P v)$
assumes $i \in \text{set } (\text{Array.interval } a) - is$
shows $\text{stable Array.modifies}_{a, is} (\lambda s. P (\text{Array.get } a \ i \ s) \ s)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

26 A concurrent variant of Imperative HOL

We model programs operating on sequentially-consistent memory with the type $(\text{heap.t}, 'v) \text{ prog}$.

Source materials:

- `$ISABELLE_HOME/src/HOL/Imperative_HOL/Heap_Monad.thy`
- `$ISABELLE_HOME/src/HOL/Imperative_HOL/Array.thy`
- `$ISABELLE_HOME/src/HOL/Imperative_HOL/Ref.thy`

– note that ImperativeHOL is deterministic and sequential

type-synonym $'v \text{ imp} = (\text{heap.t}, 'v) \text{ prog}$

$\langle ML \rangle$

definition $\text{raise} :: \text{String.literal} \Rightarrow 'a \text{ imp}$ **where** — the literal is just decoration
 $\text{raise } s = \perp$

definition $\text{assert} :: \text{bool} \Rightarrow \text{unit imp}$ **where**
 $\text{assert } P = (\text{if } P \text{ then prog.return } () \text{ else prog.raise STR "assert"})$

$\langle ML \rangle$

definition $\text{ref} :: 'a::\text{heap.rep} \Rightarrow 'a \text{ ref imp}$ **where**
 $\text{ref } v = \text{prog.action } \{(r, s, s'). (r, s') \in \text{Ref.alloc } v \ s\}$

definition $\text{lookup} :: 'a::\text{heap.rep} \text{ ref} \Rightarrow 'a \text{ imp}$ (!- 61) **where**
 $\text{lookup } r = \text{prog.read } (\text{Ref.get } r)$

definition $\text{update} :: 'a \text{ ref} \Rightarrow 'a::\text{heap.rep} \Rightarrow \text{unit imp}$ (- := - 62) **where**
 $\text{update } r \ v = \text{prog.write } (\text{Ref.set } r \ v)$

$\langle ML \rangle$

definition $\text{new} :: ('i \times 'i) \Rightarrow 'a \Rightarrow ('i::Ix, 'a::\text{heap.rep}) \text{ array imp}$ **where**
 $\text{new } b \ v = \text{prog.action } \{(\text{Array } b \ a, s, s') \mid a \ s \ s'. (a, s') \in \text{ODArray.alloc } (\text{replicate } (\text{length } (Ix.\text{interval } b)) \ v) \ s\}$

definition $\text{make} :: ('i \times 'i) \Rightarrow ('i \Rightarrow 'a) \Rightarrow ('i::Ix, 'a::\text{heap.rep}) \text{ array imp}$ **where**
 $\text{make } b \ f = \text{prog.action } \{(\text{Array } b \ a, s, s') \mid a \ s \ s'. (a, s') \in \text{ODArray.alloc } (\text{map } f \ (Ix.\text{interval } b)) \ s\}$

— Approximately Haskell’s `listArray`: “Construct an array from a pair of bounds and a list of values in index order.”

definition $\text{of-list} :: ('i \times 'i) \Rightarrow 'a \text{ list} \Rightarrow ('i::Ix, 'a::\text{heap.rep}) \text{ array imp}$ **where**
 $\text{of-list } b \ xs = \text{prog.action } \{(\text{Array } b \ a, s, s') \mid a \ s \ s'. \text{length } (Ix.\text{interval } b) \leq \text{length } xs \wedge (a, s') \in \text{ODArray.alloc } xs \ s\}$

definition $\text{nth} :: ('i::Ix, 'a::\text{heap.rep}) \text{ array} \Rightarrow 'i \Rightarrow 'a \text{ imp}$ **where**
 $\text{nth } a \ i = \text{prog.read } (\lambda s. \text{Array.get } a \ i \ s)$

definition $\text{upd} :: ('i::Ix, 'a::\text{heap.rep}) \text{ array} \Rightarrow 'i \Rightarrow 'a \Rightarrow \text{unit imp}$ **where**
 $\text{upd } a \ i \ v = \text{prog.write } (\text{Array.set } a \ i \ v)$

— derived operations; observe the lack of atomicity

definition $\text{freeze} :: ('i::Ix, 'a::\text{heap.rep}) \text{ array} \Rightarrow 'a \text{ list imp}$ **where**
 $\text{freeze } a = \text{prog.fold-mapM } (\text{prog.Array.nth } a) \ (\text{Array.interval } a)$

definition $\text{swap} :: ('i::Ix, 'a::\text{heap.rep}) \text{ array} \Rightarrow 'i \Rightarrow 'i \Rightarrow \text{unit imp}$ **where**
 $\text{swap } a \ i \ j =$
 $\text{do } \{$
 $\quad x \leftarrow \text{prog.Array.nth } a \ i;$
 $\quad y \leftarrow \text{prog.Array.nth } a \ j;$
 $\quad \text{prog.Array.upd } a \ i \ y;$
 $\quad \text{prog.Array.upd } a \ j \ x;$
 $\quad \text{prog.return } ();$
 $\}$

declare $\text{prog.raise-def}[code \ \text{del}]$


```

declare prog.Ref.ref-def[code del]
declare prog.Ref.lookup-def[code del]
declare prog.Ref.update-def[code del]
declare prog.Array.new-def[code del]
declare prog.Array.make-def[code del]
declare prog.Array.of-list-def[code del]
declare prog.Array.nth-def[code del]
declare prog.Array.upd-def[code del]
declare prog.Array.freeze-def[code del]

```

Operations on two-dimensional arrays **definition** *fst-app-chaotic* :: ('a::Ix, 'b::Ix) two-dim ⇒ ('a ⇒ ('s, unit) prog) ⇒ ('s, unit) prog **where**
fst-app-chaotic b f = prog.set-app f (set (Ix.interval (fst-bounds b)))

definition *fst-app* :: ('a::Ix, 'b::Ix) two-dim ⇒ ('a ⇒ ('s, unit) prog) ⇒ ('s, unit) prog **where**
fst-app b f = prog.app f (Ix.interval (fst-bounds b))

lemma *fst-app-fst-app-chaotic-le*:
shows prog.Array.fst-app b f ≤ prog.Array.fst-app-chaotic b f
⟨proof⟩

⟨ML⟩

lemmas *fst-app-chaotic* =
ag.prog.app-set[**where** X=set (Ix.interval (fst-bounds b)) **for** b, folded prog.Array.fst-app-chaotic-def]

lemmas *fst-app* =
ag.prog.app[**where** xs=Ix.interval (fst-bounds b) **for** b, folded prog.Array.fst-app-def]

⟨ML⟩

26.1 Code generator setup

26.1.1 Haskell

code-printing code-module *Heap* ↪ (*Haskell*)

```

<
-- Sequentially-consistent primitives
-- Arrays:
-- https://hackage.haskell.org/package/array-0.5.4.0/docs/Data-Array-IO.html
-- https://hackage.haskell.org/package/array-0.5.4.0/docs/src/Data.Array.Base.html
module Heap (
  Prog
  , Ref, newIORef, readIORef, writeIORef
  , Array, newArray, newListArray, newFunArray, readArray, writeArray
  , parallel
) where

```

```

import Control.Concurrent (forkIO)
import qualified Control.Concurrent.MVar as MVar
import qualified Data.Array.IO as Array
import Data.IORef (IORef, newIORef, readIORef, atomicWriteIORef)
import Data.List (genericLength)

```

```

type Prog a b = IO b
type Array a = Array.IOArray Integer a
type Ref a = Data.IORef.IORef a

```

```

writeIORef :: IORef a -> a -> IO ()

```

writeIORef = atomicWriteIORef -- could use the strict variant

newArray :: Integer -> a -> IO (Array a)

newArray k = Array.newArray (0, k - 1)

newFunArray :: Integer -> (Integer -> a) -> IO (Array a)

newFunArray k f = Array.newListArray (0, k - 1) (map f [0..k-1])

newListArray :: Integer -> [a] -> IO (Array a)

newListArray k xs = Array.newListArray (0, k) xs

readArray :: Array a -> Integer -> IO a

readArray = Array.readArray

writeArray :: Array a -> Integer -> a -> IO ()

writeArray = Array.writeArray -- probably should be the WMM atomic op

{-

-- 'forkIO' is reputedly cheap, but other papers imply the use of worker threads, perhaps for other reasons

-- note we don't want forkFinally as we don't model exceptions

parallel' :: IO a -> IO b -> IO (a, b)

parallel' p q = do

mvar <- MVar.newEmptyMVar

forkIO (p >>= MVar.putMVar mvar) -- note putMVar is lazy

b <- q

a <- MVar.takeMVar mvar

return (a, b)

-}

parallel :: IO () -> IO () -> IO ()

parallel p q = do

mvar <- MVar.newEmptyMVar

forkIO (p >> MVar.putMVar mvar ()) -- note putMVar is lazy

b <- q

a <- MVar.takeMVar mvar

return ()

>

code-reserved *Haskell Ix*

code-printing type-constructor *prog* \rightarrow (*Haskell*) *Heap.Prog* - -

code-monad *prog.bind Haskell*

code-printing constant *prog.return* \rightarrow (*Haskell*) *return*

code-printing constant *prog.raise* \rightarrow (*Haskell*) *error*

code-printing constant *prog.parallel* \rightarrow (*Haskell*) *Heap.parallel*

Intermediate operation avoids invariance problem in *Scala* (similar to value restriction)

$\langle ML \rangle$

definition *ref'* **where**

[code del]: ref' = prog.Ref.ref

lemma *[code]:*

prog.Ref.ref x = Ref.ref' x

$\langle proof \rangle$

$\langle ML \rangle$

code-printing type-constructor $ref \rightarrow (Haskell) Heap.Ref$ -
code-printing constant $Ref \rightarrow (Haskell) error / bare Ref$
code-printing constant $Ref.ref' \rightarrow (Haskell) Heap.newIORef$
code-printing constant $prog.Ref.lookup \rightarrow (Haskell) Heap.readIORef$
code-printing constant $prog.Ref.update \rightarrow (Haskell) Heap.writeIORef$
code-printing constant $HOL.equal :: 'a ref \Rightarrow 'a ref \Rightarrow bool \rightarrow (Haskell) infix 4 ==$
code-printing class-instance $ref :: HOL.equal \rightarrow (Haskell) -$

The target language only has to provide one-dimensional arrays indexed by *integer*.

