

Refinement for Monadic Programs

Peter Lammich

September 23, 2021

Abstract

We provide a framework for program and data refinement in Isabelle/HOL. The framework is based on a nondeterminism-monad with assertions, i.e., the monad carries a set of results or an assertion failure. Recursion is expressed by fixed points. For convenience, we also provide while and foreach combinators.

The framework provides tools to automatize canonical tasks, such as verification condition generation, finding appropriate data refinement relations, and refine an executable program to a form that is accepted by the Isabelle/HOL code generator.

Some basic usage examples can be found in this entry, but most of the examples and the userguide have been moved to the Collections AFP entry. For more advanced examples, consider the AFP entries that are based on the Refinement Framework.

Contents

1	Introduction	5
1.1	Related Work	6
2	Refinement Framework	7
2.1	Miscellaneous Lemmas and Tools	8
2.1.1	Uncategorized Lemmas	8
2.1.2	Well-Foundedness	8
2.1.3	Monotonicity and Orderings	9
2.1.4	Maps	15
2.2	Transfer between Domains	15
2.3	General Domain Theory	18
2.3.1	General Order Theory Tools	18
2.3.2	Flat Ordering	18
2.4	Generic Recursion Combinator for Complete Lattice Structured Domains	22
2.4.1	Transfer	27
2.5	Assert and Assume	28
2.6	Basic Concepts	30
2.6.1	Nondeterministic Result Lattice and Monad	31
2.6.2	VCG Setup	38
2.6.3	Data Refinement	38
2.6.4	Derived Program Constructs	41
2.6.5	Proof Rules	42
2.6.6	Relators	52
2.6.7	Autoref Setup	53
2.6.8	Convenience Rules	54
2.7	Less-Equal or Fail	62
2.8	Data Refinement Heuristics	66
2.8.1	Type Based Heuristics	66
2.8.2	Patterns	67
2.8.3	Refinement Relations	67
2.9	More Combinators	68
2.10	Generic While-Combinator	70

2.11	While-Loops	74
2.11.1	Data Refinement Rules	75
2.11.2	Autoref Setup	79
2.11.3	Invariants	81
2.11.4	Convenience	89
2.12	Deterministic Monad	90
2.12.1	Deterministic Result Lattice	90
2.13	Partial Function Package Setup	95
2.13.1	Nondeterministic Result Monad	96
2.13.2	Deterministic Result Monad	96
2.14	Transfer Setup	97
2.14.1	Transfer to Deterministic Result Lattice	97
2.14.2	Transfer to Plain Function	98
2.14.3	Total correctness in deterministic monad	98
2.14.4	Relator-Based Transfer	99
2.14.5	Post-Simplification Setup	100
2.15	Foreach Loops	100
2.15.1	Auxilliary Lemmas	100
2.15.2	Definition	101
2.15.3	Proof Rules	102
2.15.4	FOREACH with empty sets	113
2.15.5	Monotonicity	114
2.15.6	Nres-Fold with Interruption (nfoldli)	115
2.15.7	LIST FOREACH combinator	119
2.15.8	FOREACH with duplicates	122
2.15.9	Miscellaneous Utility Lemmas	123
2.16	More Automation	125
2.17	Autoref for the Refinement Monad	126
2.18	Refinement Framework	127
2.18.1	Convenience Constructs	127
2.18.2	Syntax Sugar	127
3	Examples	129
3.1	Breadth First Search	129
3.1.1	Distances in a Graph	129
3.1.2	Invariants	131
3.1.3	Algorithm	132
3.1.4	Verification Tasks	134
3.2	Machine Words	136
3.2.1	Setup	136
3.2.2	Example	137
4	Conclusion and Future Work	139

Chapter 1

Introduction

Isabelle/HOL[17] is a higher order logic theorem prover. Recently, we started to use it to implement automata algorithms (e.g., [12]). There, we do not only want to specify an algorithm and prove it correct, but we also want to obtain efficient executable code from the formalization. This can be done with Isabelle/HOL's code generator [7, 8], that converts functional specifications inside Isabelle/HOL to executable programs. In order to obtain a uniform interface to efficient data structures, we developed the Isabelle Collection Framework (ICF) [11, 13]. It provides a uniform interface to various (collection) data structures, as well as generic algorithm, that are parametrized over the data structure actually used, and can be instantiated for any data structure providing the required operations. E.g., a generic algorithm may be parametrized over a set data structure, and then instantiated with a hashtable or a red-black tree.

The ICF features a data-refinement approach to prove an algorithm correct: First, the algorithm is specified using the abstract data structures. These are usually standard datatypes on Isabelle/HOL, and thus enjoy a good tool support for proving. Hence, the correctness proof is most conveniently performed on this abstract level. In a next step, the abstract algorithm is refined to a concrete algorithm that uses some efficient data structures. Finally, it is shown that the result of the concrete algorithm is related to the result of the abstract algorithm. This last step is usually fairly straightforward.

This approach works well for simple operations. However, it is not applicable when using inherently nondeterministic operations on the abstract level, such as choosing an arbitrary element from a non-empty set. In this case, any choice of the element on the abstract level over-specifies the algorithm, as it forces the concrete algorithm to choose the same element.

One possibility is to initially specify and prove correct the algorithm on the concrete level, possibly using parametrization to leave the concrete implementation unspecified. The problem here is, that the correctness proofs

have to be performed on the concrete level, involving abstraction steps during the proof, which makes it less readable and more tedious. Moreover, this approach does not support stepwise refinement, as all operations have to work on the most concrete datatypes.

Another possibility is to use a non-deterministic algorithm on the abstract level, that is then refined to a deterministic algorithm. Here, the correctness proofs may be done on the abstract level, and stepwise refinement is properly supported.

However, as Isabelle/HOL primarily supports functions, not relations, formulating nondeterministic algorithms is more tedious. This development provides a framework for formulating nondeterministic algorithms in a monadic style, and using program and data refinement to eventually obtain an executable algorithm. The monad is defined over a set of results and a special *FAIL*-value, that indicates a failed assertion. The framework provides some tools to make reasoning about those monadic programs more comfortable.

1.1 Related Work

Data refinement dates back to Hoare [9]. Using *refinement calculus* for stepwise program refinement, including data refinement, was first proposed by Back [1]. In the last decades, these topics have been subject to extensive research. Good overviews are [2, 6], that cover the main concepts on which this formalization is based. There are various formalizations of refinement calculus within theorem provers [3, 14, 20, 22, 18]. All these works focus on imperative programs and therefore have to deal with the representation of the state space (e.g., local variables, procedure parameters). In our monadic approach, there is no need to formalize state spaces or procedures, which makes it quite simple. Note, that we achieve modularization by defining constants (or recursive functions), thus moving the burden of handling parameters and procedure calls to the underlying theorem prover, and at the same time achieving a more seamless integration of our framework into the theorem prover. In the seL4-project [5], a nondeterministic state-exception monad is used to refine the abstract specification of the kernel to an executable model. The basic concept is closely related to ours. However, as the focus is different (Verification of kernel operations vs. verification of model-checking algorithms), there are some major differences in the handling of recursion and data refinement. In [21], *refinement monads* are studied. The basic constructions there are similar to ours. However, while we focus on data refinement, they focus on introducing commands with side-effects and a predicate-transformer semantics to allow angelic nondeterminism.

Chapter 2

Refinement Framework

```
theory Refine-Mono-Prover
imports Main Automatic-Refinement.Refine-Lib
begin
  ⟨ML⟩

  locale mono-setup-loc =
    — Locale to set up monotonicity prover for given ordering operator
    fixes le :: 'a ⇒ 'a ⇒ bool
    assumes refl: le x x
  begin
    lemma monoI: (∧f g x. (∧x. le (f x) (g x)) ⇒ le (B f x) (B g x))
      ⇒ monotone (fun-ord le) (fun-ord le) B
      ⟨proof⟩

    lemma mono-if: [le t t'; le e e'] ⇒ le (If b t e) (If b t' e') ⟨proof⟩
    lemma mono-let: (∧x. le (f x) (f' x)) ⇒ le (Let x f) (Let x f') ⟨proof⟩

    lemmas mono-thms[refine-mono] = monoI mono-if mono-let refl

    ⟨ML⟩

  end

  interpretation order-mono-setup: mono-setup-loc (≤) :: 'a::preorder ⇒ -
    ⟨proof⟩

  ⟨ML⟩

  lemmas [refine-mono] =
    lfp-mono[OF le-funI, THEN le-funD]
    gfp-mono[OF le-funI, THEN le-funD]

end
```

2.1 Miscellaneous Lemmas and Tools

theory *Refine-Misc*

imports

Automatic-Refinement.Automatic-Refinement

Refine-Mono-Prover

begin

Basic configuration for monotonicity prover:

lemmas [*refine-mono*] = *monoI monotoneI[of (≤) (≤)]*

lemmas [*refine-mono*] = *TrueI le-funI order-refl*

lemma *case-prod-mono[refine-mono]*:

$\llbracket \bigwedge a b. p=(a,b) \implies f a b \leq f' a b \rrbracket \implies \text{case-prod } f p \leq \text{case-prod } f' p$
 $\langle \text{proof} \rangle$

lemma *case-option-mono[refine-mono]*:

assumes $fn \leq fn'$

assumes $\bigwedge v. x=\text{Some } v \implies fs v \leq fs' v$

shows $\text{case-option } fn fs x \leq \text{case-option } fn' fs' x$

$\langle \text{proof} \rangle$

lemma *case-list-mono[refine-mono]*:

assumes $fn \leq fn'$

assumes $\bigwedge x xs. l=x\#xs \implies fc x xs \leq fc' x xs$

shows $\text{case-list } fn fc l \leq \text{case-list } fn' fc' l$

$\langle \text{proof} \rangle$

lemma *if-mono[refine-mono]*:

assumes $b \implies m1 \leq m1'$

assumes $\neg b \implies m2 \leq m2'$

shows $(\text{if } b \text{ then } m1 \text{ else } m2) \leq (\text{if } b \text{ then } m1' \text{ else } m2')$

$\langle \text{proof} \rangle$

lemma *let-mono[refine-mono]*:

$f x \leq f' x' \implies \text{Let } x f \leq \text{Let } x' f' \langle \text{proof} \rangle$

2.1.1 Uncategorized Lemmas

lemma *all-nat-split-at*: $\forall i::'a::\text{linorder} < k. P i \implies P k \implies \forall i > k. P i$

$\implies \forall i. P i$

$\langle \text{proof} \rangle$

2.1.2 Well-Foundedness

lemma *wf-no-infinite-down-chainI*:

assumes $\bigwedge f. \llbracket \bigwedge i. (f (\text{Suc } i), f i) \in r \rrbracket \implies \text{False}$

shows $wf r$

$\langle \text{proof} \rangle$

This lemma transfers well-foundedness over a simulation relation.

lemma *sim-wf*:
assumes *WF*: $wf (S'^{-1})$
assumes *STARTR*: $(x0, x0') \in R$
assumes *SIM*: $\bigwedge s s' t. \llbracket (s, s') \in R; (s, t) \in S; (x0', s') \in S'^* \rrbracket$
 $\implies \exists t'. (s', t') \in S' \wedge (t, t') \in R$
assumes *CLOSED*: $Domain\ S \subseteq S^* \cdot \{x0\}$
shows $wf (S^{-1})$
 $\langle proof \rangle$

Well-founded relation that approximates a finite set from below.

definition *finite-psupset* $S \equiv \{ (Q', Q). Q \subset Q' \wedge Q' \subseteq S \}$
lemma *finite-psupset-wf*[*simp, intro*]: $finite\ S \implies wf (finite-psupset\ S)$
 $\langle proof \rangle$

definition *less-than-bool* $\equiv \{ (a, b). a < (b::bool) \}$
lemma *wf-less-than-bool*[*simp, intro*]: $wf (less-than-bool)$
 $\langle proof \rangle$
lemma *less-than-bool-iff*[*simp*]:
 $(x, y) \in less-than-bool \iff x = False \wedge y = True$
 $\langle proof \rangle$

definition *greater-bounded* $N \equiv inv-image\ less-than\ (\lambda x. N - x)$
lemma *wf-greater-bounded*[*simp, intro!*]: $wf (greater-bounded\ N)$ $\langle proof \rangle$

lemma *greater-bounded-Suc-iff*[*simp*]: $(Suc\ x, x) \in greater-bounded\ N \iff Suc\ x \leq N$
 $\langle proof \rangle$

2.1.3 Monotonicity and Orderings

lemma *mono-const*[*simp, intro!*]: $mono (\lambda-. c)$ $\langle proof \rangle$
lemma *mono-if*: $\llbracket mono\ S1; mono\ S2 \rrbracket \implies$
 $mono (\lambda F s. if\ b\ s\ then\ S1\ F\ s\ else\ S2\ F\ s)$
 $\langle proof \rangle$

lemma *mono-infI*: $mono\ f \implies mono\ g \implies mono (inf\ f\ g)$
 $\langle proof \rangle$

lemma *mono-infI'*:
 $mono\ f \implies mono\ g \implies mono (\lambda x. inf (f\ x) (g\ x) :: 'b::lattice)$
 $\langle proof \rangle$

lemma *mono-infArg*:
fixes $f :: 'a::lattice \Rightarrow 'b::order$
shows $mono\ f \implies mono (\lambda x. f (inf\ x\ X))$
 $\langle proof \rangle$

lemma *mono-Sup*:

fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'b::\text{complete-lattice}$
shows $\text{mono } f \Longrightarrow \text{Sup } (f'S) \leq f (\text{Sup } S)$
 $\langle \text{proof} \rangle$

lemma *mono-SupI*:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'b::\text{complete-lattice}$
assumes $\text{mono } f$
assumes $S' \subseteq f'S$
shows $\text{Sup } S' \leq f (\text{Sup } S)$
 $\langle \text{proof} \rangle$

lemma *mono-Inf*:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'b::\text{complete-lattice}$
shows $\text{mono } f \Longrightarrow f (\text{Inf } S) \leq \text{Inf } (f'S)$
 $\langle \text{proof} \rangle$

lemma *mono-funpow*: $\text{mono } (f::'a::\text{order} \Rightarrow 'a) \Longrightarrow \text{mono } (f^{\sim i})$
 $\langle \text{proof} \rangle$

lemma *mono-id[simp, intro!]*:
 $\text{mono } \text{id}$
 $\text{mono } (\lambda x. x)$
 $\langle \text{proof} \rangle$

declare *SUP-insert[simp]*

lemma (**in** *semilattice-inf*) *le-infD1*:
 $a \leq \text{inf } b \ c \Longrightarrow a \leq b \ \langle \text{proof} \rangle$

lemma (**in** *semilattice-inf*) *le-infD2*:
 $a \leq \text{inf } b \ c \Longrightarrow a \leq c \ \langle \text{proof} \rangle$

lemma (**in** *semilattice-inf*) *inf-leI*:
 $\llbracket \bigwedge x. \llbracket x \leq a; x \leq b \rrbracket \Longrightarrow x \leq c \rrbracket \Longrightarrow \text{inf } a \ b \leq c$
 $\langle \text{proof} \rangle$

lemma *top-Sup*: $(\text{top}::'a::\text{complete-lattice}) \in A \Longrightarrow \text{Sup } A = \text{top}$
 $\langle \text{proof} \rangle$

lemma *bot-Inf*: $(\text{bot}::'a::\text{complete-lattice}) \in A \Longrightarrow \text{Inf } A = \text{bot}$
 $\langle \text{proof} \rangle$

lemma *mono-compD*: $\text{mono } f \Longrightarrow x \leq y \Longrightarrow f \circ x \leq f \circ y$
 $\langle \text{proof} \rangle$

Galois Connections

locale *galois-connection* =
fixes $\alpha::'a::\text{complete-lattice} \Rightarrow 'b::\text{complete-lattice}$ **and** γ
assumes *galois*: $c \leq \gamma(a) \longleftrightarrow \alpha(c) \leq a$
begin
lemma *$\alpha\gamma\text{-defl}$* : $\alpha(\gamma(x)) \leq x$

<proof>

lemma $\gamma\alpha$ -infl: $x \leq \gamma(\alpha(x))$

<proof>

lemma α -mono: *mono* α

<proof>

lemma γ -mono: *mono* γ

<proof>

lemma *dist- γ [simp]*:

$\gamma (\inf a b) = \inf (\gamma a) (\gamma b)$

<proof>

lemma *dist- α [simp]*:

$\alpha (\sup a b) = \sup (\alpha a) (\alpha b)$

<proof>

end

Fixed Points

lemma *mono-lfp-eqI*:

assumes *MONO*: *mono* f

assumes *FIXP*: $f a \leq a$

assumes *LEAST*: $\bigwedge x. f x = x \implies a \leq x$

shows $\text{lfp } f = a$

<proof>

lemma *mono-gfp-eqI*:

assumes *MONO*: *mono* f

assumes *FIXP*: $a \leq f a$

assumes *GREATEST*: $\bigwedge x. f x = x \implies x \leq a$

shows $\text{gfp } f = a$

<proof>

lemma *lfp-le-gfp'*: *mono* $f \implies \text{lfp } f x \leq \text{gfp } f x$

<proof>

lemma *lfp-induct'*:

assumes *M*: *mono* f

assumes *IS*: $\bigwedge m. [m \leq \text{lfp } f; m \leq P] \implies f m \leq P$

shows $\text{lfp } f \leq P$

<proof>

lemma *lfp-gen-induct*:

— Induction lemma for generalized lfps

assumes *M*: *mono* f

notes $MONO[refine-mono] = monoD[OF M]$
assumes $I0: m0 \leq P$
assumes $IS: \bigwedge m. \llbracket$
 $m \leq lfp (\lambda s. sup m0 (f s));$ — Assume already established invariants
 $m \leq P;$ — Assume invariant
 $f m \leq lfp (\lambda s. sup m0 (f s))$ — Assume that step preserved est. invars
 $\rrbracket \implies f m \leq P$ — Show that step preserves invariant
shows $lfp (\lambda s. sup m0 (f s)) \leq P$
 $\langle proof \rangle$

Connecting Complete Lattices and Chain-Complete Partial Orders

lemma (in *complete-lattice*) *is-ccpo*: *class.ccpo* *Sup* (\leq) ($<$)
 $\langle proof \rangle$

lemma (in *complete-lattice*) *is-dual-ccpo*: *class.ccpo* *Inf* (\geq) ($>$)
 $\langle proof \rangle$

lemma *ccpo-mono-simp*: *monotone* (\leq) (\leq) $f \longleftrightarrow mono\ f$
 $\langle proof \rangle$

lemma *ccpo-monoI*: *mono* $f \implies monotone\ (\leq)\ (\leq)\ f$
 $\langle proof \rangle$

lemma *ccpo-monoD*: *monotone* (\leq) (\leq) $f \implies mono\ f$
 $\langle proof \rangle$

lemma *dual-ccpo-mono-simp*: *monotone* (\geq) (\geq) $f \longleftrightarrow mono\ f$
 $\langle proof \rangle$

lemma *dual-ccpo-monoI*: *mono* $f \implies monotone\ (\geq)\ (\geq)\ f$
 $\langle proof \rangle$

lemma *dual-ccpo-monoD*: *monotone* (\geq) (\geq) $f \implies mono\ f$
 $\langle proof \rangle$

lemma *ccpo-lfp-simp*: $\bigwedge f. mono\ f \implies ccpo.fixp\ Sup\ (\leq)\ f = lfp\ f$
 $\langle proof \rangle$

lemma *ccpo-gfp-simp*: $\bigwedge f. mono\ f \implies ccpo.fixp\ Inf\ (\geq)\ f = gfp\ f$
 $\langle proof \rangle$

abbreviation *chain-admissible* $P \equiv ccpo.admissible\ Sup\ (\leq)\ P$

abbreviation *is-chain* $\equiv Complete-Partial-Order.chain\ (\leq)$

lemmas *chain-admissibleI*[*intro?*] = *ccpo.admissibleI*[**where** *lub*=*Sup* **and** *ord*=(\leq)]

abbreviation *dual-chain-admissible* $P \equiv ccpo.admissible\ Inf\ (\lambda x\ y. y \leq x)\ P$

abbreviation *is-dual-chain* $\equiv Complete-Partial-Order.chain\ (\lambda x\ y. y \leq x)$

lemmas *dual-chain-admissibleI*[*intro?*] =

$ccpo.admissibleI$ [**where** *lub*=*Inf* **and** *ord*=($\lambda x\ y. y \leq x$)]

lemma *dual-chain-iff*[simp]: *is-dual-chain* $C = \textit{is-chain}$ C
 ⟨proof⟩

lemmas *chain-dualI* = *iffD1*[OF *dual-chain-iff*]
lemmas *dual-chainI* = *iffD2*[OF *dual-chain-iff*]

lemma *is-chain-empty*[simp, intro!]: *is-chain* $\{\}$
 ⟨proof⟩

lemma *is-dual-chain-empty*[simp, intro!]: *is-dual-chain* $\{\}$
 ⟨proof⟩

lemma *point-chainI*: *is-chain* $M \implies \textit{is-chain}$ $((\lambda f. f\ x)'\ M)$
 ⟨proof⟩

We transfer the admissible induction lemmas to complete lattices.

lemma *lfp-cadm-induct*:
 $\llbracket \textit{chain-admissible}$ P ; P (\textit{Sup} $\{\}$); \textit{mono} f ; $\bigwedge x. P\ x \implies P\ (f\ x) \rrbracket \implies P\ (\textit{lfp}\ f)$
 ⟨proof⟩

lemma *gfp-cadm-induct*:
 $\llbracket \textit{dual-chain-admissible}$ P ; P (\textit{Inf} $\{\}$); \textit{mono} f ; $\bigwedge x. P\ x \implies P\ (f\ x) \rrbracket \implies P\ (\textit{gfp}\ f)$
 ⟨proof⟩

Continuity and Kleene Fixed Point Theorem

definition *cont* $f \equiv \forall C. C \neq \{\} \longrightarrow f\ (\textit{Sup}\ C) = \textit{Sup}\ (f'\ C)$

definition *strict* $f \equiv f\ \textit{bot} = \textit{bot}$

definition *inf-distrib* $f \equiv \textit{strict}\ f \wedge \textit{cont}\ f$

lemma *contI*[intro?]: $\llbracket \bigwedge C. C \neq \{\} \implies f\ (\textit{Sup}\ C) = \textit{Sup}\ (f'\ C) \rrbracket \implies \textit{cont}\ f$
 ⟨proof⟩

lemma *contD*: $\textit{cont}\ f \implies C \neq \{\} \implies f\ (\textit{Sup}\ C) = \textit{Sup}\ (f'\ C)$
 ⟨proof⟩

lemma *contD'*: $\textit{cont}\ f \implies C \neq \{\} \implies f\ (\textit{Sup}\ C) = \textit{Sup}\ (f'\ C)$
 ⟨proof⟩

lemma *strictD*[dest]: $\textit{strict}\ f \implies f\ \textit{bot} = \textit{bot}$
 ⟨proof⟩

lemma *strictD-simp*[simp]: $\textit{strict}\ f \implies f\ (\textit{bot}::'a::\textit{bot}) = (\textit{bot}::'a)$
 ⟨proof⟩

lemma *strictI*[intro?]: $f\ \textit{bot} = \textit{bot} \implies \textit{strict}\ f$
 ⟨proof⟩

lemma *inf-distribD*[simp]:
 $\textit{inf-distrib}\ f \implies \textit{strict}\ f$
 $\textit{inf-distrib}\ f \implies \textit{cont}\ f$

<proof>

lemma *inf-distribI*[*intro?*]: $\llbracket \text{strict } f; \text{ cont } f \rrbracket \implies \text{inf-distrib } f$
<proof>

lemma *inf-distribD'*[*simp*]:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'b::\text{complete-lattice}$
shows $\text{inf-distrib } f \implies f (\text{Sup } C) = \text{Sup } (f' C)$
<proof>

lemma *inf-distribI'*:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'b::\text{complete-lattice}$
assumes $B: \bigwedge C. f (\text{Sup } C) = \text{Sup } (f' C)$
shows $\text{inf-distrib } f$
<proof>

lemma *cont-is-mono*[*simp*]:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'b::\text{complete-lattice}$
shows $\text{cont } f \implies \text{mono } f$
<proof>

lemma *inf-distrib-is-mono*[*simp*]:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'b::\text{complete-lattice}$
shows $\text{inf-distrib } f \implies \text{mono } f$
<proof>

Only proven for complete lattices here. Also holds for CCPOs.

theorem *gen-kleene-lfp*:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'a$
assumes *CONT*: $\text{cont } f$
shows $\text{lfp } (\lambda x. \text{sup } m (f x)) = (\text{SUP } i. (f \hat{\sim} i) m)$
<proof>

theorem *kleene-lfp*:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'a$
assumes *CONT*: $\text{cont } f$
shows $\text{lfp } f = (\text{SUP } i. (f \hat{\sim} i) \text{bot})$
<proof>

theorem
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'a$
assumes *CONT*: $\text{cont } f$
shows $\text{lfp } f = (\text{SUP } i. (f \hat{\sim} i) \text{bot})$
<proof>

lemma *SUP-funpow-contracting*:
fixes $f :: 'a \Rightarrow ('a::\text{complete-lattice})$
assumes *C*: $\text{cont } f$

shows $f (SUP\ i. (f \sim i)\ m) \leq (SUP\ i. (f \sim i)\ m)$
 ⟨proof⟩

lemma *gen-kleene-chain-conv*:

fixes $f :: 'a::complete-lattice \Rightarrow 'a$

assumes $C: cont\ f$

shows $(SUP\ i. (f \sim i)\ m) = (SUP\ i. ((\lambda x. sup\ m\ (f\ x)) \sim i)\ bot)$
 ⟨proof⟩

theorem

assumes $C: cont\ f$

shows $lfp\ (\lambda x. sup\ m\ (f\ x)) = (SUP\ i. (f \sim i)\ m)$

⟨proof⟩

lemma (in *galois-connection*) *dual-inf-dist- γ* : $\gamma (Inf\ C) = Inf\ (\gamma\ C)$
 ⟨proof⟩

lemma (in *galois-connection*) *inf-dist- α* : *inf-distrib* α
 ⟨proof⟩

2.1.4 Maps

Key-Value Set

lemma *map-to-set-simps*[*simp*]:

map-to-set $Map.empty = \{\}$

map-to-set $[a \rightarrow b] = \{(a,b)\}$

map-to-set $(m|K) = map-to-set\ m \cap K \times UNIV$

map-to-set $(m(x:=None)) = map-to-set\ m - \{x\} \times UNIV$

map-to-set $(m(x \mapsto v)) = map-to-set\ m - \{x\} \times UNIV \cup \{(x,v)\}$

map-to-set $m \cap dom\ m \times UNIV = map-to-set\ m$

$m\ k = Some\ v \implies (k,v) \in map-to-set\ m$

single-valued $(map-to-set\ m)$

⟨proof⟩

lemma *map-to-set-inj*:

$(k,v) \in map-to-set\ m \implies (k,v') \in map-to-set\ m \implies v = v'$

⟨proof⟩

end

2.2 Transfer between Domains

theory *RefineG-Transfer*

imports *../Refine-Misc*

begin

Currently, this theory is specialized to transfers that include no data refinement.

definition *REFINEG-TRANSFER-POST-SIMP* $x\ y \equiv x=y$

definition [*simp*]: *REFINEG-TRANSFER-ALIGN* $x\ y == \text{True}$

lemma *REFINEG-TRANSFER-ALIGN1*: *REFINEG-TRANSFER-ALIGN* $x\ y$ *<proof>*

lemma *START-REFINEG-TRANSFER*:

assumes *REFINEG-TRANSFER-ALIGN* $d\ c$

assumes $c \leq a$

assumes *REFINEG-TRANSFER-POST-SIMP* $c\ d$

shows $d \leq a$

<proof>

lemma *STOP-REFINEG-TRANSFER*: *REFINEG-TRANSFER-POST-SIMP* $c\ c$

<proof>

<ML>

locale *transfer* = **fixes** $\alpha :: 'c \Rightarrow 'a::\text{complete-lattice}$
begin

In the following, we define some transfer lemmas for general HOL - constructs.

lemma *transfer-if*[*refine-transfer*]:

assumes $b \Longrightarrow \alpha\ s1 \leq S1$

assumes $\neg b \Longrightarrow \alpha\ s2 \leq S2$

shows $\alpha\ (\text{if } b \text{ then } s1 \text{ else } s2) \leq (\text{if } b \text{ then } S1 \text{ else } S2)$

<proof>

lemma *transfer-prod*[*refine-transfer*]:

assumes $\bigwedge a\ b. \alpha\ (f\ a\ b) \leq F\ a\ b$

shows $\alpha\ (\text{case-prod } f\ x) \leq (\text{case-prod } F\ x)$

<proof>

lemma *transfer-Let*[*refine-transfer*]:

assumes $\bigwedge x. \alpha\ (f\ x) \leq F\ x$

shows $\alpha\ (\text{Let } x\ f) \leq \text{Let } x\ F$

<proof>

lemma *transfer-option*[*refine-transfer*]:

assumes $\alpha\ fa \leq Fa$

assumes $\bigwedge x. \alpha\ (fb\ x) \leq Fb\ x$

shows $\alpha\ (\text{case-option } fa\ fb\ x) \leq \text{case-option } Fa\ Fb\ x$

<proof>

lemma *transfer-sum*[*refine-transfer*]:

assumes $\bigwedge l. \alpha\ (fl\ l) \leq Fl\ l$

assumes $\bigwedge r. \alpha (fr\ r) \leq Fr\ r$
shows $\alpha (case\text{-}sum\ fl\ fr\ x) \leq (case\text{-}sum\ Fl\ Fr\ x)$
 $\langle proof \rangle$

lemma *transfer-list*[*refine-transfer*]:
assumes $\alpha\ fn \leq Fn$
assumes $\bigwedge x\ xs. \alpha (fc\ x\ xs) \leq Fc\ x\ xs$
shows $\alpha (case\text{-}list\ fn\ fc\ l) \leq case\text{-}list\ Fn\ Fc\ l$
 $\langle proof \rangle$

lemma *transfer-rec-list*[*refine-transfer*]:
assumes $FN: \bigwedge s. \alpha (fn\ s) \leq fn'\ s$
assumes $FC: \bigwedge x\ l\ rec\ rec'\ s. \llbracket \bigwedge s. \alpha (rec\ s) \leq (rec'\ s) \rrbracket$
 $\implies \alpha (fc\ x\ l\ rec\ s) \leq fc'\ x\ l\ rec'\ s$
shows $\alpha (rec\text{-}list\ fn\ fc\ l\ s) \leq rec\text{-}list\ fn'\ fc'\ l\ s$
 $\langle proof \rangle$

lemma *transfer-rec-nat*[*refine-transfer*]:
assumes $FN: \bigwedge s. \alpha (fn\ s) \leq fn'\ s$
assumes $FC: \bigwedge n\ rec\ rec'\ s. \llbracket \bigwedge s. \alpha (rec\ s) \leq rec'\ s \rrbracket$
 $\implies \alpha (fs\ n\ rec\ s) \leq fs'\ n\ rec'\ s$
shows $\alpha (rec\text{-}nat\ fn\ fs\ n\ s) \leq rec\text{-}nat\ fn'\ fs'\ n\ s$
 $\langle proof \rangle$

end

Transfer into complete lattice structure

locale *ordered-transfer* = *transfer* +
constrains $\alpha :: 'c::complete\text{-}lattice \Rightarrow 'a::complete\text{-}lattice$

Transfer into complete lattice structure with distributive transfer function.

locale *dist-transfer* = *ordered-transfer* +
constrains $\alpha :: 'c::complete\text{-}lattice \Rightarrow 'a::complete\text{-}lattice$
assumes $\alpha\text{-}dist: \bigwedge A. is\text{-}chain\ A \implies \alpha (Sup\ A) = Sup\ (\alpha\ 'A)$

begin

lemma $\alpha\text{-}mono$ [*simp*, *intro!*]: *mono* α
 $\langle proof \rangle$

lemma $\alpha\text{-}strict$ [*simp*]: $\alpha\ bot = bot$
 $\langle proof \rangle$

end

Transfer into ccpo

locale *ccpo-transfer* = *transfer* α **for**
 $\alpha :: 'c::ccpo \Rightarrow 'a::complete\text{-}lattice$

Transfer into ccpo with distributive transfer function.

locale *dist-ccpo-transfer* = *ccpo-transfer* α

```

for  $\alpha :: 'c::ccpo \Rightarrow 'a::complete-lattice +$ 
assumes  $\alpha\text{-dist}: \bigwedge A. \text{is-chain } A \Longrightarrow \alpha (\text{Sup } A) = \text{Sup } (\alpha 'A)$ 
begin

  lemma  $\alpha\text{-mono}[simp, \text{intro!}]: \text{mono } \alpha$ 
   $\langle \text{proof} \rangle$ 

  lemma  $\alpha\text{-strict}[simp]: \alpha (\text{Sup } \{\}) = \text{bot}$ 
   $\langle \text{proof} \rangle$ 
end

end

```

2.3 General Domain Theory

```

theory RefineG-Domain
imports ../Refine-Misc
begin

```

2.3.1 General Order Theory Tools

```

lemma chain-f-apply: Complete-Partial-Order.chain (fun-ord le) F
 $\Longrightarrow \text{Complete-Partial-Order.chain } le \{y . \exists f \in F. y = f x\}$ 
 $\langle \text{proof} \rangle$ 

lemma ccpo-lift:
  assumes class.ccpo lub le lt
  shows class.ccpo (fun-lub lub) (fun-ord le) (mk-less (fun-ord le))
   $\langle \text{proof} \rangle$ 

lemma fun-lub-simps[simp]:
   $\text{fun-lub } lub \{\} = (\lambda x. \text{lub } \{\})$ 
   $\text{fun-lub } lub \{f\} = (\lambda x. \text{lub } \{f x\})$ 
   $\langle \text{proof} \rangle$ 

```

2.3.2 Flat Ordering

```

lemma flat-ord-chain-cases:
  assumes A: Complete-Partial-Order.chain (flat-ord b) C
  obtains  $C = \{\}$ 
  |  $C = \{b\}$ 
  |  $x \text{ where } x \neq b \text{ and } C = \{x\}$ 
  |  $x \text{ where } x \neq b \text{ and } C = \{b, x\}$ 
   $\langle \text{proof} \rangle$ 

lemma flat-lub-simps[simp]:
   $\text{flat-lub } b \{\} = b$ 
   $\text{flat-lub } b \{x\} = x$ 

```

$flat-lub\ b\ (insert\ b\ X) = flat-lub\ b\ X$
 $\langle proof \rangle$

lemma $flat-ord-simps[simp]$:
 $flat-ord\ b\ b\ x$
 $\langle proof \rangle$

interpretation $flat-ord$: $ccpo\ flat-lub\ b\ flat-ord\ b\ mk-less\ (flat-ord\ b)$
 $\langle proof \rangle$

interpretation $flat-le-mono-setup$: $mono-setup-loc\ flat-ord\ b$
 $\langle proof \rangle$

Flat function Ordering

abbreviation $flatf-ord\ b == fun-ord\ (flat-ord\ b)$
abbreviation $flatf-lub\ b == fun-lub\ (flat-lub\ b)$

interpretation $flatf-ord$: $ccpo\ flatf-lub\ b\ flatf-ord\ b\ mk-less\ (flatf-ord\ b)$
 $\langle proof \rangle$

Fixed Points in Flat Ordering

Fixed points in a flat ordering are used to express recursion. The bottom element is interpreted as non-termination.

abbreviation $flat-mono\ b == monotone\ (flat-ord\ b)\ (flat-ord\ b)$
abbreviation $flatf-mono\ b == monotone\ (flatf-ord\ b)\ (flatf-ord\ b)$
abbreviation $flatf-fp\ b \equiv flatf-ord.fxp\ b$

lemma $flatf-fp-mono[refine-mono]$:
— The fixed point combinator is monotonic
assumes $flatf-mono\ b\ f$
and $flatf-mono\ b\ g$
and $\bigwedge Z\ x.\ flat-ord\ b\ (f\ Z\ x)\ (g\ Z\ x)$
shows $flat-ord\ b\ (flatf-fp\ b\ f\ x)\ (flatf-fp\ b\ g\ x)$
 $\langle proof \rangle$

lemma $flatf-admissible-pointwise$:
 $(\bigwedge x.\ P\ x\ b) \implies$
 $ccpo.admissible\ (flatf-lub\ b)\ (flatf-ord\ b)\ (\lambda g.\ \forall x.\ P\ x\ (g\ x))$
 $\langle proof \rangle$

If a property is defined pointwise, and holds for the bottom element, we can use fixed-point induction for it.

In the induction step, we can assume that the function is less or equal to the fixed-point.

This rule covers refinement and transfer properties, such as: Refinement of fixed-point combinators and transfer of fixed-point combinators to different

domains.

lemma *flatf-fp-induct-pointwise*:
 — Fixed-point induction for pointwise properties
fixes $a :: 'a$
assumes *cond-bot*: $\bigwedge a x. \text{pre } a x \implies \text{post } a x b$
assumes *MONO*: *flatf-mono* $b B$
assumes *PRE0*: $\text{pre } a x$
assumes *STEP*: $\bigwedge f a x.$
 $\llbracket \bigwedge a' x'. \text{pre } a' x' \implies \text{post } a' x' (f x'); \text{pre } a x;$
 $\text{flatf-ord } b f (\text{flatf-fp } b B) \rrbracket$
 $\implies \text{post } a x (B f x)$
shows $\text{post } a x (\text{flatf-fp } b B x)$
 $\langle \text{proof} \rangle$

The next rule covers transfer between fixed points. It allows to lift a pointwise transfer condition $P x y \longrightarrow \text{tr } (f x) (f y)$ to fixed points. Note that one of the fixed points may be an arbitrary fixed point.

lemma *flatf-fixp-transfer*:
 — Transfer rule for fixed points
assumes *TR-BOT*[*simp*]: $\bigwedge x'. \text{tr } b x'$
assumes *MONO*: *flatf-mono* $b B$
assumes *FP'*: $\text{fp}' = B' \text{fp}'$
assumes *R0*: $P x x'$
assumes *RS*: $\bigwedge f f' x x'.$
 $\llbracket \bigwedge x x'. P x x' \implies \text{tr } (f x) (f' x'); P x x'; \text{fp}' = f' \rrbracket$
 $\implies \text{tr } (B f x) (B' f' x')$
shows $\text{tr } (\text{flatf-fp } b B x) (\text{fp}' x')$
 $\langle \text{proof} \rangle$

Relation of Flat Ordering to Complete Lattices

In this section, we establish the relation between flat orderings and complete lattices. This relation is exploited to show properties of fixed points wrt. a refinement ordering.

abbreviation *flat-le* $\equiv \text{flat-ord bot}$
abbreviation *flat-ge* $\equiv \text{flat-ord top}$
abbreviation *flatf-le* $\equiv \text{flatf-ord bot}$
abbreviation *flatf-ge* $\equiv \text{flatf-ord top}$

The flat ordering implies the lattice ordering

lemma *flat-ord-compat*:
fixes $x y :: 'a :: \text{complete-lattice}$
shows
 $\text{flat-le } x y \implies x \leq y$
 $\text{flat-ge } x y \implies x \geq y$
 $\langle \text{proof} \rangle$

lemma *flatf-ord-compat*:

fixes $x\ y :: 'a \Rightarrow ('b :: \text{complete-lattice})$

shows

$\text{flatf-le } x\ y \Longrightarrow x \leq y$

$\text{flatf-ge } x\ y \Longrightarrow x \geq y$

$\langle \text{proof} \rangle$

abbreviation $\text{flat-mono-le} \equiv \text{flat-mono bot}$

abbreviation $\text{flat-mono-ge} \equiv \text{flat-mono top}$

abbreviation $\text{flatf-mono-le} \equiv \text{flatf-mono bot}$

abbreviation $\text{flatf-mono-ge} \equiv \text{flatf-mono top}$

abbreviation $\text{flatf-gfp} \equiv \text{flatf-ord.fixp top}$

abbreviation $\text{flatf-lfp} \equiv \text{flatf-ord.fixp bot}$

If a functor is monotonic wrt. both the flat and the lattice ordering, the fixed points wrt. these orderings coincide.

lemma *lfp-eq-flatf-lfp*:

assumes $FM: \text{flatf-mono-le } B$ **and** $M: \text{mono } B$

shows $\text{lfp } B = \text{flatf-lfp } B$

$\langle \text{proof} \rangle$

lemma *gfp-eq-flatf-gfp*:

assumes $FM: \text{flatf-mono-ge } B$ **and** $M: \text{mono } B$

shows $\text{gfp } B = \text{flatf-gfp } B$

$\langle \text{proof} \rangle$

The following lemma provides a well-founded induction scheme for arbitrary fixed point combinators.

lemma *wf-fixp-induct*:

— Well-Founded induction for arbitrary fixed points

fixes $a :: 'a$

assumes $\text{fixp-unfold}: \text{fp } B = B (\text{fp } B)$

assumes $WF: \text{wf } V$

assumes $P0: \text{pre } a\ x$

assumes $STEP: \bigwedge f\ a\ x. \llbracket$

$\bigwedge a'\ x'. \llbracket \text{pre } a'\ x'; (x',x) \in V \rrbracket \Longrightarrow \text{post } a'\ x' (f\ x'); \text{fp } B = f; \text{pre } a\ x$

$\rrbracket \Longrightarrow \text{post } a\ x (B\ f\ x)$

shows $\text{post } a\ x (\text{fp } B\ x)$

$\langle \text{proof} \rangle$

lemma *flatf-lfp-transfer*:

— Transfer rule for least fixed points

fixes $B::(- \Rightarrow 'a::\text{order-bot}) \Rightarrow -$

assumes $TR-BOT[\text{simp}]: \bigwedge x. \text{tr bot } x$

assumes $MONO: \text{flatf-mono-le } B$

assumes $MONO': \text{flatf-mono-le } B'$

```

assumes R0:  $P\ x\ x'$ 
assumes RS:  $\bigwedge f\ f'\ x\ x'. \llbracket \bigwedge x\ x'. P\ x\ x' \implies tr\ (f\ x)\ (f'\ x'); P\ x\ x'; flatf\text{-}lfp\ B' = f \rrbracket$ 
            $\implies tr\ (B\ f\ x)\ (B'\ f'\ x')$ 
shows  $tr\ (flatf\text{-}lfp\ B\ x)\ (flatf\text{-}lfp\ B'\ x')$ 
<proof>

```

lemma *flatf-gfp-transfer*:

```

— Transfer rule for greatest fixed points
fixes B:: $(- \Rightarrow 'a::order\text{-}top) \Rightarrow -$ 
assumes TR-TOP[simp]:  $\bigwedge x. tr\ x\ top$ 
assumes MONO: flatf-mono-ge B
assumes MONO': flatf-mono-ge B'
assumes R0:  $P\ x\ x'$ 
assumes RS:  $\bigwedge f\ f'\ x\ x'. \llbracket \bigwedge x\ x'. P\ x\ x' \implies tr\ (f\ x)\ (f'\ x'); P\ x\ x'; flatf\text{-}gfp\ B = f \rrbracket$ 
            $\implies tr\ (B\ f\ x)\ (B'\ f'\ x')$ 
shows  $tr\ (flatf\text{-}gfp\ B\ x)\ (flatf\text{-}gfp\ B'\ x')$ 
<proof>

```

lemma *meta-le-everything-if-top*: $(m=top) \implies (\bigwedge x. x \leq (m::'a::order\text{-}top))$
<proof>

```

lemmas flatf-lfp-refine = flatf-lfp-transfer[
  where  $tr = \lambda a\ b. a \leq cf\ b$  for cf, OF bot-least]
lemmas flatf-gfp-refine = flatf-gfp-transfer[
  where  $tr = \lambda a\ b. a \leq cf\ b$  for cf, OF meta-le-everything-if-top]

```

lemma *flat-ge-sup-mono*[*refine-mono*]: $\bigwedge a\ a'::'a::complete\text{-}lattice. flat\text{-}ge\ a\ a' \implies flat\text{-}ge\ b\ b' \implies flat\text{-}ge\ (sup\ a\ b)\ (sup\ a'\ b')$
<proof>

declare *sup-mono*[*refine-mono*]

end

2.4 Generic Recursion Combinator for Complete Lattice Structured Domains

```

theory RefineG-Recursion
imports ../Refine-Misc RefineG-Transfer RefineG-Domain
begin

```

We define a recursion combinator that asserts monotonicity.