$\langle ML \rangle$

definition $new' :: integer \Rightarrow 'a \Rightarrow 'a :: heap.rep\ one-dim-array\ imp\ \mathbf{where}$
 $new' k v = prog.action \{(a, s, s') \mid a\ s\ s'. (a, s') \in ODArry.alloc\ (replicate\ (nat-of-integer\ k)\ v)\ s\}$

declare $prog.Array.new'-def[code\ del]$

lemma $new-new'[code]:$

shows $prog.Array.new\ b\ v = prog.Array.new'\ (of-nat\ (length\ (Ix.interval\ b)))\ v \ggg prog.return \circ Array\ b$
 $\langle proof \rangle$

definition $make' :: integer \Rightarrow (integer \Rightarrow 'a) \Rightarrow 'a :: heap.rep\ one-dim-array\ imp\ \mathbf{where}$

$make' k f = prog.action \{(a, s, s') \mid a\ s\ s'. (a, s') \in ODArry.alloc\ (map\ (f \circ of-nat)\ [0..<nat-of-integer\ k])\ s\}$

declare $prog.Array.make'-def[code\ del]$

lemma $make-make'[code]:$

shows $prog.Array.make\ b\ f$
 $= prog.Array.make'\ (of-nat\ (length\ (Ix.interval\ b)))\ (\lambda i. f\ (Ix.interval\ b\ !\ nat-of-integer\ i))$
 $\ggg prog.return \circ Array\ b$
 $\langle proof \rangle$

definition $of-list' :: integer \Rightarrow 'a\ list \Rightarrow 'a :: heap.rep\ one-dim-array\ imp\ \mathbf{where}$

$of-list' k xs = prog.action \{(a, s, s') \mid a\ s\ s'. nat-of-integer\ k \leq length\ xs \wedge (a, s') \in ODArry.alloc\ xs\ s\}$

declare $prog.Array.of-list'-def[code\ del]$

lemma $of-list-of-list'[code]:$

shows $prog.Array.of-list\ b\ xs$
 $= prog.Array.of-list'\ (of-nat\ (length\ (Ix.interval\ b)))\ xs \ggg prog.return \circ Array\ b$
 $\langle proof \rangle$

definition $nth' :: 'a :: heap.rep\ one-dim-array \Rightarrow integer \Rightarrow 'a\ imp\ \mathbf{where}$

$nth' a i = prog.read\ (ODArry.get\ a\ (nat-of-integer\ i))$

declare $prog.Array.nth'-def[code\ del]$

lemma $nth-nth'[code]:$

shows $prog.Array.nth\ a\ i = prog.Array.nth'\ (array.arr\ a)\ (of-nat\ (Array.index\ a\ i))$
 $\langle proof \rangle$

definition $upd' :: 'a :: heap.rep\ one-dim-array \Rightarrow integer \Rightarrow 'a :: heap.rep \Rightarrow unit\ imp\ \mathbf{where}$

$upd' a i v = prog.write\ (ODArry.set\ a\ (nat-of-integer\ i)\ v)$

declare $prog.Array.upd'-def[code\ del]$

lemma $upd-upd'[code]:$

shows $prog.Array.upd\ a\ i\ v = prog.Array.upd'\ (array.arr\ a)\ (of-nat\ (Array.index\ a\ i))\ v$
 $\langle proof \rangle$

$\langle ML \rangle$

code-printing type-constructor $one\text{-}dim\text{-}array \rightarrow (Haskell) Heap.Array / -$
code-printing constant $one\text{-}dim\text{-}array.Array \rightarrow (Haskell) error / bare Array$
code-printing constant $prog.Array.new' \rightarrow (Haskell) Heap.newArray$
code-printing constant $prog.Array.make' \rightarrow (Haskell) Heap.newFunArray$
code-printing constant $prog.Array.of\text{-}list' \rightarrow (Haskell) Heap.newListArray$
code-printing constant $prog.Array.nth' \rightarrow (Haskell) Heap.readArray$
code-printing constant $prog.Array.upd' \rightarrow (Haskell) Heap.writeArray$
code-printing constant $HOL.equal :: ('i, 'a) array \Rightarrow ('i, 'a) array \Rightarrow bool \rightarrow (Haskell) \text{infix } 4 ==$
code-printing class-instance $array :: HOL.equal \rightarrow (Haskell) -$

26.2 Value-returning parallel

definition $parallelP' :: 'a::heap.rep \text{imp} \Rightarrow 'b::heap.rep \text{imp} \Rightarrow ('a \times 'b) \text{imp}$ **where**
 $parallelP' P_1 P_2 = do \{$
 $r_1 \leftarrow prog.Ref.ref\ undefined$
 $; r_2 \leftarrow prog.Ref.ref\ undefined$
 $; ((P_1 \gg= prog.Ref.update\ r_1) \parallel (P_2 \gg= prog.Ref.update\ r_2))$
 $; v_1 \leftarrow prog.Ref.lookup\ r_1$
 $; v_2 \leftarrow prog.Ref.lookup\ r_2$
 $; prog.return\ (v_1, v_2)$
 $\}$

27 Total store order (TSO)

The total store order (TSO) memory model (Owens, Sarkar, and Sewell (2009); valid on multicore x86) can be modelled as a closure as demonstrated by Jagadeesan, Petri, and Riely (2012, p182). Essentially this is done by incorporating a write buffer into each thread's local state and adding buffer draining opportunities before and after every command. The only subtlety is that the all threads involved in a parallel composition need to start and end with empty write buffers (see §27).

We configure the code generator in §27.3.

Comparison with Jagadeesan et al. (2012):

- We ignore mumbling-related issues and it doesn't make any difference
 - in our model we commit writes one at a time; mumbling allows several to be committed at once (p182) which we model as an uninterrupted sequence of individual writes
 - if we allowed *commit-writes* to commit multiple writes in a single step then *tso-closure* would not be idempotent
- their semantics is for terminating computations only; ours is for safety only
- their language is deterministic, ours is non-deterministic
- They do not provide many general laws for TSO
- Their claims that the semantics allows them to prove things (§5) is not substantiated

type-synonym $write\text{-}buffer = heap.write\ list$

definition $apply\text{-}writes :: write\text{-}buffer \Rightarrow heap.t \Rightarrow heap.t$ **where**
 $apply\text{-}writes\ ws = fold\ (\lambda w. (\circ)\ (heap.apply\text{-}write\ w))\ ws\ id$

lemma $apply\text{-}write\text{-}present$:

assumes $heap.present\ r\ s$

shows $heap.present\ r\ (heap.apply\text{-}write\ w\ s)$

$\langle \text{proof} \rangle$

lemma *apply-writes-present*:

assumes *heap.present* r s

shows *heap.present* r (*apply-writes* wb s)

$\langle \text{proof} \rangle$

$\langle ML \rangle$

type-synonym $'v$ *tso* = *write-buffer* \Rightarrow (*heap.t*, $'v \times$ *write-buffer*) *prog*

definition *bind* :: $'a$ *raw.tso* \Rightarrow ($'a \Rightarrow 'b$ *raw.tso*) \Rightarrow $'b$ *raw.tso* **where**

bind f g = ($\lambda wb.$ f wb \ggg *uncurry* g)

adhoc-overloading

Monad-Syntax.bind *raw.bind*

definition *prim-return* :: $'a \Rightarrow 'a$ *raw.tso* **where**

prim-return v = ($\lambda wb.$ *prog.return* (v , wb))

$\langle ML \rangle$

lemma *mono*:

assumes $f \leq f'$

assumes $\bigwedge x. g$ $x \leq g' x$

shows *raw.bind* f $g \leq$ *raw.bind* $f' g'$

$\langle \text{proof} \rangle$

lemma *strengthen[stg]*:

assumes *st-ord* F f f'

assumes $\bigwedge x. \text{st-ord } F (g$ $x) (g' x)$

shows *st-ord* F (*raw.bind* f g) (*raw.bind* $f' g'$)

$\langle \text{proof} \rangle$

lemma *mono2mono[cont-intro, partial-function-mono]*:

assumes *monotone* *orda* (\leq) F

assumes $\bigwedge x. \text{monotone } orda (\leq) (\lambda y. G$ y $x)$

shows *monotone* *orda* (\leq) ($\lambda f. \text{raw.bind } (F$ $f) (G$ $f)$)

$\langle \text{proof} \rangle$

lemma *botL*:

shows *raw.bind* \perp $g = \perp$

$\langle \text{proof} \rangle$

lemma *bind*:

fixes f :: - *raw.tso*

shows $f \ggg g \ggg h = f \ggg (\lambda x. g$ $x \ggg h)$

$\langle \text{proof} \rangle$

lemma *prim-return*:

shows *prim-returnL*: *raw.bind* (*raw.prim-return* v) = ($\lambda g. g$ v)

and *prim-returnR*: $f \ggg \text{raw.prim-return} = f$

$\langle \text{proof} \rangle$

lemma *supL*:

fixes g :: - \Rightarrow - *raw.tso*

shows $f_1 \sqcup f_2 \ggg g = (f_1 \ggg g) \sqcup (f_2 \ggg g)$

$\langle \text{proof} \rangle$

lemma *supR*:

fixes $f :: - \text{ raw.tso}$

shows $f \ggg (\lambda v. g_1 v \sqcup g_2 v) = (f \ggg g_1) \sqcup (f \ggg g_2)$

$\langle \text{proof} \rangle$

lemma *SUPL*:

fixes $X :: - \text{ set}$

fixes $f :: - \Rightarrow - \text{ raw.tso}$

shows $(\bigsqcup x \in X. f x) \ggg g = (\bigsqcup x \in X. f x \ggg g)$

$\langle \text{proof} \rangle$

lemma *SUPR*:

fixes $X :: - \text{ set}$

fixes $f :: - \text{ raw.tso}$

shows $f \ggg (\lambda v. \bigsqcup x \in X. g x v) = (\bigsqcup x \in X. f \ggg g x) \sqcup (f \ggg \perp)$

$\langle \text{proof} \rangle$

lemma *SUPR-not-empty*:

fixes $f :: - \text{ raw.tso}$

assumes $X \neq \{\}$

shows $f \ggg (\lambda v. \bigsqcup x \in X. g x v) = (\bigsqcup x \in X. f \ggg g x)$

$\langle \text{proof} \rangle$

lemma *mcont2mcont[cont-intro]*:

assumes $mcont \text{ luba } orda \text{ Sup } (\leq) f$

assumes $\bigwedge v. mcont \text{ luba } orda \text{ Sup } (\leq) (\lambda x. g x v)$

shows $mcont \text{ luba } orda \text{ Sup } (\leq) (\lambda x. \text{raw.bind } (f x) (g x))$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

interpretation *kleene*: $kleene \text{ raw.prim-return } () \lambda x y. \text{raw.bind } x \langle y \rangle$

$\langle \text{proof} \rangle$

primrec *commit-write* :: $unit \text{ raw.tso}$ **where**

$commit\text{-write } [] = \text{prog.return } ((), [])$

$| \text{commit-write } (w \# wb) = \text{prog.action } \{(((), wb), h, \text{heap.apply-write } w h) | h. \text{True}\}$

definition *commit-writes* :: $unit \text{ raw.tso}$ **where**

$commit\text{-writes} = \text{raw.kleene.star } \text{raw.commit-write}$

$\langle ML \rangle$

definition *cl* :: $'v \text{ raw.tso} \Rightarrow 'v \text{ raw.tso}$ **where**

$cl P = \text{raw.commit-writes} \ggg P \ggg (\lambda v. \text{raw.commit-writes} \ggg \text{raw.prim-return } v)$

$\langle ML \rangle$

definition *action* :: $(\text{write-buffer} \Rightarrow ('v \times \text{write-buffer} \times \text{heap.t} \times \text{heap.t}) \text{ set}) \Rightarrow 'v \text{ raw.tso}$ **where**

$action F = \text{raw.tso.cl } (\lambda wb. \text{prog.action } \{((v, wb @ ws), ss') | v ss' ws. (v, ws, ss') \in F wb\})$

definition *return* :: $'v \Rightarrow 'v \text{ raw.tso}$ **where**

$return v = \text{raw.action } \{\{v\} \times \{\}\} \times Id$

definition *guard* :: $(\text{write-buffer} \Rightarrow \text{heap.t pred}) \Rightarrow unit \text{ raw.tso}$ **where**