The following lemma allows to compare least fixed points wrt. different flat orderings. At any point, the fixed points are either equal or have their

2.4. GENERIC RECURSION COMBINATOR FOR COMPLETE LATTICE STRUCTURED DOMAINS

orderings bottom values.

lemma *fp-compare*:

— At any point, fixed points wrt. different orderings are either equal, or both bottom.

assumes *M1*: *flatf-mono b1 B* **and** *M2*: *flatf-mono b2 B*

shows *flatf-fp b1 B x = flatf-fp b2 B x*

\vee (*flatf-fp b1 B x = bot* \wedge *flatf-fp b2 B x = bot*)

<proof>

lemma *flat-ord-top[simp]*: *flat-ord b b x* *<proof>*

lemma *lfp-gfp-compare*:

— Least and greatest fixed point are either equal, or bot and top

assumes *MLE*: *flatf-mono-le B* **and** *MGE*: *flatf-mono-ge B*

shows *flatf-lfp B x = flatf-gfp B x*

\vee (*flatf-lfp B x = bot* \wedge *flatf-gfp B x = top*)

<proof>

definition *trimono* :: (*'a* \Rightarrow *'b*) \Rightarrow *'a* \Rightarrow (*'b*::{*bot,order,top*}) \Rightarrow *bool*

where *trimono B* \equiv ~~*flatf-mono-ge B*~~ *flatf-mono-ge B* \wedge *mono B*

lemma *trimonoI[refine-mono]*:

\llbracket *flatf-mono-ge B*; *mono B* $\rrbracket \Longrightarrow$ *trimono B*

<proof>

lemma *trimono-trigger*: *trimono B* \Longrightarrow *trimono B* *<proof>*

<ML>

lemma *trimonoD-flatf-ge*: *trimono B* \Longrightarrow *flatf-mono-ge B*

<proof>

lemma *trimonoD-mono*: *trimono B* \Longrightarrow *mono B*

<proof>

lemmas *trimonoD = trimonoD-flatf-ge trimonoD-mono*

definition *triords* \equiv {*flat-ge,(<=)*}

lemma *trimono-alt*:

trimono B \longleftrightarrow (\forall *ord* \in *fun-ord* *triords*. *monotone ord B*)

<proof>

lemma *trimonoI'*:

assumes $\bigwedge \text{ord. } \text{ord} \in \text{triords} \implies \text{monotone } (\text{fun-ord } \text{ord}) (\text{fun-ord } \text{ord}) B$
shows *trimono B*
 ⟨*proof*⟩

definition *REC* **where** $\text{REC } B x \equiv$

if (*trimono B*) *then* (*lfp B x*) *else* (*top::'a::complete-lattice*)

definition *RECT* (*RECT_T*) **where** $\text{RECT } B x \equiv$

if (*trimono B*) *then* (*flatf-gfp B x*) *else* (*top::'a::complete-lattice*)

lemma *RECT-gfp-def*: $\text{RECT } B x =$

(*if* (*trimono B*) *then* (*gfp B x*) *else* (*top::'a::complete-lattice*))
 ⟨*proof*⟩

lemma *REC-unfold*: $\text{trimono } B \implies \text{REC } B = B (\text{REC } B)$

⟨*proof*⟩

lemma *RECT-unfold*: $\llbracket \text{trimono } B \rrbracket \implies \text{RECT } B = B (\text{RECT } B)$

⟨*proof*⟩

lemma *REC-mono[refine-mono]*:

assumes [*simp*]: *trimono B*
assumes *LE*: $\bigwedge F x. (B F x) \leq (B' F x)$
shows $(\text{REC } B x) \leq (\text{REC } B' x)$
 ⟨*proof*⟩

lemma *RECT-mono[refine-mono]*:

assumes [*simp*]: *trimono B'*
assumes *LE*: $\bigwedge F x. \text{flat-ge } (B F x) (B' F x)$
shows $\text{flat-ge } (\text{RECT } B x) (\text{RECT } B' x)$
 ⟨*proof*⟩

lemma *REC-le-RECT*: $\text{REC } \text{body } x \leq \text{RECT } \text{body } x$

⟨*proof*⟩

print-statement *flatf-fp-induct-pointwise*

theorem *lfp-induct-pointwise*:

fixes *a::'a*

assumes *ADM1*: $\bigwedge a x. \text{chain-admissible } (\lambda b. \forall a x. \text{pre } a x \longrightarrow \text{post } a x (b x))$

assumes *ADM2*: $\bigwedge a x. \text{pre } a x \longrightarrow \text{post } a x \text{ bot}$

assumes *MONO*: *mono B*

assumes *P0*: *pre a x*

assumes *IS*:

$\bigwedge f a x.$

$\llbracket \bigwedge a' x'. \text{pre } a' x' \implies \text{post } a' x' (f x'); \text{pre } a x;$
 $f \leq (\text{lfp } B) \rrbracket$

2.4. GENERIC RECURSION COMBINATOR FOR COMPLETE LATTICE STRUCTURED DOMAINS

$\implies \text{post } a \ x \ (B \ f \ x)$
shows $\text{post } a \ x \ (\text{lfp } B \ x)$
 $\langle \text{proof} \rangle$

lemma *REC-rule-arb*:

fixes $x::'x$ **and** $\text{arb}::'\text{arb}$
assumes M : *trimono body*
assumes $I0$: $\text{pre } \text{arb } x$
assumes IS : $\bigwedge f \ \text{arb } x. \llbracket$
 $\bigwedge \text{arb}' \ x. \ \text{pre } \text{arb}' \ x \implies f \ x \leq M \ \text{arb}' \ x; \ \text{pre } \text{arb } x; \ f \leq \text{REC body}$
 $\rrbracket \implies \text{body } f \ x \leq M \ \text{arb } x$
shows $\text{REC body } x \leq M \ \text{arb } x$
 $\langle \text{proof} \rangle$

lemma *RECT-rule-arb*:

assumes M : *trimono body*
assumes WF : $wf \ (V::('x \times 'x) \ \text{set})$
assumes $I0$: $\text{pre } (\text{arb}::'\text{arb}) \ (x::'x)$
assumes IS : $\bigwedge f \ \text{arb } x. \llbracket$
 $\bigwedge \text{arb}' \ x'. \llbracket \text{pre } \text{arb}' \ x'; \ (x', x) \in V \rrbracket \implies f \ x' \leq M \ \text{arb}' \ x';$
 $\text{pre } \text{arb } x;$
 $\text{RECT body} = f$
 $\rrbracket \implies \text{body } f \ x \leq M \ \text{arb } x$
shows $\text{RECT body } x \leq M \ \text{arb } x$
 $\langle \text{proof} \rangle$

lemma *REC-rule*:

fixes $x::'x$
assumes M : *trimono body*
assumes $I0$: $\text{pre } x$
assumes IS : $\bigwedge f \ x. \llbracket \bigwedge x. \ \text{pre } x \implies f \ x \leq M \ x; \ \text{pre } x; \ f \leq \text{REC body} \rrbracket$
 $\implies \text{body } f \ x \leq M \ x$
shows $\text{REC body } x \leq M \ x$
 $\langle \text{proof} \rangle$

lemma *RECT-rule*:

assumes M : *trimono body*
assumes WF : $wf \ (V::('x \times 'x) \ \text{set})$
assumes $I0$: $\text{pre } (x::'x)$
assumes IS : $\bigwedge f \ x. \llbracket \bigwedge x'. \llbracket \text{pre } x'; \ (x', x) \in V \rrbracket \implies f \ x' \leq M \ x'; \ \text{pre } x;$
 $\text{RECT body} = f$
 $\rrbracket \implies \text{body } f \ x \leq M \ x$
shows $\text{RECT body } x \leq M \ x$
 $\langle \text{proof} \rangle$

lemma *REC-rule-arb2*:

assumes M : *trimono body*
assumes $I0$: $pre\ (arb::'arb)\ (arc::'arc)\ (x::'x)$
assumes IS : $\bigwedge f\ arb\ arc\ x.\ \llbracket$
 $\bigwedge arb'\ arc'\ x'.\ \llbracket pre\ arb'\ arc'\ x' \rrbracket \implies f\ x' \leq M\ arb'\ arc'\ x';$
 $pre\ arb\ arc\ x$
 $\rrbracket \implies body\ f\ x \leq M\ arb\ arc\ x$
shows *REC* $body\ x \leq M\ arb\ arc\ x$
 $\langle proof \rangle$

lemma *REC-rule-arb3*:

assumes M : *trimono body*
assumes $I0$: $pre\ (arb::'arb)\ (arc::'arc)\ (ard::'ard)\ (x::'x)$
assumes IS : $\bigwedge f\ arb\ arc\ ard\ x.\ \llbracket$
 $\bigwedge arb'\ arc'\ ard'\ x'.\ \llbracket pre\ arb'\ arc'\ ard'\ x' \rrbracket \implies f\ x' \leq M\ arb'\ arc'\ ard'\ x';$
 $pre\ arb\ arc\ ard\ x$
 $\rrbracket \implies body\ f\ x \leq M\ arb\ arc\ ard\ x$
shows *REC* $body\ x \leq M\ arb\ arc\ ard\ x$
 $\langle proof \rangle$

lemma *RECT-rule-arb2*:

assumes M : *trimono body*
assumes WF : $wf\ (V::'x\ rel)$
assumes $I0$: $pre\ (arb::'arb)\ (arc::'arc)\ (x::'x)$
assumes IS : $\bigwedge f\ arb\ arc\ x.\ \llbracket$
 $\bigwedge arb'\ arc'\ x'.\ \llbracket pre\ arb'\ arc'\ x';\ (x',x) \in V \rrbracket \implies f\ x' \leq M\ arb'\ arc'\ x';$
 $pre\ arb\ arc\ x;$
 $f \leq RECT\ body$
 $\rrbracket \implies body\ f\ x \leq M\ arb\ arc\ x$
shows *RECT* $body\ x \leq M\ arb\ arc\ x$
 $\langle proof \rangle$

lemma *RECT-rule-arb3*:

assumes M : *trimono body*
assumes WF : $wf\ (V::'x\ rel)$
assumes $I0$: $pre\ (arb::'arb)\ (arc::'arc)\ (ard::'ard)\ (x::'x)$
assumes IS : $\bigwedge f\ arb\ arc\ ard\ x.\ \llbracket$
 $\bigwedge arb'\ arc'\ ard'\ x'.\ \llbracket pre\ arb'\ arc'\ ard'\ x';\ (x',x) \in V \rrbracket \implies f\ x' \leq M\ arb'\ arc'$
 $ard'\ x';$
 $pre\ arb\ arc\ ard\ x;$
 $f \leq RECT\ body$
 $\rrbracket \implies body\ f\ x \leq M\ arb\ arc\ ard\ x$
shows *RECT* $body\ x \leq M\ arb\ arc\ ard\ x$
 $\langle proof \rangle$

2.4. GENERIC RECURSION COMBINATOR FOR COMPLETE LATTICE STRUCTURED DOMAINS

lemma *RECT-eq-REC*:

— Partial and total correct recursion are equal if total recursion does not fail.

assumes *NT*: *RECT body x* \neq *top*

shows *RECT body x* = *REC body x*

<proof>

lemma *RECT-eq-REC-tproof*:

— Partial and total correct recursion are equal if we can provide a termination proof.

fixes *a* :: 'a

assumes *M*: *trimono body*

assumes *WF*: *wf V*

assumes *I0*: *pre a x*

assumes *IS*: $\bigwedge f \text{ arb } x.$

$\llbracket \bigwedge \text{arb}' x'. \llbracket \text{pre arb}' x'; (x', x) \in V \rrbracket \implies f x' \leq M \text{ arb}' x';$
 $\text{pre arb } x; \text{REC}_T \text{ body} = f \rrbracket$

$\implies \text{body } f x \leq M \text{ arb } x$

assumes *NT*: *M a x* \neq *top*

shows *RECT body x* = *REC body x* \wedge *RECT body x* \leq *M a x*

<proof>

2.4.1 Transfer

lemma (*in transfer*) *transfer-RECT'*[*refine-transfer*]:

assumes *REC-EQ*: $\bigwedge x. \text{fr } x = b \text{ fr } x$

assumes *REF*: $\bigwedge F f x. \llbracket \bigwedge x. \alpha (f x) \leq F x \rrbracket \implies \alpha (b f x) \leq B F x$

shows $\alpha (\text{fr } x) \leq \text{RECT } B x$

<proof>

lemma (*in ordered-transfer*) *transfer-RECT*[*refine-transfer*]:

assumes *REF*: $\bigwedge F f x. \llbracket \bigwedge x. \alpha (f x) \leq F x \rrbracket \implies \alpha (b f x) \leq B F x$

assumes *M*: *trimono b*

shows $\alpha (\text{RECT } b x) \leq \text{RECT } B x$

<proof>

lemma (*in dist-transfer*) *transfer-REC*[*refine-transfer*]:

assumes *REF*: $\bigwedge F f x. \llbracket \bigwedge x. \alpha (f x) \leq F x \rrbracket \implies \alpha (b f x) \leq B F x$

assumes *M*: *trimono b*

shows $\alpha (\text{REC } b x) \leq \text{REC } B x$

<proof>

lemma *RECT-transfer-rel*:

assumes [*simp*]: *trimono F* *trimono F'*

assumes *TR-top*[*simp*]: $\bigwedge x. \text{tr } x \text{ top}$

assumes *P-start*[*simp*]: *P x x'*

assumes $IS: \bigwedge D D' x x'. \llbracket \bigwedge x x'. P x x' \implies tr (D x) (D' x'); P x x'; RECT F = D \rrbracket \implies tr (F D x) (F' D' x')$
shows $tr (RECT F x) (RECT F' x')$
 $\langle proof \rangle$

lemma *RECT-transfer-rel'*:

assumes $[simp]: trimono F \quad trimono F'$
assumes $TR-top[simp]: \bigwedge x. tr x top$
assumes $P-start[simp]: P x x'$
assumes $IS: \bigwedge D D' x x'. \llbracket \bigwedge x x'. P x x' \implies tr (D x) (D' x'); P x x' \rrbracket \implies tr (F D x) (F' D' x')$
shows $tr (RECT F x) (RECT F' x')$
 $\langle proof \rangle$

end

2.5 Assert and Assume

theory *RefineG-Assert*

imports *RefineG-Transfer*

begin

definition *iASSERT* $return \Phi \equiv if \Phi then return () else top$

definition *iASSUME* $return \Phi \equiv if \Phi then return () else bot$

locale *generic-Assert* =

fixes $bind ::$

$('mu::complete-lattice) \Rightarrow (unit \Rightarrow ('ma::complete-lattice)) \Rightarrow 'ma$

fixes $return :: unit \Rightarrow 'mu$

fixes *ASSERT*

fixes *ASSUME*

assumes *ibind-strict*:

$bind bot f = bot$

$bind top f = top$

assumes *imonad1*: $bind (return u) f = f u$

assumes *ASSERT-eq*: $ASSERT \equiv iASSERT return$

assumes *ASSUME-eq*: $ASSUME \equiv iASSUME return$

begin

lemma *ASSERT-simps* $[simp,code]$:

$ASSERT True = return ()$

$ASSERT False = top$

$\langle proof \rangle$

lemma *ASSUME-simps* $[simp,code]$:

$ASSUME True = return ()$

$ASSUME False = bot$

$\langle proof \rangle$

lemma *le-ASSERTI*: $\llbracket \Phi \implies M \leq M' \rrbracket \implies M \leq bind (ASSERT \Phi) (\lambda-. M')$

$\langle proof \rangle$

lemma *le-ASSERTI-pres*: $\llbracket \Phi \implies M \leq bind (ASSERT \Phi) (\lambda-. M') \rrbracket$
 $\implies M \leq bind (ASSERT \Phi) (\lambda-. M')$
 $\langle proof \rangle$

lemma *ASSERT-leI*: $\llbracket \Phi; \Phi \implies M \leq M' \rrbracket \implies bind (ASSERT \Phi) (\lambda-. M) \leq M'$
 $\langle proof \rangle$

lemma *ASSUME-leI*: $\llbracket \Phi \implies M \leq M' \rrbracket \implies bind (ASSUME \Phi) (\lambda-. M) \leq M'$
 $\langle proof \rangle$

lemma *ASSUME-leI-pres*: $\llbracket \Phi \implies bind (ASSUME \Phi) (\lambda-. M) \leq M' \rrbracket$
 $\implies bind (ASSUME \Phi) (\lambda-. M) \leq M'$
 $\langle proof \rangle$

lemma *le-ASSUMEI*: $\llbracket \Phi; \Phi \implies M \leq M' \rrbracket \implies M \leq bind (ASSUME \Phi) (\lambda-. M')$
 $\langle proof \rangle$

The order of these declarations does matter!

lemmas $[intro?] = ASSERT-leI le-ASSUMEI$
lemmas $[intro?] = le-ASSERTI ASSUME-leI$

lemma *ASSERT-le-iff*:
 $bind (ASSERT \Phi) (\lambda-. S) \leq S' \iff (S' \neq top \implies \Phi) \wedge S \leq S'$
 $\langle proof \rangle$

lemma *ASSUME-le-iff*:
 $bind (ASSUME \Phi) (\lambda-. S) \leq S' \iff (\Phi \implies S \leq S')$
 $\langle proof \rangle$

lemma *le-ASSERT-iff*:
 $S \leq bind (ASSERT \Phi) (\lambda-. S') \iff (\Phi \implies S \leq S')$
 $\langle proof \rangle$

lemma *le-ASSUME-iff*:
 $S \leq bind (ASSUME \Phi) (\lambda-. S') \iff (S \neq bot \implies \Phi) \wedge S \leq S'$
 $\langle proof \rangle$

end

This locale transfer's asserts and assumes. To remove them, use the next locale.

locale *transfer-generic-Assert* =
c: *generic-Assert* *cbind* *creturn* *cASSERT* *cASSUME* +
a: *generic-Assert* *abind* *areturn* *aASSERT* *aASSUME* +
ordered-transfer α
for *cbind* :: ('muc::complete-lattice)
 $\Rightarrow (unit \Rightarrow 'mac) \Rightarrow ('mac::complete-lattice)$

```

and creturn :: unit ⇒ 'muc and cASSERT and cASSUME
and abind :: ('mua::complete-lattice)
  ⇒ (unit⇒'maa) ⇒ ('maa::complete-lattice)
and areturn :: unit ⇒ 'mua and aASSERT and aASSUME
and α :: 'mac ⇒ 'maa
begin
lemma transfer-ASSERT[refine-transfer]:
   $\llbracket \Phi \implies \alpha M \leq M' \rrbracket$ 
  ⇒  $\alpha (cbind (cASSERT \Phi) (\lambda-. M)) \leq (abind (aASSERT \Phi) (\lambda-. M'))$ 
  ⟨proof⟩

lemma transfer-ASSUME[refine-transfer]:
   $\llbracket \Phi; \Phi \implies \alpha M \leq M' \rrbracket$ 
  ⇒  $\alpha (cbind (cASSUME \Phi) (\lambda-. M)) \leq (abind (aASSUME \Phi) (\lambda-. M'))$ 
  ⟨proof⟩

end

locale transfer-generic-Assert-remove =
  a: generic-Assert abind areturn aASSERT aASSUME +
  transfer α
for abind :: ('mua::complete-lattice)
  ⇒ (unit⇒'maa) ⇒ ('maa::complete-lattice)
and areturn :: unit ⇒ 'mua and aASSERT and aASSUME
and α :: 'mac ⇒ 'maa
begin
lemma transfer-ASSERT-remove[refine-transfer]:
   $\llbracket \Phi \implies \alpha M \leq M' \rrbracket \implies \alpha M \leq abind (aASSERT \Phi) (\lambda-. M')$ 
  ⟨proof⟩

lemma transfer-ASSUME-remove[refine-transfer]:
   $\llbracket \Phi; \Phi \implies \alpha M \leq M' \rrbracket \implies \alpha M \leq abind (aASSUME \Phi) (\lambda-. M')$ 
  ⟨proof⟩
end

end

```

2.6 Basic Concepts

```

theory Refine-Basic
imports Main
  HOL-Library.Monad-Syntax
  Refine-Misc
  Generic/RefineG-Recursion
  Generic/RefineG-Assert
begin

```

2.6.1 Nondeterministic Result Lattice and Monad

In this section we introduce a complete lattice of result sets with an additional top element that represents failure. On this lattice, we define a monad: The return operator models a result that consists of a single value, and the bind operator models applying a function to all results. Binding a failure yields always a failure.

In addition to the return operator, we also introduce the operator *RES*, that embeds a set of results into our lattice. Its synonym for a predicate is *SPEC*. Program correctness is expressed by refinement, i.e., the expression $M \leq SPEC \Phi$ means that M is correct w.r.t. specification Φ . This suggests the following view on the program lattice: The top-element is the result that is never correct. We call this result *FAIL*. The bottom element is the program that is always correct. It is called *SUCCEED*. An assertion can be encoded by failing if the asserted predicate is not true. Symmetrically, an assumption is encoded by succeeding if the predicate is not true.

datatype 'a nres = *FAILi* | *RES* 'a set

FAILi is only an internal notation, that should not be exposed to the user. Instead, *FAIL* should be used, that is defined later as abbreviation for the top element of the lattice.

instantiation nres :: (type) complete-lattice

begin

fun less-eq-nres **where**

- $\leq FAILi \longleftrightarrow True$ |
 (*RES* a) \leq (*RES* b) $\longleftrightarrow a \subseteq b$ |
FAILi \leq (*RES* -) $\longleftrightarrow False$

fun less-nres **where**

FAILi < - $\longleftrightarrow False$ |
 (*RES* -) < *FAILi* $\longleftrightarrow True$ |
 (*RES* a) < (*RES* b) $\longleftrightarrow a \subset b$

fun sup-nres **where**

sup - *FAILi* = *FAILi* |
 sup *FAILi* - = *FAILi* |
 sup (*RES* a) (*RES* b) = *RES* (a \cup b)

fun inf-nres **where**

inf x *FAILi* = x |
 inf *FAILi* x = x |
 inf (*RES* a) (*RES* b) = *RES* (a \cap b)

definition Sup X \equiv if *FAILi* \in X then *FAILi* else *RES* ($\bigcup \{x . RES\ x \in X\}$)

definition Inf X \equiv if $\exists x . RES\ x \in X$ then *RES* ($\bigcap \{x . RES\ x \in X\}$) else *FAILi*

definition bot $\equiv RES\ \{\}$

definition $top \equiv FAILi$

instance

$\langle proof \rangle$

end

abbreviation $FAIL \equiv top::'a\ nres$

abbreviation $SUCCEED \equiv bot::'a\ nres$

abbreviation $SPEC\ \Phi \equiv RES\ (Collect\ \Phi)$

definition $RETURN\ x \equiv RES\ \{x\}$

We try to hide the original $FAILi$ -element as well as possible.

lemma $nres-cases[case-names\ FAIL\ RES, cases\ type]:$

obtains $M=FAIL \mid X\ \mathbf{where}\ M=RES\ X$

$\langle proof \rangle$

lemma $nres-simp-internals:$

$RES\ \{\} = SUCCEED$

$FAILi = FAIL$

$\langle proof \rangle$

lemma $nres-inequalities[simp]:$

$FAIL \neq RES\ X$

$FAIL \neq SUCCEED$

$FAIL \neq RETURN\ x$

$SUCCEED \neq FAIL$

$SUCCEED \neq RETURN\ x$

$RES\ X \neq FAIL$

$RETURN\ x \neq FAIL$

$RETURN\ x \neq SUCCEED$

$\langle proof \rangle$

lemma $nres-more-simps[simp]:$

$SUCCEED = RES\ X \longleftrightarrow X = \{\}$

$RES\ X = SUCCEED \longleftrightarrow X = \{\}$

$RES\ X = RETURN\ x \longleftrightarrow X = \{x\}$

$RES\ X = RES\ Y \longleftrightarrow X = Y$

$RETURN\ x = RES\ X \longleftrightarrow \{x\} = X$

$RETURN\ x = RETURN\ y \longleftrightarrow x = y$

$\langle proof \rangle$

lemma $nres-order-simps[simp]:$

$\bigwedge M. SUCCEED \leq M$

$\bigwedge M. M \leq SUCCEED \longleftrightarrow M = SUCCEED$

$\bigwedge M. M \leq FAIL$

$\bigwedge M. FAIL \leq M \longleftrightarrow M = FAIL$

$\bigwedge X\ Y. RES\ X \leq RES\ Y \longleftrightarrow X \leq Y$

$\bigwedge X. Sup\ X = FAIL \longleftrightarrow FAIL \in X$

$$\begin{aligned}
&\bigwedge X f. \text{Sup } (f \text{ ' } X) = \text{FAIL} \longleftrightarrow \text{FAIL} \in f \text{ ' } X \\
&\bigwedge X. \text{FAIL} = \text{Sup } X \longleftrightarrow \text{FAIL} \in X \\
&\bigwedge X f. \text{FAIL} = \text{Sup } (f \text{ ' } X) \longleftrightarrow \text{FAIL} \in f \text{ ' } X \\
&\bigwedge X. \text{FAIL} \in X \implies \text{Sup } X = \text{FAIL} \\
&\bigwedge X. \text{FAIL} \in f \text{ ' } X \implies \text{Sup } (f \text{ ' } X) = \text{FAIL} \\
&\bigwedge A. \text{Sup } (\text{RES ' } A) = \text{RES } (\text{Sup } A) \\
&\bigwedge A. \text{Sup } (\text{RES ' } A) = \text{RES } (\text{Sup } A) \\
&\bigwedge A. A \neq \{\} \implies \text{Inf } (\text{RES ' } A) = \text{RES } (\text{Inf } A) \\
&\bigwedge A. A \neq \{\} \implies \text{Inf } (\text{RES ' } A) = \text{RES } (\text{Inf } A) \\
&\text{Inf } \{\} = \text{FAIL} \\
&\text{Inf } \text{UNIV} = \text{SUCCEED} \\
&\text{Sup } \{\} = \text{SUCCEED} \\
&\text{Sup } \text{UNIV} = \text{FAIL} \\
&\bigwedge x y. \text{RETURN } x \leq \text{RETURN } y \longleftrightarrow x=y \\
&\bigwedge x Y. \text{RETURN } x \leq \text{RES } Y \longleftrightarrow x \in Y \\
&\bigwedge X y. \text{RES } X \leq \text{RETURN } y \longleftrightarrow X \subseteq \{y\} \\
&\langle \text{proof} \rangle
\end{aligned}$$

lemma *Sup-eq-RESE*:

assumes $\text{Sup } A = \text{RES } B$

obtains C **where** $A = \text{RES ' } C$ **and** $B = \text{Sup } C$

$\langle \text{proof} \rangle$

declare *nres-simp-internals*[*simp*]

Pointwise Reasoning

$\langle \text{ML} \rangle$

definition *nofail* $S \equiv S \neq \text{FAIL}$

definition *inres* $S x \equiv \text{RETURN } x \leq S$

lemma *nofail-simps*[*simp*, *refine-pw-simps*]:

nofail $\text{FAIL} \longleftrightarrow \text{False}$

nofail $(\text{RES } X) \longleftrightarrow \text{True}$

nofail $(\text{RETURN } x) \longleftrightarrow \text{True}$

nofail $\text{SUCCEED} \longleftrightarrow \text{True}$

$\langle \text{proof} \rangle$

lemma *inres-simps*[*simp*, *refine-pw-simps*]:

inres $\text{FAIL} = (\lambda \cdot. \text{True})$

inres $(\text{RES } X) = (\lambda x. x \in X)$

inres $(\text{RETURN } x) = (\lambda y. x=y)$

inres $\text{SUCCEED} = (\lambda \cdot. \text{False})$

$\langle \text{proof} \rangle$

lemma *not-nofail-iff*:

$\neg \text{nofail } S \longleftrightarrow S = \text{FAIL}$ $\langle \text{proof} \rangle$

lemma *not-nofail-inres*[*simp, refine-pw-simps*]:

$$\neg \text{nofail } S \implies \text{inres } S \ x$$

<proof>

lemma *intro-nofail*[*refine-pw-simps*]:

$$S \neq \text{FAIL} \longleftrightarrow \text{nofail } S$$

$$\text{FAIL} \neq S \longleftrightarrow \text{nofail } S$$

<proof>

The following two lemmas will introduce pointwise reasoning for orderings and equalities.

lemma *pw-le-iff*:

$$S \leq S' \longleftrightarrow (\text{nofail } S' \longrightarrow (\text{nofail } S \wedge (\forall x. \text{inres } S \ x \longrightarrow \text{inres } S' \ x)))$$

<proof>

lemma *pw-eq-iff*:

$$S = S' \longleftrightarrow (\text{nofail } S = \text{nofail } S' \wedge (\forall x. \text{inres } S \ x \longleftrightarrow \text{inres } S' \ x))$$

<proof>

lemma *pw-flat-le-iff*: *flat-le* $S \ S' \longleftrightarrow$

$$(\exists x. \text{inres } S \ x) \longrightarrow (\text{nofail } S \longleftrightarrow \text{nofail } S') \wedge (\forall x. \text{inres } S \ x \longleftrightarrow \text{inres } S' \ x)$$

<proof>

lemma *pw-flat-ge-iff*: *flat-ge* $S \ S' \longleftrightarrow$

$$(\text{nofail } S) \longrightarrow \text{nofail } S' \wedge (\forall x. \text{inres } S \ x \longleftrightarrow \text{inres } S' \ x)$$

<proof>

lemmas *pw-ords-iff* = *pw-le-iff* *pw-flat-le-iff* *pw-flat-ge-iff*

lemma *pw-leI*:

$$(\text{nofail } S' \longrightarrow (\text{nofail } S \wedge (\forall x. \text{inres } S \ x \longrightarrow \text{inres } S' \ x))) \implies S \leq S'$$

<proof>

lemma *pw-leI'*:

assumes *nofail* $S' \implies \text{nofail } S$

assumes $\bigwedge x. [\text{nofail } S'; \text{inres } S \ x] \implies \text{inres } S' \ x$

shows $S \leq S'$

<proof>

lemma *pw-eqI*:

assumes *nofail* $S = \text{nofail } S'$

assumes $\bigwedge x. \text{inres } S \ x \longleftrightarrow \text{inres } S' \ x$

shows $S = S'$

<proof>

lemma *pwD1*:

assumes $S \leq S' \quad \text{nofail } S'$

shows *nofail* S

<proof>

lemma *pwD2*:
assumes $S \leq S'$ *inres* S x
shows *inres* S' x
<proof>

lemmas *pwD* = *pwD1* *pwD2*

When proving refinement, we may assume that the refined program does not fail.

lemma *le-nofailI*: $\llbracket \text{nofail } M' \implies M \leq M' \rrbracket \implies M \leq M'$
<proof>

The following lemmas push pointwise reasoning over operators, thus converting an expression over lattice operators into a logical formula.

lemma *pw-sup-nofail*[*refine-pw-simps*]:
nofail (*sup* a b) \longleftrightarrow *nofail* $a \wedge$ *nofail* b
<proof>

lemma *pw-sup-inres*[*refine-pw-simps*]:
inres (*sup* a b) $x \longleftrightarrow$ *inres* a $x \vee$ *inres* b x
<proof>

lemma *pw-Sup-inres*[*refine-pw-simps*]: *inres* (*Sup* X) $r \longleftrightarrow$ $(\exists M \in X. \text{inres } M \ r)$
<proof>

lemma *pw-SUP-inres* [*refine-pw-simps*]: *inres* (*Sup* (f ' X)) $r \longleftrightarrow$ $(\exists M \in X. \text{inres } (f \ M) \ r)$
<proof>

lemma *pw-Sup-nofail*[*refine-pw-simps*]: *nofail* (*Sup* X) \longleftrightarrow $(\forall x \in X. \text{nofail } x)$
<proof>

lemma *pw-SUP-nofail* [*refine-pw-simps*]: *nofail* (*Sup* (f ' X)) \longleftrightarrow $(\forall x \in X. \text{nofail } (f \ x))$
<proof>

lemma *pw-inf-nofail*[*refine-pw-simps*]:
nofail (*inf* a b) \longleftrightarrow *nofail* $a \vee$ *nofail* b
<proof>

lemma *pw-inf-inres*[*refine-pw-simps*]:
inres (*inf* a b) $x \longleftrightarrow$ *inres* a $x \wedge$ *inres* b x
<proof>

lemma *pw-Inf-nofail*[*refine-pw-simps*]: *nofail* (*Inf* C) \longleftrightarrow $(\exists x \in C. \text{nofail } x)$
<proof>

lemma *pw-INF-nofail* [*refine-pw-simps*]: $\text{nofail } (\text{Inf } (f \text{ ' } C)) \longleftrightarrow (\exists x \in C. \text{nofail } (f \ x))$
 ⟨*proof*⟩

lemma *pw-Inf-inres* [*refine-pw-simps*]: $\text{inres } (\text{Inf } C) \ r \longleftrightarrow (\forall M \in C. \text{inres } M \ r)$
 ⟨*proof*⟩

lemma *pw-INF-inres* [*refine-pw-simps*]: $\text{inres } (\text{Inf } (f \text{ ' } C)) \ r \longleftrightarrow (\forall M \in C. \text{inres } (f \ M) \ r)$
 ⟨*proof*⟩

lemma *nofail-RES-conv*: $\text{nofail } m \longleftrightarrow (\exists M. m = \text{RES } M)$ ⟨*proof*⟩

primrec *the-RES* **where** $\text{the-RES } (\text{RES } X) = X$

lemma *the-RES-inv* [*simp*]: $\text{nofail } m \implies \text{RES } (\text{the-RES } m) = m$
 ⟨*proof*⟩

definition [*refine-pw-simps*]: $\text{nf-inres } m \ x \equiv \text{nofail } m \ \wedge \ \text{inres } m \ x$

lemma *nf-inres-RES* [*simp*]: $\text{nf-inres } (\text{RES } X) \ x \longleftrightarrow x \in X$
 ⟨*proof*⟩

lemma *nf-inres-SPEC* [*simp*]: $\text{nf-inres } (\text{SPEC } \Phi) \ x \longleftrightarrow \Phi \ x$
 ⟨*proof*⟩

lemma *nofail-antimono-fun*: $f \leq g \implies (\text{nofail } (g \ x) \longrightarrow \text{nofail } (f \ x))$
 ⟨*proof*⟩

Monad Operators

definition *bind* **where** $\text{bind } M \ f \equiv \text{case } M \ \text{of}$
 $\text{FAIL}i \Rightarrow \text{FAIL} \mid$
 $\text{RES } X \Rightarrow \text{Sup } (f \text{ ' } X)$

lemma *bind-FAIL* [*simp*]: $\text{bind } \text{FAIL} \ f = \text{FAIL}$
 ⟨*proof*⟩

lemma *bind-SUCCEED* [*simp*]: $\text{bind } \text{SUCCEED} \ f = \text{SUCCEED}$
 ⟨*proof*⟩

lemma *bind-RES*: $\text{bind } (\text{RES } X) \ f = \text{Sup } (f \text{ ' } X)$ ⟨*proof*⟩

adhoc-overloading

Monad-Syntax.bind Refine-Basic.bind

lemma *pw-bind-nofail* [*refine-pw-simps*]:
 $\text{nofail } (\text{bind } M \ f) \longleftrightarrow (\text{nofail } M \ \wedge \ (\forall x. \text{inres } M \ x \longrightarrow \text{nofail } (f \ x)))$
 ⟨*proof*⟩

lemma *pw-bind-inres*[*refine-pw-simps*]:

$inres (bind M f) = (\lambda x. nofail M \longrightarrow (\exists y. (inres M y \wedge inres (f y) x)))$
 ⟨*proof*⟩

lemma *pw-bind-le-iff*:

$bind M f \leq S \iff (nofail S \longrightarrow nofail M) \wedge$
 $(\forall x. nofail M \wedge inres M x \longrightarrow f x \leq S)$
 ⟨*proof*⟩

lemma *pw-bind-leI*: \llbracket

$nofail S \implies nofail M; \bigwedge x. \llbracket nofail M; inres M x \rrbracket \implies f x \leq S \rrbracket$
 $\implies bind M f \leq S$
 ⟨*proof*⟩

lemma *nres-monad1*[*simp*]: $bind (RETURN x) f = f x$

⟨*proof*⟩

lemma *nres-monad2*[*simp*]: $bind M RETURN = M$

⟨*proof*⟩

lemma *nres-monad3*[*simp*]: $bind (bind M f) g = bind M (\lambda x. bind (f x) g)$

⟨*proof*⟩

lemmas *nres-monad-laws* = *nres-monad1 nres-monad2 nres-monad3*

lemma *bind-cong*:

assumes $m = m'$

assumes $\bigwedge x. RETURN x \leq m' \implies f x = f' x$

shows $bind m f = bind m' f'$

⟨*proof*⟩

lemma *bind-mono*[*refine-mono*]:

$\llbracket M \leq M'; \bigwedge x. RETURN x \leq M \implies f x \leq f' x \rrbracket \implies bind M f \leq bind M' f'$

$\llbracket flat-ge M M'; \bigwedge x. flat-ge (f x) (f' x) \rrbracket \implies flat-ge (bind M f) (bind M' f')$

⟨*proof*⟩

lemma *bind-mono1*[*simp, intro!*]: $mono (\lambda M. bind M f)$

⟨*proof*⟩

lemma *bind-mono1'*[*simp, intro!*]: $mono bind$

⟨*proof*⟩

lemma *bind-mono2'*[*simp, intro!*]: $mono (bind M)$

⟨*proof*⟩

lemma *bind-distrib-sup1*: $\text{bind } (\text{sup } M N) f = \text{sup } (\text{bind } M f) (\text{bind } N f)$
 ⟨proof⟩

lemma *bind-distrib-sup2*: $\text{bind } m (\lambda x. \text{sup } (f x) (g x)) = \text{sup } (\text{bind } m f) (\text{bind } m g)$
 ⟨proof⟩

lemma *bind-distrib-Sup1*: $\text{bind } (\text{Sup } M) f = (\text{SUP } m \in M. \text{bind } m f)$
 ⟨proof⟩

lemma *bind-distrib-Sup2*: $F \neq \{\} \implies \text{bind } m (\text{Sup } F) = (\text{SUP } f \in F. \text{bind } m f)$
 ⟨proof⟩

lemma *RES-Sup-RETURN*: $\text{Sup } (\text{RETURN } X) = \text{RES } X$
 ⟨proof⟩

2.6.2 VCG Setup

lemma *SPEC-cons-rule*:
 assumes $m \leq \text{SPEC } \Phi$
 assumes $\bigwedge x. \Phi x \implies \Psi x$
 shows $m \leq \text{SPEC } \Psi$
 ⟨proof⟩

lemmas *SPEC-trans = order-trans*[**where** $z = \text{SPEC Postcond for Postcond, zero-var-indices}$]
 ⟨ML⟩

declare *SPEC-cons-rule*[*refine-vcg-cons*]

2.6.3 Data Refinement

In this section we establish a notion of pointwise data refinement, by lifting a relation R between concrete and abstract values to our result lattice.

Given a relation R , we define a *concretization function* $\Downarrow R$ that takes an abstract result, and returns a concrete result. The concrete result contains all values that are mapped by R to a value in the abstract result.

Note that our concretization function forms no Galois connection, i.e., in general there is no α such that $m \leq \Downarrow R m'$ is equivalent to $\alpha m \leq m'$. However, we get a Galois connection for the special case of single-valued relations.

Regarding data refinement as Galois connections is inspired by [16], that

also uses the adjuncts of a Galois connection to express data refinement by program refinement.

definition *conc-fun* (\Downarrow) **where**

$$\text{conc-fun } R \ m \equiv \text{case } m \text{ of } FAILi \Rightarrow FAIL \mid RES \ X \Rightarrow RES \ (R^{-1} \text{``}X)$$

definition *abs-fun* (\Uparrow) **where**

$$\begin{aligned} \text{abs-fun } R \ m &\equiv \text{case } m \text{ of } FAILi \Rightarrow FAIL \\ &\mid RES \ X \Rightarrow \text{if } X \subseteq \text{Domain } R \text{ then } RES \ (R \text{``}X) \text{ else } FAIL \end{aligned}$$

lemma

$$\begin{aligned} \text{conc-fun-FAIL[simp]: } &\Downarrow R \ FAIL = FAIL \text{ and} \\ \text{conc-fun-RES: } &\Downarrow R \ (RES \ X) = RES \ (R^{-1} \text{``}X) \\ \langle \text{proof} \rangle & \end{aligned}$$

lemma *abs-fun-simps[simp]*:

$$\begin{aligned} \Uparrow R \ FAIL &= FAIL \\ X \subseteq \text{Domain } R &\Longrightarrow \Uparrow R \ (RES \ X) = RES \ (R \text{``}X) \\ \neg(X \subseteq \text{Domain } R) &\Longrightarrow \Uparrow R \ (RES \ X) = FAIL \\ \langle \text{proof} \rangle & \end{aligned}$$

context *fixes* R *assumes* SV : *single-valued* R **begin**

lemma *conc-abs-swap*: $m' \leq \Downarrow R \ m \longleftrightarrow \Uparrow R \ m' \leq m$
 $\langle \text{proof} \rangle$

lemma *ac-galois*: *galois-connection* ($\Uparrow R$) ($\Downarrow R$)
 $\langle \text{proof} \rangle$

end

lemma *pw-abs-nofail[refine-pw-simps]*:

$$\text{nofail } (\Uparrow R \ M) \longleftrightarrow (\text{nofail } M \wedge (\forall x. \text{inres } M \ x \longrightarrow x \in \text{Domain } R))$$

$\langle \text{proof} \rangle$

lemma *pw-abs-inres[refine-pw-simps]*:

$$\text{inres } (\Uparrow R \ M) \ a \longleftrightarrow (\text{nofail } (\Uparrow R \ M) \longrightarrow (\exists c. \text{inres } M \ c \wedge (c, a) \in R))$$

$\langle \text{proof} \rangle$

lemma *pw-conc-nofail[refine-pw-simps]*:

$$\text{nofail } (\Downarrow R \ S) = \text{nofail } S$$

$\langle \text{proof} \rangle$

lemma *pw-conc-inres[refine-pw-simps]*:

$$\begin{aligned} \text{inres } (\Downarrow R \ S') &= (\lambda s. \text{nofail } S' \\ &\longrightarrow (\exists s'. (s, s') \in R \wedge \text{inres } S' \ s')) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *abs-fun-strict[simp]*:

$$\Uparrow R \ SUCCEED = SUCCEED$$

$\langle \text{proof} \rangle$

lemma *conc-fun-strict*[simp]:
 $\Downarrow R \text{ SUCCEED} = \text{SUCCEED}$
 ⟨proof⟩

lemma *conc-fun-mono*[simp, intro!]: *mono* ($\Downarrow R$)
 ⟨proof⟩

lemma *abs-fun-mono*[simp, intro!]: *mono* ($\Uparrow R$)
 ⟨proof⟩

lemma *conc-fun-R-mono*:
assumes $R \subseteq R'$
shows $\Downarrow R M \leq \Downarrow R' M$
 ⟨proof⟩

lemma *conc-fun-chain*: $\Downarrow R (\Downarrow S M) = \Downarrow (R \circ S) M$
 ⟨proof⟩

lemma *conc-Id*[simp]: $\Downarrow \text{Id} = \text{id}$
 ⟨proof⟩

lemma *abs-Id*[simp]: $\Uparrow \text{Id} = \text{id}$
 ⟨proof⟩

lemma *conc-fun-fail-iff*[simp]:
 $\Downarrow R S = \text{FAIL} \iff S = \text{FAIL}$
 $\text{FAIL} = \Downarrow R S \iff S = \text{FAIL}$
 ⟨proof⟩

lemma *conc-trans*[trans]:
assumes $A: C \leq \Downarrow R B$ **and** $B: B \leq \Downarrow R' A$
shows $C \leq \Downarrow R (\Downarrow R' A)$
 ⟨proof⟩

lemma *abs-trans*[trans]:
assumes $A: \Uparrow R C \leq B$ **and** $B: \Uparrow R' B \leq A$
shows $\Uparrow R' (\Uparrow R C) \leq A$
 ⟨proof⟩

Transitivity Reasoner Setup

WARNING: The order of the single statements is important here!

lemma *conc-trans-additional*[trans]:
 $\bigwedge A B C. A \leq \Downarrow R B \implies B \leq C \implies A \leq \Downarrow R C$
 $\bigwedge A B C. A \leq \Downarrow \text{Id} B \implies B \leq \Downarrow R C \implies A \leq \Downarrow R C$
 $\bigwedge A B C. A \leq \Downarrow R B \implies B \leq \Downarrow \text{Id} C \implies A \leq \Downarrow R C$
 $\bigwedge A B C. A \leq \Downarrow \text{Id} B \implies B \leq \Downarrow \text{Id} C \implies A \leq C$

$$\begin{aligned} \bigwedge A B C. A \leq \Downarrow Id B &\implies B \leq C \implies A \leq C \\ \bigwedge A B C. A \leq B &\implies B \leq \Downarrow Id C \implies A \leq C \\ \langle proof \rangle \end{aligned}$$

WARNING: The order of the single statements is important here!

lemma *abs-trans-additional*[*trans*]:

$$\begin{aligned} \bigwedge A B C. \llbracket A \leq B; \uparrow R B \leq C \rrbracket &\implies \uparrow R A \leq C \\ \bigwedge A B C. \llbracket \uparrow Id A \leq B; \uparrow R B \leq C \rrbracket &\implies \uparrow R A \leq C \\ \bigwedge A B C. \llbracket \uparrow R A \leq B; \uparrow Id B \leq C \rrbracket &\implies \uparrow R A \leq C \end{aligned}$$

$$\begin{aligned} \bigwedge A B C. \llbracket \uparrow Id A \leq B; \uparrow Id B \leq C \rrbracket &\implies A \leq C \\ \bigwedge A B C. \llbracket \uparrow Id A \leq B; B \leq C \rrbracket &\implies A \leq C \\ \bigwedge A B C. \llbracket A \leq B; \uparrow Id B \leq C \rrbracket &\implies A \leq C \end{aligned}$$

<proof>

2.6.4 Derived Program Constructs

In this section, we introduce some programming constructs that are derived from the basic monad and ordering operations of our nondeterminism monad.