$guard g = \text{raw.action } (\lambda wb. \{\}\} \times \{\}\} \times \text{Diag } (g wb))$

definition *MFENCE* :: unit raw.tso **where**

MFENCE = raw.guard ($\lambda wb s. wb = []$)

definition *vmap* :: ('v \Rightarrow 'w) \Rightarrow 'v raw.tso \Rightarrow 'w raw.tso **where**

vmap *vf* *P* = ($\lambda wb. prog.vmap (map-prod\ vf\ id) (P\ wb)$)

— Parallel composition

definition *t2p* :: 'v raw.tso \Rightarrow (heap.t, 'v) prog **where**

t2p *P* = *P* [] $\ggg (\lambda(v, wb). raw.MFENCE\ wb \gg prog.return\ v)$

— Jagadeesan et al. (2012, p184 rule PAR-CMD): perform MFENCE before fork

definition *parallel* :: unit raw.tso \Rightarrow unit raw.tso \Rightarrow unit raw.tso **where**

parallel *P* *Q* = raw.MFENCE $\gg \langle (raw.t2p\ P \parallel raw.t2p\ Q) \gg prog.return\ ((), []) \rangle$

lemma *return-alt-def*:

shows raw.return = ($\lambda v. raw.tso.cl (raw.prim-return\ v)$)

$\langle proof \rangle$

$\langle ML \rangle$

lemma *return-le*:

shows raw.prim-return () $\leq raw.commit-writes$

$\langle proof \rangle$

lemma *return-le'*:

shows prog.return ((), wb) $\leq raw.commit-writes\ wb$

$\langle proof \rangle$

lemma *commit-writes*:

shows raw.commit-writes $\gg raw.commit-writes = raw.commit-writes$

$\langle proof \rangle$

lemma *Nil*:

shows raw.commit-writes [] = prog.return ((), []) (**is** ?lhs = ?rhs)

$\langle proof \rangle$

lemma *Cons*:

shows raw.commit-writes (*w* # *wb*)

= (raw.commit-write [*w*] $\gg raw.commit-writes\ wb$) $\sqcup raw.prim-return\ () (w\ \# wb)$

$\langle proof \rangle$

lemma *Cons-le*:

shows raw.commit-write [*w*] $\gg raw.commit-writes\ wb \leq raw.commit-writes (w\ \# wb)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *prim-return-Nil-le*:

shows $\langle s, [], Some\ ((),\ wb) \rangle \leq prog.p2s (raw.prim-return\ ()\ wb)$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *noop-le*:

shows $\langle s, [], Some\ ((),\ wb) \rangle \leq prog.p2s (raw.commit-writes\ wb)$

$\langle proof \rangle$

lemma *wb-suffix*:

assumes $\langle s, xs, \text{Some } (\cdot), wb \rangle \leq \text{prog.p2s } (\text{raw.commit-writes } wb)$
shows $\text{suffix } wb' \text{ } wb$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *bind-commit-writes-absorbL*:

fixes $P :: 'v \text{ raw.tso}$
shows $\text{raw.commit-writes} \gg \text{raw.tso.cl } P = \text{raw.tso.cl } P$
 $\langle \text{proof} \rangle$

lemma *bind-commit-writes-absorb-unitR*:

fixes $P :: \text{unit raw.tso}$
shows $\text{raw.tso.cl } P \gg \text{raw.commit-writes} = \text{raw.tso.cl } P$
 $\langle \text{proof} \rangle$

lemma *bind-commit-writes-absorbR*:

fixes $P :: 'v \text{ raw.tso}$
shows $\text{raw.tso.cl } P \gg (\lambda v. \text{raw.commit-writes} \gg \text{raw.prim-return } v) = \text{raw.tso.cl } P$
 $\langle \text{proof} \rangle$

lemma *bot*:

shows $\text{raw.tso.cl } \perp = \text{raw.commit-writes} \gg \perp$
 $\langle \text{proof} \rangle$

lemma *prim-return*:

shows $\text{raw.tso.cl } (\text{raw.prim-return } v) = \text{raw.commit-writes} \gg \text{raw.prim-return } v$
 $\langle \text{proof} \rangle$

lemma *Nil*:

shows $\text{raw.tso.cl } P [] = P [] \gg (\lambda v. \text{raw.commit-writes } (\text{snd } v) \gg (\lambda w. \text{prog.return } (\text{fst } v, \text{snd } w)))$
 $\langle \text{proof} \rangle$

lemma *commit*:

fixes $wb :: \text{write-buffer}$
shows $\text{raw.commit-write } [w] \gg f \text{ } wb \leq \text{raw.tso.cl } f \text{ } (w \# wb)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

interpretation *tso*: *closure-complete-distrib-lattice-distributive-class* raw.tso.cl

$\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *bind*:

fixes $f :: 'v \text{ raw.tso}$
assumes $f \in \text{raw.tso.closed}$
shows $\text{raw.tso.cl } (f \gg g) = f \gg (\lambda v. \text{raw.tso.cl } (g \ v))$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *commit-writes-absorbL*:

assumes $f \in \text{raw.tso.closed}$
shows $\text{raw.commit-writes} \gg f = f$
 $\langle \text{proof} \rangle$

lemma *commit-writes-absorb-unitR*:
assumes $f \in \text{raw.tso.closed}$
shows $f \gg \text{raw.commit-writes} = f$
 $\langle \text{proof} \rangle$

lemma *returnL*:
assumes $g \ v \in \text{raw.tso.closed}$
shows $\text{raw.return } v \ggg g = g \ v$
 $\langle \text{proof} \rangle$

lemma *returnR*:
assumes $f \in \text{raw.tso.closed}$
shows $f \ggg \text{raw.return} = f$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *commit-writes*:
shows $\text{raw.commit-writes} \in \text{raw.tso.closed}$
 $\langle \text{proof} \rangle$

lemma *bind[intro]*:
fixes $f :: 'v \text{ raw.tso}$
fixes $g :: 'v \Rightarrow 'w \text{ raw.tso}$
assumes $f \in \text{raw.tso.closed}$
assumes $\bigwedge x. g \ x \in \text{raw.tso.closed}$
shows $f \ggg g \in \text{raw.tso.closed}$
 $\langle \text{proof} \rangle$

lemma *action[intro]*:
shows $\text{raw.action } F \in \text{raw.tso.closed}$
 $\langle \text{proof} \rangle$

lemma *guard[intro]*:
shows $\text{raw.guard } g \in \text{raw.tso.closed}$
 $\langle \text{proof} \rangle$

lemma *MFENCE[intro]*:
shows $\text{raw.MFENCE} \in \text{raw.tso.closed}$
 $\langle \text{proof} \rangle$

lemma *parallel[intro]*:
assumes $P \in \text{raw.tso.closed}$
assumes $Q \in \text{raw.tso.closed}$
shows $\text{raw.parallel } P \ Q \in \text{raw.tso.closed}$
 $\langle \text{proof} \rangle$

lemma *vmap[intro]*:
assumes $P \in \text{raw.tso.closed}$
shows $\text{raw.vmap } vf \ P \in \text{raw.tso.closed}$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *bot*:
shows $\text{raw.action } \perp = \text{raw.tso.cl } \perp$
 $\langle \text{proof} \rangle$

lemma *monotone*:

shows *mono raw.action*

\langle *proof* \rangle

lemmas *strengthen[strg] = st-monotone[OF raw.action.monotone]*

lemmas *mono = monotoneD[OF raw.action.monotone]*

lemma *Sup*:

shows *raw.action* $(\sqcup Fs) = \sqcup (raw.action \text{ ' } Fs) \sqcup raw.tso.cl \perp$ (**is** *?lhs = ?rhs*)

\langle *proof* \rangle

lemma *sup*:

shows *raw.action* $(F \sqcup G) = raw.action F \sqcup raw.action G$

\langle *proof* \rangle

\langle *ML* \rangle

lemma *return-le*:

shows *raw.guard* $g \leq raw.return ()$

\langle *proof* \rangle

lemma *monotone*:

shows *mono* $(raw.guard :: (write-buffer \Rightarrow heap.t pred) \Rightarrow -)$

\langle *proof* \rangle

lemmas *strengthen[strg] = st-monotone[OF raw.guard.monotone]*

lemmas *mono = monotoneD[OF raw.guard.monotone]*

lemma *less*: — Non-triviality; essentially replay *prog.guard.less*

assumes $g < g'$

shows *raw.guard* $g < raw.guard g'$

\langle *proof* \rangle

\langle *ML* \rangle

lemma *MFENCE-alt-def*:

shows *raw.MFENCE* $= raw.commit-writes \gg (\lambda wb. prog.action (\{(), wb\} \times Diag \langle wb = [] \rangle))$

\langle *proof* \rangle

\langle *ML* \rangle

lemma *Nil*:

shows *raw.MFENCE* $[] = prog.return ((), [])$

\langle *proof* \rangle

\langle *ML* \rangle

lemma *MFENCE*:

shows *prog.p2s* $(raw.MFENCE wb) \leq \{P\}$, $A \Vdash prog.p2s (raw.MFENCE wb)$, $\{\lambda v s. snd v = []\}$

\langle *proof* \rangle

\langle *ML* \rangle

lemma *MFENCEL*:

shows *raw.MFENCE* $wb \gg g = raw.MFENCE wb \gg g ((), [])$ (**is** *?lhs = ?rhs*)

\langle *proof* \rangle

lemma *MFENCE-return*:

shows $raw.MFENCE\ wb \gg prog.return\ ((),\ []) = raw.MFENCE\ wb$
 $\langle proof \rangle$

lemma *MFENCE-MFENCE*:

shows $raw.MFENCE \gg raw.MFENCE = raw.MFENCE$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *bot*:

shows $raw.t2p\ \perp = \perp$
 $\langle proof \rangle$

lemma *cl-bot*:

shows $raw.t2p\ (raw.tso.cl\ \perp) = \perp$
 $\langle proof \rangle$

lemma *monotone*:

shows $mono\ raw.t2p$
 $\langle proof \rangle$

lemmas $strengthen[strg] = st-monotone[OF\ raw.t2p.monotone]$

lemmas $mono = monotoneD[OF\ raw.t2p.monotone]$

lemmas $mono2mono[cont-intro, partial-function-mono] = monotone2monotone[OF\ raw.t2p.monotone, simplified]$

lemma *Sup*:

shows $raw.t2p\ (\sqcup X) = \sqcup (raw.t2p\ ` X)$
 $\langle proof \rangle$

lemma *sup*:

shows $raw.t2p\ (P\ \sqcup\ Q) = raw.t2p\ P\ \sqcup\ raw.t2p\ Q$
 $\langle proof \rangle$

lemma *mcont2mcont[cont-intro]*:

fixes $P :: - \Rightarrow -\ raw.tso$
assumes $mcont\ luba\ orda\ Sup\ (\leq)\ F$
shows $mcont\ luba\ orda\ Sup\ (\leq)\ (\lambda x. raw.t2p\ (F\ x))$
 $\langle proof \rangle$

lemma *return*:

shows $raw.t2p\ (raw.return\ v) = prog.return\ v$
 $\langle proof \rangle$

lemma *MFENCE-bind*:

shows $raw.t2p\ (raw.MFENCE\ \gg\ P) = raw.t2p\ (P\ ())$
 $\langle proof \rangle$

lemma *bind-return-unit*:

shows $raw.t2p\ (\lambda wb. prog.bind\ P\ (\lambda :: unit. prog.return\ ((),\ []))) = P$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *commute*: — Jagadeesan et al. (2012, §5 (3))

shows $raw.parallel\ P\ Q = raw.parallel\ Q\ P$
 $\langle proof \rangle$

lemma *assoc*: — Jagadeesan et al. (2012, §5 (4))

shows $\text{raw.parallel } P (\text{raw.parallel } Q R) = \text{raw.parallel } (\text{raw.parallel } P Q) R$
 $\langle \text{proof} \rangle$

lemma *mono*:

assumes $P \leq P'$

assumes $Q \leq Q'$

shows $\text{raw.parallel } P Q \leq \text{raw.parallel } P' Q'$

$\langle \text{proof} \rangle$

lemma *botL*:

shows $\text{raw.parallel } (\text{raw.tso.cl } \perp) f = \text{raw.MFENCE} \gg f \gg \text{raw.MFENCE} \gg \text{raw.tso.cl } \perp$