ASSUME and ASSERT

definition *ASSERT where* $ASSERT \equiv iASSERT RETURN$

definition *ASSUME where* $ASSUME \equiv iASSUME RETURN$

interpretation *assert?: generic-Assert bind RETURN ASSERT ASSUME*

<proof>

Order matters!

lemmas [*refine-vcg*] = *ASSERT-leI*

lemmas [*refine-vcg*] = *le-ASSUMEI*

lemmas [*refine-vcg*] = *le-ASSERTI*

lemmas [*refine-vcg*] = *ASSUME-leI*

lemma *pw-ASSERT*[*refine-pw-simps*]:

nofail (*ASSERT* Φ) $\longleftrightarrow \Phi$

inres (*ASSERT* Φ) x

<proof>

lemma *pw-ASSUME*[*refine-pw-simps*]:

nofail (*ASSUME* Φ)

inres (*ASSUME* Φ) $x \longleftrightarrow \Phi$

<proof>

Recursion

lemma *pw-REC-nofail*:

shows $\text{nofail } (\text{REC } B \ x) \longleftrightarrow \text{trimono } B \wedge$
 $(\exists F. (\forall x.$
 $\quad \text{nofail } (F \ x) \longrightarrow \text{nofail } (B \ F \ x)$
 $\quad \wedge (\forall x'. \text{inres } (B \ F \ x) \ x' \longrightarrow \text{inres } (F \ x) \ x')$
 $\quad) \wedge \text{nofail } (F \ x))$
 $\langle \text{proof} \rangle$

lemma *pw-REC-inres*:

inres $(\text{REC } B \ x) \ x' = (\text{trimono } B \longrightarrow$
 $(\forall F. (\forall x''.$
 $\quad \text{nofail } (F \ x'') \longrightarrow \text{nofail } (B \ F \ x'')$
 $\quad \wedge (\forall x. \text{inres } (B \ F \ x'') \ x \longrightarrow \text{inres } (F \ x'') \ x))$
 $\quad \longrightarrow \text{inres } (F \ x) \ x')$
 $\langle \text{proof} \rangle$

lemmas $\text{pw-REC} = \text{pw-REC-inres } \text{pw-REC-nofail}$

lemma *pw-RECT-nofail*:

shows $\text{nofail } (\text{RECT } B \ x) \longleftrightarrow \text{trimono } B \wedge$
 $(\forall F. (\forall y. \text{nofail } (B \ F \ y) \longrightarrow$
 $\quad \text{nofail } (F \ y) \wedge (\forall x. \text{inres } (F \ y) \ x \longrightarrow \text{inres } (B \ F \ y) \ x)) \longrightarrow$
 $\quad \text{nofail } (F \ x))$
 $\langle \text{proof} \rangle$

lemma *pw-RECT-inres*:

shows $\text{inres } (\text{RECT } B \ x) \ x' = (\text{trimono } B \longrightarrow$
 $(\exists M. (\forall y. \text{nofail } (B \ M \ y) \longrightarrow$
 $\quad \text{nofail } (M \ y) \wedge (\forall x. \text{inres } (M \ y) \ x \longrightarrow \text{inres } (B \ M \ y) \ x)) \wedge$
 $\quad \text{inres } (M \ x) \ x')$
 $\langle \text{proof} \rangle$

lemmas $\text{pw-RECT} = \text{pw-RECT-inres } \text{pw-RECT-nofail}$

2.6.5 Proof Rules

Proving Correctness

In this section, we establish Hoare-like rules to prove that a program meets its specification.

lemma *le-SPEC-UNIV-rule* [*refine-vcg*]:

$m \leq \text{SPEC } (\lambda-. \text{True}) \implies m \leq \text{RES UNIV } \langle \text{proof} \rangle$

lemma *RETURN-rule* [*refine-vcg*]: $\Phi \ x \implies \text{RETURN } x \leq \text{SPEC } \Phi$

$\langle \text{proof} \rangle$

lemma *RES-rule* [*refine-vcg*]: $[\bigwedge x. x \in S \implies \Phi \ x] \implies \text{RES } S \leq \text{SPEC } \Phi$

$\langle proof \rangle$
lemma *SUCCEED-rule*[*refine-vcg*]: $SUCCEED \leq SPEC \Phi \langle proof \rangle$
lemma *FAIL-rule*: $False \implies FAIL \leq SPEC \Phi \langle proof \rangle$
lemma *SPEC-rule*[*refine-vcg*]: $\llbracket \bigwedge x. \Phi x \implies \Phi' x \rrbracket \implies SPEC \Phi \leq SPEC \Phi' \langle proof \rangle$

lemma *RETURN-to-SPEC-rule*[*refine-vcg*]: $m \leq SPEC ((=) v) \implies m \leq RETURN v \langle proof \rangle$

lemma *Sup-img-rule-complete*:
 $(\forall x. x \in S \longrightarrow f x \leq SPEC \Phi) \longleftrightarrow Sup (f'S) \leq SPEC \Phi \langle proof \rangle$

lemma *SUP-img-rule-complete*:
 $(\forall x. x \in S \longrightarrow f x \leq SPEC \Phi) \longleftrightarrow Sup (f ' S) \leq SPEC \Phi \langle proof \rangle$

lemma *Sup-img-rule*[*refine-vcg*]:
 $\llbracket \bigwedge x. x \in S \implies f x \leq SPEC \Phi \rrbracket \implies Sup(f'S) \leq SPEC \Phi \langle proof \rangle$

This lemma is just to demonstrate that our rule is complete.

lemma *bind-rule-complete*: $bind M f \leq SPEC \Phi \longleftrightarrow M \leq SPEC (\lambda x. f x \leq SPEC \Phi) \langle proof \rangle$

lemma *bind-rule*[*refine-vcg*]:
 $\llbracket M \leq SPEC (\lambda x. f x \leq SPEC \Phi) \rrbracket \implies bind M (\lambda x. f x) \leq SPEC \Phi$
— Note: η -expanded version helps Isabelle's unification to keep meaningful variable names from the program
 $\langle proof \rangle$

lemma *ASSUME-rule*[*refine-vcg*]: $\llbracket \Phi \implies \Psi () \rrbracket \implies ASSUME \Phi \leq SPEC \Psi \langle proof \rangle$

lemma *ASSERT-rule*[*refine-vcg*]: $\llbracket \Phi; \Phi \implies \Psi () \rrbracket \implies ASSERT \Phi \leq SPEC \Psi \langle proof \rangle$

lemma *prod-rule*[*refine-vcg*]:
 $\llbracket \bigwedge a b. p=(a,b) \implies S a b \leq SPEC \Phi \rrbracket \implies case-prod S p \leq SPEC \Phi \langle proof \rangle$

lemma *prod2-rule*[*refine-vcg*]:
assumes $\bigwedge a b c d. \llbracket ab=(a,b); cd=(c,d) \rrbracket \implies f a b c d \leq SPEC \Phi$
shows $(\lambda(a,b) (c,d). f a b c d) ab cd \leq SPEC \Phi \langle proof \rangle$

lemma *if-rule*[*refine-vcg*]:

$\llbracket b \implies S1 \leq \text{SPEC } \Phi; \neg b \implies S2 \leq \text{SPEC } \Phi \rrbracket$
 $\implies (\text{if } b \text{ then } S1 \text{ else } S2) \leq \text{SPEC } \Phi$
 <proof>

lemma *option-rule*[refine-vcg]:

$\llbracket v=\text{None} \implies S1 \leq \text{SPEC } \Phi; \bigwedge x. v=\text{Some } x \implies f2\ x \leq \text{SPEC } \Phi \rrbracket$
 $\implies \text{case-option } S1\ f2\ v \leq \text{SPEC } \Phi$
 <proof>

lemma *Let-rule*[refine-vcg]:

$f\ x \leq \text{SPEC } \Phi \implies \text{Let } x\ f \leq \text{SPEC } \Phi$ <proof>

lemma *Let-rule'*:

assumes $\bigwedge x. x=v \implies f\ x \leq \text{SPEC } \Phi$
shows $\text{Let } v\ (\lambda x. f\ x) \leq \text{SPEC } \Phi$
 <proof>

lemma *REC-le-rule*:

assumes M : trimono body
assumes $I0$: $(x, x') \in R$
assumes IS : $\bigwedge f\ x\ x'. \llbracket \bigwedge x\ x'. (x, x') \in R \implies f\ x \leq M\ x'; (x, x') \in R \rrbracket$
 $\implies \text{body } f\ x \leq M\ x'$
shows $\text{REC body } x \leq M\ x'$
 <proof>

Proving Monotonicity

lemma *nr-mono-bind*:

assumes MA : mono A **and** MB : $\bigwedge s. \text{mono } (B\ s)$
shows mono $(\lambda F\ s. \text{bind } (A\ F\ s) (\lambda s'. B\ s\ F\ s'))$
 <proof>

lemma *nr-mono-bind'*: mono $(\lambda F\ s. \text{bind } (f\ s)\ F)$

<proof>

lemmas $\text{nr-mono} = \text{nr-mono-bind nr-mono-bind' mono-const mono-if mono-id}$

Proving Refinement

In this subsection, we establish rules to prove refinement between structurally similar programs. All rules are formulated including a possible data refinement via a refinement relation. If this is not required, the refinement relation can be chosen to be the identity relation.

If we have two identical programs, this rule solves the refinement goal immediately, using the identity refinement relation.

lemma *Id-refine*[*refine0*]: $S \leq \Downarrow Id S$ *<proof>*

lemma *RES-refine*:

$\llbracket \bigwedge s. s \in S \implies \exists s' \in S'. (s, s') \in R \rrbracket \implies RES S \leq \Downarrow R (RES S')$
<proof>

lemma *SPEC-refine*:

assumes $S \leq SPEC (\lambda x. \exists x'. (x, x') \in R \wedge \Phi x')$
shows $S \leq \Downarrow R (SPEC \Phi)$
<proof>

lemma *Id-SPEC-refine*[*refine*]:

$S \leq SPEC \Phi \implies S \leq \Downarrow Id (SPEC \Phi)$ *<proof>*

lemma *RETURN-refine*[*refine*]:

assumes $(x, x') \in R$
shows $RETURN x \leq \Downarrow R (RETURN x')$
<proof>

lemma *RETURN-SPEC-refine*:

assumes $\exists x'. (x, x') \in R \wedge \Phi x'$
shows $RETURN x \leq \Downarrow R (SPEC \Phi)$
<proof>

lemma *FAIL-refine*[*refine*]: $X \leq \Downarrow R FAIL$ *<proof>*

lemma *SUCCEED-refine*[*refine*]: $SUCCEED \leq \Downarrow R X'$ *<proof>*

lemma *sup-refine*[*refine*]:

assumes $a_i \leq \Downarrow R a$
assumes $b_i \leq \Downarrow R b$
shows $\sup a_i b_i \leq \Downarrow R (\sup a b)$
<proof>

The next two rules are incomplete, but a good approximation for refining structurally similar programs.

lemma *bind-refine'*:

fixes $R' :: ('a \times 'b) \text{ set}$ **and** $R :: ('c \times 'd) \text{ set}$
assumes $R1: M \leq \Downarrow R' M'$
assumes $R2: \bigwedge x x'. \llbracket (x, x') \in R'; \text{inres } M x; \text{inres } M' x';$
 $\text{nofail } M; \text{nofail } M'$
 $\rrbracket \implies f x \leq \Downarrow R (f' x')$
shows $\text{bind } M (\lambda x. f x) \leq \Downarrow R (\text{bind } M' (\lambda x'. f' x'))$
<proof>

lemma *bind-refine*[*refine*]:

fixes $R' :: ('a \times 'b) \text{ set}$ **and** $R :: ('c \times 'd) \text{ set}$

assumes $R1: M \leq \Downarrow R' M'$
assumes $R2: \bigwedge x x'. \llbracket (x, x') \in R' \rrbracket$
 $\implies f x \leq \Downarrow R (f' x')$
shows $\text{bind } M (\lambda x. f x) \leq \Downarrow R (\text{bind } M' (\lambda x'. f' x'))$
 $\langle \text{proof} \rangle$

lemma *bind-refine-abs'*:
fixes $R' :: ('a \times 'b)$ set **and** $R :: ('c \times 'd)$ set
assumes $R1: M \leq \Downarrow R' M'$
assumes $R2: \bigwedge x x'. \llbracket (x, x') \in R'; \text{nf-inres } M' x' \rrbracket$
 $\implies f x \leq \Downarrow R (f' x')$
shows $\text{bind } M (\lambda x. f x) \leq \Downarrow R (\text{bind } M' (\lambda x'. f' x'))$
 $\langle \text{proof} \rangle$

Special cases for refinement of binding to *RES* statements

lemma *bind-refine-RES*:
 $\llbracket \text{RES } X \leq \Downarrow R' M'; \bigwedge x x'. \llbracket (x, x') \in R'; x \in X \rrbracket \rrbracket \implies f x \leq \Downarrow R (f' x')$
 $\implies \text{RES } X \ggg (\lambda x. f x) \leq \Downarrow R (M' \ggg (\lambda x'. f' x'))$

 $\llbracket M \leq \Downarrow R' (\text{RES } X'); \bigwedge x x'. \llbracket (x, x') \in R'; x' \in X' \rrbracket \rrbracket \implies f x \leq \Downarrow R (f' x')$
 $\implies M \ggg (\lambda x. f x) \leq \Downarrow R (\text{RES } X' \ggg (\lambda x'. f' x'))$

 $\llbracket \text{RES } X \leq \Downarrow R' (\text{RES } X'); \bigwedge x x'. \llbracket (x, x') \in R'; x \in X; x' \in X' \rrbracket \rrbracket \implies f x \leq \Downarrow R (f' x')$
 $\implies \text{RES } X \ggg (\lambda x. f x) \leq \Downarrow R (\text{RES } X' \ggg (\lambda x'. f' x'))$
 $\langle \text{proof} \rangle$

declare *bind-refine-RES(1,2)[refine]*
declare *bind-refine-RES(3)[refine]*

lemma *ASSERT-refine[refine]*:
 $\llbracket \Phi' \implies \Phi \rrbracket \implies \text{ASSERT } \Phi \leq \Downarrow \text{Id } (\text{ASSERT } \Phi')$
 $\langle \text{proof} \rangle$

lemma *ASSUME-refine[refine]*:
 $\llbracket \Phi \implies \Phi' \rrbracket \implies \text{ASSUME } \Phi \leq \Downarrow \text{Id } (\text{ASSUME } \Phi')$
 $\langle \text{proof} \rangle$

Assertions and assumptions are treated specially in bindings

lemma *ASSERT-refine-right*:
assumes $\Phi \implies S \leq \Downarrow R S'$
shows $S \leq \Downarrow R (\text{do } \{\text{ASSERT } \Phi; S'\})$
 $\langle \text{proof} \rangle$

lemma *ASSERT-refine-right-pres*:
assumes $\Phi \implies S \leq \Downarrow R (\text{do } \{\text{ASSERT } \Phi; S'\})$
shows $S \leq \Downarrow R (\text{do } \{\text{ASSERT } \Phi; S'\})$

<proof>

lemma *ASSERT-refine-left*:

assumes Φ
assumes $\Phi \implies S \leq \Downarrow R S'$
shows $do\{ASSERT\ \Phi; S\} \leq \Downarrow R S'$
<proof>

lemma *ASSUME-refine-right*:

assumes Φ
assumes $\Phi \implies S \leq \Downarrow R S'$
shows $S \leq \Downarrow R (do\{ASSUME\ \Phi; S'\})$
<proof>

lemma *ASSUME-refine-left*:

assumes $\Phi \implies S \leq \Downarrow R S'$
shows $do\{ASSUME\ \Phi; S\} \leq \Downarrow R S'$
<proof>

lemma *ASSUME-refine-left-pres*:

assumes $\Phi \implies do\{ASSUME\ \Phi; S\} \leq \Downarrow R S'$
shows $do\{ASSUME\ \Phi; S\} \leq \Downarrow R S'$
<proof>

Warning: The order of [*refine*]-declarations is important here, as preconditions should be generated before additional proof obligations.

lemmas [*refine0*] = *ASSUME-refine-right*

lemmas [*refine0*] = *ASSERT-refine-left*

lemmas [*refine0*] = *ASSUME-refine-left*

lemmas [*refine0*] = *ASSERT-refine-right*

For backward compatibility, as *intro refine* still seems to be used instead of *refine-recg*.

lemmas [*refine*] = *ASSUME-refine-right*

lemmas [*refine*] = *ASSERT-refine-left*

lemmas [*refine*] = *ASSUME-refine-left*

lemmas [*refine*] = *ASSERT-refine-right*

definition *lift-assn* :: ('a × 'b) set ⇒ ('b ⇒ bool) ⇒ ('a ⇒ bool)

— Lift assertion over refinement relation

where *lift-assn* R Φ s ≡ ∃ s'. (s, s') ∈ R ∧ Φ s'

lemma *lift-assnI*: $\llbracket (s, s') \in R; \Phi\ s' \rrbracket \implies \text{lift-assn } R\ \Phi\ s$

<proof>

lemma *REC-refine[refine]*:

assumes M : *trimono body*
assumes $R0$: $(x, x') \in R$
assumes RS : $\bigwedge f f' x x'. \llbracket \bigwedge x x'. (x, x') \in R \implies f x \leq \Downarrow S (f' x'); (x, x') \in R; \rrbracket$
 $REC \text{ body}' = f'$
 $\implies \text{body } f x \leq \Downarrow S (\text{body}' f' x')$
shows $REC (\lambda f x. \text{body } f x) x \leq \Downarrow S (REC (\lambda f' x'. \text{body}' f' x') x')$
<proof>

lemma $RECT$ -*refine*[*refine*]:
assumes M : *trimono body*
assumes $R0$: $(x, x') \in R$
assumes RS : $\bigwedge f f' x x'. \llbracket \bigwedge x x'. (x, x') \in R \implies f x \leq \Downarrow S (f' x'); (x, x') \in R \rrbracket$
 $\implies \text{body } f x \leq \Downarrow S (\text{body}' f' x')$
shows $RECT (\lambda f x. \text{body } f x) x \leq \Downarrow S (RECT (\lambda f' x'. \text{body}' f' x') x')$
<proof>

lemma *if-refine*[*refine*]:
assumes $b \longleftrightarrow b'$
assumes $\llbracket b; b' \rrbracket \implies S1 \leq \Downarrow R S1'$
assumes $\llbracket \neg b; \neg b' \rrbracket \implies S2 \leq \Downarrow R S2'$
shows $(\text{if } b \text{ then } S1 \text{ else } S2) \leq \Downarrow R (\text{if } b' \text{ then } S1' \text{ else } S2')$
<proof>

lemma *Let-unfold-refine*[*refine*]:
assumes $f x \leq \Downarrow R (f' x')$
shows $\text{Let } x f \leq \Downarrow R (\text{Let } x' f')$
<proof>

The next lemma is sometimes more convenient, as it prevents large let-expressions from exploding by being completely unfolded.

lemma *Let-refine*:
assumes $(m, m') \in R'$
assumes $\bigwedge x x'. (x, x') \in R' \implies f x \leq \Downarrow R (f' x')$
shows $\text{Let } m (\lambda x. f x) \leq \Downarrow R (\text{Let } m' (\lambda x'. f' x'))$
<proof>

lemma *Let-refine'*:
assumes $(m, m') \in R$
assumes $(m, m') \in R \implies f m \leq \Downarrow S (f' m')$
shows $\text{Let } m f \leq \Downarrow S (\text{Let } m' f')$
<proof>

lemma *case-option-refine*[*refine*]:
assumes $(v, v') \in \langle Ra \rangle \text{option-rel}$
assumes $\llbracket v = \text{None}; v' = \text{None} \rrbracket \implies n \leq \Downarrow Rb n'$
assumes $\bigwedge x x'. \llbracket v = \text{Some } x; v' = \text{Some } x'; (x, x') \in Ra \rrbracket$
 $\implies f x \leq \Downarrow Rb (f' x')$
shows $\text{case-option } n f v \leq \Downarrow Rb (\text{case-option } n' f' v')$

<proof>

lemma *list-case-refine*[*refine*]:

assumes $(li, l) \in \langle S \rangle \text{list-rel}$

assumes $fni \leq \Downarrow R \text{ fn}$

assumes $\bigwedge xi \ x \ xsi \ xs. \llbracket (xi, x) \in S; (xsi, xs) \in \langle S \rangle \text{list-rel}; li = xi \# xsi; l = x \# xs \rrbracket \implies fci \ xi \ xsi \leq \Downarrow R (fc \ x \ xs)$

shows $(\text{case } li \text{ of } \llbracket \implies fni \mid xi \# xsi \implies fci \ xi \ xsi \rrbracket \leq \Downarrow R (\text{case } l \text{ of } \llbracket \implies fn \mid x \# xs \rrbracket \implies fc \ x \ xs)$

<proof>

It is safe to split conjunctions in refinement goals.

declare *conjI*[*refine*]

The following rules try to compensate for some structural changes, like inlining lets or converting binds to lets.

lemma *remove-Let-refine*[*refine2*]:

assumes $M \leq \Downarrow R (f \ x)$

shows $M \leq \Downarrow R (\text{Let } x \ f) \text{ <proof>}$

lemma *intro-Let-refine*[*refine2*]:

assumes $f \ x \leq \Downarrow R \ M'$

shows $\text{Let } x \ f \leq \Downarrow R \ M' \text{ <proof>}$

lemma *bind2let-refine*[*refine2*]:

assumes $\text{RETURN } x \leq \Downarrow R' \ M'$

assumes $\bigwedge x'. (x, x') \in R' \implies f \ x \leq \Downarrow R (f' \ x')$

shows $\text{Let } x \ f \leq \Downarrow R (\text{bind } M' (\lambda x'. f' \ x'))$

<proof>

lemma *bind-Let-refine2*[*refine2*]: \llbracket

$m' \leq \Downarrow R' (\text{RETURN } x);$

$\bigwedge x'. \llbracket \text{inres } m' \ x'; (x', x) \in R' \rrbracket \implies f' \ x' \leq \Downarrow R (f \ x)$

$\rrbracket \implies m' \gg (\lambda x'. f' \ x') \leq \Downarrow R (\text{Let } x (\lambda x. f \ x))$

<proof>

lemma *bind2letRETURN-refine*[*refine2*]:

assumes $\text{RETURN } x \leq \Downarrow R' \ M'$

assumes $\bigwedge x'. (x, x') \in R' \implies \text{RETURN } (f \ x) \leq \Downarrow R (f' \ x')$

shows $\text{RETURN } (\text{Let } x \ f) \leq \Downarrow R (\text{bind } M' (\lambda x'. f' \ x'))$

<proof>

lemma *RETURN-as-SPEC-refine*[*refine2*]:

assumes $M \leq \text{SPEC } (\lambda c. (c, a) \in R)$

shows $M \leq \Downarrow R (\text{RETURN } a)$

<proof>

lemma *RETURN-as-SPEC-refine-old*:

$\bigwedge M \ R. M \leq \Downarrow R (\text{SPEC } (\lambda x. x = v)) \implies M \leq \Downarrow R (\text{RETURN } v)$

<proof>

lemma *if-RETURN-refine* [*refine2*]:

assumes $b \longleftrightarrow b'$

assumes $\llbracket b; b \rrbracket \Longrightarrow \text{RETURN } S1 \leq \Downarrow R S1'$

assumes $\llbracket \neg b; \neg b \rrbracket \Longrightarrow \text{RETURN } S2 \leq \Downarrow R S2'$

shows $\text{RETURN (if } b \text{ then } S1 \text{ else } S2) \leq \Downarrow R \text{(if } b' \text{ then } S1' \text{ else } S2')$

<proof>

lemma *RES-sng-as-SPEC-refine*[*refine2*]:

assumes $M \leq \text{SPEC } (\lambda c. (c, a) \in R)$

shows $M \leq \Downarrow R (\text{RES } \{a\})$

<proof>

lemma *intro-spec-refine-iff*:

$(\text{bind } (\text{RES } X) f \leq \Downarrow R M) \longleftrightarrow (\forall x \in X. f x \leq \Downarrow R M)$

<proof>

lemma *intro-spec-refine*[*refine2*]:

assumes $\bigwedge x. x \in X \Longrightarrow f x \leq \Downarrow R M$

shows $\text{bind } (\text{RES } X) (\lambda x. f x) \leq \Downarrow R M$

<proof>

The following rules are intended for manual application, to reflect some common structural changes, that, however, are not suited to be applied automatically.

Replacing a let by a deterministic computation

lemma *let2bind-refine*:

assumes $m \leq \Downarrow R' (\text{RETURN } m')$

assumes $\bigwedge x x'. (x, x') \in R' \Longrightarrow f x \leq \Downarrow R (f' x')$

shows $\text{bind } m (\lambda x. f x) \leq \Downarrow R (\text{Let } m' (\lambda x'. f' x'))$

<proof>

Introduce a new binding, without a structural match in the abstract program

lemma *intro-bind-refine*:

assumes $m \leq \Downarrow R' (\text{RETURN } m')$

assumes $\bigwedge x. (x, m') \in R' \Longrightarrow f x \leq \Downarrow R m''$

shows $\text{bind } m (\lambda x. f x) \leq \Downarrow R m''$

<proof>

lemma *intro-bind-refine-id*:

assumes $m \leq (\text{SPEC } ((=) m'))$

assumes $f m' \leq \Downarrow R m''$

shows $\text{bind } m f \leq \Downarrow R m''$

<proof>

The following set of rules executes a step on the LHS or RHS of a refinement proof obligation, without changing the other side. These kind of rules is useful for performing refinements with invisible steps.

lemma *lhs-step-If*:

$$\llbracket b \implies t \leq m; \neg b \implies e \leq m \rrbracket \implies \text{If } b \text{ } t \text{ } e \leq m \langle \text{proof} \rangle$$

lemma *lhs-step-RES*:

$$\llbracket \bigwedge x. x \in X \implies \text{RETURN } x \leq m \rrbracket \implies \text{RES } X \leq m \langle \text{proof} \rangle$$

lemma *lhs-step-SPEC*:

$$\llbracket \bigwedge x. \Phi x \implies \text{RETURN } x \leq m \rrbracket \implies \text{SPEC } (\lambda x. \Phi x) \leq m \langle \text{proof} \rangle$$

lemma *lhs-step-bind*:

$$\begin{aligned} &\text{fixes } m :: 'a \text{ nres and } f :: 'a \Rightarrow 'b \text{ nres} \\ &\text{assumes } \text{nofail } m' \implies \text{nofail } m \\ &\text{assumes } \bigwedge x. \text{nf-inres } m \ x \implies f \ x \leq m' \\ &\text{shows } \text{do } \{x \leftarrow m; f \ x\} \leq m' \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *rhs-step-bind*:

$$\begin{aligned} &\text{assumes } m \leq \Downarrow R \ m' \quad \text{inres } m \ x \quad \bigwedge x'. (x, x') \in R \implies \text{lhs} \leq \Downarrow S \ (f' \ x') \\ &\text{shows } \text{lhs} \leq \Downarrow S \ (m' \ggg f') \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *rhs-step-bind-RES*:

$$\begin{aligned} &\text{assumes } x' \in X' \\ &\text{assumes } m \leq \Downarrow R \ (f' \ x') \\ &\text{shows } m \leq \Downarrow R \ (\text{RES } X' \ggg f') \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *rhs-step-bind-SPEC*:

$$\begin{aligned} &\text{assumes } \Phi \ x' \\ &\text{assumes } m \leq \Downarrow R \ (f' \ x') \\ &\text{shows } m \leq \Downarrow R \ (\text{SPEC } \Phi \ggg f') \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *RES-bind-choose*:

$$\begin{aligned} &\text{assumes } x \in X \\ &\text{assumes } m \leq f \ x \\ &\text{shows } m \leq \text{RES } X \ggg f \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *pw-RES-bind-choose*:

$$\begin{aligned} &\text{nofail } (\text{RES } X \ggg f) \longleftrightarrow (\forall x \in X. \text{nofail } (f \ x)) \\ &\text{inres } (\text{RES } X \ggg f) \ y \longleftrightarrow (\exists x \in X. \text{inres } (f \ x) \ y) \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *prod-case-refine*:

assumes $(p',p) \in R1 \times_r R2$

assumes $\bigwedge x1' x2' x1 x2. \llbracket p' = (x1', x2'); p = (x1, x2); (x1', x1) \in R1; (x2', x2) \in R2 \rrbracket$
 $\implies f' x1' x2' \leq \Downarrow R (f x1 x2)$

shows $(\text{case } p' \text{ of } (x1', x2') \implies f' x1' x2') \leq \Downarrow R (\text{case } p \text{ of } (x1, x2) \implies f x1 x2)$
 $\langle \text{proof} \rangle$

2.6.6 Relators

declare *fun-relI[refine]*

definition *nres-rel where*

nres-rel-def-internal: $nres\text{-rel } R \equiv \{(c, a). c \leq \Downarrow R a\}$

lemma *nres-rel-def*: $\langle R \rangle nres\text{-rel} \equiv \{(c, a). c \leq \Downarrow R a\}$

$\langle \text{proof} \rangle$

lemma *nres-relD*: $(c, a) \in \langle R \rangle nres\text{-rel} \implies c \leq \Downarrow R a$ $\langle \text{proof} \rangle$

lemma *nres-relI[refine]*: $c \leq \Downarrow R a \implies (c, a) \in \langle R \rangle nres\text{-rel}$ $\langle \text{proof} \rangle$

lemma *nres-rel-comp*: $\langle A \rangle nres\text{-rel } O \langle B \rangle nres\text{-rel} = \langle A O B \rangle nres\text{-rel}$

$\langle \text{proof} \rangle$

lemma *pw-nres-rel-iff*: $(a, b) \in \langle A \rangle nres\text{-rel} \iff \text{nofail } (\Downarrow A b) \longrightarrow \text{nofail } a \wedge (\forall x. \text{inres } a x \longrightarrow \text{inres } (\Downarrow A b) x)$

$\langle \text{proof} \rangle$

lemma *param-SUCCEED[param]*: $(SUCCEED, SUCCEED) \in \langle R \rangle nres\text{-rel}$

$\langle \text{proof} \rangle$

lemma *param-FAIL[param]*: $(FAIL, FAIL) \in \langle R \rangle nres\text{-rel}$

$\langle \text{proof} \rangle$

lemma *param-RES[param]*:

$(RES, RES) \in \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle nres\text{-rel}$

$\langle \text{proof} \rangle$

lemma *param-RETURN[param]*:

$(RETURN, RETURN) \in R \rightarrow \langle R \rangle nres\text{-rel}$

$\langle \text{proof} \rangle$

lemma *param-bind[param]*:

$(\text{bind}, \text{bind}) \in \langle Ra \rangle nres\text{-rel} \rightarrow (Ra \rightarrow \langle Rb \rangle nres\text{-rel}) \rightarrow \langle Rb \rangle nres\text{-rel}$

$\langle \text{proof} \rangle$

lemma *param-ASSERT-bind[param]*: \llbracket

$(\Phi, \Psi) \in \text{bool-rel};$

$\llbracket \Phi; \Psi \rrbracket \implies (f, g) \in \langle R \rangle nres\text{-rel}$

$\llbracket \implies (ASSERT \Phi \gg f, ASSERT \Psi \gg g) \in \langle R \rangle nres\text{-}rel$
 $\langle proof \rangle$

2.6.7 Autoref Setup

consts *i-nres* :: *interface* \Rightarrow *interface*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of nres-rel i-nres*]

definition [*simp*]: *op-nres-ASSERT-bnd* Φ *m* \equiv *do* {*ASSERT* Φ ; *m*}

lemma *param-op-nres-ASSERT-bnd*[*param*]:

assumes $\Phi' \implies \Phi$

assumes $\llbracket \Phi'; \Phi \rrbracket \implies (m, m') \in \langle R \rangle nres\text{-}rel$

shows (*op-nres-ASSERT-bnd* Φ *m*, *op-nres-ASSERT-bnd* Φ' *m'*) $\in \langle R \rangle nres\text{-}rel$
 $\langle proof \rangle$

context begin interpretation *autoref-syn* $\langle proof \rangle$

lemma *id-ASSERT*[*autoref-op-pat-def*]:

do {*ASSERT* Φ ; *m*} \equiv *OP* (*op-nres-ASSERT-bnd* Φ)\$*m*
 $\langle proof \rangle$

definition [*simp*]: *op-nres-ASSUME-bnd* Φ *m* \equiv *do* {*ASSUME* Φ ; *m*}

lemma *id-ASSUME*[*autoref-op-pat-def*]:

do {*ASSUME* Φ ; *m*} \equiv *OP* (*op-nres-ASSUME-bnd* Φ)\$*m*
 $\langle proof \rangle$

end

lemma *autoref-SUCCEED*[*autoref-rules*]: (*SUCCEED*, *SUCCEED*) $\in \langle R \rangle nres\text{-}rel$
 $\langle proof \rangle$

lemma *autoref-FAIL*[*autoref-rules*]: (*FAIL*, *FAIL*) $\in \langle R \rangle nres\text{-}rel$
 $\langle proof \rangle$

lemma *autoref-RETURN*[*autoref-rules*]:

(*RETURN*, *RETURN*) $\in R \rightarrow \langle R \rangle nres\text{-}rel$
 $\langle proof \rangle$

lemma *autoref-bind*[*autoref-rules*]:

(*bind*, *bind*) $\in \langle R1 \rangle nres\text{-}rel \rightarrow (R1 \rightarrow \langle R2 \rangle nres\text{-}rel) \rightarrow \langle R2 \rangle nres\text{-}rel$
 $\langle proof \rangle$

context begin interpretation *autoref-syn* $\langle proof \rangle$

lemma *autoref-ASSERT*[*autoref-rules*]:

assumes $\Phi \implies (m', m) \in \langle R \rangle nres\text{-rel}$

shows (

m' ,

$(OP (op\text{-}nres\text{-}ASSERT\text{-}bnd \Phi) \text{ ::: } \langle R \rangle nres\text{-rel} \rightarrow \langle R \rangle nres\text{-rel}) \$ m) \in \langle R \rangle nres\text{-rel}$

$\langle proof \rangle$)

lemma *autoref-ASSUME*[*autoref-rules*]:

assumes *SIDE-PRECOND* Φ

assumes $\Phi \implies (m', m) \in \langle R \rangle nres\text{-rel}$

shows (

m' ,

$(OP (op\text{-}nres\text{-}ASSUME\text{-}bnd \Phi) \text{ ::: } \langle R \rangle nres\text{-rel} \rightarrow \langle R \rangle nres\text{-rel}) \$ m) \in \langle R \rangle nres\text{-rel}$

$\langle proof \rangle$)

lemma *autoref-REC*[*autoref-rules*]:

assumes $(B, B') \in (Ra \rightarrow \langle Rr \rangle nres\text{-rel}) \rightarrow Ra \rightarrow \langle Rr \rangle nres\text{-rel}$

assumes *DEFER trimono* B

shows (*REC* B ,

$(OP$ *REC*

$\text{ ::: } ((Ra \rightarrow \langle Rr \rangle nres\text{-rel}) \rightarrow Ra \rightarrow \langle Rr \rangle nres\text{-rel}) \rightarrow Ra \rightarrow \langle Rr \rangle nres\text{-rel}) \$ B'$

$) \in Ra \rightarrow \langle Rr \rangle nres\text{-rel}$

$\langle proof \rangle$)

theorem *param-RECT*[*param*]:

assumes $(B, B') \in (Ra \rightarrow \langle Rr \rangle nres\text{-rel}) \rightarrow Ra \rightarrow \langle Rr \rangle nres\text{-rel}$

and *trimono* B

shows (*RECT* B , *RECT* $B')$ $\in Ra \rightarrow \langle Rr \rangle nres\text{-rel}$

$\langle proof \rangle$)

lemma *autoref-RECT*[*autoref-rules*]:

assumes $(B, B') \in (Ra \rightarrow \langle Rr \rangle nres\text{-rel}) \rightarrow Ra \rightarrow \langle Rr \rangle nres\text{-rel}$

assumes *DEFER trimono* B

shows (*RECT* B ,

$(OP$ *RECT*

$\text{ ::: } ((Ra \rightarrow \langle Rr \rangle nres\text{-rel}) \rightarrow Ra \rightarrow \langle Rr \rangle nres\text{-rel}) \rightarrow Ra \rightarrow \langle Rr \rangle nres\text{-rel}) \$ B'$

$) \in Ra \rightarrow \langle Rr \rangle nres\text{-rel}$

$\langle proof \rangle$)

end

2.6.8 Convenience Rules

In this section, we define some lemmas that simplify common prover tasks.

lemma *ref-two-step*: $A \leq \Downarrow R B \implies B \leq C \implies A \leq \Downarrow R C$

$\langle proof \rangle$

lemma *pw-ref-iff*:
shows $S \leq \Downarrow R S'$
 \longleftrightarrow (*nofail* S'
 \longrightarrow *nofail* $S \wedge (\forall x. \text{inres } S x \longrightarrow (\exists s'. (x, s') \in R \wedge \text{inres } S' s'))$)
<proof>

lemma *pw-ref-I*:
assumes *nofail* S'
 \longrightarrow *nofail* $S \wedge (\forall x. \text{inres } S x \longrightarrow (\exists s'. (x, s') \in R \wedge \text{inres } S' s'))$
shows $S \leq \Downarrow R S'$
<proof>

Introduce an abstraction relation. Usage: *rule introR*[where $R = \text{absRel}$]

lemma *introR*: $(a, a') \in R \implies (a, a') \in R$ *<proof>*

lemma *intro-prgR*: $c \leq \Downarrow R a \implies c \leq \Downarrow R a$ *<proof>*

lemma *refine-IdI*: $m \leq m' \implies m \leq \Downarrow \text{Id } m'$ *<proof>*

lemma *le-ASSERTI-pres*:
assumes $\Phi \implies S \leq \text{do } \{\text{ASSERT } \Phi; S'\}$
shows $S \leq \text{do } \{\text{ASSERT } \Phi; S'\}$
<proof>

lemma *RETURN-ref-SPEC*:
assumes $\text{RETURN } c \leq \Downarrow R (\text{SPEC } \Phi)$
obtains a **where** $(c, a) \in R \quad \Phi a$
<proof>

lemma *RETURN-ref-RETURND*:
assumes $\text{RETURN } c \leq \Downarrow R (\text{RETURN } a)$
shows $(c, a) \in R$
<proof>

lemma *return-refine-prop-return*:
assumes *nofail* m
assumes $\text{RETURN } x \leq \Downarrow R m$
obtains x' **where** $(x, x') \in R \quad \text{RETURN } x' \leq m$
<proof>

lemma *ignore-snd-refine-conv*:
 $(m \leq \Downarrow (R \times_r \text{UNIV}) m') \longleftrightarrow m \gg (\text{RETURN } o \text{fst}) \leq \Downarrow R (m' \gg (\text{RETURN } o \text{fst}))$
<proof>

lemma *ret-le-down-conv*:
nofail $m \implies \text{RETURN } c \leq \Downarrow R m \longleftrightarrow (\exists a. (c, a) \in R \wedge \text{RETURN } a \leq m)$

<proof>

lemma *SPEC-eq-is-RETURN*:

$SPEC ((=) x) = RETURN x$

$SPEC (\lambda x. x=y) = RETURN y$

<proof>

lemma *RETURN-SPEC-conv*: $RETURN r = SPEC (\lambda x. x=r)$

<proof>

lemma *refine2spec-aux*:

$a \leq \Downarrow R b \iff (nofail b \implies a \leq SPEC (\lambda r. (\exists x. inres b x \wedge (r,x) \in R)))$

<proof>

lemma *refine2specI*:

assumes $nofail b \implies a \leq SPEC (\lambda r. (\exists x. inres b x \wedge (r,x) \in R))$

shows $a \leq \Downarrow R b$

<proof>

lemma *specify-left*:

assumes $m \leq SPEC \Phi$

assumes $\bigwedge x. \Phi x \implies f x \leq M$

shows $do \{ x \leftarrow m; f x \} \leq M$

<proof>

lemma *build-rel-SPEC*:

$M \leq SPEC (\lambda x. \Phi (\alpha x) \wedge I x) \implies M \leq \Downarrow (build-rel \alpha I) (SPEC \Phi)$

<proof>

lemma *build-rel-SPEC-conv*: $\Downarrow (br \alpha I) (SPEC \Phi) = SPEC (\lambda x. I x \wedge \Phi (\alpha x))$

<proof>

lemma *refine-IdD*: $c \leq \Downarrow Id a \implies c \leq a$ *<proof>*

lemma *bind-sim-select-rule*:

assumes $m \gg f' \leq SPEC \Psi$

assumes $\bigwedge x. \llbracket nofail m; inres m x; f' x \leq SPEC \Psi \rrbracket \implies f x \leq SPEC \Phi$

shows $m \gg f \leq SPEC \Phi$

— Simultaneously select a result from assumption and verification goal. Useful to work with assumptions that restrict the current program to be verified.

<proof>

lemma *assert-bind-spec-conv*: $ASSERT \Phi \gg m \leq SPEC \Psi \iff (\Phi \wedge m \leq SPEC \Psi)$

— Simplify a bind-assert verification condition. Useful if this occurs in the assumptions, and considerably faster than using pointwise reasoning, which may cause a blowup for many chained assertions.

<proof>

lemma *summarize-ASSERT-conv*: $do \{ ASSERT \Phi; ASSERT \Psi; m \} = do \{ ASSERT (\Phi \wedge \Psi); m \}$
 ⟨proof⟩

lemma *bind-ASSERT-eq-if*: $do \{ ASSERT \Phi; m \} = (if \Phi then m else FAIL)$
 ⟨proof⟩

lemma *le-RES-nofailI*:

assumes $a \leq RES x$

shows *nofail a*

⟨proof⟩

lemma *add-invar-refineI*:

assumes $f x \leq \Downarrow R (f' x')$

and *nofail* $(f x) \implies f x \leq SPEC I$

shows $f x \leq \Downarrow \{ (c, a). (c, a) \in R \wedge I c \} (f' x')$

⟨proof⟩

lemma *bind-RES-RETURN-eq*: $bind (RES X) (\lambda x. RETURN (f x)) = RES \{ f x \mid x. x \in X \}$
 ⟨proof⟩

lemma *bind-RES-RETURN2-eq*: $bind (RES X) (\lambda(x,y). RETURN (f x y)) = RES \{ f x y \mid x y. (x,y) \in X \}$
 ⟨proof⟩

lemma *le-SPEC-bindI*:

assumes Φx

assumes $m \leq f x$

shows $m \leq SPEC \Phi \gg= f$

⟨proof⟩

lemma *bind-assert-refine*:

assumes $m1 \leq SPEC \Phi$

assumes $\bigwedge x. \Phi x \implies m2 x \leq m'$

shows $do \{ x \leftarrow m1; ASSERT (\Phi x); m2 x \} \leq m'$

⟨proof⟩

lemma *RETURN-refine-iff[simp]*: $RETURN x \leq \Downarrow R (RETURN y) \longleftrightarrow (x,y) \in R$
 ⟨proof⟩

lemma *RETURN-RES-refine-iff*:

$RETURN x \leq \Downarrow R (RES Y) \longleftrightarrow (\exists y \in Y. (x,y) \in R)$

⟨proof⟩

lemma *RETURN-RES-refine*:

assumes $\exists x'. (x, x') \in R \wedge x' \in X$
shows $RETURN\ x \leq \Downarrow R (RES\ X)$
 $\langle proof \rangle$

lemma *in-nres-rel-iff*: $(a, b) \in (R) nres\text{-}rel \iff a \leq \Downarrow R\ b$
 $\langle proof \rangle$

lemma *inf-RETURN-RES*:
 $inf (RETURN\ x) (RES\ X) = (if\ x \in X\ then\ RETURN\ x\ else\ SUCCEED)$
 $inf (RES\ X) (RETURN\ x) = (if\ x \in X\ then\ RETURN\ x\ else\ SUCCEED)$
 $\langle proof \rangle$

lemma *inf-RETURN-SPEC[simp]*:
 $inf (RETURN\ x) (SPEC\ (\lambda y. \Phi\ y)) = SPEC\ (\lambda y. y = x \wedge \Phi\ x)$
 $inf (SPEC\ (\lambda y. \Phi\ y)) (RETURN\ x) = SPEC\ (\lambda y. y = x \wedge \Phi\ x)$
 $\langle proof \rangle$

lemma *RES-sng-eq-RETURN*: $RES\ \{x\} = RETURN\ x$
 $\langle proof \rangle$

lemma *nofail-inf-serialize*:
 $\llbracket nofail\ a; nofail\ b \rrbracket \implies inf\ a\ b = do\ \{x \leftarrow a; ASSUME\ (inres\ b\ x); RETURN\ x\}$
 $\langle proof \rangle$

lemma *conc-fun-SPEC*:
 $\Downarrow R (SPEC\ (\lambda x. \Phi\ x)) = SPEC\ (\lambda y. \exists x. (y, x) \in R \wedge \Phi\ x)$
 $\langle proof \rangle$

lemma *conc-fun-RETURN*:
 $\Downarrow R (RETURN\ x) = SPEC\ (\lambda y. (y, x) \in R)$
 $\langle proof \rangle$

lemma *use-spec-rule*:
assumes $m \leq SPEC\ \Psi$
assumes $m \leq SPEC\ (\lambda s. \Psi\ s \longrightarrow \Phi\ s)$
shows $m \leq SPEC\ \Phi$
 $\langle proof \rangle$

lemma *strengthen-SPEC*: $m \leq SPEC\ \Phi \implies m \leq SPEC(\lambda s. inres\ m\ s \wedge nofail\ m \wedge \Phi\ s)$
— Strengthen SPEC by adding trivial upper bound for result
 $\langle proof \rangle$

lemma *weaken-SPEC*:
 $m \leq SPEC\ \Phi \implies (\bigwedge x. \Phi\ x \implies \Psi\ x) \implies m \leq SPEC\ \Psi$
 $\langle proof \rangle$

lemma *bind-le-nofailI*:

assumes *nofail m*
assumes $\bigwedge x. \text{RETURN } x \leq m \implies f x \leq m'$
shows $m \ggg f \leq m'$
 $\langle \text{proof} \rangle$

lemma *bind-le-shift*:

$\text{bind } m f \leq m'$
 $\longleftrightarrow m \leq (\text{if } \text{nofail } m' \text{ then } \text{SPEC } (\lambda x. f x \leq m') \text{ else } \text{FAIL})$
 $\langle \text{proof} \rangle$

lemma *If-bind-distrib[simp]*:

fixes $t e :: 'a \text{ nres}$
shows $(\text{If } b \ t \ e \ggg (\lambda x. f x)) = (\text{If } b \ (t \ggg (\lambda x. f x)) \ (e \ggg (\lambda x. f x)))$
 $\langle \text{proof} \rangle$

lemma *unused-bind-conv*:

assumes *NO-MATCH (ASSERT Φ) m*
assumes *NO-MATCH (ASSUME Φ) m*
shows $(m \ggg (\lambda x. c)) = (\text{ASSERT } (\text{nofail } m) \ggg (\lambda x. \text{ASSUME } (\exists x. \text{inres } m \ x) \ggg (\lambda x. c)))$
 $\langle \text{proof} \rangle$

The following rules are useful for massaging programs before the refinement takes place

lemma *let-to-bind-conv*:

$\text{Let } x \ f = \text{RETURN } x \ggg f$
 $\langle \text{proof} \rangle$

lemmas *bind-to-let-conv = let-to-bind-conv[symmetric]*

lemma *pull-out-let-conv*: $\text{RETURN } (\text{Let } x \ f) = \text{Let } x \ (\lambda x. \text{RETURN } (f x))$

$\langle \text{proof} \rangle$

lemma *push-in-let-conv*:

$\text{Let } x \ (\lambda x. \text{RETURN } (f x)) = \text{RETURN } (\text{Let } x \ f)$
 $\text{Let } x \ (\text{RETURN } o \ f) = \text{RETURN } (\text{Let } x \ f)$
 $\langle \text{proof} \rangle$

lemma *pull-out-RETURN-case-option*:

$\text{case-option } (\text{RETURN } a) \ (\lambda v. \text{RETURN } (f v)) \ x = \text{RETURN } (\text{case-option } a \ f \ x)$
 $\langle \text{proof} \rangle$

lemma *if-bind-cond-refine*:

assumes $ci \leq \text{RETURN } b$
assumes $b \implies ti \leq \Downarrow R \ t$

assumes $\neg b \implies ei \leq \Downarrow R e$
shows $do \{ b \leftarrow ci; \text{if } b \text{ then } ti \text{ else } ei \} \leq \Downarrow R (\text{if } b \text{ then } t \text{ else } e)$
 $\langle proof \rangle$

lemma *intro-RETURN-Let-refine*:
assumes $RETURN (f x) \leq \Downarrow R M'$
shows $RETURN (Let x f) \leq \Downarrow R M'$
 $\langle proof \rangle$

lemma *ife-FAIL-to-ASSERT-cnv*:
 $(\text{if } \Phi \text{ then } m \text{ else } FAIL) = op\text{-}nres\text{-}ASSERT\text{-}bnd \ \Phi \ m$
 $\langle proof \rangle$

lemma *nres-bind-let-law*: $(do \{ x \leftarrow do \{ let y=v; f y \}; g x \} :: - nres)$
 $= do \{ let y=v; x \leftarrow f y; g x \} \langle proof \rangle$

lemma *unused-bind-RES-ne[simp]*: $X \neq \{\} \implies do \{ - \leftarrow RES X; m \} = m$
 $\langle proof \rangle$

lemma *le-ASSERT-defI1*:
assumes $c \equiv do \{ ASSERT \ \Phi; m \}$
assumes $\Phi \implies m' \leq c$
shows $m' \leq c$
 $\langle proof \rangle$

lemma *refine-ASSERT-defI1*:
assumes $c \equiv do \{ ASSERT \ \Phi; m \}$
assumes $\Phi \implies m' \leq \Downarrow R c$
shows $m' \leq \Downarrow R c$
 $\langle proof \rangle$

lemma *le-ASSERT-defI2*:
assumes $c \equiv do \{ ASSERT \ \Phi; ASSERT \ \Psi; m \}$
assumes $\llbracket \Phi; \Psi \rrbracket \implies m' \leq c$
shows $m' \leq c$
 $\langle proof \rangle$

lemma *refine-ASSERT-defI2*:
assumes $c \equiv do \{ ASSERT \ \Phi; ASSERT \ \Psi; m \}$
assumes $\llbracket \Phi; \Psi \rrbracket \implies m' \leq \Downarrow R c$
shows $m' \leq \Downarrow R c$
 $\langle proof \rangle$

lemma *ASSERT-le-defI*:
assumes $c \equiv do \{ ASSERT \ \Phi; m \}$
assumes Φ
assumes $\Phi \implies m' \leq m$

shows $c \leq m$
 ⟨proof⟩

lemma *ASSERT-same-eq-conv*: $(\text{ASSERT } \Phi \gg m) = (\text{ASSERT } \Phi \gg n) \longleftrightarrow (\Phi \longrightarrow m=n)$
 ⟨proof⟩

lemma *case-prod-bind-simp*[simp]:
 $(\lambda x. (\text{case } x \text{ of } (a, b) \Rightarrow f a b) \leq \text{SPEC } \Phi) = (\lambda(a,b). f a b \leq \text{SPEC } \Phi)$
 ⟨proof⟩

lemma *RECT-eq-REC'*: $\text{nofail } (\text{RECT } B x) \Longrightarrow \text{RECT } B x = \text{REC } B x$
 ⟨proof⟩

lemma *rel2p-nres-RETURN*[rel2p]: $\text{rel2p } (\langle A \rangle \text{nres-rel}) (\text{RETURN } x) (\text{RETURN } y) = \text{rel2p } A x y$
 ⟨proof⟩

Boolean Operations on Specifications

lemma *SPEC-iff*:
assumes $P \leq \text{SPEC } (\lambda s. Q s \longrightarrow R s)$
and $P \leq \text{SPEC } (\lambda s. \neg Q s \longrightarrow \neg R s)$
shows $P \leq \text{SPEC } (\lambda s. Q s \longleftrightarrow R s)$
 ⟨proof⟩

lemma *SPEC-rule-conjI*:
assumes $A \leq \text{SPEC } P$ **and** $A \leq \text{SPEC } Q$
shows $A \leq \text{SPEC } (\lambda v. P v \wedge Q v)$
 ⟨proof⟩

lemma *SPEC-rule-conjunct1*:
assumes $A \leq \text{SPEC } (\lambda v. P v \wedge Q v)$
shows $A \leq \text{SPEC } P$
 ⟨proof⟩

lemma *SPEC-rule-conjunct2*:
assumes $A \leq \text{SPEC } (\lambda v. P v \wedge Q v)$
shows $A \leq \text{SPEC } Q$
 ⟨proof⟩

Pointwise Reasoning

lemma *inres-if*:
 $\llbracket \text{inres } (\text{if } P \text{ then } Q \text{ else } R) x; \llbracket P; \text{inres } Q x \rrbracket \Longrightarrow S; \llbracket \neg P; \text{inres } R x \rrbracket \Longrightarrow S \rrbracket$
 $\Longrightarrow S$
 ⟨proof⟩

lemma *inres-SPEC*:

inres $M\ x \Longrightarrow M \leq \text{SPEC } \Phi \Longrightarrow \Phi\ x$
 ⟨proof⟩

lemma *SPEC-nofail*:
 $X \leq \text{SPEC } \Phi \Longrightarrow \text{nofail } X$
 ⟨proof⟩

lemma *nofail-SPEC*: $\text{nofail } m \Longrightarrow m \leq \text{SPEC } (\lambda\cdot. \text{True})$
 ⟨proof⟩

lemma *nofail-SPEC-iff*: $\text{nofail } m \longleftrightarrow m \leq \text{SPEC } (\lambda\cdot. \text{True})$
 ⟨proof⟩

lemma *nofail-SPEC-triv-refine*: $\llbracket \text{nofail } m; \bigwedge x. \Phi\ x \rrbracket \Longrightarrow m \leq \text{SPEC } \Phi$
 ⟨proof⟩

end

2.7 Less-Equal or Fail

theory *Refine-Leof*
imports *Refine-Basic*
begin

A predicate that states refinement or that the LHS fails.

definition *le-or-fail* :: $'a\ \text{nres} \Rightarrow 'a\ \text{nres} \Rightarrow \text{bool}$ (**infix** \leq_n 50) **where**
 $m \leq_n m' \equiv \text{nofail } m \longrightarrow m \leq m'$

lemma *leofI*[*intro?*]:
assumes $\text{nofail } m \Longrightarrow m \leq m'$ **shows** $m \leq_n m'$
 ⟨proof⟩

lemma *leofD*:
assumes $\text{nofail } m$
assumes $m \leq_n m'$
shows $m \leq m'$
 ⟨proof⟩

lemma *pw-leof-iff*:
 $m \leq_n m' \longleftrightarrow (\text{nofail } m \longrightarrow (\forall x. \text{inres } m\ x \longrightarrow \text{inres } m'\ x))$
 ⟨proof⟩

lemma *le-by-leofI*: $\llbracket \text{nofail } m' \Longrightarrow \text{nofail } m; m \leq_n m' \rrbracket \Longrightarrow m \leq m'$
 ⟨proof⟩

lemma *inres-leof-mono*: $m \leq_n m' \implies \text{nofail } m \implies \text{inres } m \ x \implies \text{inres } m' \ x$
 ⟨proof⟩

lemma *leof-trans[trans]*: $\llbracket a \leq_n \text{RES } X; \text{RES } X \leq_n c \rrbracket \implies a \leq_n c$
 ⟨proof⟩

lemma *leof-trans-nofail*: $\llbracket a \leq_n b; \text{nofail } b; b \leq_n c \rrbracket \implies a \leq_n c$
 ⟨proof⟩

lemma *leof-refl[simp]*: $a \leq_n a$
 ⟨proof⟩

lemma *leof-RES-UNIV[simp, intro!]*: $m \leq_n \text{RES UNIV}$
 ⟨proof⟩

lemma *leof-FAIL[simp, intro!]*: $m \leq_n \text{FAIL}$ ⟨proof⟩

lemma *FAIL-leof[simp, intro!]*: $\text{FAIL} \leq_n m$
 ⟨proof⟩

lemma *leof-lift*:
 $m \leq F \implies m \leq_n F$
 ⟨proof⟩

lemma *leof-RETURN-rule[refine-vcg]*:
 $\Phi \ m \implies \text{RETURN } m \leq_n \text{SPEC } \Phi$ ⟨proof⟩

lemma *leof-bind-rule[refine-vcg]*:
 $\llbracket m \leq_n \text{SPEC } (\lambda x. f \ x \leq_n \text{SPEC } \Phi) \rrbracket \implies m \gg f \leq_n \text{SPEC } \Phi$
 ⟨proof⟩

lemma *RETURN-leof-RES-iff[simp]*: $\text{RETURN } x \leq_n \text{RES } Y \longleftrightarrow x \in Y$
 ⟨proof⟩

lemma *RES-leof-RES-iff[simp]*: $\text{RES } X \leq_n \text{RES } Y \longleftrightarrow X \subseteq Y$
 ⟨proof⟩

lemma *leof-Let-rule[refine-vcg]*: $f \ x \leq_n \text{SPEC } \Phi \implies \text{Let } x \ f \leq_n \text{SPEC } \Phi$
 ⟨proof⟩

lemma *leof-If-rule[refine-vcg]*:
 $\llbracket c \implies t \leq_n \text{SPEC } \Phi; \neg c \implies e \leq_n \text{SPEC } \Phi \rrbracket \implies \text{If } c \ t \ e \leq_n \text{SPEC } \Phi$
 ⟨proof⟩

lemma *leof-RES-rule[refine-vcg]*:
 $\llbracket \bigwedge x. \Psi \ x \implies \Phi \ x \rrbracket \implies \text{SPEC } \Psi \leq_n \text{SPEC } \Phi$
 $\llbracket \bigwedge x. x \in X \implies \Phi \ x \rrbracket \implies \text{RES } X \leq_n \text{SPEC } \Phi$
 ⟨proof⟩

lemma *leof-True-rule*: $\llbracket \bigwedge x. \Phi \ x \rrbracket \implies m \leq_n \text{SPEC } \Phi$

$\langle proof \rangle$

lemma *sup-leof-iff*: $(sup\ a\ b\ \leq_n\ m) \longleftrightarrow (nofail\ a \wedge nofail\ b \longrightarrow a \leq_n m \wedge b \leq_n m)$

$\langle proof \rangle$

lemma *sup-leof-rule[refine-vcg]*:

assumes $\llbracket nofail\ a; nofail\ b \rrbracket \Longrightarrow a \leq_n m$

assumes $\llbracket nofail\ a; nofail\ b \rrbracket \Longrightarrow b \leq_n m$

shows $sup\ a\ b\ \leq_n\ m$

$\langle proof \rangle$

lemma *leof-option-rule[refine-vcg]*:

$\llbracket v = None \Longrightarrow S1 \leq_n SPEC\ \Phi; \bigwedge x. v = Some\ x \Longrightarrow f2\ x \leq_n SPEC\ \Phi \rrbracket$

$\Longrightarrow (case\ v\ of\ None \Rightarrow S1 \mid Some\ x \Rightarrow f2\ x) \leq_n SPEC\ \Phi$

$\langle proof \rangle$

lemma *ASSERT-leof-rule[refine-vcg]*:

assumes $\Phi \Longrightarrow m \leq_n m'$

shows $do\ \{ASSERT\ \Phi; m\} \leq_n m'$

$\langle proof \rangle$

lemma *leof-ASSERT-rule[refine-vcg]*: $\llbracket \Phi \Longrightarrow m \leq_n m' \rrbracket \Longrightarrow m \leq_n ASSERT\ \Phi \gg m'$

$\langle proof \rangle$

lemma *leof-ASSERT-refine-rule[refine]*: $\llbracket \Phi \Longrightarrow m \leq_n \Downarrow R\ m' \rrbracket \Longrightarrow m \leq_n \Downarrow R\ (ASSERT\ \Phi \gg m')$

$\langle proof \rangle$

lemma *ASSUME-leof-iff*: $(ASSUME\ \Phi \leq_n SPEC\ \Psi) \longleftrightarrow (\Phi \longrightarrow \Psi\ ())$

$\langle proof \rangle$

lemma *ASSUME-leof-rule[refine-vcg]*:

assumes $\Phi \Longrightarrow \Psi\ ()$

shows $ASSUME\ \Phi \leq_n SPEC\ \Psi$

$\langle proof \rangle$

lemma *SPEC-rule-conj-leofI1*:

assumes $m \leq SPEC\ \Phi$

assumes $m \leq_n SPEC\ \Psi$

shows $m \leq SPEC\ (\lambda s. \Phi\ s \wedge \Psi\ s)$

$\langle proof \rangle$

lemma *SPEC-rule-conj-leofI2*:

assumes $m \leq_n SPEC\ \Phi$

assumes $m \leq SPEC\ \Psi$

shows $m \leq SPEC\ (\lambda s. \Phi\ s \wedge \Psi\ s)$

<proof>

lemma *SPEC-rule-leof-conjI*:

assumes $m \leq_n \text{SPEC } \Phi \quad m \leq_n \text{SPEC } \Psi$
shows $m \leq_n \text{SPEC } (\lambda x. \Phi x \wedge \Psi x)$
<proof>

lemma *leof-use-spec-rule*:

assumes $m \leq_n \text{SPEC } \Psi$
assumes $m \leq_n \text{SPEC } (\lambda s. \Psi s \longrightarrow \Phi s)$
shows $m \leq_n \text{SPEC } \Phi$
<proof>

lemma *use-spec-leof-rule*:

assumes $m \leq_n \text{SPEC } \Psi$
assumes $m \leq \text{SPEC } (\lambda s. \Psi s \longrightarrow \Phi s)$
shows $m \leq \text{SPEC } \Phi$
<proof>

lemma *leof-strengthen-SPEC*:

$m \leq_n \text{SPEC } \Phi \implies m \leq_n \text{SPEC } (\lambda x. \text{inres } m x \wedge \Phi x)$
<proof>

lemma *build-rel-SPEC-leof*:

assumes $m \leq_n \text{SPEC } (\lambda x. I x \wedge \Phi (\alpha x))$
shows $m \leq_n \Downarrow(\text{br } \alpha I) (\text{SPEC } \Phi)$
<proof>

lemma *RETURN-as-SPEC-refine-leof[refine2]*:

assumes $M \leq_n \text{SPEC } (\lambda c. (c, a) \in R)$
shows $M \leq_n \Downarrow R (\text{RETURN } a)$
<proof>

lemma *ASSERT-leof-defI*:

assumes $c \equiv \text{do } \{ \text{ASSERT } \Phi; m' \}$
assumes $\Phi \implies m' \leq_n m$
shows $c \leq_n m$
<proof>

lemma *leof-fun-conv-le*:

$(f x \leq_n M x) \longleftrightarrow (f x \leq (\text{if nofail } (f x) \text{ then } M x \text{ else FAIL}))$
<proof>

lemma *leof-add-nofailI*: $\llbracket \text{nofail } m \implies m \leq_n m' \rrbracket \implies m \leq_n m'$

<proof>

lemma *leof-cons-rule[refine-vcg-cons]*:

assumes $m \leq_n \text{SPEC } Q$
assumes $\bigwedge x. Q x \implies P x$

shows $m \leq_n SPEC P$
 $\langle proof \rangle$

lemma *RECT-rule-leof*:

assumes *WF*: $wf (V :: ('x \times 'x) \text{ set})$

assumes *I0*: $pre (x :: 'x)$

assumes *IS*: $\bigwedge f x. [\bigwedge x'. \llbracket pre\ x'; (x', x) \in V \rrbracket \implies f\ x' \leq_n M\ x'; pre\ x;$
RECT body = f

$\rrbracket \implies body\ f\ x \leq_n M\ x$

shows *RECT body* $x \leq_n M\ x$

$\langle proof \rangle$

end

2.8 Data Refinement Heuristics

theory *Refine-Heuristics*

imports *Refine-Basic*

begin

This theory contains some heuristics to automatically prove data refinement goals that are left over by the refinement condition generator.

The theorem collection *refine-hsimp* contains additional simplifier rules that are useful to discharge typical data refinement goals.

$\langle ML \rangle$

2.8.1 Type Based Heuristics

This heuristics instantiates schematic data refinement relations based on their type. Both, the left hand side and right hand side type are considered.

The heuristics works by proving goals of the form *RELATES ?R*, thereby instantiating *?R*.

definition *RELATES* :: $('a \times 'b) \text{ set} \Rightarrow \text{bool}$ **where** *RELATES* $R \equiv \text{True}$

lemma *RELATESI*: *RELATES* $R \langle proof \rangle$

$\langle ML \rangle$

2.8.2 Patterns

This section defines the patterns that are recognized as data refinement goals.

lemma *RELATESI-memb*[*refine-dref-pattern*]:
 $RELATES R \implies (a,b) \in R \implies (a,b) \in R$ *<proof>*

lemma *RELATESI-refspec*[*refine-dref-pattern*]:
 $RELATES R \implies S \leq \Downarrow R S' \implies S \leq \Downarrow R S'$ *<proof>*

Allows refine-rules to add *RELATES* goals if they introduce hidden relations

lemma *RELATES-pattern*[*refine-dref-pattern*]: $RELATES R \implies RELATES R$ *<proof>*

lemmas [*refine-hsimp*] = *RELATES-def*

2.8.3 Refinement Relations

In this section, we define some general purpose refinement relations, e.g., for product types and sets.

lemma *Id-RELATES* [*refine-dref-RELATES*]: $RELATES Id$ *<proof>*

lemma *prod-rel-RELATES*[*refine-dref-RELATES*]:
 $RELATES Ra \implies RELATES Rb \implies RELATES (\langle Ra, Rb \rangle prod-rel)$
<proof>

declare *prod-rel-sv*[*refine-hsimp*]
lemma *prod-rel-iff*[*refine-hsimp*]:
 $((a,b), (a',b')) \in \langle A, B \rangle prod-rel \iff (a,a') \in A \wedge (b,b') \in B$
<proof>

lemmas [*refine-hsimp*] = *prod-rel-id-simp*

lemma *option-rel-RELATES*[*refine-dref-RELATES*]:
 $RELATES Ra \implies RELATES (\langle Ra \rangle option-rel)$
<proof>

declare *option-rel-sv*[*refine-hsimp*]

lemmas [*refine-hsimp*] = *option-rel-id-simp*

lemmas [*refine-hsimp*] = *set-rel-sv set-rel-csv*

lemma *set-rel-RELATES*[*refine-dref-RELATES*]:
 $RELATES R \implies RELATES (\langle R \rangle set-rel)$ *<proof>*

lemma *set-rel-empty-eq*: $(S, S') \in \langle X \rangle set-rel \implies S = \{\} \iff S' = \{\}$
<proof>

lemma *set-rel-sngD*: $(\{a\}, \{b\}) \in \langle R \rangle set-rel \implies (a,b) \in R$
<proof>

lemma *Image-insert*[*refine-hsimp*]:
 $(a,b) \in R \implies \text{single-valued } R \implies R \text{ ``insert } a \ A = \text{insert } b \ (R \text{ ``} A)$
 ⟨*proof*⟩

lemmas [*refine-hsimp*] = *Image-Un*

lemma *Image-Diff*[*refine-hsimp*]:
 $\text{single-valued } (\text{converse } R) \implies R \text{ ``}(A - B) = R \text{ ``} A - R \text{ ``} B$
 ⟨*proof*⟩

lemma *Image-Inter*[*refine-hsimp*]:
 $\text{single-valued } (\text{converse } R) \implies R \text{ ``}(A \cap B) = R \text{ ``} A \cap R \text{ ``} B$
 ⟨*proof*⟩

lemma *list-rel-RELATES*[*refine-dref-RELATES*]:
 $\text{RELATES } R \implies \text{RELATES } (\langle R \rangle \text{list-rel})$ ⟨*proof*⟩

lemmas [*refine-hsimp*] = *list-rel-sv-iff list-rel-simp*

lemma *RELATES-nres-rel*[*refine-dref-RELATES*]: $\text{RELATES } R \implies \text{RELATES } (\langle R \rangle \text{nres-rel})$
 ⟨*proof*⟩

end

2.9 More Combinators

theory *Refine-More-Comb*
imports *Refine-Basic Refine-Heuristics Refine-Leaf*
begin

OBTAIN Combinator

Obtain value with given property, asserting that there exists one.

definition *OBTAIN* :: $(a \Rightarrow \text{bool}) \Rightarrow a \ \text{nres}$
where
 $\text{OBTAIN } P \equiv \text{ASSERT } (\exists x. P \ x) \gg \text{SPEC } P$

lemma *OBTAIN-nofail*[*refine-pw-simps*]: $\text{nofail } (\text{OBTAIN } P) \longleftrightarrow (\exists x. P \ x)$
 ⟨*proof*⟩

lemma *OBTAIN-inres*[*refine-pw-simps*]: $\text{inres } (\text{OBTAIN } P) \ x \longleftrightarrow (\forall x. \neg P \ x) \vee P \ x$

<proof>
lemma *OBTAIN-rule[refine-vcg]*: $\llbracket \exists x. P x; \bigwedge x. P x \implies Q x \rrbracket \implies \text{OBTAIN } P \leq \text{SPEC } Q$
<proof>
lemma *OBTAIN-refine-iff*: $\text{OBTAIN } P \leq \Downarrow R (\text{OBTAIN } Q) \iff (Ex Q \longrightarrow Ex P \wedge \text{Collect } P \subseteq R^{-1} \text{“Collect } Q$
<proof>

lemma *OBTAIN-refine[refine]*:
assumes *RELATES* *R*
assumes $\bigwedge x. Q x \implies Ex P$
assumes $\bigwedge x y. \llbracket Q x; P y \rrbracket \implies \exists x'. (y, x') \in R \wedge Q x'$
shows $\text{OBTAIN } P \leq \Downarrow R (\text{OBTAIN } Q)$
<proof>

SELECT Combinator

Select some value with given property, or *None* if there is none.

definition *SELECT* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ option nres}$
where $\text{SELECT } P \equiv \text{if } \exists x. P x \text{ then RES } \{\text{Some } x \mid x. P x\} \text{ else RETURN } \text{None}$

lemma *SELECT-rule[refine-vcg]*:
assumes $\bigwedge x. P x \implies Q (\text{Some } x)$
assumes $\forall x. \neg P x \implies Q \text{None}$
shows $\text{SELECT } P \leq \text{SPEC } Q$
<proof>

lemma *SELECT-refine-iff*: $(\text{SELECT } P \leq \Downarrow ((R) \text{option-rel}) (\text{SELECT } P'))$
 \iff (
 $(Ex P' \longrightarrow Ex P) \wedge$
 $(\forall x. P x \longrightarrow (\exists x'. (x, x') \in R \wedge P' x'))$
 $)$
<proof>

lemma *SELECT-refine[refine]*:
assumes *RELATES* *R*
assumes $\bigwedge x'. P' x' \implies \exists x. P x$
assumes $\bigwedge x. P x \implies \exists x'. (x, x') \in R \wedge P' x'$
shows $\text{SELECT } P \leq \Downarrow ((R) \text{option-rel}) (\text{SELECT } P')$
<proof>

lemma *SELECT-as-SPEC*: $\text{SELECT } P = \text{SPEC } (\lambda \text{None} \Rightarrow \forall x. \neg P x \mid \text{Some } x \Rightarrow P x)$
<proof>

lemma *SELECT-pw[refine-pw-simps]*:
nofail $(\text{SELECT } P)$
inres $(\text{SELECT } P) r \iff (r = \text{None} \longrightarrow (\forall x. \neg P x)) \wedge (\forall x. r = \text{Some } x \longrightarrow P$

x)
 $\langle proof \rangle$

lemma *SELECT-pw-simps*[simp]:
 nofail (*SELECT P*)
 inres (*SELECT P*) None $\longleftrightarrow (\forall x. \neg P x)$
 inres (*SELECT P*) (Some x) $\longleftrightarrow P x$
 $\langle proof \rangle$

end

2.10 Generic While-Combinator

theory *RefineG-While*

imports

RefineG-Recursion

HOL-Library.While-Combinator

begin

definition

WHILEI-body bind return I b f \equiv
 $(\lambda W s.$
 if $I s$ then
 if $b s$ then $bind (f s) W$ else $return s$
 else top)

definition

iWHILEI bind return I b f s0 $\equiv REC (WHILEI-body bind return I b f) s0$

definition

WHILEIT bind return I b f s0 $\equiv RECT (WHILEI-body bind return I b f) s0$

definition *iWHILE bind return* $\equiv iWHILEI bind return (\lambda-. True)$

definition *iWHILET bind return* $\equiv iWHILEIT bind return (\lambda-. True)$

lemma *mono-prover-monoI*[refine-mono]:

monotone (fun-ord (\leq)) (fun-ord (\leq)) B $\implies mono B$
 $\langle proof \rangle$

locale *generic-WHILE* =

fixes $bind :: 'm \Rightarrow ('a \Rightarrow 'm) \Rightarrow ('m :: complete-lattice)$

fixes $return :: 'a \Rightarrow 'm$

fixes *WHILEIT WHILEI WHILET WHILE*

assumes *imonad1*: $bind (return x) f = f x$

assumes *imonad2*: $bind M return = M$

assumes *imonad3*: $bind (bind M f) g = bind M (\lambda x. bind (f x) g)$

assumes *ibind-mono-ge*: $\llbracket flat-ge m m'; \bigwedge x. flat-ge (f x) (f' x) \rrbracket$

$\implies flat-ge (bind m f) (bind m' f')$

assumes *ibind-mono*: $\llbracket (\leq) m m'; \bigwedge x. (\leq) (f x) (f' x) \rrbracket$

$\implies (\leq) (bind m f) (bind m' f')$

assumes *WHILEIT-eq*: $WHILEIT \equiv iWHILEIT$ bind return
assumes *WHILEI-eq*: $WHILEI \equiv iWHILEI$ bind return
assumes *WHILET-eq*: $WHILET \equiv iWHILET$ bind return
assumes *WHILE-eq*: $WHILE \equiv iWHILE$ bind return
begin

lemmas *WHILEIT-def* = *WHILEIT-eq*[*unfolded iWHILEIT-def* [*abs-def*]]
lemmas *WHILEI-def* = *WHILEI-eq*[*unfolded iWHILEI-def* [*abs-def*]]
lemmas *WHILET-def* = *WHILET-eq*[*unfolded iWHILET-def*, *folded WHILEIT-eq*]
lemmas *WHILE-def* = *WHILE-eq*[*unfolded iWHILE-def* [*abs-def*], *folded WHILEI-eq*]

lemmas *imonad-laws* = *imonad1 imonad2 imonad3*

lemmas [*refine-mono*] = *ibind-mono-ge ibind-mono*

lemma *WHILEI-body-trimono*: *trimono* (*WHILEI-body* bind return *I b f*)
<proof>

lemmas *WHILEI-mono* = *trimonoD-mono*[*OF WHILEI-body-trimono*]
lemmas *WHILEI-mono-ge* = *trimonoD-flatf-ge*[*OF WHILEI-body-trimono*]

lemma *WHILEI-unfold*: $WHILEI I b f x = ($
if (*I x*) *then* (*if b x then bind (f x) (WHILEI I b f) else return x*) *else top*)
<proof>

lemma *REC-mono-ref*[*refine-mono*]:
 $\llbracket trimono B; \bigwedge D x. B D x \leq B' D x \rrbracket \implies REC B x \leq REC B' x$
<proof>

lemma *RECT-mono-ref*[*refine-mono*]:
 $\llbracket trimono B; \bigwedge D x. B D x \leq B' D x \rrbracket \implies RECT B x \leq RECT B' x$
<proof>

lemma *WHILEI-weaken*:
assumes *IW*: $\bigwedge x. I x \implies I' x$
shows $WHILEI I' b f x \leq WHILEI I b f x$
<proof>

lemma *WHILEIT-unfold*: $WHILEIT I b f x = ($
if (*I x*) *then*
(if b x then bind (f x) (WHILEIT I b f) else return x)
else top)
<proof>

lemma *WHILEIT-weaken*:

assumes $IW: \bigwedge x. I x \implies I' x$
shows $WHILEIT I' b f x \leq WHILEIT I b f x$
 $\langle proof \rangle$

lemma $WHILEI-le-WHILEIT: WHILEI I b f s \leq WHILEIT I b f s$
 $\langle proof \rangle$

While without Annotated Invariant

lemma $WHILE-unfold$:
 $WHILE b f s = (if b s then bind (f s) (WHILE b f) else return s)$
 $\langle proof \rangle$

lemma $WHILET-unfold$:
 $WHILET b f s = (if b s then bind (f s) (WHILET b f) else return s)$
 $\langle proof \rangle$

lemma $transfer-WHILEIT-esc[refine-transfer]$:
assumes $REF: \bigwedge x. return (f x) \leq F x$
shows $return (while b f x) \leq WHILEIT I b F x$
 $\langle proof \rangle$

lemma $transfer-WHILET-esc[refine-transfer]$:
assumes $REF: \bigwedge x. return (f x) \leq F x$
shows $return (while b f x) \leq WHILET b F x$
 $\langle proof \rangle$

lemma $WHILE-mono-prover-rule[refine-mono]$:
 $\llbracket \bigwedge x. f x \leq f' x \rrbracket \implies WHILE b f s0 \leq WHILE b f' s0$
 $\llbracket \bigwedge x. f x \leq f' x \rrbracket \implies WHILEI I b f s0 \leq WHILEI I b f' s0$
 $\llbracket \bigwedge x. f x \leq f' x \rrbracket \implies WHILET b f s0 \leq WHILET b f' s0$
 $\llbracket \bigwedge x. f x \leq f' x \rrbracket \implies WHILEIT I b f s0 \leq WHILEIT I b f' s0$
 $\llbracket \bigwedge x. flat-ge (f x) (f' x) \rrbracket \implies flat-ge (WHILET b f s0) (WHILET b f' s0)$
 $\llbracket \bigwedge x. flat-ge (f x) (f' x) \rrbracket \implies flat-ge (WHILEIT I b f s0) (WHILEIT I b f' s0)$
 $\langle proof \rangle$

end

locale $transfer-WHILE =$
 $c: generic-WHILE cbind creturn cWHILEIT cWHILEI cWHILET cWHILE +$
 $a: generic-WHILE abind areturn aWHILEIT aWHILEI aWHILET aWHILE +$
 $dist-transfer \alpha$
for $cbind$ **and** $creturn::'a \Rightarrow 'mc::complete-lattice$
and $cWHILEIT cWHILEI cWHILET cWHILE$
and $abind$ **and** $areturn::'a \Rightarrow 'ma::complete-lattice$
and $aWHILEIT aWHILEI aWHILET aWHILE$

and $\alpha :: 'mc \Rightarrow 'ma +$
assumes *transfer-bind*: $\llbracket \alpha \ m \leq M; \bigwedge x. \alpha \ (f \ x) \leq F \ x \rrbracket$
 $\implies \alpha \ (cbind \ m \ f) \leq abind \ M \ F$
assumes *transfer-return*: $\alpha \ (creturn \ x) \leq areturn \ x$
begin

lemma *transfer-WHILEIT[refine-transfer]*:
assumes *REF*: $\bigwedge x. \alpha \ (f \ x) \leq F \ x$
shows $\alpha \ (cWHILEIT \ I \ b \ f \ x) \leq aWHILEIT \ I \ b \ F \ x$
 $\langle proof \rangle$

lemma *transfer-WHILEI[refine-transfer]*:
assumes *REF*: $\bigwedge x. \alpha \ (f \ x) \leq F \ x$
shows $\alpha \ (cWHILEI \ I \ b \ f \ x) \leq aWHILEI \ I \ b \ F \ x$
 $\langle proof \rangle$

lemma *transfer-WHILE[refine-transfer]*:
assumes *REF*: $\bigwedge x. \alpha \ (f \ x) \leq F \ x$
shows $\alpha \ (cWHILE \ b \ f \ x) \leq aWHILE \ b \ F \ x$
 $\langle proof \rangle$

lemma *transfer-WHILET[refine-transfer]*:
assumes *REF*: $\bigwedge x. \alpha \ (f \ x) \leq F \ x$
shows $\alpha \ (cWHILET \ b \ f \ x) \leq aWHILET \ b \ F \ x$
 $\langle proof \rangle$

end

locale *generic-WHILE-rules* =
generic-WHILE *bind* *return* *WHILEIT* *WHILEI* *WHILET* *WHILE*
for *bind* *return* *SPEC* *WHILEIT* *WHILEI* *WHILET* *WHILE* +
assumes *iSPEC-eq*: $SPEC \ \Phi = Sup \ \{return \ x \mid x. \Phi \ x\}$
assumes *ibind-rule*: $\llbracket M \leq SPEC \ (\lambda x. f \ x \leq SPEC \ \Phi) \rrbracket \implies bind \ M \ f \leq SPEC \ \Phi$
begin

lemma *ireturn-eq*: $return \ x = SPEC \ ((=) \ x)$
 $\langle proof \rangle$

lemma *iSPEC-rule*: $(\bigwedge x. \Phi \ x \implies \Psi \ x) \implies SPEC \ \Phi \leq SPEC \ \Psi$
 $\langle proof \rangle$

lemma *ireturn-rule*: $\Phi \ x \implies return \ x \leq SPEC \ \Phi$
 $\langle proof \rangle$

lemma *WHILEI-rule*:
assumes *I0*: $I \ s$
assumes *ISTEP*: $\bigwedge s. \llbracket I \ s; b \ s \rrbracket \implies f \ s \leq SPEC \ I$
assumes *CONS*: $\bigwedge s. \llbracket I \ s; \neg b \ s \rrbracket \implies \Phi \ s$

shows $WHILEI\ I\ b\ f\ s \leq SPEC\ \Phi$
<proof>

lemma *WHILEIT-rule:*

assumes $WF: wf\ R$
assumes $I0: I\ s$
assumes $IS: \bigwedge s. \llbracket I\ s; b\ s \rrbracket \implies f\ s \leq SPEC\ (\lambda s'. I\ s' \wedge (s',s) \in R)$
assumes $PHI: \bigwedge s. \llbracket I\ s; \neg b\ s \rrbracket \implies \Phi\ s$
shows $WHILEIT\ I\ b\ f\ s \leq SPEC\ \Phi$

<proof>

lemma *WHILE-rule:*

assumes $I0: I\ s$
assumes $ISTEP: \bigwedge s. \llbracket I\ s; b\ s \rrbracket \implies f\ s \leq SPEC\ I$
assumes $CONS: \bigwedge s. \llbracket I\ s; \neg b\ s \rrbracket \implies \Phi\ s$
shows $WHILE\ b\ f\ s \leq SPEC\ \Phi$
<proof>

lemma *WHILET-rule:*

assumes $WF: wf\ R$
assumes $I0: I\ s$
assumes $IS: \bigwedge s. \llbracket I\ s; b\ s \rrbracket \implies f\ s \leq SPEC\ (\lambda s'. I\ s' \wedge (s',s) \in R)$
assumes $PHI: \bigwedge s. \llbracket I\ s; \neg b\ s \rrbracket \implies \Phi\ s$
shows $WHILET\ b\ f\ s \leq SPEC\ \Phi$
<proof>

end

end

2.11 While-Loops

theory *Refine-While*

imports

Refine-Basic Refine-Leaf Generic/RefineG-While

begin

definition $WHILEIT$ ($WHILE_T^-$) **where**

$WHILEIT \equiv iWHILEIT\ bind\ RETURN$

definition $WHILEI$ ($WHILE^-$) **where** $WHILEI \equiv iWHILEI\ bind\ RETURN$

definition $WHILET$ ($WHILE_T$) **where** $WHILET \equiv iWHILET\ bind\ RETURN$

definition $WHILE$ **where** $WHILE \equiv iWHILE\ bind\ RETURN$

interpretation *while?: generic-WHILE-rules bind RETURN SPEC*

$WHILEIT\ WHILEI\ WHILET\ WHILE$

<proof>

lemmas $[refine\text{-}vcg] = \text{WHILEI-rule}$
lemmas $[refine\text{-}vcg] = \text{WHILEIT-rule}$