$\langle \text{proof} \rangle$

lemma *returnL*:

shows $\text{raw.parallel } (\text{raw.return } ()) P = \text{raw.MFENCE} \gg (\lambda-. P) \gg (\lambda-. \text{raw.MFENCE})$

$\langle \text{proof} \rangle$

lemma *SupL-not-empty*:

assumes $\forall x \in X. x \in \text{raw.tso.closed}$

assumes $Q \in \text{raw.tso.closed}$

assumes $X \neq \{\}$

shows $\text{raw.parallel } (\bigsqcup X \sqcup \text{raw.tso.cl } \perp) Q = (\bigsqcup P \in X. \text{raw.parallel } P Q) \sqcup \text{raw.tso.cl } \perp$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

typedef $'v \text{ tso} = \text{raw.tso.closed} :: 'v \text{ raw.tso set}$

morphisms $t2p' \text{ Abs-tso}$

$\langle \text{proof} \rangle$

setup-lifting *type-definition-tso*

instantiation $\text{tso} :: (\text{type}) \text{ complete-distrib-lattice}$

begin

lift-definition $\text{bot-tso} :: 'v \text{ tso is raw.tso.cl } \perp \langle \text{proof} \rangle$

lift-definition $\text{top-tso} :: 'v \text{ tso is } \top \langle \text{proof} \rangle$

lift-definition $\text{sup-tso} :: 'v \text{ tso} \Rightarrow 'v \text{ tso} \Rightarrow 'v \text{ tso is sup} \langle \text{proof} \rangle$

lift-definition $\text{inf-tso} :: 'v \text{ tso} \Rightarrow 'v \text{ tso} \Rightarrow 'v \text{ tso is inf} \langle \text{proof} \rangle$

lift-definition $\text{less-eq-tso} :: 'v \text{ tso} \Rightarrow 'v \text{ tso} \Rightarrow \text{bool is less-eq} \langle \text{proof} \rangle$

lift-definition $\text{less-tso} :: 'v \text{ tso} \Rightarrow 'v \text{ tso} \Rightarrow \text{bool is less} \langle \text{proof} \rangle$

lift-definition $\text{Inf-tso} :: 'v \text{ tso set} \Rightarrow 'v \text{ tso is Inf} \langle \text{proof} \rangle$

lift-definition $\text{Sup-tso} :: 'v \text{ tso set} \Rightarrow 'v \text{ tso is } \lambda X. \text{Sup } X \sqcup \text{raw.tso.cl } \perp \langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

$\langle ML \rangle$

lift-definition $\text{bind} :: 'v \text{ tso} \Rightarrow ('v \Rightarrow 'w \text{ tso}) \Rightarrow 'w \text{ tso is raw.bind} \langle \text{proof} \rangle$

lift-definition $\text{action} :: (\text{write-buffer} \Rightarrow ('v \times \text{write-buffer} \times \text{heap.t} \times \text{heap.t}) \text{ set}) \Rightarrow 'v \text{ tso is raw.action} \langle \text{proof} \rangle$

lift-definition $\text{MFENCE} :: \text{unit tso is raw.MFENCE} \langle \text{proof} \rangle$

lift-definition $\text{parallel} :: \text{unit tso} \Rightarrow \text{unit tso} \Rightarrow \text{unit tso is raw.parallel} \langle \text{proof} \rangle$

lift-definition $\text{vmap} :: ('v \Rightarrow 'w) \Rightarrow 'v \text{ tso} \Rightarrow 'w \text{ tso is raw.vmap} \langle \text{proof} \rangle$

lift-definition $\text{t2p} :: 'v \text{ tso} \Rightarrow (\text{heap.t}, 'v) \text{ prog is raw.t2p} \langle \text{proof} \rangle$

adhoc-overloading

Monad-Syntax.bind tso.bind

adhoc-overloading

parallel tso.parallel

definition *return* :: 'v ⇒ 'v tso **where**

return v = tso.action ⟨{v} × {} × Id⟩

definition *guard* :: (write-buffer ⇒ heap.t pred) ⇒ unit tso **where**

guard g = tso.action (λwb. {} × {} × Diag (g wb))

abbreviation (input) *read* :: (heap.t ⇒ 'v) ⇒ 'v tso **where**

read f ≡ tso.action (λwb. {(f (apply-writes wb s), [], s, s) | s. True})

abbreviation (input) *write* :: (heap.t ⇒ heap.write) ⇒ unit tso **where**

write f ≡ tso.action ⟨{(), [f s], s, s} | s. True⟩

lemma *return-alt-def*:

shows *tso.return v = tso.read* ⟨v⟩

⟨proof⟩

declare *tso.bind-def*[code del]

declare *tso.action-def*[code del]

declare *tso.return-def*[code del]

declare *tso.MFENCE-def*[code del]

declare *tso.parallel-def*[code del]

declare *tso.vmap-def*[code del]

⟨ML⟩

lemma *transfer*[transfer-rule]:

shows *rel-fun* (=) *cr-tso raw.return tso.return*

⟨proof⟩

⟨ML⟩

lemma *empty*:

shows *bot*: *tso.action* ⊥ = ⊥

and *tso.action* (λ-. {}) = ⊥

⟨proof⟩

lemmas *monotone* = *raw.action.monotone*[transferred]

lemmas *strengthen*[strg] = *st-monotone*[OF *tso.action.monotone*]

lemmas *mono* = *monotoneD*[OF *tso.action.monotone*]

lemmas *mono2mono*[cont-intro, partial-function-mono] = *monotone2monotone*[OF *tso.action.monotone*, simplified]

lemma *Sup*:

shows *tso.action* (⊔ Fs) = ⊔ (tso.action ‘ Fs)

⟨proof⟩

lemmas *sup* = *tso.action.Sup*[**where** Fs={F, G} **for** F G, simplified]

⟨ML⟩

lemmas *if-distrL* = *if-distrib*[**where** f=λf. *tso.bind f g* **for** g] — Jagadeesan et al. (2012, §5 (5))

lemmas *mono* = *raw.bind.mono*[transferred]

lemma *strengthen*[*strg*]:

assumes *st-ord* $F f f'$

assumes $\bigwedge x. \textit{st-ord } F (g x) (g' x)$

shows *st-ord* $F (\textit{tso.bind } f g) (\textit{tso.bind } f' g')$

$\langle \textit{proof} \rangle$

lemmas *mono2mono*[*cont-intro, partial-function-mono*] = *raw.bind.mono2mono*[*transferred*]

lemma *bind*: — Jagadeesan et al. (2012, §5 (2))

shows $f \ggg g \ggg h = \textit{tso.bind } f (\lambda x. g x \ggg h)$

$\langle \textit{proof} \rangle$

lemma *return*: — Jagadeesan et al. (2012, §5 (1))

shows *returnL*: $\textit{tso.return } v \ggg g = g v$

and *returnR*: $f \ggg \textit{tso.return} = f$

$\langle \textit{proof} \rangle$

lemma *botL*:

shows $\textit{tso.bind } \perp g = \perp$

$\langle \textit{proof} \rangle$

lemma *botR-le*:

shows $\textit{tso.bind } f \langle \perp \rangle \leq f$ (**is** *?thesis1*)

and $\textit{tso.bind } f \perp \leq f$ (**is** *?thesis2*)

$\langle \textit{proof} \rangle$

lemma

fixes $f :: - \textit{tso}$

fixes $f_1 :: - \textit{tso}$

shows *supL*: $(f_1 \sqcup f_2) \ggg g = (f_1 \ggg g) \sqcup (f_2 \ggg g)$

and *supR*: $f \ggg (\lambda x. g_1 x \sqcup g_2 x) = (f \ggg g_1) \sqcup (f \ggg g_2)$

$\langle \textit{proof} \rangle$

lemma *SUPL*:

fixes $X :: - \textit{set}$

fixes $f :: - \Rightarrow - \textit{tso}$

shows $(\bigsqcup_{x \in X}. f x) \ggg g = (\bigsqcup_{x \in X}. f x \ggg g)$

$\langle \textit{proof} \rangle$

lemma *SUPR*:

fixes $X :: - \textit{set}$

fixes $f :: - \textit{tso}$

shows $f \ggg (\lambda v. \bigsqcup_{x \in X}. g x v) = (\bigsqcup_{x \in X}. f \ggg g x) \sqcup (f \ggg \perp)$

$\langle \textit{proof} \rangle$

lemma *SupR*:

fixes $X :: - \textit{set}$

fixes $f :: - \textit{tso}$

shows $f \gg (\bigsqcup X) = (\bigsqcup_{x \in X}. f \gg x) \sqcup (f \gg \perp)$

$\langle \textit{proof} \rangle$

lemma *SUPR-not-empty*:

fixes $f :: - \textit{tso}$

assumes $X \neq \{\}$

shows $f \ggg (\lambda v. \bigsqcup_{x \in X}. g x v) = (\bigsqcup_{x \in X}. f \ggg g x)$

$\langle \textit{proof} \rangle$

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* (\leq) *f*
assumes $\bigwedge v. mcont luba orda Sup$ (\leq) $(\lambda x. g x v)$
shows *mcont luba orda Sup* (\leq) $(\lambda x. tso.bind (f x) (g x))$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *transfer*[*transfer-rule*]:
shows *rel-fun* (=) *cr-tso raw.guard tso.guard*
 $\langle proof \rangle$

lemma *bot*:
shows *tso.guard* $\perp = \perp$
and *tso.guard* $(\lambda - . False) = \perp$
 $\langle proof \rangle$

lemma *top*:
shows *tso.guard* $\top = tso.return ()$ (**is** *?thesis1*)
and *tso.guard* $(\lambda -. \top) = tso.return ()$ (**is** *?thesis2*)
and *tso.guard* $(\lambda - . True) = tso.return ()$ (**is** *?thesis3*)
 $\langle proof \rangle$

lemma *return-le*:
shows *tso.guard g* $\leq tso.return ()$
 $\langle proof \rangle$

lemma *monotone*:
shows *mono tso.guard*
 $\langle proof \rangle$

lemmas *strengthen*[*strg*] = *st-monotone*[*OF tso.guard.monotone*]
lemmas *mono* = *monotoneD*[*OF tso.guard.monotone*]
lemmas *mono2mono*[*cont-intro, partial-function-mono*] = *monotone2monotone*[*OF tso.guard.monotone, simplified*]

lemma *less*: — Non-triviality
assumes $g < g'$
shows *tso.guard g* $< tso.guard g'$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *commute*: — Jagadeesan et al. (2012, §5 (3))
shows *tso.parallel P Q* = *tso.parallel Q P*
 $\langle proof \rangle$

lemma *assoc*: — Jagadeesan et al. (2012, §5 (4))
shows *tso.parallel P (tso.parallel Q R)* = *tso.parallel (tso.parallel P Q) R*
 $\langle proof \rangle$