2.11.1 Data Refinement Rules

lemma *ref-WHILEI-invarI*:
assumes $I\ s \implies M \leq \Downarrow R\ (\text{WHILEI}\ I\ b\ f\ s)$
shows $M \leq \Downarrow R\ (\text{WHILEI}\ I\ b\ f\ s)$
<proof>

lemma *ref-WHILEIT-invarI*:
assumes $I\ s \implies M \leq \Downarrow R\ (\text{WHILEIT}\ I\ b\ f\ s)$
shows $M \leq \Downarrow R\ (\text{WHILEIT}\ I\ b\ f\ s)$
<proof>

lemma *WHILEI-le-rule*:
assumes $R0: (s, s') \in R$
assumes $RS: \bigwedge W\ s\ s'. \llbracket \bigwedge s\ s'. (s, s') \in R \implies W\ s \leq M\ s'; (s, s') \in R \rrbracket \implies$
do $\{\text{ASSERT}\ (I\ s); \text{if}\ b\ s\ \text{then}\ \text{bind}\ (f\ s)\ W\ \text{else}\ \text{RETURN}\ s\} \leq M\ s'$
shows $\text{WHILEI}\ I\ b\ f\ s \leq M\ s'$
<proof>

WHILE-refinement rule with invisible concrete steps. Intuitively, a concrete step may either refine an abstract step, or must not change the corresponding abstract state.

lemma *WHILEI-invisible-refine-genR*:
assumes $R0: I'\ s' \implies (s, s') \in R'$
assumes $RI: \bigwedge s\ s'. \llbracket (s, s') \in R'; I'\ s' \rrbracket \implies I\ s$
assumes $RB: \bigwedge s\ s'. \llbracket (s, s') \in R'; I'\ s'; I\ s; b'\ s' \rrbracket \implies b\ s$
assumes $RS: \bigwedge s\ s'. \llbracket (s, s') \in R'; I'\ s'; I\ s; b\ s \rrbracket$
 $\implies f\ s \leq \text{sup}\ (\Downarrow R'\ (\text{do}\ \{\text{ASSUME}\ (b'\ s'); f'\ s'\}))\ (\Downarrow R'\ (\text{RETURN}\ s'))$
assumes $R\text{-REF}$:
 $\bigwedge x\ x'. \llbracket (x, x') \in R'; \neg b\ x; \neg b'\ x'; I\ x; I'\ x' \rrbracket \implies (x, x') \in R$
shows $\text{WHILEI}\ I\ b\ f\ s \leq \Downarrow R\ (\text{WHILEI}\ I'\ b'\ f'\ s')$
<proof>

lemma *WHILEI-refine-genR*:
assumes $R0: I'\ x' \implies (x, x') \in R'$
assumes $IREF: \bigwedge x\ x'. \llbracket (x, x') \in R'; I'\ x' \rrbracket \implies I\ x$
assumes $COND\text{-REF}: \bigwedge x\ x'. \llbracket (x, x') \in R'; I\ x; I'\ x' \rrbracket \implies b\ x = b'\ x'$
assumes $STEP\text{-REF}$:
 $\bigwedge x\ x'. \llbracket (x, x') \in R'; b\ x; b'\ x'; I\ x; I'\ x' \rrbracket \implies f\ x \leq \Downarrow R'\ (f'\ x')$
assumes $R\text{-REF}$:
 $\bigwedge x\ x'. \llbracket (x, x') \in R'; \neg b\ x; \neg b'\ x'; I\ x; I'\ x' \rrbracket \implies (x, x') \in R$
shows $\text{WHILEI}\ I\ b\ f\ x \leq \Downarrow R\ (\text{WHILEI}\ I'\ b'\ f'\ x')$
<proof>

lemma *WHILE-invisible-refine-genR*:

assumes $R0: (s, s') \in R'$
assumes $RB: \bigwedge s s'. \llbracket (s, s') \in R'; b' s' \rrbracket \implies b s$
assumes $RS: \bigwedge s s'. \llbracket (s, s') \in R'; b s \rrbracket$
 $\implies f s \leq \text{sup} (\Downarrow R' (\text{do} \{ \text{ASSUME} (b' s'); f' s' \})) (\Downarrow R' (\text{RETURN} s'))$
assumes $R\text{-REF}$:
 $\bigwedge x x'. \llbracket (x, x') \in R'; \neg b x; \neg b' x' \rrbracket \implies (x, x') \in R$
shows $\text{WHILE } b f s \leq \Downarrow R (\text{WHILE } b' f' s')$
<proof>

lemma *WHILE-refine-genR*:

assumes $R0: (x, x') \in R'$
assumes $COND\text{-REF}: \bigwedge x x'. \llbracket (x, x') \in R' \rrbracket \implies b x = b' x'$
assumes $STEP\text{-REF}$:
 $\bigwedge x x'. \llbracket (x, x') \in R'; b x; b' x' \rrbracket \implies f x \leq \Downarrow R' (f' x')$
assumes $R\text{-REF}$:
 $\bigwedge x x'. \llbracket (x, x') \in R'; \neg b x; \neg b' x' \rrbracket \implies (x, x') \in R$
shows $\text{WHILE } b f x \leq \Downarrow R (\text{WHILE } b' f' x')$
<proof>

lemma *WHILE-refine-genR'*:

assumes $R0: (x, x') \in R'$
assumes $COND\text{-REF}: \bigwedge x x'. \llbracket (x, x') \in R'; I' x' \rrbracket \implies b x = b' x'$
assumes $STEP\text{-REF}$:
 $\bigwedge x x'. \llbracket (x, x') \in R'; b x; b' x'; I' x' \rrbracket \implies f x \leq \Downarrow R' (f' x')$
assumes $R\text{-REF}$:
 $\bigwedge x x'. \llbracket (x, x') \in R'; \neg b x; \neg b' x' \rrbracket \implies (x, x') \in R$
shows $\text{WHILE } b f x \leq \Downarrow R (\text{WHILEI } I' b' f' x')$
<proof>

WHILE-refinement rule with invisible concrete steps. Intuitively, a concrete step may either refine an abstract step, or must not change the corresponding abstract state.

lemma *WHILEI-invisible-refine*:

assumes $R0: I' s' \implies (s, s') \in R$
assumes $RI: \bigwedge s s'. \llbracket (s, s') \in R; I' s' \rrbracket \implies I s$
assumes $RB: \bigwedge s s'. \llbracket (s, s') \in R; I' s'; I s; b' s' \rrbracket \implies b s$
assumes $RS: \bigwedge s s'. \llbracket (s, s') \in R; I' s'; I s; b s \rrbracket$
 $\implies f s \leq \text{sup} (\Downarrow R (\text{do} \{ \text{ASSUME} (b' s'); f' s' \})) (\Downarrow R (\text{RETURN} s'))$
shows $\text{WHILEI } I b f s \leq \Downarrow R (\text{WHILEI } I' b' f' s')$
<proof>

lemma *WHILEI-refine[refine]*:

assumes $R0: I' x' \implies (x, x') \in R$
assumes $IREF: \bigwedge x x'. \llbracket (x, x') \in R; I' x' \rrbracket \implies I x$
assumes $COND\text{-REF}: \bigwedge x x'. \llbracket (x, x') \in R; I x; I' x' \rrbracket \implies b x = b' x'$
assumes $STEP\text{-REF}$:
 $\bigwedge x x'. \llbracket (x, x') \in R; b x; b' x'; I x; I' x' \rrbracket \implies f x \leq \Downarrow R (f' x')$

shows $WHILEI\ I\ b\ f\ x \leq \Downarrow R\ (WHILEI\ I'\ b'\ f'\ x')$
<proof>

lemma *WHILE-invisible-refine*:

assumes $R0: (s, s') \in R$
assumes $RB: \bigwedge s\ s'. \llbracket (s, s') \in R; b'\ s' \rrbracket \implies b\ s$
assumes $RS: \bigwedge s\ s'. \llbracket (s, s') \in R; b\ s \rrbracket$
 $\implies f\ s \leq \text{sup} (\Downarrow R\ (\text{do}\ \{ASSUME\ (b'\ s');\ f'\ s'\})) (\Downarrow R\ (RETURN\ s'))$
shows $WHILE\ b\ f\ s \leq \Downarrow R\ (WHILE\ b'\ f'\ s')$
<proof>

lemma *WHILE-le-rule*:

assumes $R0: (s, s') \in R$
assumes $RS: \bigwedge W\ s\ s'. \llbracket \bigwedge s\ s'. (s, s') \in R \implies W\ s \leq M\ s'; (s, s') \in R \rrbracket \implies$
 $\text{do}\ \{if\ b\ s\ \text{then}\ \text{bind}\ (f\ s)\ \text{else}\ RETURN\ s\} \leq M\ s'$
shows $WHILE\ b\ f\ s \leq M\ s'$
<proof>

lemma *WHILE-refine[refine]*:

assumes $R0: (x, x') \in R$
assumes $COND-REF: \bigwedge x\ x'. \llbracket (x, x') \in R \rrbracket \implies b\ x = b'\ x'$
assumes $STEP-REF:$
 $\bigwedge x\ x'. \llbracket (x, x') \in R; b\ x; b'\ x' \rrbracket \implies f\ x \leq \Downarrow R\ (f'\ x')$
shows $WHILE\ b\ f\ x \leq \Downarrow R\ (WHILE\ b'\ f'\ x')$
<proof>

lemma *WHILE-refine'[refine]*:

assumes $R0: I'\ x' \implies (x, x') \in R$
assumes $COND-REF: \bigwedge x\ x'. \llbracket (x, x') \in R; I'\ x' \rrbracket \implies b\ x = b'\ x'$
assumes $STEP-REF:$
 $\bigwedge x\ x'. \llbracket (x, x') \in R; b\ x; b'\ x'; I'\ x' \rrbracket \implies f\ x \leq \Downarrow R\ (f'\ x')$
shows $WHILE\ b\ f\ x \leq \Downarrow R\ (WHILEI\ I'\ b'\ f'\ x')$
<proof>

lemma *AIF-leI*: $\llbracket \Phi; \Phi \implies S \leq S' \rrbracket \implies (if\ \Phi\ \text{then}\ S\ \text{else}\ FAIL) \leq S'$
<proof>

lemma *ref-AIFI*: $\llbracket \Phi \implies S \leq \Downarrow R\ S' \rrbracket \implies S \leq \Downarrow R\ (if\ \Phi\ \text{then}\ S'\ \text{else}\ FAIL)$
<proof>

Refinement with generalized refinement relation. Required to exploit the fact that the condition does not hold at the end of the loop.

lemma *WHILEIT-refine-genR*:

assumes $R0: I'\ x' \implies (x, x') \in R'$
assumes $IREF: \bigwedge x\ x'. \llbracket (x, x') \in R'; I'\ x' \rrbracket \implies I\ x$
assumes $COND-REF: \bigwedge x\ x'. \llbracket (x, x') \in R'; I\ x; I'\ x' \rrbracket \implies b\ x = b'\ x'$
assumes $STEP-REF:$
 $\bigwedge x\ x'. \llbracket (x, x') \in R'; b\ x; b'\ x'; I\ x; I'\ x' \rrbracket \implies f\ x \leq \Downarrow R'\ (f'\ x')$
assumes $R-REF:$
 $\bigwedge x\ x'. \llbracket (x, x') \in R'; \neg b\ x; \neg b'\ x'; I\ x; I'\ x' \rrbracket \implies (x, x') \in R$

shows $WHILEIT\ I\ b\ f\ x \leq \Downarrow R$ ($WHILEIT\ I'\ b'\ f'\ x'$)
 ⟨proof⟩

lemma $WHILET-refine-genR$:

assumes $R0: (x, x') \in R'$
 assumes $COND-REF: \bigwedge x\ x'. (x, x') \in R' \implies b\ x = b'\ x'$
 assumes $STEP-REF$:
 $\bigwedge x\ x'. \llbracket (x, x') \in R'; b\ x; b'\ x' \rrbracket \implies f\ x \leq \Downarrow R' (f'\ x')$
 assumes $R-REF$:
 $\bigwedge x\ x'. \llbracket (x, x') \in R'; \neg b\ x; \neg b'\ x' \rrbracket \implies (x, x') \in R$
 shows $WHILET\ b\ f\ x \leq \Downarrow R$ ($WHILET\ b'\ f'\ x'$)
 ⟨proof⟩

lemma $WHILET-refine-genR'$:

assumes $R0: I'\ x' \implies (x, x') \in R'$
 assumes $COND-REF: \bigwedge x\ x'. \llbracket (x, x') \in R'; I'\ x' \rrbracket \implies b\ x = b'\ x'$
 assumes $STEP-REF$:
 $\bigwedge x\ x'. \llbracket (x, x') \in R'; b\ x; b'\ x'; I'\ x' \rrbracket \implies f\ x \leq \Downarrow R' (f'\ x')$
 assumes $R-REF$:
 $\bigwedge x\ x'. \llbracket (x, x') \in R'; \neg b\ x; \neg b'\ x'; I'\ x' \rrbracket \implies (x, x') \in R$
 shows $WHILET\ b\ f\ x \leq \Downarrow R$ ($WHILEIT\ I'\ b'\ f'\ x'$)
 ⟨proof⟩

lemma $WHILEIT-refine[refine]$:

assumes $R0: I'\ x' \implies (x, x') \in R$
 assumes $IREF: \bigwedge x\ x'. \llbracket (x, x') \in R; I'\ x' \rrbracket \implies I\ x$
 assumes $COND-REF: \bigwedge x\ x'. \llbracket (x, x') \in R; I\ x; I'\ x' \rrbracket \implies b\ x = b'\ x'$
 assumes $STEP-REF$:
 $\bigwedge x\ x'. \llbracket (x, x') \in R; b\ x; b'\ x'; I\ x; I'\ x' \rrbracket \implies f\ x \leq \Downarrow R (f'\ x')$
 shows $WHILEIT\ I\ b\ f\ x \leq \Downarrow R$ ($WHILEIT\ I'\ b'\ f'\ x'$)
 ⟨proof⟩

lemma $WHILET-refine[refine]$:

assumes $R0: (x, x') \in R$
 assumes $COND-REF: \bigwedge x\ x'. \llbracket (x, x') \in R \rrbracket \implies b\ x = b'\ x'$
 assumes $STEP-REF$:
 $\bigwedge x\ x'. \llbracket (x, x') \in R; b\ x; b'\ x' \rrbracket \implies f\ x \leq \Downarrow R (f'\ x')$
 shows $WHILET\ b\ f\ x \leq \Downarrow R$ ($WHILET\ b'\ f'\ x'$)
 ⟨proof⟩

lemma $WHILET-refine'[refine]$:

assumes $R0: I'\ x' \implies (x, x') \in R$
 assumes $COND-REF: \bigwedge x\ x'. \llbracket (x, x') \in R; I'\ x' \rrbracket \implies b\ x = b'\ x'$
 assumes $STEP-REF$:
 $\bigwedge x\ x'. \llbracket (x, x') \in R; b\ x; b'\ x'; I'\ x' \rrbracket \implies f\ x \leq \Downarrow R (f'\ x')$
 shows $WHILET\ b\ f\ x \leq \Downarrow R$ ($WHILEIT\ I'\ b'\ f'\ x'$)
 ⟨proof⟩

lemma *WHILEI-refine-new-invar*:

assumes $R0: I' x' \Longrightarrow (x, x') \in R$

assumes $INV0: \llbracket I' x'; (x, x') \in R \rrbracket \Longrightarrow I x$

assumes $COND-REF: \bigwedge x x'. \llbracket (x, x') \in R; I x; I' x' \rrbracket \Longrightarrow b x = b' x'$

assumes *STEP-REF*:

$\bigwedge x x'. \llbracket (x, x') \in R; b x; b' x'; I x; I' x' \rrbracket \Longrightarrow f x \leq \Downarrow R (f' x')$

assumes *STEP-INV*:

$\bigwedge x x'. \llbracket (x, x') \in R; b x; b' x'; I x; I' x'; \text{nofail} (f x) \rrbracket \Longrightarrow f x \leq \text{SPEC } I$

shows $\text{WHILEI } I b f x \leq \Downarrow R (\text{WHILEI } I' b' f' x')$

<proof>

lemma *WHILEIT-refine-new-invar*:

assumes $R0: I' x' \Longrightarrow (x, x') \in R$

assumes $INV0: \llbracket I' x'; (x, x') \in R \rrbracket \Longrightarrow I x$

assumes $COND-REF: \bigwedge x x'. \llbracket (x, x') \in R; I x; I' x' \rrbracket \Longrightarrow b x = b' x'$

assumes *STEP-REF*:

$\bigwedge x x'. \llbracket (x, x') \in R; b x; b' x'; I x; I' x' \rrbracket \Longrightarrow f x \leq \Downarrow R (f' x')$

assumes *STEP-INV*:

$\bigwedge x x'. \llbracket \text{nofail} (f x); (x, x') \in R; b x; b' x'; I x; I' x' \rrbracket \Longrightarrow f x \leq \text{SPEC } I$

shows $\text{WHILEIT } I b f x \leq \Downarrow R (\text{WHILEIT } I' b' f' x')$

<proof>

2.11.2 Autoref Setup

context begin interpretation *autoref-syn* *<proof>*

lemma [*autoref-op-pat-def*]:

$\text{WHILEIT } I \equiv \text{OP } (\text{WHILEIT } I)$

$\text{WHILEI } I \equiv \text{OP } (\text{WHILEI } I)$

<proof>

lemma *autoref-WHILET*[*autoref-rules*]:

assumes $\bigwedge x x'. \llbracket (x, x') \in R \rrbracket \Longrightarrow (c x, c' \$ x') \in \text{Id}$

assumes $\bigwedge x x'. \llbracket \text{REMOVE-INTERNAL } c' x'; (x, x') \in R \rrbracket$
 $\Longrightarrow (f x, f' \$ x') \in \langle R \rangle \text{nres-rel}$

assumes $(s, s') \in R$

shows $(\text{WHILET } c f s,$

$(\text{OP } \text{WHILET} ::: (R \rightarrow \text{Id}) \rightarrow (R \rightarrow \langle R \rangle \text{nres-rel}) \rightarrow R \rightarrow \langle R \rangle \text{nres-rel}) \$ c' \$ f' \$ s'$

$\in \langle R \rangle \text{nres-rel}$

<proof>

lemma *autoref-WHILEIT*[*autoref-rules*]:

assumes $\bigwedge x x'. \llbracket I x'; (x, x') \in R \rrbracket \Longrightarrow (c x, c' \$ x') \in \text{Id}$

assumes $\bigwedge x x'. \llbracket \text{REMOVE-INTERNAL } c' x'; I x'; (x, x') \in R \rrbracket \Longrightarrow (f x, f' \$ x') \in$
 $\langle R \rangle \text{nres-rel}$

assumes $I s' \Longrightarrow (s, s') \in R$

shows $(\text{WHILET } c f s,$

$(\text{OP } (\text{WHILEIT } I) ::: (R \rightarrow \text{Id}) \rightarrow (R \rightarrow \langle R \rangle \text{nres-rel}) \rightarrow R \rightarrow \langle R \rangle \text{nres-rel}) \$ c' \$ f' \$ s'$

) $\in\langle R\rangle nres\text{-}rel$
 <proof>

lemma *autoref-WHILEIT'*[*autoref-rules*]:

assumes $\bigwedge x x'. \llbracket (x,x')\in R; I x \rrbracket \implies (c x, c'\$x') \in Id$
assumes $\bigwedge x x'. \llbracket REMOVE\text{-}INTERNAL\ c' x'; (x,x')\in R; I x \rrbracket$
 $\implies (f x, f'\$x') \in \langle R\rangle nres\text{-}rel$
shows (*WHILEIT* $c\ f$,
 (*OP* (*WHILEIT* I) $::: (R\rightarrow Id) \rightarrow (R\rightarrow\langle R\rangle nres\text{-}rel) \rightarrow R \rightarrow \langle R\rangle nres\text{-}rel$) $\$c'\f')
) $\in R \rightarrow \langle R\rangle nres\text{-}rel$
 <proof>

lemma *autoref-WHILE*[*autoref-rules*]:

assumes $\bigwedge x x'. \llbracket (x,x')\in R \rrbracket \implies (c x, c'\$x') \in Id$
assumes $\bigwedge x x'. \llbracket REMOVE\text{-}INTERNAL\ c' x'; (x,x')\in R \rrbracket$
 $\implies (f x, f'\$x') \in \langle R\rangle nres\text{-}rel$
assumes $(s, s')\in R$
shows (*WHILE* $c\ f\ s$,
 (*OP* *WHILE* $::: (R\rightarrow Id) \rightarrow (R\rightarrow\langle R\rangle nres\text{-}rel) \rightarrow R \rightarrow \langle R\rangle nres\text{-}rel$) $\$c'\$f'\$s'$)
) $\in\langle R\rangle nres\text{-}rel$
 <proof>

lemma *autoref-WHILE'*[*autoref-rules*]:

assumes $\bigwedge x x'. \llbracket (x,x')\in R \rrbracket \implies (c x, c'\$x') \in Id$
assumes $\bigwedge x x'. \llbracket REMOVE\text{-}INTERNAL\ c' x'; (x,x')\in R \rrbracket$
 $\implies (f x, f'\$x') \in \langle R\rangle nres\text{-}rel$
shows (*WHILE* $c\ f$,
 (*OP* *WHILE* $::: (R\rightarrow Id) \rightarrow (R\rightarrow\langle R\rangle nres\text{-}rel) \rightarrow R \rightarrow \langle R\rangle nres\text{-}rel$) $\$c'\f')
) $\in R \rightarrow \langle R\rangle nres\text{-}rel$
 <proof>

lemma *autoref-WHILEI*[*autoref-rules*]:

assumes $\bigwedge x x'. \llbracket I x'; (x,x')\in R \rrbracket \implies (c x, c'\$x') \in Id$
assumes $\bigwedge x x'. \llbracket REMOVE\text{-}INTERNAL\ c' x'; I x'; (x,x')\in R \rrbracket \implies (f x, f'\$x') \in \langle R\rangle nres\text{-}rel$
assumes $I s' \implies (s, s')\in R$
shows (*WHILEI* $c\ f\ s$,
 (*OP* (*WHILEI* I) $::: (R\rightarrow Id) \rightarrow (R\rightarrow\langle R\rangle nres\text{-}rel) \rightarrow R \rightarrow \langle R\rangle nres\text{-}rel$) $\$c'\$f'\$s'$)
) $\in\langle R\rangle nres\text{-}rel$
 <proof>

lemma *autoref-WHILEI'*[*autoref-rules*]:

assumes $\bigwedge x x'. \llbracket (x,x')\in R; I x \rrbracket \implies (c x, c'\$x') \in Id$
assumes $\bigwedge x x'. \llbracket REMOVE\text{-}INTERNAL\ c' x'; (x,x')\in R; I x \rrbracket$
 $\implies (f x, f'\$x') \in \langle R\rangle nres\text{-}rel$
shows (*WHILEI* $c\ f$,
 (*OP* (*WHILEI* I) $::: (R\rightarrow Id) \rightarrow (R\rightarrow\langle R\rangle nres\text{-}rel) \rightarrow R \rightarrow \langle R\rangle nres\text{-}rel$) $\$c'\f')
) $\in R \rightarrow \langle R\rangle nres\text{-}rel$

<proof>

end

2.11.3 Invariants

Tail Recursion

context begin

private lemma *tailrec-transform-aux1*:

assumes $RETURN\ s \leq m$

shows $REC\ (\lambda D\ s.\ sup\ (a\ s \gg D)\ (b\ s))\ s \leq lfp\ (\lambda x.\ sup\ m\ (x \gg a)) \gg b$
 (is $REC\ ?F\ s \leq lfp\ ?f \gg b$)

<proof> corollary *tailrec-transform1*:

fixes $m :: 'a\ nres$

shows $m \gg REC\ (\lambda D\ s.\ sup\ (a\ s \gg D)\ (b\ s)) \leq lfp\ (\lambda x.\ sup\ m\ (x \gg a)) \gg b$

b

<proof> lemma *tailrec-transform-aux2*:

fixes $m :: 'a\ nres$

shows $lfp\ (\lambda x.\ sup\ m\ (x \gg a))$
 $\leq m \gg REC\ (\lambda D\ s.\ sup\ (a\ s \gg D)\ (RETURN\ s))$
 (is $lfp\ ?f \leq m \gg REC\ ?F$)

<proof> lemma *tailrec-transform-aux3*: $REC\ (\lambda D\ s.\ sup\ (a\ s \gg D)\ (RETURN\ s))\ s \gg b$

$\leq REC\ (\lambda D\ s.\ sup\ (a\ s \gg D)\ (b\ s))\ s$

<proof> lemma *tailrec-transform2*:

$lfp\ (\lambda x.\ sup\ m\ (bind\ x\ a)) \gg b \leq m \gg REC\ (\lambda D\ s.\ sup\ (a\ s \gg D)\ (b\ s))$

<proof>

definition *tailrec-body* $a\ b \equiv (\lambda D\ s.\ sup\ (bind\ (a\ s)\ D)\ (b\ s))$

theorem *tailrec-transform*:

$m \gg REC\ (\lambda D\ s.\ sup\ (a\ s \gg D)\ (b\ s)) = lfp\ (\lambda x.\ sup\ m\ (bind\ x\ a)) \gg b$
<proof>

theorem *tailrec-transform'*:

$m \gg REC\ (tailrec-body\ a\ b) = lfp\ (\lambda x.\ sup\ m\ (bind\ x\ a)) \gg b$
<proof>

lemma *WHILE* $c\ f =$

$REC\ (tailrec-body$
 $(\lambda s.\ do\ \{ASSUME\ (c\ s);\ f\ s\})$
 $(\lambda s.\ do\ \{ASSUME\ (\neg c\ s);\ RETURN\ s\})$
 $)$

<proof>

lemma *WHILEI-tailrec-conv*: $WHILEI\ I\ c\ f =$

REC (*tailrec-body*
 $(\lambda s. \text{do } \{ \text{ASSERT } (I \ s); \text{ ASSUME } (c \ s); f \ s \})$
 $(\lambda s. \text{do } \{ \text{ASSERT } (I \ s); \text{ ASSUME } (\neg c \ s); \text{ RETURN } s \})$
 \rangle
 $\langle \text{proof} \rangle$)

lemma *WHILEIT-tailrec-conv*: $\text{WHILEIT } I \ c \ f =$
 $\text{RECT } (\text{tailrec-body}$
 $(\lambda s. \text{do } \{ \text{ASSERT } (I \ s); \text{ ASSUME } (c \ s); f \ s \})$
 $(\lambda s. \text{do } \{ \text{ASSERT } (I \ s); \text{ ASSUME } (\neg c \ s); \text{ RETURN } s \})$
 \rangle
 $\langle \text{proof} \rangle$)

definition *WHILEI-lfp-body* $I \ m \ c \ f \equiv$
 $(\lambda x. \text{sup } m \ (\text{do } \{$
 $\quad s \leftarrow x;$
 $\quad - \leftarrow \text{ASSERT } (I \ s);$
 $\quad - \leftarrow \text{ASSUME } (c \ s);$
 $\quad f \ s$
 $\quad \}))$

lemma *WHILEI-lfp-conv*: $m \gg= \text{WHILEI } I \ c \ f =$
 $\text{do } \{$
 $\quad s \leftarrow \text{lfp } (\text{WHILEI-lfp-body } I \ m \ c \ f);$
 $\quad \text{ASSERT } (I \ s);$
 $\quad \text{ASSUME } (\neg c \ s);$
 $\quad \text{RETURN } s$
 $\}$
 $\langle \text{proof} \rangle$

end

Most Specific Invariant

definition *msii* — Most specific invariant for WHILE-loop
where $\text{msii } I \ m \ c \ f \equiv \text{lfp } (\text{WHILEI-lfp-body } I \ m \ c \ f)$

lemma [*simp, intro!*]: $\text{mono } (\text{WHILEI-lfp-body } I \ m \ c \ f)$
 $\langle \text{proof} \rangle$

definition *filter-ASSUME* $c \ m \equiv \text{do } \{ x \leftarrow m; \text{ASSUME } (c \ x); \text{RETURN } x \}$

definition *filter-ASSERT* $c \ m \equiv \text{do } \{ x \leftarrow m; \text{ASSERT } (c \ x); \text{RETURN } x \}$

lemma [*refine-pw-simps*]: $\text{nofail } (\text{filter-ASSUME } c \ m) \longleftrightarrow \text{nofail } m$
 $\langle \text{proof} \rangle$

lemma [*refine-pw-simps*]: $\text{inres } (\text{filter-ASSUME } c \ m) \ x$
 $\longleftrightarrow (\text{nofail } m \longrightarrow \text{inres } m \ x \wedge c \ x)$

<proof>

lemma *msii-is-invar*:

$m \leq msii\ I\ m\ c\ f$
 $m \leq msii\ I\ m\ c\ f \implies bind\ (filter\text{-}ASSUME\ c\ (filter\text{-}ASSERT\ I\ m))\ f \leq msii\ I\ m\ c\ f$
<proof>

lemma *WHILE-msii-conv*: $m \gg= WHILEI\ I\ c\ f$

$= filter\text{-}ASSUME\ (Not\ o\ c)\ (filter\text{-}ASSERT\ I\ (msii\ I\ m\ c\ f))$
<proof>

lemma *msii-induct*:

assumes $I0$: $m0 \leq P$
assumes IS : $\bigwedge m. \llbracket m \leq msii\ I\ m0\ c\ f; m \leq P;$
 $filter\text{-}ASSUME\ c\ (filter\text{-}ASSERT\ I\ m) \gg= f \leq msii\ I\ m0\ c\ f$
 $\rrbracket \implies filter\text{-}ASSUME\ c\ (filter\text{-}ASSERT\ I\ m) \gg= f \leq P$
shows $msii\ I\ m0\ c\ f \leq P$
<proof>

Reachable without fail

Reachable states in a while loop, ignoring failing states

inductive *rwof* :: $'a\ nres \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a\ nres) \Rightarrow 'a \Rightarrow bool$
for $m0\ cond\ step$
where
 $init$: $\llbracket m0 = RES\ X; x \in X \rrbracket \implies rwof\ m0\ cond\ step\ x$
 $| step$: $\llbracket rwof\ m0\ cond\ step\ x; cond\ x; step\ x = RES\ Y; y \in Y \rrbracket$
 $\implies rwof\ m0\ cond\ step\ y$

lemma *establish-rwof-invar*:

assumes I : $m0 \leq_n SPEC\ I$
assumes S : $\bigwedge s. \llbracket rwof\ m0\ cond\ step\ s; I\ s; cond\ s \rrbracket$
 $\implies step\ s \leq_n SPEC\ I$
assumes $rwof\ m0\ cond\ step\ s$
shows $I\ s$
<proof>

definition *is-rwof-invar* $m0\ cond\ step\ I \equiv$

$(m0 \leq_n SPEC\ I)$
 $\wedge (\forall s. rwof\ m0\ cond\ step\ s \wedge I\ s \wedge cond\ s$
 $\longrightarrow step\ s \leq_n SPEC\ I)$

lemma *is-rwof-invarI*[*intro?*]:

assumes I : $m0 \leq_n SPEC\ I$
assumes S : $\bigwedge s. \llbracket rwof\ m0\ cond\ step\ s; I\ s; cond\ s \rrbracket$
 $\implies step\ s \leq_n SPEC\ I$

shows *is-rwof-invar* $m0$ *cond* *step* I
 $\langle proof \rangle$

lemma *rwof-cons*: $\llbracket is-rwof-invar\ m0\ cond\ step\ I; rwof\ m0\ cond\ step\ s \rrbracket \implies I\ s$
 $\langle proof \rangle$

lemma *rwof-WHILE-rule*:

assumes $I0$: $m0 \leq SPEC\ I$

assumes S : $\bigwedge s. \llbracket rwof\ m0\ cond\ step\ s; I\ s; cond\ s \rrbracket \implies step\ s \leq SPEC\ I$

shows $m0 \ggg WHILE\ cond\ step \leq SPEC\ (\lambda s. rwof\ m0\ cond\ step\ s \wedge \neg cond\ s \wedge I\ s)$
 $\langle proof \rangle$

Filtering out states that satisfy the loop condition

definition *filter-nb* :: $('a \Rightarrow bool) \Rightarrow 'a\ nres \Rightarrow 'a\ nres$ **where**
 $filter-nb\ b\ I \equiv do\ \{s \leftarrow I; ASSUME\ (\neg b\ s); RETURN\ s\}$

lemma *pw-filter-nb[refine-pw-simps]*:

$nofail\ (filter-nb\ b\ I) \longleftrightarrow nofail\ I$

$inres\ (filter-nb\ b\ I)\ x \longleftrightarrow (nofail\ I \longrightarrow inres\ I\ x \wedge \neg b\ x)$

$\langle proof \rangle$

lemma *filter-nb-mono*: $m \leq m' \implies filter-nb\ cond\ m \leq filter-nb\ cond\ m'$
 $\langle proof \rangle$

lemma *filter-nb-cont*:

$filter-nb\ cond\ (Sup\ M) = Sup\ \{filter-nb\ cond\ m \mid m. m \in M\}$

$\langle proof \rangle$

lemma *filter-nb-FAIL[simp]*: $filter-nb\ cond\ FAIL = FAIL$

$\langle proof \rangle$

lemma *filter-nb-RES[simp]*: $filter-nb\ cond\ (RES\ X) = RES\ \{x \in X. \neg cond\ x\}$

$\langle proof \rangle$

Bounded while-loop

lemma *WHILE-rule-gen-le*:

assumes $I0$: $m0 \leq I$

assumes $ISTEP$: $\bigwedge s. \llbracket RETURN\ s \leq I; b\ s \rrbracket \implies f\ s \leq I$

shows $m0 \ggg WHILE\ b\ f \leq filter-nb\ b\ I$

$\langle proof \rangle$

primrec *bounded-WHILE'*

:: $nat \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a\ nres) \Rightarrow 'a\ nres \Rightarrow 'a\ nres$

where

```

  bounded-WHILE' 0 cond step m = m
| bounded-WHILE' (Suc n) cond step m = do {
  x ← m;
  if cond x then bounded-WHILE' n cond step (step x)
  else RETURN x
}

```

primrec bounded-WHILE

```

:: nat ⇒ ('a ⇒ bool) ⇒ ('a ⇒ 'a nres) ⇒ 'a nres ⇒ 'a nres

```

where

```

  bounded-WHILE 0 cond step m = m
| bounded-WHILE (Suc n) cond step m = do {
  x ← bounded-WHILE n cond step m;
  if cond x then step x
  else RETURN x
}

```

lemma bounded-WHILE-shift: do {

```

  x ← m;
  if cond x then bounded-WHILE n cond step (step x) else RETURN x
} = do {
  x ← bounded-WHILE n cond step m;
  if cond x then step x else RETURN x
}

```

⟨proof⟩

lemma bounded-WHILE'-eq:

```

bounded-WHILE' n cond step m = bounded-WHILE n cond step m
⟨proof⟩

```

lemma mWHILE-unfold: $m \gg= \text{WHILE cond step} = \text{do } \{$

```

  x ← m;
  if cond x then step x ≫= WHILE cond step
  else RETURN x
}

```

⟨proof⟩

lemma WHILE-bounded-aux1:

```

filter-nb cond (bounded-WHILE n cond step m) ≤ m ≫= WHILE cond step
⟨proof⟩

```

lemma WHILE-bounded-aux2:

```

m ≫= WHILE cond step
≤ filter-nb cond (Sup {bounded-WHILE n cond step m | n. True})
⟨proof⟩

```

lemma WHILE-bounded:

$m \gg= \text{WHILE } \text{cond } \text{step}$
 $= \text{filter-nb } \text{cond } (\text{Sup } \{\text{bounded-WHILE } n \text{ cond } \text{step } m \mid n. \text{True}\})$
 $\langle \text{proof} \rangle$

Relation to rwof

lemma *rwof-in-bounded-WHILE*:
assumes *rwof m0 cond step s*
shows $\exists n. \text{RETURN } s \leq (\text{bounded-WHILE } n \text{ cond } \text{step } m0)$
 $\langle \text{proof} \rangle$

lemma *bounded-WHILE-FAIL-rwof*:
assumes *bounded-WHILE n cond step m0 = FAIL*
assumes *M0: m0 ≠ FAIL*
shows $\exists n' < n. \exists x X.$
 $\text{bounded-WHILE } n' \text{ cond } \text{step } m0 = \text{RES } X$
 $\wedge x \in X \wedge \text{cond } x \wedge \text{step } x = \text{FAIL}$
 $\langle \text{proof} \rangle$

lemma *bounded-WHILE-RES-rwof*:
assumes *bounded-WHILE n cond step m0 = RES X*
assumes *x ∈ X*
shows *rwof m0 cond step x*
 $\langle \text{proof} \rangle$

lemma *rwof-FAIL-imp-WHILE-FAIL*:
assumes *RW: rwof m0 cond step s*
and *C: cond s*
and *S: step s = FAIL*
shows $m0 \gg= \text{WHILE } \text{cond } \text{step} = \text{FAIL}$
 $\langle \text{proof} \rangle$

lemma *pw-bounded-WHILE-RES-rwof*: $\llbracket \text{nofail } (\text{bounded-WHILE } n \text{ cond } \text{step } m0);$
 $\text{inres } (\text{bounded-WHILE } n \text{ cond } \text{step } m0) x \rrbracket \implies \text{rwof } m0 \text{ cond } \text{step } x$
 $\langle \text{proof} \rangle$

corollary *WHILE-nofail-imp-rwof-nofail*:
assumes *nofail (m0 ≫= WHILE cond step)*
assumes *RW: rwof m0 cond step s*
assumes *C: cond s*
shows *nofail (step s)*
 $\langle \text{proof} \rangle$

lemma *WHILE-le-WHILEI*: $\text{WHILE } b \text{ f } s \leq \text{WHILEI } I \text{ b } f \text{ s}$
 $\langle \text{proof} \rangle$

corollary *WHILEI-nofail-imp-rwof-nofail*:
assumes NF : *nofail* ($m0 \ggg \text{WHILEI } I \text{ cond step}$)
assumes RW : *rwof* $m0 \text{ cond step } s$
assumes C : *cond* s
shows *nofail* (*step* s)
<proof>

corollary *WHILET-nofail-imp-rwof-nofail*:
assumes NF : *nofail* ($m0 \ggg \text{WHILET cond step}$)
assumes RW : *rwof* $m0 \text{ cond step } s$
assumes C : *cond* s
shows *nofail* (*step* s)
<proof>

corollary *WHILEIT-nofail-imp-rwof-nofail*:
assumes NF : *nofail* ($m0 \ggg \text{WHILEIT } I \text{ cond step}$)
assumes RW : *rwof* $m0 \text{ cond step } s$
assumes C : *cond* s
shows *nofail* (*step* s)
<proof>

lemma *pw-rwof-in-bounded-WHILE*:
rwof $m0 \text{ cond step } x \implies \exists n. \text{inres} (\text{bounded-WHILE } n \text{ cond step } m0) x$
<proof>

WHILE-loops in the nofail-case

lemma *nofail-WHILE-eq-rwof*:
assumes NF : *nofail* ($m0 \ggg \text{WHILE cond step}$)
shows $m0 \ggg \text{WHILE cond step} = \text{SPEC} (\lambda s. \text{rwof } m0 \text{ cond step } s \wedge \neg \text{cond } s)$
<proof>

lemma *WHILE-refine-rwof*:
assumes *nofail* ($m \ggg \text{WHILE } c f$) $\implies mi \leq \text{SPEC} (\lambda s. \text{rwof } m \text{ } c f s \wedge \neg c$
 $s)$
shows $mi \leq m \ggg \text{WHILE } c f$
<proof>

lemma *pw-rwof-init*:
assumes NF : *nofail* ($m0 \ggg \text{WHILE cond step}$)
shows *inres* $m0 s \implies \text{rwof } m0 \text{ cond step } s$ **and** *nofail* $m0$
<proof>

lemma *rwof-init*:
assumes NF : *nofail* ($m0 \ggg \text{WHILE cond step}$)
shows $m0 \leq \text{SPEC} (\text{rwof } m0 \text{ cond step})$
<proof>

lemma *pw-rwof-step'*:
assumes *NF*: *nofail (step s)*
assumes *R*: *rwof m0 cond step s*
assumes *C*: *cond s*
shows *inres (step s) s' \implies rwof m0 cond step s'*
 \langle *proof* \rangle

lemma *rwof-step'*:
 \llbracket *nofail (step s); rwof m0 cond step s; cond s* \rrbracket
 \implies *step s \leq SPEC (rwof m0 cond step)*
 \langle *proof* \rangle

lemma *pw-rwof-step*:
assumes *NF*: *nofail (m0 \ggg WHILE cond step)*
assumes *R*: *rwof m0 cond step s*
assumes *C*: *cond s*
shows *inres (step s) s' \implies rwof m0 cond step s'*
and *nofail (step s)*
 \langle *proof* \rangle

lemma *rwof-step*:
 \llbracket *nofail (m0 \ggg WHILE cond step); rwof m0 cond step s; cond s* \rrbracket
 \implies *step s \leq SPEC (rwof m0 cond step)*
 \langle *proof* \rangle

lemma (**in** $-$) *rwof-leof-init*: *m \leq_n SPEC (rwof m c f)*
 \langle *proof* \rangle

lemma (**in** $-$) *rwof-leof-step*: \llbracket *rwof m c f s; c s* $\rrbracket \implies$ *f s \leq_n SPEC (rwof m c f)*
 \langle *proof* \rangle

lemma (**in** $-$) *rwof-step-refine*:
assumes *NF*: *nofail (m0 \ggg WHILE cond step)*
assumes *A*: *rwof m0 cond step' s*
assumes *FR*: $\bigwedge s. \llbracket$ *rwof m0 cond step s; cond s* $\rrbracket \implies$ *step' s \leq_n step s*
shows *rwof m0 cond step s*
 \langle *proof* \rangle

Adding Invariants

lemma *WHILE-eq-I-rwof*: *m \ggg WHILE c f = m \ggg WHILEI (rwof m c f) c f*
 \langle *proof* \rangle

lemma *WHILET-eq-I-rwof*: *m \ggg WHILET c f = m \ggg WHILEIT (rwof m c f) c f*
 \langle *proof* \rangle

Refinement**lemma** *rwof-refine*:**assumes** *RW*: *rwof m c f s***assumes** *NF*: *nofail (m' >> WHILE c' f')***assumes** *M*: $m \leq_n \Downarrow R m'$ **assumes** *C*: $\bigwedge s s'. \llbracket (s, s') \in R; \text{rwof } m \text{ c f s}; \text{rwof } m' \text{ c' f' s'} \rrbracket \implies c s = c' s'$ **assumes** *S*: $\bigwedge s s'. \llbracket (s, s') \in R; \text{rwof } m \text{ c f s}; \text{rwof } m' \text{ c' f' s'}; c s; c' s' \rrbracket \implies f s \leq_n \Downarrow R (f' s')$ **shows** $\exists s'. (s, s') \in R \wedge \text{rwof } m' \text{ c' f' s'}$ *<proof>***Total Correct Loops**

In this theory, we show that every non-failing total-correct while loop gives rise to a compatible well-founded relation

definition *rwof-rel*

— Transitions in a while-loop as relation

where *rwof-rel init cond step* $\equiv \{(s, s'). \text{rwof init cond step } s \wedge \text{cond } s \wedge \text{inres (step } s) \text{ s'}\}$ **lemma** *rwof-rel-spec*: $\llbracket \text{rwof init cond step } s; \text{cond } s \rrbracket$ $\implies \text{step } s \leq_n \text{SPEC } (\lambda s'. (s, s') \in \text{rwof-rel init cond step})$ *<proof>***lemma** *rwof-reachable*:**assumes** *rwof init cond step s***shows** $\exists s0. \text{inres init } s0 \wedge (s0, s) \in (\text{rwof-rel init cond step})^*$ *<proof>***theorem** *nofail-WHILEIT-wf-rel*:**assumes** *NF*: *nofail (init >> WHILEIT I cond step)***shows** *wf ((rwof-rel init cond step)⁻¹)**<proof>***2.11.4 Convenience****Iterate over range of list****lemma** *range-set-WHILE*:**assumes** *CEQ*: $\bigwedge i s. c (i, s) \longleftrightarrow i < u$ **assumes** *F0*: $F \{\} s0 = s0$ **assumes** *Fs*: $\bigwedge s i X. \llbracket l \leq i; i < u \rrbracket$ $\implies f (i, (F X s)) \leq \text{SPEC } (\lambda (i', r). i' = i + 1 \wedge r = F (\text{insert (list!} i) X) s)$ **shows** *WHILET c f (l, s0)* $\leq \text{SPEC } (\lambda (-, r). r = F \{\text{list!} i \mid i. l \leq i \wedge i < u\} s0)$ *<proof>*

end

2.12 Deterministic Monad

```

theory Refine-Det
imports
  HOL-Library.Monad-Syntax
  Generic/RefineG-Assert
  Generic/RefineG-While
begin

```

2.12.1 Deterministic Result Lattice

We define the flat complete lattice of deterministic program results:

```

datatype 'a dres =
  dSUCCEEDi — No result
| dFAILi — Failure
| dRETURN 'a — Regular result

instantiation dres :: (type) complete-lattice
begin
  definition top-dres  $\equiv$  dFAILi
  definition bot-dres  $\equiv$  dSUCCEEDi
  fun sup-dres where
    sup dFAILi - = dFAILi |
    sup - dFAILi = dFAILi |
    sup (dRETURN a) (dRETURN b) = (if a=b then dRETURN b else dFAILi) |
    sup dSUCCEEDi x = x |
    sup x dSUCCEEDi = x

  lemma sup-dres-addsimps[simp]:
    sup x dFAILi = dFAILi
    sup x dSUCCEEDi = x
    <proof>

  fun inf-dres where
    inf dFAILi x = x |
    inf x dFAILi = x |
    inf (dRETURN a) (dRETURN b) = (if a=b then dRETURN b else dSUC-
CEEDi) |
    inf dSUCCEEDi - = dSUCCEEDi |
    inf - dSUCCEEDi = dSUCCEEDi

  lemma inf-dres-addsimps[simp]:
    inf x dSUCCEEDi = dSUCCEEDi

```

$\text{inf } dSUCCEEDi \ x = dSUCCEEDi$
 $\text{inf } x \ dFAILi = x$
 $\text{inf } (dRETURN \ v) \ x \neq dFAILi$
 ⟨proof⟩

definition *Sup-dres* $S \equiv$
 if $S \subseteq \{dSUCCEEDi\}$ then $dSUCCEEDi$
 else if $dFAILi \in S$ then $dFAILi$
 else if $\exists a \ b. a \neq b \wedge dRETURN \ a \in S \wedge dRETURN \ b \in S$ then $dFAILi$
 else $dRETURN \ (THE \ x. dRETURN \ x \in S)$

definition *Inf-dres* $S \equiv$
 if $S \subseteq \{dFAILi\}$ then $dFAILi$
 else if $dSUCCEEDi \in S$ then $dSUCCEEDi$
 else if $\exists a \ b. a \neq b \wedge dRETURN \ a \in S \wedge dRETURN \ b \in S$ then $dSUCCEEDi$
 else $dRETURN \ (THE \ x. dRETURN \ x \in S)$

fun *less-eq-dres* **where**
 $\text{less-eq-dres } dSUCCEEDi \ - \longleftrightarrow \ True \ |$
 $\text{less-eq-dres } \ - \ dFAILi \longleftrightarrow \ True \ |$
 $\text{less-eq-dres } (dRETURN \ (a::'a)) \ (dRETURN \ b) \longleftrightarrow \ a=b \ |$
 $\text{less-eq-dres } \ - \ - \longleftrightarrow \ False$

definition *less-dres* **where** $\text{less-dres } (a::'a \ \text{dres}) \ b \longleftrightarrow a \leq b \wedge \neg b \leq a$

lemma *less-eq-dres-split-conv*:
 $a \leq b \longleftrightarrow (\text{case } (a,b) \ \text{of}$
 $(dSUCCEEDi, -) \Rightarrow \ True$
 $| \ (-, dFAILi) \Rightarrow \ True$
 $| \ (dRETURN \ (a::'a), dRETURN \ b) \Rightarrow \ a=b$
 $| \ - \Rightarrow \ False$
 $)$
 ⟨proof⟩

lemma *inf-dres-split-conv*:
 $\text{inf } a \ b = (\text{case } (a,b) \ \text{of}$
 $(dFAILi, x) \Rightarrow \ x$
 $| \ (x, dFAILi) \Rightarrow \ x$
 $| \ (dRETURN \ a, dRETURN \ b) \Rightarrow \ (\text{if } a=b \ \text{then } dRETURN \ b \ \text{else } dSUCCEEDi)$
 $| \ - \Rightarrow \ dSUCCEEDi)$
 ⟨proof⟩

lemma *sup-dres-split-conv*:
 $\text{sup } a \ b = (\text{case } (a,b) \ \text{of}$
 $(dSUCCEEDi, x) \Rightarrow \ x$
 $| \ (x, dSUCCEEDi) \Rightarrow \ x$
 $| \ (dRETURN \ a, dRETURN \ b) \Rightarrow \ (\text{if } a=b \ \text{then } dRETURN \ b \ \text{else } dFAILi)$
 $| \ - \Rightarrow \ dFAILi)$
 ⟨proof⟩

instance
 ⟨*proof*⟩

end

abbreviation $dSUCCEED \equiv (bot::'a \text{ dres})$

abbreviation $dFAIL \equiv (top::'a \text{ dres})$

lemma *dres-cases*[*cases type, case-names dSUCCEED dRETURN dFAIL*]:

obtains $x=dSUCCEED \mid r \text{ where } x=dRETURN \ r \mid \ x=dFAIL$
 ⟨*proof*⟩

lemmas [*simp*] = *dres.case(1,2)[folded top-dres-def bot-dres-def]*

lemma *dres-order-simps*[*simp*]:

$x \leq dSUCCEED \longleftrightarrow x = dSUCCEED$

$dFAIL \leq x \longleftrightarrow x = dFAIL$

$dRETURN \ r \neq dFAIL$

$dRETURN \ r \neq dSUCCEED$

$dFAIL \neq dRETURN \ r$

$dSUCCEED \neq dRETURN \ r$

$dFAIL \neq dSUCCEED$

$dSUCCEED \neq dFAIL$

$x=y \implies \text{inf } (dRETURN \ x) \ (dRETURN \ y) = dRETURN \ y$

$x \neq y \implies \text{inf } (dRETURN \ x) \ (dRETURN \ y) = dSUCCEED$

$x=y \implies \text{sup } (dRETURN \ x) \ (dRETURN \ y) = dRETURN \ y$

$x \neq y \implies \text{sup } (dRETURN \ x) \ (dRETURN \ y) = dFAIL$

⟨*proof*⟩

lemma *dres-Sup-cases*:

obtains $S \subseteq \{dSUCCEED\}$ **and** $\text{Sup } S = dSUCCEED$

$\mid dFAIL \in S$ **and** $\text{Sup } S = dFAIL$

$\mid a \ b \text{ where } a \neq b \quad dRETURN \ a \in S \quad dRETURN \ b \in S \quad dFAIL \notin S \quad \text{Sup } S = dFAIL$

$\mid a \text{ where } S \subseteq \{dSUCCEED, dRETURN \ a\} \quad dRETURN \ a \in S \quad \text{Sup } S = dRETURN \ a$

⟨*proof*⟩

lemma *dres-Inf-cases*:

obtains $S \subseteq \{dFAIL\}$ **and** $\text{Inf } S = dFAIL$

$\mid dSUCCEED \in S$ **and** $\text{Inf } S = dSUCCEED$

$\mid a \ b \text{ where } a \neq b \quad dRETURN \ a \in S \quad dRETURN \ b \in S \quad dSUCCEED \notin S \quad \text{Inf } S = dSUCCEED$

$\mid a \text{ where } S \subseteq \{dFAIL, dRETURN \ a\} \quad dRETURN \ a \in S \quad \text{Inf } S = dRETURN \ a$

⟨*proof*⟩

lemma *dres-chain-eq-res*:

is-chain $M \implies$
 $dRETURN\ r \in M \implies dRETURN\ s \in M \implies r=s$
 ⟨*proof*⟩

lemma *dres-Sup-chain-cases*:

assumes *CHAIN*: *is-chain* M
obtains $M \subseteq \{dSUCCEED\}$ $Sup\ M = dSUCCEED$
 | r **where** $M \subseteq \{dSUCCEED, dRETURN\ r\}$ $dRETURN\ r \in M$ $Sup\ M =$
 $dRETURN\ r$
 | $dFAIL \in M$ $Sup\ M = dFAIL$
 ⟨*proof*⟩

lemma *dres-Inf-chain-cases*:

assumes *CHAIN*: *is-chain* M
obtains $M \subseteq \{dFAIL\}$ $Inf\ M = dFAIL$
 | r **where** $M \subseteq \{dFAIL, dRETURN\ r\}$ $dRETURN\ r \in M$ $Inf\ M = dRETURN$
 r
 | $dSUCCEED \in M$ $Inf\ M = dSUCCEED$
 ⟨*proof*⟩

lemma *dres-internal-simps[simp]*:

$dSUCCEEDi = dSUCCEED$
 $dFAILi = dFAIL$
 ⟨*proof*⟩

Monad Operations

function *dbind where*

$dbind\ dFAIL\ - = dFAIL$
 | $dbind\ dSUCCEED\ - = dSUCCEED$
 | $dbind\ (dRETURN\ x)\ f = f\ x$
 ⟨*proof*⟩

termination ⟨*proof*⟩

adhoc-overloading

Monad-Syntax.bind *dbind*

lemma [*code*]:

$dbind\ (dRETURN\ x)\ f = f\ x$
 $dbind\ (dSUCCEEDi)\ f = dSUCCEEDi$
 $dbind\ (dFAILi)\ f = dFAILi$
 ⟨*proof*⟩

lemma *dres-monad1[simp]*: $dbind\ (dRETURN\ x)\ f = f\ x$

⟨*proof*⟩

lemma *dres-monad2[simp]*: $dbind\ M\ dRETURN = M$

⟨*proof*⟩

lemma *dres-monad3[simp]*: $dbind\ (dbind\ M\ f)\ g = dbind\ M\ (\lambda x. dbind\ (f\ x)\ g)$

<proof>

lemmas *dres-monad-laws* = *dres-monad1 dres-monad2 dres-monad3*

lemma *dbind-mono[refine-mono]*:

$\llbracket M \leq M'; \bigwedge x. dRETURN\ x \leq M \implies f\ x \leq f'\ x \rrbracket \implies dbind\ M\ f \leq dbind\ M'\ f'$
 $\llbracket flat-ge\ M\ M'; \bigwedge x. flat-ge\ (f\ x)\ (f'\ x) \rrbracket \implies flat-ge\ (dbind\ M\ f)\ (dbind\ M'\ f')$
<proof>

lemma *dbind-mono1[simp, intro!]*: *mono dbind*

<proof>

lemma *dbind-mono2[simp, intro!]*: *mono (dbind M)*

<proof>

lemma *dr-mono-bind*:

assumes *MA: mono A and MB: $\bigwedge s. mono\ (B\ s)$*

shows *mono $(\lambda F\ s. dbind\ (A\ F\ s)\ (\lambda s'. B\ s\ F\ s'))$*

<proof>

lemma *dr-mono-bind'*: *mono $(\lambda F\ s. dbind\ (f\ s)\ F)$*

<proof>

lemmas *dr-mono* = *mono-if dr-mono-bind dr-mono-bind' mono-const mono-id*

lemma *[refine-mono]*:

dbind dSUCCEED f = dSUCCEED

dbind dFAIL f = dFAIL

<proof>

definition *dASSERT* \equiv *iASSERT dRETURN*

definition *dASSUME* \equiv *iASSUME dRETURN*

interpretation *dres-assert*: *generic-Assert dbind dRETURN dASSERT dASSUME*

<proof>

definition *dWHILEIT* \equiv *iWHILEIT dbind dRETURN*

definition *dWHILEI* \equiv *iWHILEI dbind dRETURN*

definition *dWHILET* \equiv *iWHILET dbind dRETURN*

definition *dWHILE* \equiv *iWHILE dbind dRETURN*

interpretation *dres-while*: *generic-WHILE dbind dRETURN*

dWHILEIT dWHILEI dWHILET dWHILE

<proof>

lemmas *[code]* =

dres-while.WHILEIT-unfold

dres-while.WHILEI-unfold

dres-while.WHILET-unfold
dres-while.WHILE-unfold

Syntactic criteria to prove $s \neq dSUCCEED$

lemma *dres-ne-bot-basic[refine-transfer]*:

$dFAIL \neq dSUCCEED$
 $\bigwedge x. dRETURN\ x \neq dSUCCEED$
 $\bigwedge m\ f. \llbracket m \neq dSUCCEED; \bigwedge x. f\ x \neq dSUCCEED \rrbracket \implies dbind\ m\ f \neq dSUCCEED$
 $\bigwedge \Phi. dASSERT\ \Phi \neq dSUCCEED$
 $\bigwedge b\ m1\ m2. \llbracket m1 \neq dSUCCEED; m2 \neq dSUCCEED \rrbracket \implies If\ b\ m1\ m2 \neq dSUCCEED$
 $\bigwedge x\ f. \llbracket \bigwedge x. f\ x \neq dSUCCEED \rrbracket \implies Let\ x\ f \neq dSUCCEED$
 $\bigwedge g\ p. \llbracket \bigwedge x1\ x2. g\ x1\ x2 \neq dSUCCEED \rrbracket \implies case-prod\ g\ p \neq dSUCCEED$
 $\bigwedge fn\ fs\ x.$
 $\llbracket fn \neq dSUCCEED; \bigwedge v. fs\ v \neq dSUCCEED \rrbracket \implies case-option\ fn\ fs\ x \neq dSUCCEED$
 $\bigwedge fn\ fc\ x.$
 $\llbracket fn \neq dSUCCEED; \bigwedge x\ xs. fc\ x\ xs \neq dSUCCEED \rrbracket \implies case-list\ fn\ fc\ x \neq dSUCCEED$
<proof>

lemma *dres-ne-bot-RECT[refine-transfer]*:

assumes $A: \bigwedge f\ x. \llbracket \bigwedge x. f\ x \neq dSUCCEED \rrbracket \implies B\ f\ x \neq dSUCCEED$
shows $RECT\ B\ x \neq dSUCCEED$
<proof>

lemma *dres-ne-bot-dWHILEIT[refine-transfer]*:

assumes $\bigwedge x. f\ x \neq dSUCCEED$
shows $dWHILEIT\ I\ b\ f\ s \neq dSUCCEED$ *<proof>*

lemma *dres-ne-bot-dWHILET[refine-transfer]*:

assumes $\bigwedge x. f\ x \neq dSUCCEED$
shows $dWHILET\ b\ f\ s \neq dSUCCEED$ *<proof>*

end

2.13 Partial Function Package Setup

theory *Refine-Pfun*

imports *Refine-Basic Refine-Det*

begin

In this theory, we set up the partial function package to be used with our refinement framework.

2.13.1 Nondeterministic Result Monad

interpretation *nrec*:

partial-function-definitions (\leq) *Sup*::'a nres set \Rightarrow 'a nres
 ⟨proof⟩

lemma *nrec-admissible*: *nrec.admissible* ($\lambda(f::'a \Rightarrow 'b \text{ nres}).$
 ($\forall x0. f x0 \leq \text{SPEC } (P x0)$))
 ⟨proof⟩

⟨ML⟩

lemma *bind-mono-pfun*[*partial-function-mono*]:

fixes $C :: 'a \Rightarrow ('b \Rightarrow 'c \text{ nres}) \Rightarrow ('d \text{ nres})$

shows

$\llbracket \text{monotone } (\text{fun-ord } (\leq)) (\leq) B;$
 $\bigwedge y. \text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda f. C y f) \rrbracket \Longrightarrow$
 $\text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda f. \text{bind } (B f) (\lambda y. C y f))$
 ⟨proof⟩

2.13.2 Deterministic Result Monad

interpretation *drec*:

partial-function-definitions (\leq) *Sup*::'a dres set \Rightarrow 'a dres
 ⟨proof⟩

lemma *drec-admissible*: *drec.admissible* ($\lambda(f::'a \Rightarrow 'b \text{ dres}).$
 ($\forall x. P x \longrightarrow$
 ($f x \neq \text{dFAIL} \wedge$
 ($\forall r. f x = \text{dRETURN } r \longrightarrow Q x r$))))
 ⟨proof⟩

⟨ML⟩

lemma *drec-bind-mono-pfun*[*partial-function-mono*]:

fixes $C :: 'a \Rightarrow ('b \Rightarrow 'c \text{ dres}) \Rightarrow ('d \text{ dres})$

shows

$\llbracket \text{monotone } (\text{fun-ord } (\leq)) (\leq) B;$
 $\bigwedge y. \text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda f. C y f) \rrbracket \Longrightarrow$
 $\text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda f. \text{dbind } (B f) (\lambda y. C y f))$
 ⟨proof⟩

end

2.14 Transfer Setup

```

theory Refine-Transfer
imports
  Refine-Basic
  Refine-While
  Refine-Det
  Generic/RefineG-Transfer
begin

```

2.14.1 Transfer to Deterministic Result Lattice

TODO: Once lattice and ccpo are connected, also transfer to option monad, that is a ccpo, but no complete lattice!

Connecting Deterministic and Non-Deterministic Result Lattices

definition *nres-of* $r \equiv$ case r of
 $dSUCCEEDi \Rightarrow SUCCEED$
 $| dFAILi \Rightarrow FAIL$
 $| dRETURN x \Rightarrow RETURN x$

lemma *nres-of-simps*[*simp*]:
 $nres-of dSUCCEED = SUCCEED$
 $nres-of dFAIL = FAIL$
 $nres-of (dRETURN x) = RETURN x$
 ⟨*proof*⟩

lemma *nres-of-mono*: *mono nres-of*
 ⟨*proof*⟩

lemma *nres-transfer*:
 $nres-of dSUCCEED = SUCCEED$
 $nres-of dFAIL = FAIL$
 $nres-of a \leq nres-of b \longleftrightarrow a \leq b$
 $nres-of a < nres-of b \longleftrightarrow a < b$
 $is-chain A \Longrightarrow nres-of (Sup A) = Sup (nres-of' A)$
 $is-chain A \Longrightarrow nres-of (Inf A) = Inf (nres-of' A)$
 ⟨*proof*⟩

lemma *nres-correctD*:
assumes $nres-of S \leq SPEC \Phi$
shows
 $S = dRETURN x \Longrightarrow \Phi x$
 $S \neq dFAIL$
 ⟨*proof*⟩

Transfer Theorems Setup

interpretation *dres*: *dist-transfer nres-of*
 $\langle \text{proof} \rangle$

lemma *nres-of-transfer[refine-transfer]*: $nres\text{-of } x \leq nres\text{-of } x \langle \text{proof} \rangle$

lemma *det-FAIL[refine-transfer]*: $nres\text{-of } (dFAIL) \leq FAIL \langle \text{proof} \rangle$

lemma *det-SUCCEED[refine-transfer]*: $nres\text{-of } (dSUCCEED) \leq SUCCEED \langle \text{proof} \rangle$

lemma *det-SPEC*: $\Phi x \implies nres\text{-of } (dRETURN x) \leq SPEC \Phi \langle \text{proof} \rangle$

lemma *det-RETURN[refine-transfer]*:
 $nres\text{-of } (dRETURN x) \leq RETURN x \langle \text{proof} \rangle$

lemma *det-bind[refine-transfer]*:
assumes $nres\text{-of } m \leq M$
assumes $\bigwedge x. nres\text{-of } (f x) \leq F x$
shows $nres\text{-of } (dbind m f) \leq bind M F$
 $\langle \text{proof} \rangle$

interpretation *det-assert*: *transfer-generic-Assert-remove*
 $bind RETURN ASSERT ASSUME$
 $nres\text{-of}$
 $\langle \text{proof} \rangle$

interpretation *det-while*: *transfer-WHILE*
 $dbind dRETURN dWHILEIT dWHILEI dWHILET dWHILE$
 $bind RETURN WHILEIT WHILEI WHILET WHILE nres\text{-of}$
 $\langle \text{proof} \rangle$

2.14.2 Transfer to Plain Function

interpretation *plain*: *transfer RETURN* $\langle \text{proof} \rangle$

lemma *plain-RETURN[refine-transfer]*: $RETURN a \leq RETURN a \langle \text{proof} \rangle$

lemma *plain-bind[refine-transfer]*:
 $\llbracket RETURN x \leq M; \bigwedge x. RETURN (f x) \leq F x \rrbracket \implies RETURN (Let x f) \leq bind$
 $M F$
 $\langle \text{proof} \rangle$

interpretation *plain-assert*: *transfer-generic-Assert-remove*
 $bind RETURN ASSERT ASSUME$
 $RETURN$
 $\langle \text{proof} \rangle$

2.14.3 Total correctness in deterministic monad

Sometimes one cannot extract total correct programs to executable plain Isabelle functions, for example, if the total correctness only holds for certain preconditions. In those cases, one can still show $RETURN (the\text{-res } S) \leq S'$. Here, *the-res* extracts the result from a deterministic monad. As *the-res*

is executable, the above shows that (*the-res* S) is always a correct result.

fun *the-res* **where** *the-res* (*dRETURN* x) = x

The following lemma converts a proof-obligation with result extraction to a transfer proof obligation, and a proof obligation that the program yields not bottom.

Note that this rule has to be applied manually, as, otherwise, it would interfere with the default setup, that tries to generate a plain function.

lemma *the-resI*:

assumes *nres-of* $S \leq S'$

assumes $S \neq dSUCCEED$

shows *RETURN* (*the-res* S) $\leq S'$

<proof>

The following rule sets up a refinement goal, a transfer goal, and a final optimization goal.

definition *detTAG* $x \equiv x$

lemma *detTAGI*: $x = detTAG\ x$ *<proof>*

lemma *autoref-detI*:

assumes $(b,a) \in \langle R \rangle nres-rel$

assumes *RETURN* $c \leq b$

assumes $c = detTAG\ d$

shows $(RETURN\ d, a) \in \langle R \rangle nres-rel$

<proof>

2.14.4 Relator-Based Transfer

definition *dres-nres-rel-internal-def*:

dres-nres-rel $R \equiv \{(c,a). nres-of\ c \leq \Downarrow R\ a\}$

lemma *dres-nres-rel-def*: $\langle R \rangle dres-nres-rel \equiv \{(c,a). nres-of\ c \leq \Downarrow R\ a\}$

<proof>

lemma *dres-nres-relI*[*intro?*]: $nres-of\ c \leq \Downarrow R\ a \implies (c,a) \in \langle R \rangle dres-nres-rel$

<proof>

lemma *dres-nres-relD*: $(c,a) \in \langle R \rangle dres-nres-rel \implies nres-of\ c \leq \Downarrow R\ a$

<proof>

lemma *dres-nres-rel-as-br-conv*:

$\langle R \rangle dres-nres-rel = br\ nres-of\ (\lambda-. True)\ O\ \langle R \rangle nres-rel$

<proof>

definition *plain-nres-rel-internal-def*:

plain-nres-rel $R \equiv \{(c,a). RETURN\ c \leq \Downarrow R\ a\}$

lemma *plain-nres-rel-def*: $\langle R \rangle plain-nres-rel \equiv \{(c,a). RETURN\ c \leq \Downarrow R\ a\}$

<proof>

lemma *plain-nres-rel*[*intro?*]: $RETURN\ c \leq \Downarrow R\ a \implies (c,a) \in \langle R \rangle\text{plain-nres-rel}$
<proof>

lemma *plain-nres-relD*: $(c,a) \in \langle R \rangle\text{plain-nres-rel} \implies RETURN\ c \leq \Downarrow R\ a$
<proof>

lemma *plain-nres-rel-as-br-conv*:
 $\langle R \rangle\text{plain-nres-rel} = br\ RETURN\ (\lambda-. True)\ O\ \langle R \rangle\text{nres-rel}$
<proof>

2.14.5 Post-Simplification Setup

lemma *dres-unit-simps*[*refine-transfer-post-simp*]:
 $dbind\ (dRETURN\ (u::unit))\ f = f\ ()$
<proof>

lemma *Let-dRETURN-simp*[*refine-transfer-post-simp*]:
 $Let\ m\ dRETURN = dRETURN\ m$ *<proof>*

lemmas [*refine-transfer-post-simp*] = *dres-monad-laws*

end

2.15 Foreach Loops

theory *Refine-Foreach*

imports

Refine-While
Refine-Pfun
Refine-Transfer
Refine-Heuristics

begin

A common pattern for loop usage is iteration over the elements of a set. This theory provides the *FOREACH*-combinator, that iterates over each element of a set.

2.15.1 Auxilliary Lemmas

The following lemma is commonly used when reasoning about iterator invariants. It helps converting the set of elements that remain to be iterated over to the set of elements already iterated over.

lemma *it-step-insert-iff*:

$$it \subseteq S \implies x \in it \implies S - (it - \{x\}) = insert\ x\ (S - it) \text{ (proof)}$$

2.15.2 Definition

Foreach-loops come in different versions, depending on whether they have an annotated invariant (I), a termination condition (C), and an order (O).

Note that asserting that the set is finite is not necessary to guarantee termination. However, we currently provide only iteration over finite sets, as this also matches the ICF concept of iterators.

definition *FOREACH-body* $f \equiv \lambda(xs, \sigma). do \{$
 $let\ x = hd\ xs; \sigma' \leftarrow f\ x\ \sigma; RETURN\ (tl\ xs, \sigma')$
 $\}$

definition *FOREACH-cond where FOREACH-cond* $c \equiv (\lambda(xs, \sigma). xs \neq [] \wedge c\ \sigma)$

Foreach with continuation condition, order and annotated invariant:

definition *FOREACHoci* ($FOREACH_{OC}^-$) **where** *FOREACHoci* $R\ \Phi\ S\ c\ f\ \sigma\ \theta$
 $\equiv do \{$
 $ASSERT\ (finite\ S);$
 $xs \leftarrow SPEC\ (\lambda xs. distinct\ xs \wedge S = set\ xs \wedge sorted-wrt\ R\ xs);$
 $(-, \sigma) \leftarrow WHILEIT$
 $(\lambda(it, \sigma). \exists xs'. xs = xs' @ it \wedge \Phi\ (set\ it)\ \sigma)\ (FOREACH-cond\ c)\ (FOREACH-body$
 $f)\ (xs, \sigma\ \theta);$
 $RETURN\ \sigma \}$

Foreach with continuation condition and annotated invariant:

definition *FOREACHci* ($FOREACH_C^-$) **where** *FOREACHci* $\equiv FOREACHoci$
 $(\lambda-. True)$

Foreach with continuation condition:

definition *FOREACHc* ($FOREACH_C$) **where** *FOREACHc* $\equiv FOREACHci$ ($\lambda-. True$)

Foreach with annotated invariant:

definition *FOREACHi* ($FOREACH^-$) **where**
 $FOREACHi\ \Phi\ S \equiv FOREACHci\ \Phi\ S\ (\lambda-. True)$

Foreach with annotated invariant and order:

definition *FOREACHoi* ($FOREACH_{O^-}$) **where**
 $FOREACHoi\ R\ \Phi\ S \equiv FOREACHoci\ R\ \Phi\ S\ (\lambda-. True)$

Basic foreach

definition *FOREACH* $S \equiv FOREACHc\ S\ (\lambda-. True)$

lemmas *FOREACH-to-oci-unfold*

$= FOREACHci-def\ FOREACHc-def\ FOREACHi-def\ FOREACHoi-def\ FOREACH-def$

2.15.3 Proof Rules

lemma *FOREACHoci-rule[refine-vcg]*:

assumes *FIN*: finite *S*

assumes *I0*: $I\ S\ \sigma\ 0$

assumes *IP*:

$\bigwedge x\ it\ \sigma. \llbracket c\ \sigma; x \in it; it \subseteq S; I\ it\ \sigma; \forall y \in it - \{x\}. R\ x\ y;$
 $\quad \forall y \in S - it. R\ y\ x \rrbracket \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it - \{x\}))$

assumes *II1*: $\bigwedge \sigma. \llbracket I\ \{\} \sigma \rrbracket \Longrightarrow P\ \sigma$

assumes *II2*: $\bigwedge it\ \sigma. \llbracket it \neq \{\}; it \subseteq S; I\ it\ \sigma; \neg c\ \sigma;$
 $\quad \forall x \in it. \forall y \in S - it. R\ y\ x \rrbracket \Longrightarrow P\ \sigma$

shows *FOREACHoci* $R\ I\ S\ c\ f\ \sigma\ 0 \leq SPEC\ P$

<proof>

lemma *FOREACHoi-rule[refine-vcg]*:

assumes *FIN*: finite *S*

assumes *I0*: $I\ S\ \sigma\ 0$

assumes *IP*:

$\bigwedge x\ it\ \sigma. \llbracket x \in it; it \subseteq S; I\ it\ \sigma; \forall y \in it - \{x\}. R\ x\ y;$
 $\quad \forall y \in S - it. R\ y\ x \rrbracket \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it - \{x\}))$

assumes *II1*: $\bigwedge \sigma. \llbracket I\ \{\} \sigma \rrbracket \Longrightarrow P\ \sigma$

shows *FOREACHoi* $R\ I\ S\ f\ \sigma\ 0 \leq SPEC\ P$

<proof>

lemma *FOREACHci-rule[refine-vcg]*:

assumes *FIN*: finite *S*

assumes *I0*: $I\ S\ \sigma\ 0$

assumes *IP*:

$\bigwedge x\ it\ \sigma. \llbracket x \in it; it \subseteq S; I\ it\ \sigma; c\ \sigma \rrbracket \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it - \{x\}))$

assumes *II1*: $\bigwedge \sigma. \llbracket I\ \{\} \sigma \rrbracket \Longrightarrow P\ \sigma$

assumes *II2*: $\bigwedge it\ \sigma. \llbracket it \neq \{\}; it \subseteq S; I\ it\ \sigma; \neg c\ \sigma \rrbracket \Longrightarrow P\ \sigma$

shows *FOREACHci* $I\ S\ c\ f\ \sigma\ 0 \leq SPEC\ P$

<proof>

Refinement:

Refinement rule using a coupling invariant over sets of remaining items and the state.

lemma *FOREACHoci-refine-genR*:

fixes $\alpha :: 'S \Rightarrow 'Sa$ — Abstraction mapping of elements

fixes $S :: 'S\ set$ — Concrete set

fixes $S' :: 'Sa\ set$ — Abstract set

fixes $\sigma\ 0 :: 'S$

fixes $\sigma\ 0' :: 'Sa$

fixes $R :: (('S\ set \times 'S) \times ('Sa\ set \times 'Sa))\ set$

assumes *INJ*: inj-on $\alpha\ S$

assumes *REFS[simp]*: $S' = \alpha\ S$

assumes *RR-OK*: $\bigwedge x\ y. \llbracket x \in S; y \in S; RR\ x\ y \rrbracket \Longrightarrow RR'\ (\alpha\ x)\ (\alpha\ y)$

assumes *REF0*: $((S, \sigma\ 0), (\alpha\ S, \sigma\ 0')) \in R$

assumes *REFC*: $\bigwedge it \ \sigma \ it' \ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi' it' \ \sigma'; \Phi it \ \sigma;$
 $\forall x \in S - it. \forall y \in it. RR \ x \ y; \forall x \in S' - it'. \forall y \in it'. RR' \ x \ y;$
 $it' = \alpha' it; ((it, \sigma), (it', \sigma')) \in R$
 $\rrbracket \implies c \ \sigma \longleftrightarrow c' \ \sigma'$
assumes *REFPHI*: $\bigwedge it \ \sigma \ it' \ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi' it' \ \sigma';$
 $\forall x \in S - it. \forall y \in it. RR \ x \ y; \forall x \in S' - it'. \forall y \in it'. RR' \ x \ y;$
 $it' = \alpha' it; ((it, \sigma), (it', \sigma')) \in R$
 $\rrbracket \implies \Phi it \ \sigma$
assumes *REFSTEP*: $\bigwedge x \ it \ \sigma \ x' \ it' \ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi it \ \sigma; \Phi' it' \ \sigma';$
 $\forall x \in S - it. \forall y \in it. RR \ x \ y; \forall x \in S' - it'. \forall y \in it'. RR' \ x \ y;$
 $x' = \alpha \ x; it' = \alpha' it; ((it, \sigma), (it', \sigma')) \in R;$
 $x \in it; \forall y \in it - \{x\}. RR \ x \ y;$
 $x' \in it'; \forall y' \in it' - \{x'\}. RR' \ x' \ y';$
 $c \ \sigma; c' \ \sigma'$
 $\rrbracket \implies f \ x \ \sigma$
 $\leq \Downarrow(\{\{\sigma, \sigma'\}. ((it - \{x\}, \sigma), (it' - \{x'\}, \sigma')) \in R\}) (f' \ x' \ \sigma')$
assumes *REF-R-DONE*: $\bigwedge \sigma \ \sigma'. \llbracket \Phi \ \{\} \ \sigma; \Phi' \ \{\} \ \sigma'; ((\{\}, \sigma), (\{\}, \sigma')) \in R \rrbracket$
 $\implies (\sigma, \sigma') \in R'$
assumes *REF-R-BRK*: $\bigwedge it \ \sigma \ it' \ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi it \ \sigma; \Phi' it' \ \sigma';$
 $\forall x \in S - it. \forall y \in it. RR \ x \ y; \forall x \in S' - it'. \forall y \in it'. RR' \ x \ y;$
 $it' = \alpha' it; ((it, \sigma), (it', \sigma')) \in R;$
 $it \neq \{\}; it' \neq \{\};$
 $\neg c \ \sigma; \neg c' \ \sigma'$
 $\rrbracket \implies (\sigma, \sigma') \in R'$
shows *FOREACHoci* $RR \ \Phi \ S \ c \ f \ \sigma \ 0 \leq \Downarrow R' (FOREACHoci \ RR' \ \Phi' \ S' \ c' \ f' \ \sigma \ 0')$

(proof)

lemma *FOREACHoci-refine*:

fixes $\alpha :: 'S \Rightarrow 'Sa$

fixes $S :: 'S \text{ set}$

fixes $S' :: 'Sa \text{ set}$

assumes *INJ*: *inj-on* $\alpha \ S$

assumes *REFS*: $S' = \alpha' S$

assumes *REF0*: $(\sigma \ 0, \sigma \ 0') \in R$

assumes *RR-OK*: $\bigwedge x \ y. \llbracket x \in S; y \in S; RR \ x \ y \rrbracket \implies RR' (\alpha \ x) (\alpha \ y)$

assumes *REFPHI0*: $\Phi'' S \ \sigma \ 0 (\alpha' S) \ \sigma \ 0'$

assumes *REFC*: $\bigwedge it \ \sigma \ it' \ \sigma'. \llbracket$

$it' = \alpha' it; it \subseteq S; it' \subseteq S'; \Phi' it' \ \sigma'; \Phi'' it \ \sigma \ it' \ \sigma'; \Phi it \ \sigma; (\sigma, \sigma') \in R$

$\rrbracket \implies c \ \sigma \longleftrightarrow c' \ \sigma'$

assumes *REFPHI*: $\bigwedge it \ \sigma \ it' \ \sigma'. \llbracket$

$it' = \alpha' it; it \subseteq S; it' \subseteq S'; \Phi' it' \ \sigma'; \Phi'' it \ \sigma \ it' \ \sigma'; (\sigma, \sigma') \in R$

$\rrbracket \implies \Phi it \ \sigma$

assumes *REFSTEP*: $\bigwedge x \ it \ \sigma \ x' \ it' \ \sigma'. \llbracket \forall y \in it - \{x\}. RR \ x \ y;$

$x' = \alpha \ x; x \in it; x' \in it'; it' = \alpha' it; it \subseteq S; it' \subseteq S';$

$\Phi \text{ it } \sigma; \Phi' \text{ it}' \sigma'; \Phi'' \text{ it } \sigma \text{ it}' \sigma'; c \sigma; c' \sigma';$
 $(\sigma, \sigma') \in R$
 $\Downarrow \Longrightarrow f x \sigma$
 $\leq \Downarrow (\{(\sigma, \sigma'). (\sigma, \sigma') \in R \wedge \Phi'' (\text{it} - \{x\}) \sigma (\text{it}' - \{x'\}) \sigma'\}) (f' x' \sigma')$
shows $\text{FOREACHoci } RR \Phi S c f \sigma \leq \Downarrow R (\text{FOREACHoci } RR' \Phi' S' c' f' \sigma')$
(proof)

lemma *FOREACHoci-refine-rcg[refine]:*

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes *INJ*: *inj-on* αS
assumes *REFS*: $S' = \alpha S$
assumes *REF0*: $(\sigma \theta, \sigma \theta') \in R$
assumes *RR-OK*: $\bigwedge x y. \llbracket x \in S; y \in S; RR x y \rrbracket \Longrightarrow RR' (\alpha x) (\alpha y)$
assumes *REFC*: $\bigwedge \text{it } \sigma \text{ it}' \sigma'. \llbracket$
 $\text{it}' = \alpha \text{ it}; \text{it} \subseteq S; \text{it}' \subseteq S'; \Phi' \text{ it}' \sigma'; \Phi \text{ it } \sigma; (\sigma, \sigma') \in R$
 $\rrbracket \Longrightarrow c \sigma \longleftrightarrow c' \sigma'$
assumes *REFPHI*: $\bigwedge \text{it } \sigma \text{ it}' \sigma'. \llbracket$
 $\text{it}' = \alpha \text{ it}; \text{it} \subseteq S; \text{it}' \subseteq S'; \Phi' \text{ it}' \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \Longrightarrow \Phi \text{ it } \sigma$
assumes *REFSTEP*: $\bigwedge x \text{ it } \sigma x' \text{ it}' \sigma'. \llbracket \forall y \in \text{it} - \{x\}. RR x y;$
 $x' = \alpha x; x \in \text{it}; x' \in \text{it}'; \text{it}' = \alpha \text{ it}; \text{it} \subseteq S; \text{it}' \subseteq S';$
 $\Phi \text{ it } \sigma; \Phi' \text{ it}' \sigma'; c \sigma; c' \sigma';$
 $(\sigma, \sigma') \in R$
 $\rrbracket \Longrightarrow f x \sigma \leq \Downarrow R (f' x' \sigma')$
shows $\text{FOREACHoci } RR \Phi S c f \sigma \leq \Downarrow R (\text{FOREACHoci } RR' \Phi' S' c' f' \sigma')$
(proof)

lemma *FOREACHoci-weaken:*

assumes *IREF*: $\bigwedge \text{it } \sigma. \text{it} \subseteq S \Longrightarrow I \text{ it } \sigma \Longrightarrow I' \text{ it } \sigma$
shows $\text{FOREACHoci } RR I' S c f \sigma \leq \text{FOREACHoci } RR I S c f \sigma$
(proof)

lemma *FOREACHoci-weaken-order:*

assumes *RRREF*: $\bigwedge x y. x \in S \Longrightarrow y \in S \Longrightarrow RR x y \Longrightarrow RR' x y$
shows $\text{FOREACHoci } RR I S c f \sigma \leq \text{FOREACHoci } RR' I S c f \sigma$
(proof)

Rules for Derived Constructs

lemma *FOREACHoi-refine-genR:*

fixes $\alpha :: 'S \Rightarrow 'Sa$ — Abstraction mapping of elements
fixes $S :: 'S \text{ set}$ — Concrete set
fixes $S' :: 'Sa \text{ set}$ — Abstract set
fixes $\sigma \theta :: 'S$
fixes $\sigma \theta' :: 'Sa$
fixes $R :: ((S \text{ set} \times 'S) \times (Sa \text{ set} \times 'Sa)) \text{ set}$

assumes *INJ*: $\text{inj-on } \alpha S$
assumes *REFS*[*simp*]: $S' = \alpha' S$
assumes *RR-OK*: $\bigwedge x y. \llbracket x \in S; y \in S; RR x y \rrbracket \implies RR' (\alpha x) (\alpha y)$
assumes *REF0*: $((S, \sigma 0), (\alpha' S, \sigma 0')) \in R$
assumes *REFPHI*: $\bigwedge it \sigma it' \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi' it' \sigma';$
 $\forall x \in S - it. \forall y \in it. RR x y; \forall x \in S' - it'. \forall y \in it'. RR' x y;$
 $it' = \alpha' it; ((it, \sigma), (it', \sigma')) \in R$
 $\rrbracket \implies \Phi it \sigma$
assumes *REFSTEP*: $\bigwedge x it \sigma x' it' \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi it \sigma; \Phi' it' \sigma';$
 $\forall x \in S - it. \forall y \in it. RR x y; \forall x \in S' - it'. \forall y \in it'. RR' x y;$
 $x' = \alpha x; it' = \alpha' it; ((it, \sigma), (it', \sigma')) \in R;$
 $x \in it; \forall y \in it - \{x\}. RR x y;$
 $x' \in it'; \forall y' \in it' - \{x'\}. RR' x' y'$
 $\rrbracket \implies f x \sigma$
 $\leq \Downarrow(\{(\sigma, \sigma'). ((it - \{x\}, \sigma), (it' - \{x'\}, \sigma')) \in R\}) (f' x' \sigma')$
assumes *REF-R-DONE*: $\bigwedge \sigma \sigma'. \llbracket \Phi \{ \} \sigma; \Phi' \{ \} \sigma'; ((\{ \}, \sigma), (\{ \}, \sigma')) \in R \rrbracket$
 $\implies (\sigma, \sigma') \in R'$
shows *FOREACHoi* $RR \Phi S f \sigma 0 \leq \Downarrow R' (FOREACHoi RR' \Phi' S' f' \sigma 0')$
<proof>

lemma *FOREACHoi-refine*:

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes *INJ*: $\text{inj-on } \alpha S$
assumes *REFS*: $S' = \alpha' S$
assumes *REF0*: $(\sigma 0, \sigma 0') \in R$
assumes *RR-OK*: $\bigwedge x y. \llbracket x \in S; y \in S; RR x y \rrbracket \implies RR' (\alpha x) (\alpha y)$
assumes *REFPHI0*: $\Phi'' S \sigma 0 (\alpha' S) \sigma 0'$
assumes *REFPHI*: $\bigwedge it \sigma it' \sigma'. \llbracket$
 $it' = \alpha' it; it \subseteq S; it' \subseteq S'; \Phi' it' \sigma'; \Phi'' it \sigma it' \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \implies \Phi it \sigma$
assumes *REFSTEP*: $\bigwedge x it \sigma x' it' \sigma'. \llbracket \forall y \in it - \{x\}. RR x y;$
 $x' = \alpha x; x \in it; x' \in it'; it' = \alpha' it; it \subseteq S; it' \subseteq S';$
 $\Phi it \sigma; \Phi' it' \sigma'; \Phi'' it \sigma it' \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \implies f x \sigma$
 $\leq \Downarrow(\{(\sigma, \sigma'). (\sigma, \sigma') \in R \wedge \Phi'' (it - \{x\}) \sigma (it' - \{x'\}) \sigma' \}) (f' x' \sigma')$
shows *FOREACHoi* $RR \Phi S f \sigma 0 \leq \Downarrow R (FOREACHoi RR' \Phi' S' f' \sigma 0')$
<proof>

lemma *FOREACHoi-refine-rcg*[*refine*]:

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes *INJ*: $\text{inj-on } \alpha S$
assumes *REFS*: $S' = \alpha' S$
assumes *REF0*: $(\sigma 0, \sigma 0') \in R$

assumes *RR-OK*: $\bigwedge x y. \llbracket x \in S; y \in S; RR\ x\ y \rrbracket \implies RR' (\alpha\ x) (\alpha\ y)$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'. \llbracket$
 $it' = \alpha\ it; it \subseteq S; it' \subseteq S'; \Phi' it'\ \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \implies \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'. \llbracket \forall y \in it - \{x\}. RR\ x\ y;$
 $x' = \alpha\ x; x \in it; x' \in it'; it' = \alpha\ it; it \subseteq S; it' \subseteq S';$
 $\Phi\ it\ \sigma; \Phi' it'\ \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \implies f\ x\ \sigma \leq \Downarrow R (f'\ x'\ \sigma')$
shows *FOREACHoi* $RR\ \Phi\ S\ f\ \sigma\ \theta \leq \Downarrow R (FOREACHoi\ RR'\ \Phi'\ S'\ f'\ \sigma\ \theta')$
<proof>

lemma *FOREACHci-refine-genR*:

fixes $\alpha :: 'S \Rightarrow 'Sa$ — Abstraction mapping of elements
fixes $S :: 'S\ set$ — Concrete set
fixes $S' :: 'Sa\ set$ — Abstract set
fixes $\sigma\ \theta :: 'S$
fixes $\sigma\ \theta' :: 'Sa$
fixes $R :: (('S\ set \times 'S) \times ('Sa\ set \times 'Sa))\ set$
assumes *INJ*: *inj-on* $\alpha\ S$
assumes *REFS[simp]*: $S' = \alpha\ S$
assumes *REF0*: $((S, \sigma\ \theta), (\alpha\ S, \sigma\ \theta')) \in R$
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi' it'\ \sigma'; \Phi\ it\ \sigma;$
 $it' = \alpha\ it; ((it, \sigma), (it', \sigma')) \in R$
 $\rrbracket \implies c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi' it'\ \sigma';$
 $it' = \alpha\ it; ((it, \sigma), (it', \sigma')) \in R$
 $\rrbracket \implies \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi\ it\ \sigma; \Phi' it'\ \sigma';$
 $x' = \alpha\ x; it' = \alpha\ it; ((it, \sigma), (it', \sigma')) \in R;$
 $x \in it; x' \in it';$
 $c\ \sigma; c'\ \sigma'$
 $\rrbracket \implies f\ x\ \sigma$
 $\leq \Downarrow (\{(\sigma, \sigma'). ((it - \{x\}, \sigma), (it' - \{x'\}, \sigma')) \in R\}) (f'\ x'\ \sigma')$
assumes *REF-R-DONE*: $\bigwedge \sigma\ \sigma'. \llbracket \Phi\ \{\}\ \sigma; \Phi'\ \{\}\ \sigma'; ((\{\}, \sigma), (\{\}, \sigma')) \in R \rrbracket$
 $\implies (\sigma, \sigma') \in R'$
assumes *REF-R-BRK*: $\bigwedge it\ \sigma\ it'\ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi\ it\ \sigma; \Phi' it'\ \sigma';$
 $it' = \alpha\ it; ((it, \sigma), (it', \sigma')) \in R;$
 $it \neq \{\}; it' \neq \{\};$
 $\neg c\ \sigma; \neg c'\ \sigma'$
 $\rrbracket \implies (\sigma, \sigma') \in R'$
shows *FOREACHci* $\Phi\ S\ c\ f\ \sigma\ \theta \leq \Downarrow R' (FOREACHci\ \Phi'\ S'\ c'\ f'\ \sigma\ \theta')$
<proof>

lemma *FOREACHci-refine*:

fixes $\alpha :: 'S \Rightarrow 'Sa$

fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes $INJ: inj\text{-on } \alpha S$
assumes $REFS: S' = \alpha'S$
assumes $REF0: (\sigma 0, \sigma 0') \in R$
assumes $REFPHI0: \Phi'' S \sigma 0 (\alpha'S) \sigma 0'$
assumes $REFC: \bigwedge it \sigma it' \sigma'. \llbracket$
 $it' = \alpha'it; it \subseteq S; it' \subseteq S'; \Phi' it' \sigma'; \Phi'' it \sigma it' \sigma'; \Phi it \sigma; (\sigma, \sigma') \in R$
 $\rrbracket \implies c \sigma \longleftrightarrow c' \sigma'$
assumes $REFPHI: \bigwedge it \sigma it' \sigma'. \llbracket$
 $it' = \alpha'it; it \subseteq S; it' \subseteq S'; \Phi' it' \sigma'; \Phi'' it \sigma it' \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \implies \Phi it \sigma$
assumes $REFSTEP: \bigwedge x it \sigma x' it' \sigma'. \llbracket$
 $x' = \alpha x; x \in it; x' \in it'; it' = \alpha'it; it \subseteq S; it' \subseteq S';$
 $\Phi it \sigma; \Phi' it' \sigma'; \Phi'' it \sigma it' \sigma'; c \sigma; c' \sigma';$
 $(\sigma, \sigma') \in R$
 $\rrbracket \implies f x \sigma$
 $\leq \Downarrow (\{(\sigma, \sigma'). (\sigma, \sigma') \in R \wedge \Phi'' (it - \{x\}) \sigma (it' - \{x'\}) \sigma'\}) (f' x' \sigma')$
shows $FOREACHci \Phi S c f \sigma 0 \leq \Downarrow R (FOREACHci \Phi' S' c' f' \sigma 0')$
 $\langle proof \rangle$

lemma $FOREACHci\text{-refine}\text{-rcg}[refine]:$

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes $INJ: inj\text{-on } \alpha S$
assumes $REFS: S' = \alpha'S$
assumes $REF0: (\sigma 0, \sigma 0') \in R$
assumes $REFC: \bigwedge it \sigma it' \sigma'. \llbracket$
 $it' = \alpha'it; it \subseteq S; it' \subseteq S'; \Phi' it' \sigma'; \Phi it \sigma; (\sigma, \sigma') \in R$
 $\rrbracket \implies c \sigma \longleftrightarrow c' \sigma'$
assumes $REFPHI: \bigwedge it \sigma it' \sigma'. \llbracket$
 $it' = \alpha'it; it \subseteq S; it' \subseteq S'; \Phi' it' \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \implies \Phi it \sigma$
assumes $REFSTEP: \bigwedge x it \sigma x' it' \sigma'. \llbracket$
 $x' = \alpha x; x \in it; x' \in it'; it' = \alpha'it; it \subseteq S; it' \subseteq S';$
 $\Phi it \sigma; \Phi' it' \sigma'; c \sigma; c' \sigma';$
 $(\sigma, \sigma') \in R$
 $\rrbracket \implies f x \sigma \leq \Downarrow R (f' x' \sigma')$
shows $FOREACHci \Phi S c f \sigma 0 \leq \Downarrow R (FOREACHci \Phi' S' c' f' \sigma 0')$
 $\langle proof \rangle$

lemma $FOREACHci\text{-weaken}:$

assumes $IREF: \bigwedge it \sigma. it \subseteq S \implies I it \sigma \implies I' it \sigma$
shows $FOREACHci I' S c f \sigma 0 \leq FOREACHci I S c f \sigma 0$
 $\langle proof \rangle$

lemma $FOREACHi\text{-rule}[refine\text{-vcg}]:$

assumes $FIN: finite S$

assumes $I0: I S \sigma 0$
assumes $IP:$
 $\bigwedge x \text{ it } \sigma. \llbracket x \in \text{it}; \text{it} \subseteq S; I \text{ it } \sigma \rrbracket \implies f x \sigma \leq \text{SPEC } (I (\text{it} - \{x\}))$
assumes $II: \bigwedge \sigma. \llbracket I \{ \} \sigma \rrbracket \implies P \sigma$
shows $\text{FOREACH}_i I S f \sigma 0 \leq \text{SPEC } P$
 $\langle \text{proof} \rangle$

lemma FOREACH_c -rule:

assumes $FIN: \text{finite } S$
assumes $I0: I S \sigma 0$
assumes $IP:$
 $\bigwedge x \text{ it } \sigma. \llbracket x \in \text{it}; \text{it} \subseteq S; I \text{ it } \sigma; c \sigma \rrbracket \implies f x \sigma \leq \text{SPEC } (I (\text{it} - \{x\}))$
assumes $III: \bigwedge \sigma. \llbracket I \{ \} \sigma \rrbracket \implies P \sigma$
assumes $II2: \bigwedge \text{it } \sigma. \llbracket \text{it} \neq \{ \}; \text{it} \subseteq S; I \text{ it } \sigma; \neg c \sigma \rrbracket \implies P \sigma$
shows $\text{FOREACH}_c S c f \sigma 0 \leq \text{SPEC } P$
 $\langle \text{proof} \rangle$

lemma FOREACH -rule:

assumes $FIN: \text{finite } S$
assumes $I0: I S \sigma 0$
assumes $IP:$
 $\bigwedge x \text{ it } \sigma. \llbracket x \in \text{it}; \text{it} \subseteq S; I \text{ it } \sigma \rrbracket \implies f x \sigma \leq \text{SPEC } (I (\text{it} - \{x\}))$
assumes $II: \bigwedge \sigma. \llbracket I \{ \} \sigma \rrbracket \implies P \sigma$
shows $\text{FOREACH } S f \sigma 0 \leq \text{SPEC } P$
 $\langle \text{proof} \rangle$

lemma FOREACH_c -refine-genR:

fixes $\alpha :: 'S \Rightarrow 'Sa$ — Abstraction mapping of elements
fixes $S :: 'S \text{ set}$ — Concrete set
fixes $S' :: 'Sa \text{ set}$ — Abstract set
fixes $\sigma 0 :: ' \sigma$
fixes $\sigma 0' :: ' \sigma a$
fixes $R :: (('S \text{ set} \times ' \sigma) \times ('Sa \text{ set} \times ' \sigma a)) \text{ set}$
assumes $INJ: \text{inj-on } \alpha S$
assumes $REFS[\text{simp}]: S' = \alpha 'S$
assumes $REF0: ((S, \sigma 0), (\alpha 'S, \sigma 0')) \in R$
assumes $REFC: \bigwedge \text{it } \sigma \text{ it}' \sigma'. \llbracket$
 $\text{it} \subseteq S; \text{it}' \subseteq S';$
 $\text{it}' = \alpha ' \text{it}; ((\text{it}, \sigma), (\text{it}', \sigma')) \in R$
 $\rrbracket \implies c \sigma \longleftrightarrow c' \sigma'$
assumes $REFSTEP: \bigwedge x \text{ it } \sigma x' \text{ it}' \sigma'. \llbracket$
 $\text{it} \subseteq S; \text{it}' \subseteq S';$
 $x' = \alpha x; \text{it}' = \alpha ' \text{it}; ((\text{it}, \sigma), (\text{it}', \sigma')) \in R;$
 $x \in \text{it}; x' \in \text{it}';$
 $c \sigma; c' \sigma'$
 $\rrbracket \implies f x \sigma$
 $\leq \Downarrow (\{(\sigma, \sigma'). ((\text{it} - \{x\}, \sigma), (\text{it}' - \{x'\}, \sigma')) \in R\}) (f' x' \sigma')$
assumes $REF\text{-}R\text{-}DONE: \bigwedge \sigma \sigma'. \llbracket (\{ \}, \sigma), (\{ \}, \sigma') \in R \rrbracket$

$\implies (\sigma, \sigma') \in R'$
assumes *REF-R-BRK*: $\bigwedge it \ \sigma \ it' \ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S';$
 $it' = \alpha it; ((it, \sigma), (it', \sigma')) \in R;$
 $it \neq \{\}; it' \neq \{\};$
 $\neg c \ \sigma; \neg c' \ \sigma'$
 $\rrbracket \implies (\sigma, \sigma') \in R'$
shows $FOREACHc \ S \ c \ f \ \sigma \theta \leq \Downarrow R' (FOREACHc \ S' \ c' \ f' \ \sigma \theta')$
<proof>

lemma *FOREACHc-refine*:

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes *INJ*: *inj-on* $\alpha \ S$
assumes *REFS*: $S' = \alpha S$
assumes *REF0*: $(\sigma \theta, \sigma \theta') \in R$
assumes *REFPHI0*: $\Phi'' \ S \ \sigma \theta \ (\alpha S) \ \sigma \theta'$
assumes *REFC*: $\bigwedge it \ \sigma \ it' \ \sigma'. \llbracket$
 $it' = \alpha it; it \subseteq S; it' \subseteq S'; \Phi'' \ it \ \sigma \ it' \ \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \implies c \ \sigma \longleftrightarrow c' \ \sigma'$
assumes *REFSTEP*: $\bigwedge x \ it \ \sigma \ x' \ it' \ \sigma'. \llbracket$
 $x' = \alpha x; x \in it; x' \in it'; it' = \alpha it; it \subseteq S; it' \subseteq S';$
 $\Phi'' \ it \ \sigma \ it' \ \sigma'; c \ \sigma; c' \ \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \implies f \ x \ \sigma$
 $\leq \Downarrow (\{\sigma, \sigma'\}. (\sigma, \sigma') \in R \wedge \Phi'' (it - \{x\}) \ \sigma (it' - \{x'\}) \ \sigma')$ $(f' \ x' \ \sigma')$
shows $FOREACHc \ S \ c \ f \ \sigma \theta \leq \Downarrow R (FOREACHc \ S' \ c' \ f' \ \sigma \theta')$
<proof>

lemma *FOREACHc-refine-rcg[refine]*:

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes *INJ*: *inj-on* $\alpha \ S$
assumes *REFS*: $S' = \alpha S$
assumes *REF0*: $(\sigma \theta, \sigma \theta') \in R$
assumes *REFC*: $\bigwedge it \ \sigma \ it' \ \sigma'. \llbracket$
 $it' = \alpha it; it \subseteq S; it' \subseteq S'; (\sigma, \sigma') \in R$
 $\rrbracket \implies c \ \sigma \longleftrightarrow c' \ \sigma'$
assumes *REFSTEP*: $\bigwedge x \ it \ \sigma \ x' \ it' \ \sigma'. \llbracket$
 $x' = \alpha x; x \in it; x' \in it'; it' = \alpha it; it \subseteq S; it' \subseteq S'; c \ \sigma; c' \ \sigma';$
 $(\sigma, \sigma') \in R$
 $\rrbracket \implies f \ x \ \sigma \leq \Downarrow R (f' \ x' \ \sigma')$
shows $FOREACHc \ S \ c \ f \ \sigma \theta \leq \Downarrow R (FOREACHc \ S' \ c' \ f' \ \sigma \theta')$
<proof>

lemma *FOREACHi-refine-genR*:

fixes $\alpha :: 'S \Rightarrow 'Sa$ — Abstraction mapping of elements
fixes $S :: 'S \text{ set}$ — Concrete set

fixes $S' :: 'Sa \text{ set}$ — Abstract set
fixes $\sigma 0 :: ' \sigma$
fixes $\sigma 0' :: ' \sigma a$
fixes $R :: (('S \text{ set} \times ' \sigma) \times ('Sa \text{ set} \times ' \sigma a)) \text{ set}$
assumes INJ : $\text{inj-on } \alpha S$
assumes $REFS[simp]$: $S' = \alpha 'S$
assumes $REF0$: $((S, \sigma 0), (\alpha 'S, \sigma 0')) \in R$
assumes $REFPHI$: $\bigwedge it \sigma it' \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi' it' \sigma';$
 $it' = \alpha 'it; ((it, \sigma), (it', \sigma')) \in R$
 $\rrbracket \implies \Phi it \sigma$
assumes $REFSTEP$: $\bigwedge x it \sigma x' it' \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi it \sigma; \Phi' it' \sigma';$
 $x' = \alpha x; it' = \alpha 'it; ((it, \sigma), (it', \sigma')) \in R;$
 $x \in it; x' \in it'$
 $\rrbracket \implies f x \sigma$
 $\leq \Downarrow (\{(\sigma, \sigma'). ((it - \{x\}, \sigma), (it' - \{x'\}, \sigma')) \in R\}) (f' x' \sigma')$
assumes $REF-R-DONE$: $\bigwedge \sigma \sigma'. \llbracket \Phi \{\} \sigma; \Phi' \{\} \sigma'; ((\{\}, \sigma), (\{\}, \sigma')) \in R \rrbracket$
 $\implies (\sigma, \sigma') \in R'$
shows $FOREACHi \Phi S f \sigma 0 \leq \Downarrow R' (FOREACHi \Phi' S' f' \sigma 0')$
 $\langle \text{proof} \rangle$

lemma $FOREACHi\text{-refine}$:

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes INJ : $\text{inj-on } \alpha S$
assumes $REFS$: $S' = \alpha 'S$
assumes $REF0$: $(\sigma 0, \sigma 0') \in R$
assumes $REFPHI0$: $\Phi'' S \sigma 0 (\alpha 'S) \sigma 0'$
assumes $REFPHI$: $\bigwedge it \sigma it' \sigma'. \llbracket$
 $it' = \alpha 'it; it \subseteq S; it' \subseteq S'; \Phi' it' \sigma'; \Phi'' it \sigma it' \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \implies \Phi it \sigma$
assumes $REFSTEP$: $\bigwedge x it \sigma x' it' \sigma'. \llbracket$
 $x' = \alpha x; x \in it; x' \in it'; it' = \alpha 'it; it \subseteq S; it' \subseteq S';$
 $\Phi it \sigma; \Phi' it' \sigma'; \Phi'' it \sigma it' \sigma';$
 $(\sigma, \sigma') \in R$
 $\rrbracket \implies f x \sigma$
 $\leq \Downarrow (\{(\sigma, \sigma'). (\sigma, \sigma') \in R \wedge \Phi'' (it - \{x\}) \sigma (it' - \{x'\}) \sigma'\}) (f' x' \sigma')$
shows $FOREACHi \Phi S f \sigma 0 \leq \Downarrow R (FOREACHi \Phi' S' f' \sigma 0')$
 $\langle \text{proof} \rangle$

lemma $FOREACHi\text{-refine-rcg}[refine]$:

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes INJ : $\text{inj-on } \alpha S$
assumes $REFS$: $S' = \alpha 'S$
assumes $REF0$: $(\sigma 0, \sigma 0') \in R$

assumes *REFPHI*: $\bigwedge it \ \sigma \ it' \ \sigma'. \llbracket$
 $it' = \alpha' it; it \subseteq S; it' \subseteq S'; \Phi' it' \ \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \implies \Phi it \ \sigma$
assumes *REFSTEP*: $\bigwedge x \ it \ \sigma \ x' \ it' \ \sigma'. \llbracket$
 $x' = \alpha x; x \in it; x' \in it'; it' = \alpha' it; it \subseteq S; it' \subseteq S';$
 $\Phi it \ \sigma; \Phi' it' \ \sigma';$
 $(\sigma, \sigma') \in R$
 $\rrbracket \implies f x \ \sigma \leq \Downarrow R (f' x' \ \sigma')$
shows *FOREACHi* $\Phi S f \ \sigma \theta \leq \Downarrow R (FOREACHi \ \Phi' S' f' \ \sigma \theta')$
<proof>

lemma *FOREACH-refine-genR*:

fixes $\alpha :: 'S \Rightarrow 'Sa$ — Abstraction mapping of elements
fixes $S :: 'S \text{ set}$ — Concrete set
fixes $S' :: 'Sa \text{ set}$ — Abstract set
fixes $\sigma \theta :: ' \sigma$
fixes $\sigma \theta' :: ' \sigma a$
fixes $R :: (('S \text{ set} \times ' \sigma) \times ('Sa \text{ set} \times ' \sigma a)) \text{ set}$
assumes *INJ*: *inj-on* αS
assumes *REFS[simp]*: $S' = \alpha' S$
assumes *REF0*: $((S, \sigma \theta), (\alpha' S, \sigma \theta')) \in R$
assumes *REFSTEP*: $\bigwedge x \ it \ \sigma \ x' \ it' \ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S';$
 $x' = \alpha x; it' = \alpha' it; ((it, \sigma), (it', \sigma')) \in R;$
 $x \in it; x' \in it'$
 $\rrbracket \implies f x \ \sigma$
 $\leq \Downarrow (\{(\sigma, \sigma'). ((it - \{x\}, \sigma), (it' - \{x'\}, \sigma')) \in R\}) (f' x' \ \sigma')$
assumes *REF-R-DONE*: $\bigwedge \sigma \ \sigma'. \llbracket (\{\}, \sigma), (\{\}, \sigma') \in R \rrbracket$
 $\implies (\sigma, \sigma') \in R'$
shows *FOREACH* $S f \ \sigma \theta \leq \Downarrow R' (FOREACH S' f' \ \sigma \theta')$
<proof>

lemma *FOREACH-refine*:

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes *INJ*: *inj-on* αS
assumes *REFS*: $S' = \alpha' S$
assumes *REF0*: $(\sigma \theta, \sigma \theta') \in R$
assumes *REFPHI0*: $\Phi'' S \ \sigma \theta (\alpha' S) \ \sigma \theta'$
assumes *REFSTEP*: $\bigwedge x \ it \ \sigma \ x' \ it' \ \sigma'. \llbracket$
 $x' = \alpha x; x \in it; x' \in it'; it' = \alpha' it; it \subseteq S; it' \subseteq S';$
 $\Phi'' it \ \sigma \ it' \ \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \implies f x \ \sigma$
 $\leq \Downarrow (\{(\sigma, \sigma'). (\sigma, \sigma') \in R \wedge \Phi'' (it - \{x\}) \ \sigma (it' - \{x'\}) \ \sigma'\}) (f' x' \ \sigma')$
shows *FOREACH* $S f \ \sigma \theta \leq \Downarrow R (FOREACH S' f' \ \sigma \theta')$
<proof>

lemma *FOREACH-refine-rcg[refine]*:

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes $INJ: inj\text{-on } \alpha S$
assumes $REFS: S' = \alpha'S$
assumes $REF0: (\sigma 0, \sigma 0') \in R$
assumes $REFSTEP: \bigwedge x \text{ it } \sigma \ x' \ \text{it}' \ \sigma'. \llbracket$
 $x' = \alpha x; x \in \text{it}; x' \in \text{it}'; \text{it}' = \alpha \text{it}; \text{it} \subseteq S; \text{it}' \subseteq S';$
 $(\sigma, \sigma') \in R$
 $\rrbracket \Rightarrow f \ x \ \sigma \leq \Downarrow R \ (f' \ x' \ \sigma')$
shows $FOREACH \ S \ f \ \sigma 0 \leq \Downarrow R \ (FOREACH \ S' \ f' \ \sigma 0')$
 $\langle proof \rangle$

lemma $FOREACHci\text{-refine}\text{-rcg}'[refine]:$

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes $INJ: inj\text{-on } \alpha S$
assumes $REFS: S' = \alpha'S$
assumes $REF0: (\sigma 0, \sigma 0') \in R$
assumes $REFC: \bigwedge \text{it } \sigma \ \text{it}' \ \sigma'. \llbracket$
 $\text{it}' = \alpha \text{it}; \text{it} \subseteq S; \text{it}' \subseteq S'; \Phi' \ \text{it}' \ \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \Rightarrow c \ \sigma \longleftrightarrow c' \ \sigma'$
assumes $REFSTEP: \bigwedge x \ \text{it } \sigma \ x' \ \text{it}' \ \sigma'. \llbracket$
 $x' = \alpha x; x \in \text{it}; x' \in \text{it}'; \text{it}' = \alpha \text{it}; \text{it} \subseteq S; \text{it}' \subseteq S';$
 $\Phi' \ \text{it}' \ \sigma'; c \ \sigma; c' \ \sigma';$
 $(\sigma, \sigma') \in R$
 $\rrbracket \Rightarrow f \ x \ \sigma \leq \Downarrow R \ (f' \ x' \ \sigma')$
shows $FOREACHc \ S \ c \ f \ \sigma 0 \leq \Downarrow R \ (FOREACHci \ \Phi' \ S' \ c' \ f' \ \sigma 0')$
 $\langle proof \rangle$

lemma $FOREACHi\text{-refine}\text{-rcg}'[refine]:$

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes $INJ: inj\text{-on } \alpha S$
assumes $REFS: S' = \alpha'S$
assumes $REF0: (\sigma 0, \sigma 0') \in R$
assumes $REFSTEP: \bigwedge x \ \text{it } \sigma \ x' \ \text{it}' \ \sigma'. \llbracket$
 $x' = \alpha x; x \in \text{it}; x' \in \text{it}'; \text{it}' = \alpha \text{it}; \text{it} \subseteq S; \text{it}' \subseteq S';$
 $\Phi' \ \text{it}' \ \sigma';$
 $(\sigma, \sigma') \in R$
 $\rrbracket \Rightarrow f \ x \ \sigma \leq \Downarrow R \ (f' \ x' \ \sigma')$
shows $FOREACH \ S \ f \ \sigma 0 \leq \Downarrow R \ (FOREACHi \ \Phi' \ S' \ f' \ \sigma 0')$
 $\langle proof \rangle$

Alternative set of FOREACHc-rules

Here, we provide an alternative set of FOREACH rules with interruption. In some cases, they are easier to use, as they avoid redundancy between the final cases for interruption and non-interruption

lemma *FOREACHoci-rule'*:

assumes *FIN*: finite *S*

assumes *I0*: $I\ S\ \sigma\ \theta$

assumes *IP*:

$$\bigwedge x\ it\ \sigma. \llbracket c\ \sigma; x \in it; it \subseteq S; I\ it\ \sigma; \forall y \in it - \{x\}. R\ x\ y; \\ \forall y \in S - it. R\ y\ x \rrbracket \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it - \{x\}))$$

assumes *III1*: $\bigwedge \sigma. \llbracket I\ \{\} \sigma; c\ \sigma \rrbracket \Longrightarrow P\ \sigma$

assumes *III2*: $\bigwedge it\ \sigma. \llbracket it \subseteq S; I\ it\ \sigma; \neg c\ \sigma; \\ \forall x \in it. \forall y \in S - it. R\ y\ x \rrbracket \Longrightarrow P\ \sigma$

shows *FOREACHoci* $R\ I\ S\ c\ f\ \sigma\ \theta \leq SPEC\ P$

<proof>

lemma *FOREACHci-rule'*[*refine-vcg*]:

assumes *FIN*: finite *S*

assumes *I0*: $I\ S\ \sigma\ \theta$

assumes *IP*:

$$\bigwedge x\ it\ \sigma. \llbracket x \in it; it \subseteq S; I\ it\ \sigma; c\ \sigma \rrbracket \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it - \{x\}))$$

assumes *III1*: $\bigwedge \sigma. \llbracket I\ \{\} \sigma; c\ \sigma \rrbracket \Longrightarrow P\ \sigma$

assumes *III2*: $\bigwedge it\ \sigma. \llbracket it \subseteq S; I\ it\ \sigma; \neg c\ \sigma \rrbracket \Longrightarrow P\ \sigma$

shows *FOREACHci* $I\ S\ c\ f\ \sigma\ \theta \leq SPEC\ P$

<proof>

lemma *FOREACHc-rule'*:

assumes *FIN*: finite *S*

assumes *I0*: $I\ S\ \sigma\ \theta$

assumes *IP*:

$$\bigwedge x\ it\ \sigma. \llbracket x \in it; it \subseteq S; I\ it\ \sigma; c\ \sigma \rrbracket \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it - \{x\}))$$

assumes *III1*: $\bigwedge \sigma. \llbracket I\ \{\} \sigma; c\ \sigma \rrbracket \Longrightarrow P\ \sigma$

assumes *III2*: $\bigwedge it\ \sigma. \llbracket it \subseteq S; I\ it\ \sigma; \neg c\ \sigma \rrbracket \Longrightarrow P\ \sigma$

shows *FOREACHc* $S\ c\ f\ \sigma\ \theta \leq SPEC\ P$

<proof>

2.15.4 FOREACH with empty sets

lemma *FOREACHoci-emp* [*simp*] :

FOREACHoci $R\ \Phi\ \{\} c\ f\ \sigma = do\ \{ASSERT\ (\Phi\ \{\})\ \sigma; RETURN\ \sigma\}$

<proof>

lemma *FOREACHoi-emp* [*simp*] :

FOREACHoi $R\ \Phi\ \{\} f\ \sigma = do\ \{ASSERT\ (\Phi\ \{\})\ \sigma; RETURN\ \sigma\}$

<proof>

lemma *FOREACHci-emp* [*simp*] :

FOREACHci $\Phi\ \{\} c\ f\ \sigma = do\ \{ASSERT\ (\Phi\ \{\})\ \sigma; RETURN\ \sigma\}$

<proof>

lemma *FOREACHc-emp* [*simp*] :
 $FOREACHc \{ \} c f \sigma = RETURN \sigma$
<proof>

lemma *FOREACH-emp* [*simp*] :
 $FOREACH \{ \} f \sigma = RETURN \sigma$
<proof>

lemma *FOREACHi-emp* [*simp*] :
 $FOREACHi \Phi \{ \} f \sigma = do \{ ASSERT (\Phi \{ \} \sigma); RETURN \sigma \}$
<proof>

2.15.5 Monotonicity

definition *lift-refl* $P c f g == \forall x. P c (f x) (g x)$

definition *lift-mono* $P c f g == \forall x y. c x y \longrightarrow P c (f x) (g y)$

definition *lift-mono1* $P c f g == \forall x y. (\forall a. c (x a) (y a)) \longrightarrow P c (f x) (g y)$

definition *lift-mono2* $P c f g == \forall x y. (\forall a b. c (x a b) (y a b)) \longrightarrow P c (f x) (g y)$

definition *trimono-spec* $L f == ((L id (\leq) f f) \wedge (L id flat-ge f f))$

lemmas *trimono-atomize* = *atomize-imp atomize-conj atomize-all*

lemmas *trimono-deatomize* = *trimono-atomize[symmetric]*

lemmas *trimono-spec-defs* = *trimono-spec-def lift-refl-def[abs-def] comp-def id-def*
lift-mono-def[abs-def] lift-mono1-def[abs-def] lift-mono2-def[abs-def]
trimono-deatomize

locale *trimono-spec* **begin**

abbreviation $R \equiv lift-refl$

abbreviation $M \equiv lift-mono$

abbreviation $M1 \equiv lift-mono1$

abbreviation $M2 \equiv lift-mono2$

end

context **begin** **interpretation** *trimono-spec* *<proof>*

lemma *FOREACHoci-mono*[*unfolded trimono-spec-defs,refine-mono*]:

trimono-spec $(R o R o R o R o M2 o R) FOREACHoci$

trimono-spec $(R o R o R o M2 o R) FOREACHoi$

trimono-spec $(R o R o R o M2 o R) FOREACHci$

trimono-spec $(R o R o M2 o R) FOREACHc$

trimono-spec $(R o R o M2 o R) FOREACHi$

trimono-spec $(R o M2 o R) FOREACH$

<proof>

end

2.15.6 Nres-Fold with Interruption (nfoldli)

A foreach-loop can be conveniently expressed as an operation that converts the set to a list, followed by folding over the list.

This representation is handy for automatic refinement, as the complex foreach-operation is expressed by two relatively simple operations.

We first define a fold-function in the nres-monad

partial-function (*nrec*) *nfoldli* **where**
 $nfoldli\ l\ c\ f\ s = (case\ l\ of$
 $\ \ [] \Rightarrow RETURN\ s$
 $\ | x\#\!ls \Rightarrow if\ c\ s\ then\ do\ \{ s\leftarrow f\ x\ s;\ nfoldli\ ls\ c\ f\ s\}\ else\ RETURN\ s$
 $\)$

lemma *nfoldli-simps*[*simp*]:
 $nfoldli\ []\ c\ f\ s = RETURN\ s$
 $nfoldli\ (x\#\!ls)\ c\ f\ s =$
 $\ (if\ c\ s\ then\ do\ \{ s\leftarrow f\ x\ s;\ nfoldli\ ls\ c\ f\ s\}\ else\ RETURN\ s)$
 $\langle proof \rangle$

lemma *param-nfoldli*[*param*]:
shows $(nfoldli, nfoldli) \in$
 $\langle Ra \rangle list\text{-}rel \rightarrow (Rb \rightarrow Id) \rightarrow (Ra \rightarrow Rb \rightarrow \langle Rb \rangle nres\text{-}rel) \rightarrow Rb \rightarrow \langle Rb \rangle nres\text{-}rel$
 $\langle proof \rangle$

lemma *nfoldli-no-ctd*[*simp*]: $\neg ctd\ s \implies nfoldli\ l\ ctd\ f\ s = RETURN\ s$
 $\langle proof \rangle$

lemma *nfoldli-append*[*simp*]: $nfoldli\ (l1\ @\ l2)\ ctd\ f\ s = nfoldli\ l1\ ctd\ f\ s \gg= nfoldli$
 $\ l2\ ctd\ f$
 $\langle proof \rangle$

lemma *nfoldli-map*: $nfoldli\ (map\ f\ l)\ ctd\ g\ s = nfoldli\ l\ ctd\ (g\ o\ f)\ s$
 $\langle proof \rangle$

lemma *nfoldli-nfoldli-prod-conv*:
 $nfoldli\ l2\ ctd\ (\lambda i. nfoldli\ l1\ ctd\ (f\ i))\ s = nfoldli\ (List.product\ l2\ l1)\ ctd\ (\lambda(i,j).$
 $\ f\ i\ j)\ s$
 $\langle proof \rangle$

The fold-function over the nres-monad is transferred to a plain foldli function

lemma *nfoldli-transfer-plain*[*refine-transfer*]:
assumes $\bigwedge x\ s. RETURN\ (f\ x\ s) \leq f'\ x\ s$
shows $RETURN\ (foldli\ l\ c\ f\ s) \leq (nfoldli\ l\ c\ f'\ s)$
 $\langle proof \rangle$

lemma *nfoldli-transfer-dres*[*refine-transfer*]:
fixes $l :: 'a \text{ list}$ **and** $c :: 'b \Rightarrow \text{bool}$
assumes $FR: \bigwedge x s. \text{nres-of } (f x s) \leq f' x s$
shows nres-of
 $(\text{foldli } l \text{ (case-dres False False } c) (\lambda x s. s \gg f x) (\text{dRETURN } s))$
 $\leq (\text{nfoldli } l c f' s)$
 $\langle \text{proof} \rangle$

lemma *nfoldli-mono*[*refine-mono*]:
 $\llbracket \bigwedge x s. f x s \leq f' x s \rrbracket \Longrightarrow \text{nfoldli } l c f \sigma \leq \text{nfoldli } l c f' \sigma$
 $\llbracket \bigwedge x s. \text{flat-ge } (f x s) (f' x s) \rrbracket \Longrightarrow \text{flat-ge } (\text{nfoldli } l c f \sigma) (\text{nfoldli } l c f' \sigma)$
 $\langle \text{proof} \rangle$

We relate our fold-function to the while-loop that we used in the original definition of the foreach-loop

lemma *nfoldli-while*: $\text{nfoldli } l c f \sigma$
 \leq
 $(\text{WHILE}_T^I$
 $(\text{FOREACH-cond } c) (\text{FOREACH-body } f) (l, \sigma) \gg$
 $(\lambda(-, \sigma). \text{RETURN } \sigma))$
 $\langle \text{proof} \rangle$

lemma *while-nfoldli*:
 $\text{do } \{$
 $(-, \sigma) \leftarrow \text{WHILE}_T (\text{FOREACH-cond } c) (\text{FOREACH-body } f) (l, \sigma);$
 $\text{RETURN } \sigma$
 $\} \leq \text{nfoldli } l c f \sigma$
 $\langle \text{proof} \rangle$

lemma *while-eq-nfoldli*: $\text{do } \{$
 $(-, \sigma) \leftarrow \text{WHILE}_T (\text{FOREACH-cond } c) (\text{FOREACH-body } f) (l, \sigma);$
 $\text{RETURN } \sigma$
 $\} = \text{nfoldli } l c f \sigma$
 $\langle \text{proof} \rangle$

lemma *nfoldli-rule*:
assumes $I0: I \llbracket l0 \sigma0$
assumes $IS: \bigwedge x l1 l2 \sigma. \llbracket l0 = l1 @ x \# l2; I l1 (x \# l2) \sigma; c \sigma \rrbracket \Longrightarrow f x \sigma \leq \text{SPEC}$
 $(I (l1 @ [x]) l2)$
assumes $FNC: \bigwedge l1 l2 \sigma. \llbracket l0 = l1 @ l2; I l1 l2 \sigma; \neg c \sigma \rrbracket \Longrightarrow P \sigma$
assumes $FC: \bigwedge \sigma. \llbracket I l0 \rrbracket \sigma; c \sigma \rrbracket \Longrightarrow P \sigma$
shows $\text{nfoldli } l0 c f \sigma0 \leq \text{SPEC } P$
 $\langle \text{proof} \rangle$

lemma *nfoldli-leof-rule*:
assumes $I0: I \llbracket l0 \sigma0$
assumes $IS: \bigwedge x l1 l2 \sigma. \llbracket l0 = l1 @ x \# l2; I l1 (x \# l2) \sigma; c \sigma \rrbracket \Longrightarrow f x \sigma \leq_n \text{SPEC}$
 $(I (l1 @ [x]) l2)$
assumes $FNC: \bigwedge l1 l2 \sigma. \llbracket l0 = l1 @ l2; I l1 l2 \sigma; \neg c \sigma \rrbracket \Longrightarrow P \sigma$

assumes $FC: \bigwedge \sigma. \llbracket I \ l0 \rrbracket \sigma; c \ \sigma \rrbracket \Longrightarrow P \ \sigma$
shows $\text{nfoldli } l0 \ c \ f \ \sigma 0 \leq_n \text{SPEC } P$
 <proof>

lemma $\text{nfoldli-refine}[\text{refine}]$:
assumes $(li, l) \in \langle S \rangle \text{list-rel}$
and $(si, s) \in R$
and $CR: (ci, c) \in R \rightarrow \text{bool-rel}$
and $[\text{refine}]: \bigwedge xi \ x \ si \ s. \llbracket (xi, x) \in S; (si, s) \in R; c \ s \rrbracket \Longrightarrow fi \ xi \ si \leq \Downarrow R \ (f \ x \ s)$
shows $\text{nfoldli } li \ ci \ fi \ si \leq \Downarrow R \ (\text{nfoldli } l \ c \ f \ s)$
 <proof>

lemma $\text{nfoldli-invar-refine}$:
assumes $(li, l) \in \langle S \rangle \text{list-rel}$
assumes $(si, s) \in R$
assumes $I \llbracket li \ si$
assumes $COND: \bigwedge l1i \ l2i \ l1 \ l2 \ si \ s. \llbracket$
 $li = l1i @ l2i; l = l1 @ l2; (l1i, l1) \in \langle S \rangle \text{list-rel}; (l2i, l2) \in \langle S \rangle \text{list-rel};$
 $I \ l1i \ l2i \ si; (si, s) \in R \rrbracket \Longrightarrow (ci \ si, c \ s) \in \text{bool-rel}$
assumes $INV: \bigwedge l1i \ xi \ l2i \ si. \llbracket li = l1i @ xi \# l2i; I \ l1i \ (xi \# l2i) \ si \rrbracket \Longrightarrow fi \ xi \ si \leq_n$
 $\text{SPEC } (I \ (l1i @ [xi]) \ l2i)$
assumes $STEP: \bigwedge l1i \ xi \ l2i \ l1 \ x \ l2 \ si \ s. \llbracket$
 $li = l1i @ xi \# l2i; l = l1 @ x \# l2; (l1i, l1) \in \langle S \rangle \text{list-rel}; (xi, x) \in S; (l2i, l2) \in \langle S \rangle \text{list-rel};$
 $I \ l1i \ (xi \# l2i) \ si; (si, s) \in R \rrbracket \Longrightarrow fi \ xi \ si \leq \Downarrow R \ (f \ x \ s)$
shows $\text{nfoldli } li \ ci \ fi \ si \leq \Downarrow R \ (\text{nfoldli } l \ c \ f \ s)$
 <proof>

lemma $\text{foldli-mono-dres-aux1}$:
fixes $\sigma :: 'a :: \{\text{order-bot}, \text{order-top}\}$
assumes $COND: \bigwedge \sigma \ \sigma'. \sigma \leq \sigma' \Longrightarrow c \ \sigma \neq c \ \sigma' \Longrightarrow \sigma = \text{bot} \vee \sigma' = \text{top}$
assumes $STRICT: \bigwedge x. f \ x \ \text{bot} = \text{bot} \quad \bigwedge x. f' \ x \ \text{top} = \text{top}$
assumes $B: \sigma \leq \sigma'$
assumes $A: \bigwedge a \ x \ x'. x \leq x' \Longrightarrow f \ a \ x \leq f' \ a \ x'$
shows $\text{foldli } l \ c \ f \ \sigma \leq \text{foldli } l \ c \ f' \ \sigma'$
 <proof>

lemma $\text{foldli-mono-dres-aux2}$:
assumes $STRICT: \bigwedge x. f \ x \ \text{bot} = \text{bot} \quad \bigwedge x. f' \ x \ \text{top} = \text{top}$
assumes $A: \bigwedge a \ x \ x'. x \leq x' \Longrightarrow f \ a \ x \leq f' \ a \ x'$
shows $\text{foldli } l \ (\text{case-dres } \text{False } \text{False } c) \ f \ \sigma$
 $\leq \text{foldli } l \ (\text{case-dres } \text{False } \text{False } c) \ f' \ \sigma$
 <proof>

lemma $\text{foldli-mono-dres}[\text{refine-mono}]$:
assumes $A: \bigwedge a \ x. f \ a \ x \leq f' \ a \ x$

shows $\text{foldli } l \text{ (case-dres False False } c) (\lambda x s. \text{dbind } s (f x)) \sigma$
 $\leq \text{foldli } l \text{ (case-dres False False } c) (\lambda x s. \text{dbind } s (f' x)) \sigma$
 $\langle \text{proof} \rangle$

partial-function (*drec*) *dfoldli* **where**

$\text{dfoldli } l \text{ } c \text{ } f \text{ } s = (\text{case } l \text{ of}$
 $\quad \square \Rightarrow \text{dRETURN } s$
 $\quad | x\#ls \Rightarrow \text{if } c \text{ } s \text{ then do } \{ s \leftarrow f \text{ } x \text{ } s; \text{dfoldli } ls \text{ } c \text{ } f \text{ } s \} \text{ else dRETURN } s$
 $\quad)$

lemma *dfoldli-simps*[*simp*]:

$\text{dfoldli } \square \text{ } c \text{ } f \text{ } s = \text{dRETURN } s$
 $\text{dfoldli } (x\#ls) \text{ } c \text{ } f \text{ } s =$
 $\quad (\text{if } c \text{ } s \text{ then do } \{ s \leftarrow f \text{ } x \text{ } s; \text{dfoldli } ls \text{ } c \text{ } f \text{ } s \} \text{ else dRETURN } s)$
 $\langle \text{proof} \rangle$

lemma *dfoldli-mono*[*refine-mono*]:

$\llbracket \bigwedge x s. f \text{ } x \text{ } s \leq f' \text{ } x \text{ } s \rrbracket \Longrightarrow \text{dfoldli } l \text{ } c \text{ } f \text{ } \sigma \leq \text{dfoldli } l \text{ } c \text{ } f' \text{ } \sigma$
 $\llbracket \bigwedge x s. \text{flat-ge } (f \text{ } x \text{ } s) (f' \text{ } x \text{ } s) \rrbracket \Longrightarrow \text{flat-ge } (\text{dfoldli } l \text{ } c \text{ } f \text{ } \sigma) (\text{dfoldli } l \text{ } c \text{ } f' \text{ } \sigma)$
 $\langle \text{proof} \rangle$

lemma *foldli-dres-pres-FAIL*[*simp*]:

$\text{foldli } l \text{ (case-dres False False } c) (\lambda x s. \text{dbind } s (f x)) \text{dFAIL} = \text{dFAIL}$
 $\langle \text{proof} \rangle$

lemma *foldli-dres-pres-SUCCEED*[*simp*]:

$\text{foldli } l \text{ (case-dres False False } c) (\lambda x s. \text{dbind } s (f x)) \text{dSUCCEED} = \text{dSUCCEED}$
 $\langle \text{proof} \rangle$

lemma *dfoldli-by-foldli*: $\text{dfoldli } l \text{ } c \text{ } f \text{ } \sigma$

$= \text{foldli } l \text{ (case-dres False False } c) (\lambda x s. \text{dbind } s (f x)) (\text{dRETURN } \sigma)$
 $\langle \text{proof} \rangle$

lemma *foldli-mono-dres-flat*[*refine-mono*]:

assumes $A: \bigwedge a x. \text{flat-ge } (f \text{ } a \text{ } x) (f' \text{ } a \text{ } x)$
shows $\text{flat-ge } (\text{foldli } l \text{ (case-dres False False } c) (\lambda x s. \text{dbind } s (f x)) \sigma)$
 $\quad (\text{foldli } l \text{ (case-dres False False } c) (\lambda x s. \text{dbind } s (f' x)) \sigma)$
 $\langle \text{proof} \rangle$

lemma *dres-foldli-ne-bot*[*refine-transfer*]:

assumes 1: $\sigma \neq \text{dSUCCEED}$
assumes 2: $\bigwedge x \sigma. f \text{ } x \text{ } \sigma \neq \text{dSUCCEED}$
shows $\text{foldli } l \text{ } c \text{ } (\lambda x s. s \gg= f \text{ } x) \sigma \neq \text{dSUCCEED}$
 $\langle \text{proof} \rangle$

2.15.7 LIST FOREACH combinator

Foreach-loops are mapped to the combinator *LIST-FOREACH*, that takes as first argument an explicit *to-list* operation. This mapping is done during operation identification. It is then the responsibility of the various implementations to further map the *to-list* operations to custom *to-list* operations, like *set-to-list*, *map-to-list*, *nodes-to-list*, etc.