lemmas *mono* = *raw.parallel.mono*[*transferred*]

lemma *strengthen*[*strg*]:
assumes *st-ord F P P'*
assumes *st-ord F Q Q'*
shows *st-ord F (tso.parallel P Q) (tso.parallel P' Q')*
 $\langle proof \rangle$

lemma *mono2mono*[*cont-intro, partial-function-mono*]:
assumes *monotone orda* (\leq) F
assumes *monotone orda* (\leq) G
shows *monotone orda* (\leq) $(\lambda f. \text{tso.parallel } (F f) (G f))$
 $\langle \text{proof} \rangle$

lemma *bot*:
shows *parallel-botL*: $\text{tso.parallel } \perp f = \text{tso.MFENCE} \gg f \gg \text{tso.MFENCE} \gg \perp$ (**is** *?thesis1*)
and *parallel-botR*: $\text{tso.parallel } f \perp = \text{tso.MFENCE} \gg f \gg \text{tso.MFENCE} \gg \perp$ (**is** *?thesis2*)
 $\langle \text{proof} \rangle$

lemma *return*: — Jagadeesan et al. (2012, unnumbered)
shows *returnL*: $\text{tso.return } () \parallel P = \text{tso.MFENCE} \gg P \gg \text{tso.MFENCE}$ (**is** *?thesis1*)
and *returnR*: $P \parallel \text{tso.return } () = \text{tso.MFENCE} \gg P \gg \text{tso.MFENCE}$ (**is** *?thesis2*)
 $\langle \text{proof} \rangle$

lemma *Sup-not-empty*:
fixes $X :: \text{unit tso set}$
assumes $X \neq \{\}$
shows *SupL-not-empty*: $\bigsqcup X \parallel Q = (\bigsqcup P \in X. P \parallel Q)$ (**is** *?thesis1* Q)
and *SupR-not-empty*: $P \parallel \bigsqcup X = (\bigsqcup Q \in X. P \parallel Q)$ (**is** *?thesis2*)
 $\langle \text{proof} \rangle$

lemma *sup*:
fixes $P :: \text{unit tso}$
shows *supL*: $P \sqcup Q \parallel R = (P \parallel R) \sqcup (Q \parallel R)$
and *supR*: $P \parallel Q \sqcup R = (P \parallel Q) \sqcup (P \parallel R)$
 $\langle \text{proof} \rangle$

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* (\leq) P
assumes *mcont luba orda Sup* (\leq) Q
shows *mcont luba orda Sup* (\leq) $(\lambda x. \text{tso.parallel } (P x) (Q x))$
 $\langle \text{proof} \rangle$
 $\langle \text{ML} \rangle$

lemmas *MFENCE-MFENCE* = *raw.bind.MFENCE-MFENCE*[*transferred*]
 $\langle \text{ML} \rangle$

lemma *monotone*:
shows *mono* $(\lambda t. \text{t2p}' t \text{wb})$
 $\langle \text{proof} \rangle$

lemmas *strengthen*[*strg*] = *st-monotone*[*OF tso.t2p'.monotone*]
lemmas *mono* = *monotoneD*[*OF tso.t2p'.monotone*]

lemmas *action* = *tso.action.rep-eq*
lemma *return*:
shows $\text{t2p}' (\text{tso.return } v) = \text{raw.return } v$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

Combinators $\langle \text{ML} \rangle$

abbreviation $guardM :: bool \Rightarrow unit\ tso\ \mathbf{where}$

$guardM\ b \equiv \text{if } b \text{ then } \perp \text{ else } tso.\text{return } ()$

abbreviation $unlessM :: bool \Rightarrow unit\ tso \Rightarrow unit\ tso\ \mathbf{where}$

$unlessM\ b\ c \equiv \text{if } b \text{ then } tso.\text{return } () \text{ else } c$

abbreviation $whenM :: bool \Rightarrow unit\ tso \Rightarrow unit\ tso\ \mathbf{where}$

$whenM\ b\ c \equiv \text{if } b \text{ then } c \text{ else } tso.\text{return } ()$

definition $app :: ('a \Rightarrow unit\ tso) \Rightarrow 'a\ list \Rightarrow unit\ tso\ \mathbf{where}$ — Haskell's $mapM$ -

$app\ f\ xs = \text{foldr } (\lambda x\ m. f\ x \gg m)\ xs\ (tso.\text{return } ())$

primrec $fold\text{-}mapM :: ('a \Rightarrow 'b\ tso) \Rightarrow 'a\ list \Rightarrow 'b\ list\ tso\ \mathbf{where}$

$fold\text{-}mapM\ f\ [] = tso.\text{return } []$

| $fold\text{-}mapM\ f\ (x \# xs) = \text{do } \{$

$y \leftarrow f\ x;$

$ys \leftarrow fold\text{-}mapM\ f\ xs;$

$tso.\text{return } (y \# ys)$

$\}$

— Jagadeesan et al. (2012, §5 (6) is $tso.\text{while}.\text{simps}$)

partial-function $(lfp)\ \text{while} :: ('k \Rightarrow ('k + 'v)\ tso) \Rightarrow 'k \Rightarrow 'v\ tso\ \mathbf{where}$

$\text{while}\ c\ k = c\ k \gg (\lambda rv. \text{case } rv \text{ of } Inl\ k' \Rightarrow \text{while}\ c\ k' \mid Inr\ v \Rightarrow tso.\text{return } v)$

abbreviation $(input)\ \text{while}' :: ((unit + 'v)\ tso) \Rightarrow 'v\ tso\ \mathbf{where}$

$\text{while}'\ c \equiv tso.\text{while } \langle c \rangle ()$

definition $\text{raise} :: String.\text{literal} \Rightarrow 'v\ tso\ \mathbf{where}$

$\text{raise } s = \perp$

definition $\text{assert} :: bool \Rightarrow unit\ tso\ \mathbf{where}$

$\text{assert } P = (\text{if } P \text{ then } tso.\text{return } () \text{ else } tso.\text{raise } STR\ \text{"assert"})$

declare $tso.\text{raise}\text{-}def[\text{code del}]$

$\langle ML \rangle$

lemma bot :

shows $tso.\text{fold}\text{-}mapM\ \perp = (\lambda xs. \text{case } xs \text{ of } [] \Rightarrow tso.\text{return } [] \mid - \Rightarrow \perp)$

$\langle proof \rangle$

lemma $append$:

shows $tso.\text{fold}\text{-}mapM\ f\ (xs\ @\ ys) = tso.\text{fold}\text{-}mapM\ f\ xs \gg (\lambda xs. tso.\text{fold}\text{-}mapM\ f\ ys \gg (\lambda ys. tso.\text{return } (xs\ @\ ys)))$

$\langle proof \rangle$

$\langle ML \rangle$

lemma bot :

shows $tso.\text{app } \perp = (\lambda xs. \text{case } xs \text{ of } [] \Rightarrow tso.\text{return } () \mid - \Rightarrow \perp)$

and $tso.\text{app } (\lambda -. \perp) = (\lambda xs. \text{case } xs \text{ of } [] \Rightarrow tso.\text{return } () \mid - \Rightarrow \perp)$

$\langle proof \rangle$

lemma Nil :

shows $tso.\text{app } f\ [] = tso.\text{return } ()$

$\langle proof \rangle$

lemma $Cons$:

shows $tso.app\ f\ (x\ \# \ xs) = f\ x \gg tso.app\ f\ xs$
 ⟨proof⟩

lemmas $simps =$
 $tso.app.bot$
 $tso.app.Nil$
 $tso.app.Cons$

lemma $append:$
shows $tso.app\ f\ (xs\ @\ ys) = tso.app\ f\ xs \gg tso.app\ f\ ys$
 ⟨proof⟩

lemma $monotone:$
shows $mono\ (\lambda f. tso.app\ f\ xs)$
 ⟨proof⟩

lemmas $strengthen[stg] = st-monotone[OF\ tso.app.monotone]$

lemmas $mono = monotoneD[OF\ tso.app.monotone]$

lemmas $mono2mono[cont-intro, partial-function-mono] = monotone2monotone[OF\ tso.app.monotone, simplified,$
 $of\ orda\ P\ for\ orda\ P]$

lemma $Sup-le:$
shows $(\bigsqcup f \in X. tso.app\ f\ xs) \leq tso.app\ (\bigsqcup X)\ xs$
 ⟨proof⟩

⟨ML⟩

27.1 References

Observe that allocation is global in this model. We allow the memory location to have an arbitrary value and enqueue the initialising write in the TSO buffer.

⟨ML⟩

definition $ref :: 'a::heap.rep \Rightarrow 'a\ ref\ tso\ \mathbf{where}$
 $ref\ v = tso.action\ (\lambda wb. \{(r, [heap.Write\ (ref.addr-of\ r)\ 0\ (heap.rep.to\ v)]), s, s'\}$
 $\mid r\ s\ s'\ v'. (r, s') \in Ref.alloc\ v'\ s\})$

definition $lookup :: 'a::heap.rep\ ref \Rightarrow 'a\ tso\ (!- 61)\ \mathbf{where}$
 $lookup\ r = tso.read\ (Ref.get\ r)$

definition $update :: 'a\ ref \Rightarrow 'a::heap.rep \Rightarrow unit\ tso\ (- := - 62)\ \mathbf{where}$
 $update\ r\ v = tso.write\ \langle heap.Write\ (ref.addr-of\ r)\ 0\ (heap.rep.to\ v) \rangle$

declare $tso.Ref.ref-def[code\ del]$
declare $tso.Ref.lookup-def[code\ del]$
declare $tso.Ref.update-def[code\ del]$

⟨ML⟩

27.2 Inhabitation

In order to obtain compositional rules we need to make the write buffer explicit.

⟨ML⟩

definition $t2s :: write-buffer \Rightarrow 'v\ tso \Rightarrow (sequential, heap.t, 'v \times write-buffer)\ spec\ \mathbf{where}$
 $t2s\ wb\ P = prog.p2s\ (tso.t2p'\ P\ wb)$

⟨ML⟩

lemma *t2s-commit*:

assumes $\langle \text{heap.apply-write } w \ s, \ xs, \ v \rangle \leq \text{tso.t2s } wb \ f$

shows $\langle s, \ (\text{self}, \ \text{heap.apply-write } w \ s) \ \# \ xs, \ v \rangle \leq \text{tso.t2s } (w \ \# \ wb) \ f$

⟨proof⟩

⟨ML⟩

lemma *t2s-le*:

shows $\text{spec.idle} \leq \text{tso.t2s } wb \ P$

⟨proof⟩

⟨ML⟩

lemmas $\text{minimal}[\text{iff}] = \text{order.trans}[\text{OF } \text{spec.idle.minimal-le } \text{spec.idle.tso.t2s-le}]$

⟨ML⟩

lemma *t2s-le*:

shows $\text{spec.rel } (\{ \text{env} \} \times \text{UNIV}) \ggg (\lambda :: \text{unit. tso.t2s } wb \ P) \leq \text{tso.t2s } wb \ P$

⟨proof⟩

⟨ML⟩

lemma *t2p[prog.p2s.simps]*:

shows $\text{prog.p2s } (\text{tso.t2p } P)$

$= \text{tso.t2s } [] \ P \ggg (\lambda vwb. \ \text{prog.p2s } (\text{raw.MFENCE } (\text{snd } vwb) \ggg \text{prog.return } (\text{fst } vwb)))$

⟨proof⟩

⟨ML⟩

lemma *bind*:

shows $\text{tso.t2s } wb \ (f \ggg g) = \text{tso.t2s } wb \ f \ggg (\lambda x. \ \text{tso.t2s } (\text{snd } x) \ (g \ (\text{fst } x)))$

⟨proof⟩

lemma *parallel*:

shows $\text{tso.t2s } [] \ (P \parallel Q) = \text{prog.p2s } ((\text{tso.t2p } P \parallel \text{tso.t2p } Q) \ggg \text{prog.return } ((), \ []))$

⟨proof⟩

lemma *return*:

shows $\text{tso.t2s } [] \ (\text{tso.return } v) = \text{prog.p2s } (\text{prog.return } (v, \ []))$

⟨proof⟩

⟨ML⟩

Inhabitation rules. ⟨ML⟩

lemma *bind*:

assumes $\text{tso.t2s } wb \ f \ -s, \ xs \rightarrow \text{tso.t2s } wb' \ f'$

shows $\text{tso.t2s } wb \ (f \ggg g) \ -s, \ xs \rightarrow \text{tso.t2s } wb' \ (f' \ggg g)$

⟨proof⟩

lemma *commit*:

shows $\text{tso.t2s } (w \ \# \ wb) \ f \ -s, \ [(\text{self}, \ \text{heap.apply-write } w \ s)] \rightarrow \text{tso.t2s } wb \ f$

⟨proof⟩

⟨ML⟩

lemma *ref*:
fixes $r :: 'a::\text{heap.rep } \text{ref}$
fixes $s :: \text{heap.t}$
fixes $v :: 'a$
fixes $v' :: 'a$
assumes $\neg \text{heap.present } r \ s$
shows $\text{tso.t2s } wb \ (\text{tso.Ref.ref } v)$
 $\quad -s, [(self, \text{Ref.set } r \ v' \ s)] \rightarrow$
 $\quad \text{tso.t2s } (wb \ @ \ [\text{heap.Write } (\text{ref.addr-of } r) \ 0 \ (\text{heap.rep.to } v)]) \ (\text{tso.return } r) \ (\text{is } ?lhs \ -s, \ ?step \rightarrow \ ?rhs)$
 $\langle \text{proof} \rangle$

lemma *lookup*:
fixes $r :: 'a::\text{heap.rep } \text{ref}$
shows $\text{tso.t2s } wb \ (!r) \ -s, [] \rightarrow \text{tso.t2s } wb \ (\text{tso.return } (\text{Ref.get } r \ (\text{apply-writes } wb \ s)))$
 $\langle \text{proof} \rangle$

lemma *update*:
fixes $r :: 'a::\text{heap.rep } \text{ref}$
shows $\text{tso.t2s } wb \ (r := v)$
 $\quad -s, [] \rightarrow$
 $\quad \text{tso.t2s } (wb \ @ \ [\text{heap.Write } (\text{ref.addr-of } r) \ 0 \ (\text{heap.rep.to } v)]) \ (\text{tso.return } ())$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemmas $\text{bind}' = \text{inhabits.trans}[OF \ \text{inhabits.tso.bind}]$
lemmas $\text{commit}' = \text{inhabits.trans}[OF \ \text{inhabits.tso.commit}]$

$\langle ML \rangle$

27.3 Code generator setup for TSO

The following is only sound if the generated code runs on a machine with a TSO memory model such as:

- x86
- x86 code running on macOS under Rosetta 2 (ask Google)

Notes:

- Haskell: GHC exposes unfenced operations for references and some kinds of arrays
 - GHC has a zoo of arrays; for now we use the general but inefficient boxed array type
- SML: Poly/ML appears to have committed to release/acquire (see email with subject “Git master update: ARM64, PIE and new bootstrap process”)
 - on x86 this is TSO
- Scala: beyond the scope of this work

TODO:

- support a CAS-like operation
 - Haskell: <https://stackoverflow.com/questions/10102881/haskell-how-does-atomicmodifyioref-work>

27.3.1 Haskell

Adaption layer

code-printing code-module $T\text{SOHeap} \rightarrow (\text{Haskell})$

```
<
module TSOHeap (
  TSO
  , IORef, newIORef, readIORef, writeIORef
  , Array, newArray, newListArray, newFunArray, lengthArray, readArray, writeArray
  , parallel
  ) where

import Control.Concurrent (forkIO)
import qualified Control.Concurrent.MVar as MVar
import qualified Data.Array.IO as Array -- FIXME boxed, contemplate the menagerie of other arrays; perhaps
type families might help here
import Data.IORef (IORef, newIORef, readIORef, writeIORef)
import Data.List (genericLength)

type TSO a = IO a
type Array a = Array.IOArray Integer a
type Ref a = Data.IORef.IORef a

writeIORef :: IORef a -> a -> IO ()
writeIORef = writeIORef -- FIXME strict variant?

newArray :: Integer -> a -> IO (Array a)
newArray k = Array.newArray (0, k - 1)

newListArray :: [a] -> IO (Array a)
newListArray xs = Array.newListArray (0, genericLength xs - 1) xs

newFunArray :: Integer -> (Integer -> a) -> IO (Array a)
newFunArray k f = Array.newListArray (0, k - 1) (map f [0..k-1])

lengthArray :: Array a -> IO Integer
lengthArray a = Array.getBounds a >>= return . (\(-, l) -> l + 1)

readArray :: Array a -> Integer -> IO a
readArray = Array.readArray

writeArray :: Array a -> Integer -> a -> IO ()
writeArray = Array.writeArray

-- note we don't want forkFinally as we don't model exceptions
parallel :: IO () -> IO () -> IO ()
parallel p q = do
  mvar <- MVar.newEmptyMVar
  forkIO (p >> MVar.putMVar mvar ()) -- FIXME putMVar is lazy
  b <- q
  a <- MVar.takeMVar mvar
  return ()
>
```

code-reserved Haskell TSOHeap

Monad

code-printing type-constructor $t\text{so} \rightarrow (\text{Haskell}) T\text{SOHeap.TSO}$ -

code-monad *tso.bind Haskell*
code-printing constant *tso.return* \rightarrow (*Haskell*) *return*
code-printing constant *tso.raise* \rightarrow (*Haskell*) *error*
code-printing constant *tso.parallel* \rightarrow (*Haskell*) *TSOHeap.parallel*

Intermediate operation avoids invariance problem in *Scala* (similar to value restriction)

$\langle ML \rangle$

definition *ref'* where

[*code del*]: *ref' = tso.Ref.ref*

lemma [*code*]:

tso.Ref.ref x = tso.Ref.ref' x

$\langle proof \rangle$

$\langle ML \rangle$

Haskell

code-printing type-constructor *ref* \rightarrow (*Haskell*) *TSOHeap.Ref* -
code-printing constant *Ref* \rightarrow (*Haskell*) *error/ bare Ref*
code-printing constant *tso.Ref.ref'* \rightarrow (*Haskell*) *TSOHeap.newIORef*
code-printing constant *tso.Ref.lookup* \rightarrow (*Haskell*) *TSOHeap.readIORef*
code-printing constant *tso.Ref.update* \rightarrow (*Haskell*) *TSOHeap.writeIORef*
code-printing constant *HOL.equal* :: '*a ref* \Rightarrow '*a ref* \Rightarrow *bool* \rightarrow (*Haskell*) **infix** 4 ==
code-printing class-instance *ref* :: *HOL.equal* \rightarrow (*Haskell*) -

27.4 A TSO litmus test

The classic TSO litmus test Owens et al. (2009, §1): write buffering allows both threads to read zero, which is impossible under sequential consistency.

definition *iwp2-3-a* :: (*nat* \times *nat*) *tso* where

```
iwp2-3-a = do {
  x  $\leftarrow$  tso.Ref.ref 0
; y  $\leftarrow$  tso.Ref.ref 0
; xvr  $\leftarrow$  tso.Ref.ref 0
; yvr  $\leftarrow$  tso.Ref.ref 0
; ( ( do { x := 1 ; yv  $\leftarrow$  !y ; yvr := yv } )
  || ( do { y := 1 ; xv  $\leftarrow$  !x ; xvr := xv } ) )
; xv  $\leftarrow$  !xvr
; yv  $\leftarrow$  !yvr
; tso.return (xv, yv)
}
```

code-thms *iwp2-3-a*

export-code *iwp2-3-a* in *Haskell*

schematic-goal *iwp2-3-a*: — “Can terminate with both threads reading 0”

shows $\langle heap.empty, ?xs, Some (0, 0) \rangle \leq prog.p2s (tso.t2p iwp2-3-a)$

$\langle proof \rangle$

thm *iwp2-3-a*[*simplified apply-writes-def, simplified*]

28 Floyd-Warshall all-pairs shortest paths

The Floyd-Warshall algorithm computes the lengths of the shortest paths between all pairs of nodes by updating an adjacency (square) matrix that represents the edge weights. Our goal here is to present it at a very abstract

level to exhibit the data dependencies.

Source materials:

- https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
- \$AFP/Floyd_Warshall/Floyd_Warshall.thy
 - a proof by refinement yielding a thorough correctness result including negative weights but not the absence of edges
- Dingel (2002, §6.2)
 - Overly parallelised, which is not practically useful but does reveal the data dependencies
 - the refinement is pretty much the same as the direct partial correctness proof here
 - the equivalent to *fw-update* is a single expression

We are not very ambitious here. This theory:

- does not track the actual shortest paths here but it is easy to add another array to do so
- ignores numeric concerns
- assumes the graph is complete

A further step would be to refine the parallel program to the classic three-loop presentation.

definition *fw-update* :: ('i::Ix × 'i, nat) array ⇒ 'i × 'i ⇒ 'i ⇒ unit imp **where**

```
fw-update = (λa (i, j) k. do {
  ij ← prog.Array.nth a (i, j);
  ik ← prog.Array.nth a (i, k);
  kj ← prog.Array.nth a (k, j);
  prog.whenM (ik + kj < ij) (prog.Array.upd a (i, j) (ik + kj))
})
```

— top-level specification: we can process the nodes in an arbitrary order

definition *fw-chaotic* :: ('i::Ix × 'i, nat) array ⇒ unit imp **where**

```
fw-chaotic a =
  (let b = array.bounds a in
  prog.Array.fst-app-chaotic b (λk. ||(i, j)∈set (Ix.interval b). fw-update a (i, j) k))
```

— executable version

definition *fw* :: ('i::Ix × 'i, nat) array ⇒ unit imp **where**

```
fw a =
  (let b = array.bounds a in
  prog.Array.fst-app b (λk. ||(i, j)∈set (Ix.interval b). fw-update a (i, j) k))
```

lemma *fw-fw-chaotic-le*: — the executable program refines the specification

shows *fw a ≤ fw-chaotic a*

⟨proof⟩

Safety proof type-synonym 'i matrix = 'i × 'i ⇒ nat

— The weight of the given path

fun *path-weight* :: 'i matrix ⇒ 'i × 'i ⇒ 'i list ⇒ nat **where**

```
path-weight m ij [] = m ij
| path-weight m ij (k # xs) = m (fst ij, k) + path-weight m (k, snd ij) xs
```

— The set of acyclic paths from *i* to *j* using the nodes *ks*

definition *paths* :: 'i × 'i ⇒ 'i set ⇒ 'i list set **where**

$paths\ ij\ ks = \{p. set\ p \subseteq ks \wedge fst\ ij \notin set\ p \wedge snd\ ij \notin set\ p \wedge distinct\ p\}$

— The minimum weight of a path from i to j using the nodes ks . See \$AFP/Floyd_Warshall/Floyd_Warshall.thy for proof that these are minimal amongst all paths.

definition $min\text{-}path\text{-}weight :: 'i\ matrix \Rightarrow 'i \times 'i \Rightarrow 'i\ set \Rightarrow nat$ **where**
 $min\text{-}path\text{-}weight\ m\ ij\ ks = Min\ (path\text{-}weight\ m\ ij\ `paths\ ij\ ks)$

context

fixes $a :: ('i::Ix \times 'i, nat)$ array

fixes $m :: 'i\ matrix$

begin

definition $fw\text{-}p\text{-}inv :: 'i \times 'i \Rightarrow 'i\ set \Rightarrow heap.t\ pred$ **where** — process invariant
 $fw\text{-}p\text{-}inv\ ij\ ks = (heap.rep\text{-}inv\ a \wedge Array.get\ a\ ij = \langle min\text{-}path\text{-}weight\ m\ ij\ ks \rangle)$

definition $fw\text{-}inv :: 'i\ set \Rightarrow heap.t\ pred$ **where** — loop invariant
 $fw\text{-}inv\ ks = (\forall\ ij. \langle ij \in set\ (Array.interval\ a) \rangle \longrightarrow fw\text{-}p\text{-}inv\ ij\ ks)$

definition $fw\text{-}pre :: heap.t\ pred$ **where** — overall precondition
 $fw\text{-}pre = (\langle Array.square\ a \rangle \wedge heap.rep\text{-}inv\ a$
 $\wedge (\forall\ ij. \langle ij \in set\ (Array.interval\ a) \rangle \longrightarrow Array.get\ a\ ij = \langle m\ ij \rangle))$

definition $fw\text{-}post :: unit \Rightarrow heap.t\ pred$ **where** — overall postcondition
 $fw\text{-}post = fw\text{-}inv\ (set\ (Ix.interval\ (fst\ bounds\ (array.bounds\ a))))$

end

$\langle ML \rangle$

lemma I :

assumes $set\ p \subseteq ks$

assumes $i \notin set\ p$

assumes $j \notin set\ p$

assumes $distinct\ p$

shows $p \in paths\ (i, j)\ ks$

$\langle proof \rangle$

lemma Nil :

shows $[] \in paths\ ij\ ks$

$\langle proof \rangle$

lemma $empty$:

shows $paths\ ij\ \{\} = \{\}\}$

$\langle proof \rangle$

lemma $not\text{-}empty$:

shows $paths\ ij\ ks \neq \{\}$

$\langle proof \rangle$

lemma $monotone$:

shows $mono\ (paths\ ij)$

$\langle proof \rangle$

lemmas $mono = monoD[OF\ paths.monotone]$

lemmas $strengthen[strg] = st\text{-}monotone[OF\ paths.monotone]$

lemma $finite$:

assumes $finite\ ks$

shows *finite* (*paths ij ks*)
⟨*proof*⟩

lemma *unused*:

assumes $p \in \text{paths } ij \text{ (insert } k \text{ } ks)$
assumes $k \notin \text{set } p$
shows $p \in \text{paths } ij \text{ } ks$
⟨*proof*⟩

lemma *decompE*:

assumes $p \in \text{paths } (i, j) \text{ (insert } k \text{ } ks)$
assumes $k \in \text{set } p$
obtains $r \ s$
where $p = r @ k \# s$
and $r \in \text{paths } (i, k) \text{ } ks$ **and** $s \in \text{paths } (k, j) \text{ } ks$
and $\text{distinct } (r @ s)$ **and** $i \notin \text{set } (r @ k \# s)$ **and** $j \notin \text{set } (r @ k \# s)$
⟨*proof*⟩

⟨*ML*⟩

lemma *append*:

shows $\text{path-weight } m \ ij \ (xs @ y \# ys) = \text{path-weight } m \ (fst \ ij, y) \ xs + \text{path-weight } m \ (y, snd \ ij) \ ys$
⟨*proof*⟩

⟨*ML*⟩

lemmas *min-path-weightI* = *trans[OF min-path-weight-def Min-eqI]*

⟨*ML*⟩

lemma *fw-update*:

assumes $m: \text{min-path-weight } m \ (i, k) \ ks + \text{min-path-weight } m \ (k, j) \ ks < \text{min-path-weight } m \ (i, j) \ ks$
assumes *finite ks*
shows $\text{min-path-weight } m \ (i, j) \text{ (insert } k \text{ } ks)$
 $= \text{min-path-weight } m \ (i, k) \ ks + \text{min-path-weight } m \ (k, j) \ ks$ (**is** ?lhs = ?rhs)
⟨*proof*⟩

lemma *return*:

assumes $m: \neg(\text{min-path-weight } m \ (i, k) \ ks + \text{min-path-weight } m \ (k, j) \ ks < \text{min-path-weight } m \ (i, j) \ ks)$
assumes *finite ks*
shows $\text{min-path-weight } m \ (i, j) \text{ (insert } k \text{ } ks) = \text{min-path-weight } m \ (i, j) \ ks$
⟨*proof*⟩

⟨*ML*⟩

lemma *Id-on-fw-inv*:

shows $\text{stable heap.}Id_{\{a\}} \ (fw\text{-inv } a \ m \ ys)$
⟨*proof*⟩

lemma *Id-on-fw-p-inv*:

shows $\text{stable heap.}Id_{\{a\}} \ (fw\text{-p-inv } a \ m \ ij \ ks)$
⟨*proof*⟩

lemma *modifies-fw-p-inv*:

assumes $ij \in \text{set } (\text{Array.interval } a) - is$
shows $\text{stable Array.modifies}_{a, is} \ (fw\text{-p-inv } a \ m \ ij \ ks)$
⟨*proof*⟩

$\langle ML \rangle$

lemma *fw-p-inv-cong*:

assumes $a = a'$
assumes $m = m'$
assumes $ij = ij'$
assumes $ks = ks'$
assumes $s (\text{heap.addr-of } a) = s' (\text{heap.addr-of } a')$
shows $\text{fw-p-inv } a \ m \ ij \ ks \ s = \text{fw-p-inv } a' \ m' \ ij' \ ks' \ s'$

$\langle \text{proof} \rangle$

lemma *fw-p-invD*:

assumes $\text{fw-p-inv } a \ m \ ij \ ks \ s$
shows $\text{heap.rep-inv } a \ s$
and $\text{Array.get } a \ ij \ s = \text{min-path-weight } m \ ij \ ks$

$\langle \text{proof} \rangle$

lemma *fw-p-inv-fw-update*:

assumes *finite* ks
assumes $ij \in \text{set } (\text{Array.interval } a)$
assumes $\text{fw-p-inv } a \ m \ ij \ ks \ s$
assumes $\text{min-path-weight } m \ (\text{fst } ij, k) \ ks + \text{min-path-weight } m \ (k, \text{snd } ij) \ ks < \text{min-path-weight } m \ ij \ ks$
shows $\text{fw-p-inv } a \ m \ ij \ (\text{insert } k \ ks) \ (\text{Array.set } a \ ij \ (\text{min-path-weight } m \ (\text{fst } ij, k) \ ks + \text{min-path-weight } m \ (k, \text{snd } ij) \ ks) \ s)$

$\langle \text{proof} \rangle$

lemma *fw-p-inv-return*:

assumes *finite* ks
assumes $\text{fw-p-inv } a \ m \ ij \ ks \ s$
assumes $\neg(\text{min-path-weight } m \ (\text{fst } ij, k) \ ks + \text{min-path-weight } m \ (k, \text{snd } ij) \ ks < \text{min-path-weight } m \ ij \ ks)$
shows $\text{fw-p-inv } a \ m \ ij \ (\text{insert } k \ ks) \ s$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

Dingel (2000, p109) key intuition: when processing index k , neither $a[i, k]$ and $a[k, j]$ change.

- his argument is bogus: it is enough to observe that shortest paths never get shorter by adding edges
- he unnecessarily assumes that $\delta(i, i) = 0$ for all i

lemma *fw-update*:

assumes $\text{insert } k \ ks \subseteq \text{set } (\text{Ix.interval } (\text{fst-bounds } (\text{array.bounds } a)))$
assumes $\text{Array.square } a$
assumes $ij: ij \in \text{set } (\text{Array.interval } a)$
defines $\bigwedge ij. G \ ij \equiv \text{Array.modifies}_a, \{ij \mid \text{unit}. k \notin \{\text{fst } ij, \text{snd } ij\}\}$
defines $A \equiv \text{heap.Id}_{\{a\}} \cup \bigcup (G \ ' (\text{set } (\text{Array.interval } a) - \{ij\}))$
shows $\text{prog.p2s } (\text{fw-update } a \ ij \ k)$
 $\leq \{\{\text{fw-p-inv } a \ m \ ij \ ks \wedge \text{fw-p-inv } a \ m \ (\text{fst } ij, k) \ ks \wedge \text{fw-p-inv } a \ m \ (k, \text{snd } ij) \ ks\}, A$
 $\vdash G \ ij, \{\lambda-. \text{fw-p-inv } a \ m \ ij \ (\text{insert } k \ ks)\}$

$\langle \text{proof} \rangle$

lemma *fw-chaotic*:

fixes $a :: ('i::Ix \times 'i, \text{nat}) \text{array}$
fixes $m :: 'i \text{matrix}$
shows $\text{prog.p2s } (\text{fw-chaotic } a) \leq \{\{\text{fw-pre } a \ m\}, \text{heap.Id}_{\{a\}} \vdash \text{heap.modifies}_{\{a\}}, \{\{\text{fw-post } a \ m\}\}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

References

- M. Abadi. An axiomatization of lamport's temporal logic of actions. In *CONCUR '90*, volume 458 of *LNCS*, pages 57–69. Springer, 1990. doi: 10.1007/BFB0039051.
- M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. doi: 10.1016/0304-3975(91)90224-P.
- M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995. doi: 10.1145/203095.201069.
- M. Abadi and S. Merz. An abstract account of composition. In *MFCS'95*, volume 969 of *LNCS*, pages 499–508. Springer, 1995. doi: 10.1007/3-540-60246-1_155.
- M. Abadi and S. Merz. On TLA as a logic. In Manfred Broy, editor, *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, pages 235–271. IOS Press, 1996. ISBN 3-540-60947-4.
- M. Abadi and G. D. Plotkin. A logical view of composition and refinement. In *POPL'1991*, pages 323–332. ACM Press, 1991. doi: 10.1145/99583.99626.
- M. Abadi and G. D. Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, 1993. doi: 10.1016/0304-3975(93)90151-I.
- B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985. doi: 10.1016/0020-0190(85)90056-0.
- B. Alpern, A. J. Demers, and F. B. Schneider. Safety without stuttering. *Information Processing Letters*, 23(4):177–180, 1986. doi: 10.1016/0020-0190(86)90132-8.
- K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer, 2009. ISBN 978-1-84882-744-8. doi: 10.1007/978-1-84882-745-5.
- A. Armstrong, V. B. F. Gomes, and G. Struth. Algebraic principles for rely-guarantee style concurrency verification tools. In *FM 2014*, volume 8442 of *LNCS*, pages 78–93. Springer, 2014. doi: 10.1007/978-3-319-06410-9_6.
- R. C. Backhouse. Galois connections and fixed point calculus. In R.C. Backhouse, R. L. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, pages 89–148. Springer, 2000. doi: 10.1007/3-540-47797-7_4.
- S. D. Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, 1996. doi: 10.1006/inco.1996.0056.
- A. Cau and P. Collette. Parallel composition of assumption-commitment specifications: A unifying approach for shared variable and distributed message passing concurrency. *Acta Informatica*, 33(2):153–176, 1996. doi: 10.1007/s002360050039.
- K. M. Chandy and J. Misra. *Parallel program design - a foundation*. Addison-Wesley, 1989. ISBN 978-0-201-05866-6.
- B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order, Second Edition*. Cambridge University Press, 2002. ISBN 978-0-521-78451-1. doi: 10.1017/CBO9780511809088.
- G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI'13*, pages 854–860. IJCAI/AAAI, 2013.
- W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
- W.-P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001. ISBN 0-521-80608-9.
- J. Dingel. Modular verification for shared-variable concurrent programs. In *CONCUR '96*, volume 1119 of *LNCS*, pages 703–718. Springer, 1996. doi: 10.1007/3-540-61604-7_85.

- J. Dingel. *Systematic parallel programming*. PhD thesis, Carnegie Mellon University, May 2000. CMU Tech Report CS-99-172.
- J. Dingel. A refinement calculus for shared-variable parallel and distributed programming. *Formal Aspects of Computing*, 14(2):123–197, 2002. doi: 10.1007/s001650200032.
- E. A. Emerson. Alternative semantics for temporal logics. *Theoretical Computer Science*, 26:121–130, 1983. doi: 10.1016/0304-3975(83)90082-8.
- L. Esakia, G. Bezhanishvili, W. H. Holliday, and A. Evseev. *Heyting Algebras: Duality Theory*. Springer, 1st edition, 2019. ISBN 3030120953.
- S. Foster, J. Baxter, A. Cavalcanti, J. Woodcock, and F. Zeyda. Unifying semantic foundations for automated verification tools in isabelle/utp. *Science of Computer Programming*, 197:102510, 2020. ISSN 0167-6423. doi: 10.1016/j.scico.2020.102510.
- G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *Continuous Lattices and Domains*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2003. doi: 10.1017/CBO9780511542725.
- R. Goldblatt. *Logics of Time and Computation*. Number 7 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, 2 edition, 1992.
- I. J. Hayes. Generalised rely-guarantee concurrency: an algebraic foundation. *Formal Aspects of Computing*, 28(6):1057–1078, 2016. doi: 10.1007/s00165-016-0384-0.
- I. J. Hayes and C. B. Jones. A guide to rely/guarantee thinking. In *SETSS 2017*, volume 11174 of *LNCS*, pages 1–38. Springer, 2017. doi: 10.1007/978-3-030-02928-9_1.
- C. A. R. Hoare and J. He. The weakest prespecification. *Information Processing Letters*, 24(2):127–132, 1987. doi: 10.1016/0020-0190(87)90106-2. Oxford Technical Monograph PRG-44.
- C. A. R. Hoare, I. J. Hayes, J. He, C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987a. doi: 10.1145/27651.27653.
- C. A. R. Hoare, J. He, and J. W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71–76, 1987b. doi: 10.1016/0020-0190(87)90224-9.
- C. A. R. Hoare, J. He, and A. Sampaio. Algebraic derivation of an operational semantics. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 77–98. The MIT Press, 2000.
- T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent kleene algebra and its foundations. *Journal of Logic and Algebraic Programming*, 80(6):266–296, 2011. doi: 10.1016/j.jlap.2011.04.005.
- P. B. Jackson. Verifying a garbage collection algorithm. In *TPHOLs*, volume 1479 of *LNCS*, pages 225–244. Springer, 1998. doi: 10.1007/BFb0055139.
- R. Jagadeesan, G. Petri, and J. Riely. Brookes is relaxed, almost! In Lars Birkedal, editor, *FOSSACS 2012*, volume 7213 of *LNCS*, pages 180–194. Springer, 2012. doi: 10.1007/978-3-642-28729-9_12.
- C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983. doi: 10.1145/69575.69577.
- B. Jonsson and Y.-K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167(1&2):47–72, 1996. doi: 10.1016/0304-3975(96)00069-2.
- R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969. doi: 10.1016/S0022-0000(69)80011-5.
- E. Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53(30):268–272, 6 1994.
- D. Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994. doi: 10.1006/inco.1994.1037.

- F. Kröger and S. Merz. *Temporal Logic and State Systems*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008. ISBN 978-3-540-67401-6.
- L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3): 872–923, 1994. doi: 10.1145/177492.177726.
- L. Lamport. Specifying concurrent systems in TLA⁺. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series, III: Computer and Systems Sciences*, pages 183–247. IOS Press, January 2000. ISBN 9789051994599. Proceedings of Marktoberdorf 1998.
- H. Liang, X. Feng, and M. Fu. Rely-guarantee-based simulation for compositional verification of concurrent program transformations. *ACM Transactions on Programming Languages and Systems*, 36(1):3:1–3:55, 2014. doi: 10.1145/2576235.
- N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. ISBN 1-55860-348-4.
- P. Maier. A set-theoretic framework for assume-guarantee reasoning. In *ICALP’2001*, volume 2076 of *LNCS*, pages 821–834. Springer, 2001. doi: 10.1007/3-540-48224-5_67.
- P. Maier. Intuitionistic LTL and a new characterization of safety and liveness. In *CSL 2004*, volume 3210 of *LNCS*, pages 295–309. Springer, 2004. doi: 10.1007/978-3-540-30124-0_24.
- Z. Manna and A. Pnueli. The anchored version of the temporal framework. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*, volume 354 of *LNCS*, pages 201–284. Springer, 1988. doi: 10.1007/BFb0013024.
- Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. In R. F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, pages 121–156. ACM Press and Addison-Wesley, 1991. Also Technical Report STAN-CS-90-1321.
- P. Manolios and R. J. Treffer. A lattice-theoretic characterization of safety and liveness. In E. Borowsky and S. Rajsbaum, editors, *PODC’2003*, pages 325–333. ACM, 2003. doi: 10.1145/872035.872083.
- A. Melton, D. A. Schmidt, and G. E. Strecker. Calois connections and computer science applications. In *Category Theory and Computer Programming*, volume 240 of *LNCS*, pages 299–312. Springer, 1985. doi: 10.1007/3-540-17162-2_130.
- M. Müller-Olm. *Modular Compiler Verification - A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *LNCS*. Springer, 1997. doi: 10.1007/BFb0027453.
- H. Ono. *Proof Theory and Algebra in Logic*. Short Textbooks in Logic. Springer, 2019. doi: 10.1007/978-981-13-7997-0.
- S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs’2009*, volume 5674 of *LNCS*, pages 391–407. Springer, 2009. doi: 10.1007/978-3-642-03359-9_27. URL <https://www.cl.cam.ac.uk/~pes20/weakmemory/x86tso-paper.pdf>.
- S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976. doi: 10.1007/BF00268134.
- S. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982. doi: 10.1145/357172.357178.
- J. L. Pfaltz and J. Šlapal. Transformations of discrete closure systems. *Acta Mathematica Hungarica*, 138(4): 386–405, 2013. doi: 10.1007/s10474-012-0262-z.
- V. R. Pratt. Action logic and pure induction. In J. van Eijck, editor, *Logics in AI, European Workshop, JELIA ’90, Amsterdam, The Netherlands, September 10-14, 1990, Proceedings*, volume 478 of *LNCS*, pages 97–120. Springer, 1990. doi: 10.1007/BFb0018436.
- L. Prensa Nieto. The rely-guarantee method in isabelle/hol. In *ESOP 2003*, volume 2618 of *LNCS*, pages 348–362. Springer, 2003. doi: 10.1007/3-540-36575-3_24.

- B. K. Rosen. Correctness of parallel programs: The church-rosser approach. *Theoretical Computer Science*, 2(2): 183–207, 1976. doi: 10.1016/0304-3975(76)90032-3.
- F. B. Schneider. Decomposing properties into safety and liveness using predicate logic. Technical Report 87-874, Department of Computer Science, Cornell University, October 1987.
- D. S. Scott. *The Kleene Symposium*, volume 101 of *Studies in Logic and the Foundations of Mathematics*, chapter Lambda Calculus: Some Models, Some Philosophy, pages 223–265. Elsevier, 1980. doi: 10.1016/S0049-237X(08)71262-X.
- A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–512, 1994. doi: 10.1007/BF01211865.
- M. H. Stone. Topological representations of distributive lattices and Brouwerian logics. *Časopis pro pěstování matematiky a fyziky*, 67(1):1–25, 1938. doi: 10.21136/CPMF.1938.124080.
- V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, UK, 2008.
- D. van Dalen. *Logic and structure (4. ed.)*. Universitext. Springer, 2004. ISBN 978-3-540-57839-0.
- R. van Glabbeek and P. Höfner. Progress, justness, and fairness. *ACM Computing Surveys*, 52(4):69:1–69:38, 2019. doi: 10.1145/3329125.
- S. van Staden. Constructing the views framework. In *UTP 2014*, volume 8963 of *LNCS*, pages 62–83. Springer, 2014. doi: 10.1007/978-3-319-14806-9_4.
- S. van Staden. On rely-guarantee reasoning. In *MPC 2015*, volume 9129 of *LNCS*, pages 30–49. Springer, 2015. doi: 10.1007/978-3-319-19797-5_2.
- S. Vickers. *Topology via Logic*. Cambridge University Press, 1989. ISBN 0521360625.
- J. S. Warford, D. Vega, and S. M. Staley. A calculational deductive system for linear temporal logic. *ACM Computing Surveys*, 53(3):53:1–53:38, 2020. doi: 10.1145/3387109.
- J. Wickerson, M. Dodds, and M. J. Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. In *ESOP 2010*, volume 6012 of *LNCS*, pages 610–629. Springer, 2010. doi: 10.1007/978-3-642-11957-6_32. URL <https://johnwickerson.github.io/expstab.thy.html>. Extended version in UCAM-CL-TR-774.
- Q. Xu and J. He. A theory of state-based parallel programming: Part 1. In Joseph M. Morris and Roger C. Shaw, editors, *4th Refinement Workshop*, pages 326–359. Springer, 1991.
- Q. Xu and J. He. Laws of parallel programming with shared variables. In David Till, editor, *6th Refinement Workshop, Proceedings of the 6th Refinement Workshop, organised by BCS-FACS, London, UK, 5-7 January 1994*, Workshops in Computing, pages 205–216. Springer, 1994. doi: 10.1007/978-1-4471-3240-0_11.
- Q. Xu, A. Cau, and P. Collette. On unifying assumption-commitment style proof rules for concurrency. In B. Jonsson and J. Parrow, editors, *CONCUR '94*, volume 836 of *LNCS*, pages 267–282. Springer, 1994. doi: 10.1007/978-3-540-48654-1_22.
- Q. Xu, W.-P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997. doi: 10.1007/BF01211617.
- Y. Zakowski, D. Cachera, D. Demange, G. Petri, D. Pichardie, S. Jagannathan, and J. Vitek. Verifying a concurrent garbage collector with a rely-guarantee methodology. *Journal of Automated Reasoning*, 63(2):489–515, 2019. doi: 10.1007/s10817-018-9489-x.
- J. Zwiers. *Compositionality, Concurrency and Partial Correctness - Proof Theories for Networks of Processes, and Their Relationship*, volume 321 of *LNCS*. Springer, 1989. ISBN 3-540-50845-7. doi: 10.1007/BFb0020836.