We define a relation between distinct lists and sets.

definition [*to-relAPP*]: *list-set-rel* $R \equiv \langle R \rangle \text{list-rel } O \text{ br set distinct}$

lemma *autoref-nfoldli*[*autoref-rules*]:

shows (*nfoldli*, *nfoldli*)
 $\in \langle Ra \rangle \text{list-rel} \rightarrow (Rb \rightarrow \text{bool-rel}) \rightarrow (Ra \rightarrow Rb \rightarrow \langle Rb \rangle \text{nres-rel}) \rightarrow Rb \rightarrow \langle Rb \rangle \text{nres-rel}$
<proof>

This constant is a placeholder to be converted to custom operations by pattern rules

definition *it-to-sorted-list* $R \ s$

$\equiv \text{SPEC } (\lambda l. \text{distinct } l \wedge s = \text{set } l \wedge \text{sorted-wrt } R \ l)$

definition *LIST-FOREACH* $\Phi \ \text{tsl} \ c \ f \ \sigma \ 0 \equiv \text{do } \{$

$\ \text{xs} \leftarrow \text{tsl};$
 $\ (-, \sigma) \leftarrow \text{WHILE}_T^{\lambda(it, \sigma). \exists \text{xs}'. \text{xs} = \text{xs}' @ it \wedge \Phi(\text{set } it) \sigma}$
 $\ (\text{FOREACH-cond } c) (\text{FOREACH-body } f) (\text{xs}, \sigma 0);$
 $\ \text{RETURN } \sigma \}$

lemma *FOREACHoci-by-LIST-FOREACH*:

$\text{FOREACHoci } R \ \Phi \ S \ c \ f \ \sigma \ 0 = \text{do } \{$
 $\ \text{ASSERT } (\text{finite } S);$
 $\ \text{LIST-FOREACH } \Phi \ (\text{it-to-sorted-list } R \ S) \ c \ f \ \sigma \ 0$
 $\ \}$
<proof>

Patterns that convert FOREACH-constructs to *LIST-FOREACH*

context begin interpretation *autoref-syn* *<proof>*

lemma *FOREACH-patterns*[*autoref-op-pat-def*]:

$\text{FOREACH}^I \ s \ f \equiv \text{FOREACH}_{OC}^{\lambda-. \text{True}, I} \ s \ (\lambda-. \text{True}) \ f$
 $\text{FOREACHci} \ I \ s \ c \ f \equiv \text{FOREACHoci} \ (\lambda-. \text{True}) \ I \ s \ c \ f$
 $\text{FOREACH}_{OC}^{R, \Phi} \ s \ c \ f \equiv \lambda \sigma. \text{do } \{$
 $\ \text{ASSERT } (\text{finite } s);$
 $\ \text{Autoref-Tagging.OP } (\text{LIST-FOREACH } \Phi) (\text{it-to-sorted-list } R \ s) \ c \ f \ \sigma$
 $\ \}$
 $\text{FOREACH} \ s \ f \equiv \text{FOREACHoci} \ (\lambda-. \text{True}) \ (\lambda-. \text{True}) \ s \ (\lambda-. \text{True}) \ f$
 $\text{FOREACHoi} \ R \ I \ s \ f \equiv \text{FOREACHoci} \ R \ I \ s \ (\lambda-. \text{True}) \ f$

$FOREACHc\ s\ c\ f \equiv FOREACHoci\ (\lambda\ -. \ True)\ (\lambda\ -. \ True)\ s\ c\ f$
 $\langle proof \rangle$

end

definition $LIST-FOREACH'\ tsl\ c\ f\ \sigma \equiv do\ \{xs \leftarrow\ tsl;\ nfoldli\ xs\ c\ f\ \sigma\}$

lemma $LIST-FOREACH'\text{-param}[param]:$

shows $(LIST-FOREACH', LIST-FOREACH')$
 $\in (\langle\langle Rv \rangle list-rel \rangle nres-rel \rightarrow (R\sigma \rightarrow bool-rel)$
 $\rightarrow (Rv \rightarrow R\sigma \rightarrow \langle R\sigma \rangle nres-rel) \rightarrow R\sigma \rightarrow \langle R\sigma \rangle nres-rel)$
 $\langle proof \rangle$

lemma $LIST-FOREACH\text{-autoref}[autoref-rules]:$

shows $(LIST-FOREACH', LIST-FOREACH\ \Phi) \in$
 $(\langle\langle Rv \rangle list-rel \rangle nres-rel \rightarrow (R\sigma \rightarrow bool-rel)$
 $\rightarrow (Rv \rightarrow R\sigma \rightarrow \langle R\sigma \rangle nres-rel) \rightarrow R\sigma \rightarrow \langle R\sigma \rangle nres-rel)$
 $\langle proof \rangle$

context begin interpretation $trimono-spec\ \langle proof \rangle$

lemma $LIST-FOREACH'\text{-mono}[unfolded\ trimono-spec-defs, refine-mono]:$

$trimono-spec\ (R\ o\ R\ o\ M2\ o\ R)\ LIST-FOREACH'$
 $\langle proof \rangle$

end

lemma $LIST-FOREACH'\text{-transfer-plain}[refine-transfer]:$

assumes $RETURN\ tsl \leq\ tsl'$
assumes $\bigwedge x\ \sigma. RETURN\ (f\ x\ \sigma) \leq\ f'\ x\ \sigma$
shows $RETURN\ (foldli\ tsl\ c\ f\ \sigma) \leq\ LIST-FOREACH'\ tsl'\ c\ f'\ \sigma$
 $\langle proof \rangle$

thm $refine-transfer$

lemma $LIST-FOREACH'\text{-transfer-nres}[refine-transfer]:$

assumes $nres-of\ tsl \leq\ tsl'$
assumes $\bigwedge x\ \sigma. nres-of\ (f\ x\ \sigma) \leq\ f'\ x\ \sigma$
shows $nres-of\ ($
 $do\ \{$
 $xs \leftarrow\ tsl;$
 $foldli\ xs\ (case-dres\ False\ False\ c)\ (\lambda x\ s. s \gg\! =\ f\ x)\ (dRETURN\ \sigma)$
 $\}) \leq\ LIST-FOREACH'\ tsl'\ c\ f'\ \sigma$
 $\langle proof \rangle$

Simplification rules to summarize iterators

lemma $[refine-transfer-post-simp]:$

$do\ \{$
 $xs \leftarrow\ dRETURN\ tsl;$

$\text{foldli } xs \ c \ f \ \sigma$
 $\} = \text{foldli } \text{tsl } c \ f \ \sigma$
 $\langle \text{proof} \rangle$

lemma [*refine-transfer-post-simp*]:
 $(\text{let } xs = \text{tsl in foldli } xs \ c \ f \ \sigma) = \text{foldli } \text{tsl } c \ f \ \sigma$
 $\langle \text{proof} \rangle$

lemma *LFO-pre-refine*:
assumes $(li, l) \in \langle A \rangle \text{list-set-rel}$
assumes $(ci, c) \in R \rightarrow \text{bool-rel}$
assumes $(fi, f) \in A \rightarrow R \rightarrow \langle R \rangle \text{nres-rel}$
assumes $(s0i, s0) \in R$
shows $\text{LIST-FOREACH}' (\text{RETURN } li) \ ci \ fi \ s0i \leq \Downarrow R (\text{FOREACH} \ ci \ l \ c \ f \ s0)$
 $\langle \text{proof} \rangle$

lemma *LFOci-refine*:
assumes $(li, l) \in \langle A \rangle \text{list-set-rel}$
assumes $\bigwedge s \ si. (si, s) \in R \implies ci \ si \longleftrightarrow c \ s$
assumes $\bigwedge x \ xi \ s \ si. \llbracket (xi, x) \in A; (si, s) \in R \rrbracket \implies fi \ xi \ si \leq \Downarrow R (f \ x \ s)$
assumes $(s0i, s0) \in R$
shows $\text{nfoldli } li \ ci \ fi \ s0i \leq \Downarrow R (\text{FOREACH} \ ci \ l \ c \ f \ s0)$
 $\langle \text{proof} \rangle$

lemma *LFOc-refine*:
assumes $(li, l) \in \langle A \rangle \text{list-set-rel}$
assumes $\bigwedge s \ si. (si, s) \in R \implies ci \ si \longleftrightarrow c \ s$
assumes $\bigwedge x \ xi \ s \ si. \llbracket (xi, x) \in A; (si, s) \in R \rrbracket \implies fi \ xi \ si \leq \Downarrow R (f \ x \ s)$
assumes $(s0i, s0) \in R$
shows $\text{nfoldli } li \ ci \ fi \ s0i \leq \Downarrow R (\text{FOREACH} \ c \ l \ c \ f \ s0)$
 $\langle \text{proof} \rangle$

lemma *LFO-refine*:
assumes $(li, l) \in \langle A \rangle \text{list-set-rel}$
assumes $\bigwedge x \ xi \ s \ si. \llbracket (xi, x) \in A; (si, s) \in R \rrbracket \implies fi \ xi \ si \leq \Downarrow R (f \ x \ s)$
assumes $(s0i, s0) \in R$
shows $\text{nfoldli } li \ (\lambda-. \text{True}) \ fi \ s0i \leq \Downarrow R (\text{FOREACH} \ l \ f \ s0)$
 $\langle \text{proof} \rangle$

lemma *LFOi-refine*:
assumes $(li, l) \in \langle A \rangle \text{list-set-rel}$
assumes $\bigwedge x \ xi \ s \ si. \llbracket (xi, x) \in A; (si, s) \in R \rrbracket \implies fi \ xi \ si \leq \Downarrow R (f \ x \ s)$
assumes $(s0i, s0) \in R$
shows $\text{nfoldli } li \ (\lambda-. \text{True}) \ fi \ s0i \leq \Downarrow R (\text{FOREACH} \ i \ l \ f \ s0)$
 $\langle \text{proof} \rangle$

lemma *LIST-FOREACH'-refine*: $LIST-FOREACH' \text{ tsl}' c' f' \sigma' \leq LIST-FOREACH \Phi \text{ tsl}' c' f' \sigma'$
 <proof>

lemma *LIST-FOREACH'-eq*: $LIST-FOREACH (\lambda-. True) \text{ tsl}' c' f' \sigma' = (LIST-FOREACH' \text{ tsl}' c' f' \sigma')$
 <proof>

2.15.8 FOREACH with duplicates

definition *FOREACHcd* $S c f \sigma \equiv do \{$
ASSERT (*finite* S);
 $l \leftarrow SPEC (\lambda l. set\ l = S)$;
nfoldli $l c f \sigma$
 $\}$

lemma *FOREACHcd-rule*:

assumes *finite* S_0
assumes *I0*: $I \{ \} S_0 \sigma_0$
assumes *STEP*: $\bigwedge S1\ S2\ x\ \sigma. \llbracket S_0 = insert\ x\ (S1 \cup S2); I\ S1\ (insert\ x\ S2)\ \sigma; c\ \sigma \rrbracket \implies f\ x\ \sigma \leq SPEC\ (I\ (insert\ x\ S1)\ S2)$
assumes *INTR*: $\bigwedge S1\ S2\ \sigma. \llbracket S_0 = S1 \cup S2; I\ S1\ S2\ \sigma; \neg c\ \sigma \rrbracket \implies \Phi\ \sigma$
assumes *COMPL*: $\bigwedge \sigma. \llbracket I\ S_0\ \{ \} \sigma; c\ \sigma \rrbracket \implies \Phi\ \sigma$
shows $FOREACHcd\ S_0\ c\ f\ \sigma_0 \leq SPEC\ \Phi$
 <proof>

definition *FOREACHcdi*

$:: ('a\ set \implies 'a\ set \implies 'b \implies bool)$
 $\implies 'a\ set \implies ('b \implies bool) \implies ('a \implies 'b \implies 'b\ nres) \implies 'b \implies 'b\ nres$

where

$FOREACHcdi\ I \equiv FOREACHcd$

lemma *FOREACHcdi-rule[refine-vcg]*:

assumes *finite* S_0
assumes *I0*: $I \{ \} S_0 \sigma_0$
assumes *STEP*: $\bigwedge S1\ S2\ x\ \sigma. \llbracket S_0 = insert\ x\ (S1 \cup S2); I\ S1\ (insert\ x\ S2)\ \sigma; c\ \sigma \rrbracket \implies f\ x\ \sigma \leq SPEC\ (I\ (insert\ x\ S1)\ S2)$
assumes *INTR*: $\bigwedge S1\ S2\ \sigma. \llbracket S_0 = S1 \cup S2; I\ S1\ S2\ \sigma; \neg c\ \sigma \rrbracket \implies \Phi\ \sigma$
assumes *COMPL*: $\bigwedge \sigma. \llbracket I\ S_0\ \{ \} \sigma; c\ \sigma \rrbracket \implies \Phi\ \sigma$
shows $FOREACHcdi\ I\ S_0\ c\ f\ \sigma_0 \leq SPEC\ \Phi$
 <proof>

lemma *FOREACHcd-refine[refine]*:

assumes [*simp*]: *finite* s'
assumes $S: (s', s) \in \langle S \rangle\ set\ rel$
assumes $SV: single\ valued\ S$
assumes $R0: (\sigma', \sigma) \in R$
assumes $C: \bigwedge \sigma' \sigma. (\sigma', \sigma) \in R \implies (c' \sigma', c \sigma) \in bool\ rel$
assumes $F: \bigwedge x' x \sigma' \sigma. \llbracket (x', x) \in S; (\sigma', \sigma) \in R \rrbracket$

$\implies f' x' \sigma' \leq \Downarrow R (f x \sigma)$
shows $FOREACHcd s' c' f' \sigma' \leq \Downarrow R (FOREACHcdi I s c f \sigma)$
 ⟨proof⟩

lemma *FOREACHc-refines-FOREACHcd-aux*:
shows $FOREACHc s c f \sigma \leq FOREACHcd s c f \sigma$
 ⟨proof⟩

lemmas *FOREACHc-refines-FOREACHcd[refine]*
 $= \text{order-trans}[OF FOREACHc-refines-FOREACHcd-aux FOREACHcd-refine]$

2.15.9 Miscellaneous Utility Lemmas

lemma *map-foreach*:
assumes *finite S*
shows $FOREACH S (\lambda x \sigma. RETURN (insert (f x) \sigma)) R0 \leq SPEC ((=) (R0 \cup f'S))$
 ⟨proof⟩

lemma *map-sigma-foreach*:
fixes $f :: 'a \times 'b \Rightarrow 'c$
assumes *finite A*
assumes $\bigwedge x. x \in A \implies \text{finite } (B x)$
shows $FOREACH A (\lambda a \sigma. FOREACH (B a) (\lambda b \sigma. RETURN (insert (f (a,b)) \sigma)) \sigma) R0 \leq SPEC ((=) (R0 \cup f'Sigma A B))$
 ⟨proof⟩

lemma *map-sigma-sigma-foreach*:
fixes $f :: 'a \times ('b \times 'c) \Rightarrow 'd$
assumes *finite A*
assumes $\bigwedge a. a \in A \implies \text{finite } (B a)$
assumes $\bigwedge a b. \llbracket a \in A; b \in B a \rrbracket \implies \text{finite } (C a b)$
shows $FOREACH A (\lambda a \sigma. FOREACH (B a) (\lambda b \sigma. FOREACH (C a b) (\lambda c \sigma. RETURN (insert (f (a,(b,c))) \sigma)) \sigma) \sigma) R0 \leq SPEC ((=) (R0 \cup f'Sigma A (\lambda a. Sigma (B a) (C a))))$
 ⟨proof⟩

lemma *bij-set-rel-for-inj*:
fixes R
defines $\alpha \equiv \text{fun-of-rel } R$
assumes *bijective R* $(s,s') \in \langle R \rangle \text{set-rel}$
shows *inj-on* $\alpha s \quad s' = \alpha s$
 — To be used when generating refinement conditions for foreach-loops

<proof>

lemma *nfoldli-by-idx-gen:*

shows $nfoldli\ (drop\ k\ l)\ c\ f\ s = nfoldli\ [k..<length\ l]\ c\ (\lambda i\ s.\ do\ \{\$
 $\quad ASSERT\ (i < length\ l);$
 $\quad let\ x = !i;$
 $\quad f\ x\ s$
 $\quad \})\ s$

<proof>

lemma *nfoldli-by-idx:*

$nfoldli\ l\ c\ f\ s = nfoldli\ [0..<length\ l]\ c\ (\lambda i\ s.\ do\ \{\$
 $\quad ASSERT\ (i < length\ l);$
 $\quad let\ x = !i;$
 $\quad f\ x\ s$
 $\quad \})\ s$

<proof>

lemma *nfoldli-map-inv:*

assumes *inj g*
shows $nfoldli\ l\ c\ f = nfoldli\ (map\ g\ l)\ c\ (\lambda x\ s.\ f\ (the\ inv\ g\ x)\ s)$
<proof>

lemma *nfoldli-shift:*

fixes *ofs :: nat*
shows $nfoldli\ l\ c\ f = nfoldli\ (map\ (\lambda i.\ i + ofs)\ l)\ c\ (\lambda x\ s.\ do\ \{\ ASSERT\ (x \geq ofs);$
 $f\ (x - ofs)\ s\ \})$
<proof>

lemma *nfoldli-foreach-shift:*

shows $nfoldli\ [a..<b]\ c\ f = nfoldli\ [a + ofs..<b + ofs]\ c\ (\lambda x\ s.\ do\ \{\ ASSERT\ (x \geq ofs);$
 $f\ (x - ofs)\ s\ \})$
<proof>

lemma *member-by-nfoldli:* $nfoldli\ l\ (\lambda f.\ \neg f)\ (\lambda y\ -. \ RETURN\ (y=x))\ False \leq$
 $SPEC\ (\lambda r.\ r \longleftrightarrow x \in set\ l)$
<proof>

definition *sum-impl :: ('a \Rightarrow 'b::comm-monoid-add nres) \Rightarrow 'a set \Rightarrow 'b nres*
where

$sum\text{-impl}\ g\ S \equiv FOREACH\ S\ (\lambda x\ a.\ do\ \{\ b \leftarrow g\ x; \ RETURN\ (a+b)\ \})\ 0$

lemma *sum-impl-correct:*

assumes [*simp*]: *finite S*

```

assumes [refine-vcg]:  $\bigwedge x. x \in S \implies gi\ x \leq SPEC\ (\lambda r. r = g\ x)$ 
shows sum-impl  $gi\ S \leq SPEC\ (\lambda r. r = sum\ g\ S)$ 
  <proof>

```

end

2.16 More Automation

```

theory Refine-Automation
imports Refine-Basic Refine-Transfer
keywords concrete-definition :: thy-decl
  and prepare-code-thms :: thy-decl
  and uses
begin

```

This theory provides a tool for extracting definitions from terms, and for generating code equations for recursion combinators.

<ML>

Command: *concrete-definition name [attribs] for params uses thm is patterns* where *attribs*, *for*, and *is*-parts are optional.

Declares a new constant *name* by matching the theorem *thm* against a pattern.

If the *for* clause is given, it lists variables in the theorem, and thus determines the order of parameters of the defined constant. Otherwise, parameters will be in order of occurrence.

If the *is* clause is given, it lists patterns. The conclusion of the theorem will be matched against each of these patterns. For the first matching pattern, the constant will be declared to be the term that matches the first non-dummy variable of the pattern. If no *is*-clause is specified, the default patterns will be tried.

Attribute: *cd-patterns pats*. Declaration attribute. Declares default patterns for the *concrete-definition* command.

```

declare [[ cd-patterns (?f,-)∈- ]]
declare [[ cd-patterns RETURN ?f ≤ - nres-of ?f ≤ - ]]
declare [[ cd-patterns (RETURN ?f,-)∈- (nres-of ?f,-)∈- ]]
declare [[ cd-patterns - = ?f - == ?f ]]

```

<ML>

Command: *prepare-code-thms (modes) thm* where the *(mode)*-part is optional.

Set up code-equations for recursions in constant defined by *thm*. The optional *modes* is a comma-separated list of extraction modes.

lemma *gen-code-thm-RECT*:

fixes x
assumes $D: f \equiv RECT\ B$
assumes $M: trimono\ B$
shows $f\ x \equiv B\ f\ x$
 $\langle proof \rangle$

lemma *gen-code-thm-REC*:

fixes x
assumes $D: f \equiv REC\ B$
assumes $M: trimono\ B$
shows $f\ x \equiv B\ f\ x$
 $\langle proof \rangle$

$\langle ML \rangle$

Method *vc-solve* (*no-pre*) *clasimp-modifiers* *rec* (*add/del*): ... *solve* (*add/del*):
 ... Named theorems *vcs-rec* and *vcs-solve*.

This method is specialized to solve verification conditions. It first *clarsimps* all goals, then it tries to apply a set of safe introduction rules (*vcs-rec*, *rec add*). Finally, it applies introduction rules (*vcs-solve*, *solve add*) and tries to discharge all emerging subgoals by *auto*. If this does not succeed, it backtracks over the application of the *solve*-rule.

$\langle ML \rangle$

end

2.17 Autoref for the Refinement Monad

theory *Autoref-Monadic*
imports *Refine-Transfer*
begin

Default setup of the autoref-tool for the monadic framework.

lemma *autoref-monadicI1*:

assumes $(b,a) \in \langle R \rangle nres\text{-}rel$
assumes $RETURN\ c \leq b$
shows $(RETURN\ c, a) \in \langle R \rangle nres\text{-}rel \quad RETURN\ c \leq \Downarrow R\ a$
 $\langle proof \rangle$

lemma *autoref-monadicI2*:

assumes $(b,a) \in \langle R \rangle nres\text{-}rel$
assumes $nres\text{-}of\ c \leq b$
shows $(nres\text{-}of\ c, a) \in \langle R \rangle nres\text{-}rel \quad nres\text{-}of\ c \leq \Downarrow R\ a$
 $\langle proof \rangle$

lemmas *autoref-monadicI* = *autoref-monadicI1* *autoref-monadicI2*

$\langle ML \rangle$

end

2.18 Refinement Framework

theory *Refine-Monadic*

imports

Refine-Chapter

Refine-Basic

Refine-Leaf

Refine-Heuristics

Refine-More-Comb

Refine-While

Refine-Foreach

Refine-Transfer

Refine-Pfun

Refine-Automation

Autoref-Monadic

begin

This theory summarizes all default theories of the refinement framework.

2.18.1 Convenience Constructs

definition *REC-annot* *pre post body x* \equiv

REC ($\lambda D x. do \{ ASSERT (pre\ x); r \leftarrow body\ D\ x; ASSERT (post\ x\ r); RETURN\ r \}$) *x*

theorem *REC-annot-rule*[*refine-vcg*]:

assumes *M*: *trimono body*

and *P*: *pre x*

and *S*: $\bigwedge f x. \llbracket \bigwedge x. pre\ x \implies f\ x \leq SPEC\ (post\ x); pre\ x \rrbracket$

$\implies body\ f\ x \leq SPEC\ (post\ x)$

and *C*: $\bigwedge r. post\ x\ r \implies \Phi\ r$

shows *REC-annot pre post body x* $\leq SPEC\ \Phi$

$\langle proof \rangle$

2.18.2 Syntax Sugar

locale *Refine-Monadic-Syntax* **begin**

notation *SPEC* (**binder** *spec* 10)

notation *ASSERT* (*assert*)

notation *RETURN* (*return*)
notation *FOREACH* (*foreach*)
notation *WHILE* (*while*)
notation *WHILE_T* (*while_T*)
notation *WHILE⁻* (*while⁻*)
notation *WHILE_T* (*while_T*)
notation *WHILEIT* (*while_T⁻*)

notation *RECT* (**binder** *rec_T* 10)
notation *REC* (**binder** *rec* 10)

notation *SELECT* (**binder** *select* 10)

end

end

Chapter 3

Examples

This chapter contains some examples of using the Refinement Framework. Examples of how to use data refinement to collection data structures can be found in the examples directory of the Isabelle Collection Framework.

3.1 Breadth First Search

```
theory Breadth-First-Search  
imports ../Refine-Monadic  
begin
```

This is a slightly modified version of Task 5 of our submission to the VSTTE 2011 verification competition (<https://sites.google.com/site/vstte2012/compet>). The task was to formalize a breadth-first-search algorithm.

With Isabelle's locale-construct, we put ourselves into a context where the *succ*-function is fixed. We assume finitely branching graphs here, as our *foreach*-construct is only defined for finite sets.

```
locale Graph =  
  fixes succ :: 'vertex  $\Rightarrow$  'vertex set  
  assumes [simp, intro!]: finite (succ v)  
begin
```

3.1.1 Distances in a Graph

We start over by defining the basic notions of paths and shortest paths.

A path is expressed by the *dist*-predicate. Intuitively, *dist* *v* *d* *v'* means that there is a path of length *d* between *v* and *v'*.

The definition of the *dist*-predicate is done inductively, i.e., as the least solution of the following constraints:

```
inductive dist :: 'vertex  $\Rightarrow$  nat  $\Rightarrow$  'vertex  $\Rightarrow$  bool where
```

dist-z: $\text{dist } v \ 0 \ v \mid$
dist-suc: $\llbracket \text{dist } v \ d \ v h; v' \in \text{succ } v h \rrbracket \Longrightarrow \text{dist } v \ (\text{Suc } d) \ v'$

Next, we define a predicate that expresses that the shortest path between v and v' has length d . This is the case if there is a path of length d , but there is no shorter path.

definition *min-dist* $v \ v' = (\text{LEAST } d. \text{dist } v \ d \ v')$

definition *conn* $v \ v' = (\exists d. \text{dist } v \ d \ v')$

Properties

In this subsection, we prove some properties of paths.

lemma

shows *connI*[*intro*]: $\text{dist } v \ d \ v' \Longrightarrow \text{conn } v \ v'$
and *connI-id*[*intro*]: $\text{conn } v \ v$
and *connI-succ*[*intro*]: $\text{conn } v \ v' \Longrightarrow v'' \in \text{succ } v' \Longrightarrow \text{conn } v \ v''$
 <proof>

lemma *min-distI2*:

$\llbracket \text{conn } v \ v'; \bigwedge d. \llbracket \text{dist } v \ d \ v'; \bigwedge d'. \text{dist } v \ d' \ v' \Longrightarrow d \leq d' \rrbracket \Longrightarrow Q \ d \rrbracket$
 $\Longrightarrow Q \ (\text{min-dist } v \ v')$
 <proof>

lemma *min-distI-eq*:

$\llbracket \text{dist } v \ d \ v'; \bigwedge d'. \text{dist } v \ d' \ v' \Longrightarrow d \leq d' \rrbracket \Longrightarrow \text{min-dist } v \ v' = d$
 <proof>

Two nodes are connected by a path of length 0, iff they are equal.

lemma *dist-z-iff*[*simp*]: $\text{dist } v \ 0 \ v' \longleftrightarrow v'=v$
 <proof>

The same holds for *min-dist*, i.e., the shortest path between two nodes has length 0, iff these nodes are equal.

lemma *min-dist-z*[*simp*]: $\text{min-dist } v \ v = 0$
 <proof>

lemma *min-dist-z-iff*[*simp*]: $\text{conn } v \ v' \Longrightarrow \text{min-dist } v \ v' = 0 \longleftrightarrow v'=v$
 <proof>

lemma *min-dist-is-dist*: $\text{conn } v \ v' \Longrightarrow \text{dist } v \ (\text{min-dist } v \ v') \ v'$
 <proof>

lemma *min-dist-minD*: $\text{dist } v \ d \ v' \Longrightarrow \text{min-dist } v \ v' \leq d$
 <proof>

We also provide introduction and destruction rules for the pattern *min-dist* $v \ v' = \text{Suc } d$.

lemma *min-dist-succ*:

$\llbracket \text{conn } v \ v'; v'' \in \text{succ } v' \rrbracket \implies \text{min-dist } v \ v'' \leq \text{Suc } (\text{min-dist } v \ v')$
 <proof>

lemma *min-dist-suc*:

assumes $c: \text{conn } v \ v' \quad \text{min-dist } v \ v' = \text{Suc } d$
shows $\exists v''. \text{conn } v \ v'' \wedge v' \in \text{succ } v'' \wedge \text{min-dist } v \ v'' = d$
 <proof>

If there is a node with a shortest path of length d , then, for any $d' < d$, there is also a node with a shortest path of length d' .

lemma *min-dist-less*:

assumes $\text{conn } \text{src } v \quad \text{min-dist } \text{src } v = d$ **and** $d' < d$
shows $\exists v'. \text{conn } \text{src } v' \wedge \text{min-dist } \text{src } v' = d'$
 <proof>

Lemma *min-dist-less* can be weakened to $d' \leq d$.

corollary *min-dist-le*:

assumes $c: \text{conn } \text{src } v$ **and** $d': d' \leq \text{min-dist } \text{src } v$
shows $\exists v'. \text{conn } \text{src } v' \wedge \text{min-dist } \text{src } v' = d'$
 <proof>

3.1.2 Invariants

In our framework, it is convenient to annotate the invariants and auxiliary assertions into the program. Thus, we have to define the invariants first.

The invariant for the outer loop is split into two parts: The first part already holds before the *if* $C = \{\}$ check, the second part only holds again at the end of the loop body.

The first part of the invariant, *bfs-invar'*, intuitively states the following: If the loop is not *broken*, then we have:

- The next-node set N is a subset of V , and the destination node is not contained into $V - (C \cup N)$,
- all nodes in the current-node set C have a shortest path of length d ,
- all nodes in the next-node set N have a shortest path of length $d+1$,
- all nodes in the visited set V have a shortest path of length at most $d+1$,
- all nodes with a path shorter than d are already in V , and
- all nodes with a shortest path of length $d+1$ are either in the next-node set N , or they are undiscovered successors of a node in the current-node set.

If the loop has been *broken*, d is the distance of the shortest path between src and dst .

definition $bfs-invar' src dst \sigma \equiv let (f, V, C, N, d) = \sigma in$
 $(\neg f \rightarrow ($
 $N \subseteq V \wedge dst \notin V - (C \cup N) \wedge$
 $(\forall v \in C. conn src v \wedge min-dist src v = d) \wedge$
 $(\forall v \in N. conn src v \wedge min-dist src v = Suc d) \wedge$
 $(\forall v \in V. conn src v \wedge min-dist src v \leq Suc d) \wedge$
 $(\forall v. conn src v \wedge min-dist src v \leq d \rightarrow v \in V) \wedge$
 $(\forall v. conn src v \wedge min-dist src v = Suc d \rightarrow v \in N \cup ((\bigcup (succ' C)) - V))$
 $)) \wedge ($
 $f \rightarrow conn src dst \wedge min-dist src dst = d$
 $)$

The second part of the invariant, *empty-assm*, just states that C can only be empty if N is also empty.

definition $empty-assm \sigma \equiv let (f, V, C, N, d) = \sigma in$
 $C = \{\} \rightarrow N = \{\}$

Finally, we define the invariant of the outer loop, *bfs-invar*, as the conjunction of both parts:

definition $bfs-invar src dst \sigma \equiv bfs-invar' src dst \sigma \wedge$
 $empty-assm \sigma$

The invariant of the inner foreach-loop states that the successors that have already been processed ($succ v - it$), have been added to V and have also been added to N' if they are not in V .

definition $FE-invar V N v it \sigma \equiv let (V', N') = \sigma in$
 $V' = V \cup (succ v - it) \wedge$
 $N' = N \cup ((succ v - it) - V)$

3.1.3 Algorithm

The following algorithm is a straightforward transcription of the algorithm given in the assignment to the monadic style featured by our framework. We briefly explain the (mainly syntactic) differences:

- The initialization of the variables occur after the loop in our formulation. This is just a syntactic difference, as our loop construct has the form $WHILE I c f \sigma_0$, where σ_0 is the initial state, and I is the loop invariant;
- We translated the textual specification *remove one vertex v from C* as accurately as possible: The statement $v \leftarrow SPEC (\lambda v. v \in C)$ non-deterministically assigns a node from C to v , that is then removed in the next statement;

- In our monad, we have no notion of loop-breaking (yet). Hence we added an additional boolean variable f that indicates that the loop shall terminate. The *RETURN*-statements used in our program are the return-operator of the monad, and must not be mixed up with the return-statement given in the original program, that is modeled by breaking the loop. The if-statement after the loop takes care to return the right value;
- We added an else-branch to the if-statement that checks whether we reached the destination node;
- We added an assertion of the first part of the invariant to the program text, moreover, we annotated invariants at the loops. We also added an assertion $w \notin N$ into the inner loop. This is merely an optimization, that will allow us to implement the insert operation more efficiently;
- Each conditional branch in the loop body ends with a *RETURN*-statement. This is required by the monadic style;
- Failure is modeled by an option-datatype. The result *Some d* means that the integer d is returned, the result *None* means that a failure is returned.

```

definition bfs :: 'vertex  $\Rightarrow$  'vertex  $\Rightarrow$ 
  (nat option nres)
where bfs src dst  $\equiv$  do {
  (f, -, -, -, d)  $\leftarrow$  WHILEI (bfs-invar src dst) ( $\lambda(f, V, C, N, d). f = \text{False} \wedge C \neq \{\}$ )
  ( $\lambda(f, V, C, N, d). do$  {
    v  $\leftarrow$  SPEC ( $\lambda v. v \in C$ ); let C = C - {v};
    if v = dst then RETURN (True, { }, { }, { }, d)
    else do {
      (V, N)  $\leftarrow$  FOREACHi (FE-invar V N v) (succ v) ( $\lambda w (V, N).$ 
        if (w  $\notin$  V) then do {
          ASSERT (w  $\notin$  N);
          RETURN (insert w V, insert w N)
        } else RETURN (V, N)
      ) (V, N);
      ASSERT (bfs-invar' src dst (f, V, C, N, d));
      if (C = { }) then do {
        let C = N;
        let N = { };
        let d = d + 1;
        RETURN (f, V, C, N, d)
      } else RETURN (f, V, C, N, d)
    }
  }
  )
  (False, {src}, {src}, { }, 0 :: nat);
  if f then RETURN (Some d) else RETURN None
}

```

3.1.4 Verification Tasks

In order to make the proof more readable, we have extracted the difficult verification conditions and proved them in separate lemmas. The other verification conditions are proved automatically by Isabelle/HOL during the proof of the main theorem.

Due to the timing constraints of the competition, the verification conditions are mostly proved in Isabelle's apply-style, that is faster to write for the experienced user, but harder to read by a human.

Exemplarily, we formulated the last proof in the proof language *Isar*, that allows one to write human-readable proofs and verify them with Isabelle/HOL.

The first part of the invariant is preserved if we take a node from C , and add its successors that are not in V to N . This is the verification condition for the assertion after the foreach-loop.

```

lemma invar-succ-step:
  assumes bfs-invar' src dst (False, V, C, N, d)
  assumes v  $\in$  C
  assumes v  $\neq$  dst
  shows bfs-invar' src dst

```

(*False*, $V \cup \text{succ } v$, $C - \{v\}$, $N \cup (\text{succ } v - V)$, d)
 ⟨*proof*⟩

The first part of the invariant is preserved if the *if* $C=\{\}$ -statement is executed. This is the verification condition for the loop-invariant. Note that preservation of the second part of the invariant is proven easily inside the main proof.

lemma *invar-empty-step*:
assumes *bfs-invar'* *src dst* (*False*, V , $\{\}$, N , d)
shows *bfs-invar'* *src dst* (*False*, V , N , $\{\}$, *Suc* d)
 ⟨*proof*⟩

The invariant holds initially.

lemma *invar-init*: *bfs-invar src dst* (*False*, $\{\text{src}\}$, $\{\text{src}\}$, $\{\}$, 0)
 ⟨*proof*⟩

The invariant is preserved if we break the loop.

lemma *invar-break*:
assumes *bfs-invar src dst* (*False*, V , C , N , d)
assumes $\text{dst} \in C$
shows *bfs-invar src dst* (*True*, $\{\}$, $\{\}$, $\{\}$, d)
 ⟨*proof*⟩

If we have *broken* the loop, the invariant implies that we, indeed, returned the shortest path.

lemma *invar-final-succeed*:
assumes *bfs-invar'* *src dst* (*True*, V , C , N , d)
shows $\text{min-dist } \text{src } \text{dst} = d$
 ⟨*proof*⟩

If the loop terminated normally, there is no path between *src* and *dst*.

The lemma is formulated as deriving a contradiction from the fact that there is a path and the loop terminated normally.

Note the proof language *Isar* that was used here. It allows one to write human-readable proofs in a theorem prover.

lemma *invar-final-fail*:
assumes $C: \text{conn } \text{src } \text{dst}$ — There is a path between *src* and *dst*.
assumes *INV*: *bfs-invar'* *src dst* (*False*, V , $\{\}$, $\{\}$, d)
shows *False*
 ⟨*proof*⟩

Finally, we prove our algorithm correct: The following theorem solves both verification tasks.

Note that a proposition of the form $S \sqsubseteq \text{SPEC } \Phi$ states partial correctness in our framework, i.e., S refines the specification Φ .

The actual specification that we prove here precisely reflects the two verification tasks: *If the algorithm fails, there is no path between src and dst, otherwise it returns the length of the shortest path.*

The proof of this theorem first applies the verification condition generator (*apply (intro refine-vcg)*), and then uses the lemmas proved beforehand to discharge the verification conditions. During the *auto*-methods, some trivial verification conditions, e.g., those concerning the invariant of the inner loop, are discharged automatically. During the proof, we gradually unfold the definition of the loop invariant.

definition *bfs-spec src dst* \equiv *SPEC* (
 $\lambda r.$ *case r of None* $\Rightarrow \neg \text{conn src dst}$
 $\mid \text{Some } d \Rightarrow \text{conn src dst} \wedge \text{min-dist src dst} = d$)
theorem *bfs-correct: bfs src dst* \leq *bfs-spec src dst*
 $\langle \text{proof} \rangle$

end

end

3.2 Machine Words

theory *WordRefine*
imports *../Refine-Monadic HOL-Library.Word*
begin

This theory provides a simple example to show refinement of natural numbers to machine words. The setup is not yet very elaborated, but shows the direction to go.

3.2.1 Setup

definition [*simp*]: *word-nat-rel* \equiv *build-rel (unat) ($\lambda.$ True)*

lemma *word-nat-RELEATES[refine-dref-RELATES]*:
RELATES word-nat-rel $\langle \text{proof} \rangle$

lemma [*simp, relator-props*]:
single-valued word-nat-rel $\langle \text{proof} \rangle$

lemma [*simp*]: *single-valuedp* ($\lambda c a. a = \text{unat } c$)
 $\langle \text{proof} \rangle$

lemma [*simp, relator-props*]: *single-valued (converse word-nat-rel)*
 $\langle \text{proof} \rangle$

lemmas [*refine-hsimp*] =
word-less-nat-alt word-le-nat-alt unat-sub iffD1[OF unat-add-lem]

3.2.2 Example

type-synonym $word32 = 32 \text{ word}$

definition $test :: nat \Rightarrow nat \Rightarrow nat \text{ set nres}$ **where** $test \ x0 \ y0 \equiv do \{$
 $let \ S = \{\};$
 $(S, -, -) \leftarrow WHILE \ (\lambda(S, x, y). \ x > 0) \ (\lambda(S, x, y). \ do \{$
 $let \ S = S \cup \{y\};$
 $let \ x = x - 1;$
 $ASSERT \ (y < x0 + y0);$
 $let \ y = y + 1;$
 $RETURN \ (S, x, y)$
 $\}) \ (S, x0, y0);$
 $RETURN \ S$
 $\}$

lemma $y0 > 0 \implies test \ x0 \ y0 \leq SPEC \ (\lambda S. \ S = \{y0 .. y0 + x0 - 1\})$
 — Chosen pre-condition to get least trouble when proving
 $\langle proof \rangle$

definition $test-impl :: word32 \Rightarrow word32 \Rightarrow word32 \text{ set nres}$ **where**
 $test-impl \ x \ y \equiv do \{$
 $let \ S = \{\};$
 $(S, -, -) \leftarrow WHILE \ (\lambda(S, x, y). \ x > 0) \ (\lambda(S, x, y). \ do \{$
 $let \ S = S \cup \{y\};$
 $let \ x = x - 1;$
 $let \ y = y + 1;$
 $RETURN \ (S, x, y)$
 $\}) \ (S, x, y);$
 $RETURN \ S$
 $\}$

lemma $test-impl-refine:$

assumes $x' + y' < 2 \wedge LENGTH(32)$

assumes $(x, x') \in word-nat-rel$

assumes $(y, y') \in word-nat-rel$

shows $test-impl \ x \ y \leq \Downarrow((word-nat-rel) \text{ set-rel}) \ (test \ x' \ y')$

$\langle proof \rangle$

end

Chapter 4

Conclusion and Future Work

We have presented a framework for program and data refinement. The notion of a program is based on a nondeterminism monad, and we provided tools for verification condition generation, finding data refinement relations, and for generating executable code by Isabelle/HOL's code generator [7, 8]. We illustrated the usability of our framework by various examples, among others a breadth-first search algorithm, which was our solution to task 5 of the VSTTE 2012 verification competition.

There is lots of possible future work. We sketch some major directions here:

- Some of our refinement rules (e.g. for while-loops) are only applicable for single-valued relations. This seems to be related to the monadic structure of our programs, which focuses on single values. A direction of future research is to understand this connection better, and to develop usable rules for non single-valued abstraction relations.
- Currently, transfer for partial correct programs is done to a complete-lattice domain. However, as assertions need not to be included in the transferred program, we could also transfer to a ccpo-domain, as, e.g., the option monad that is integrated into Isabelle/HOL by default. This is, however, only a technical problem, as ccpo and lattice type-classes are not properly linked¹. Moreover, with the partial function package [10], Isabelle/HOL has a powerful tool to express arbitrary recursion schemes over monadic programs. Currently, we have done the basic setup for the partial function package, i.e., we can define recursions over our monad. However, induction-rule generation does not yet work, and there is potential for more tool-support regarding refinement and transfer to deterministic programs.
- Finally, our framework only supports functional programs. However, as shown in Imperative/HOL [4], monadic programs are well-suited to

¹This has also been fixed in the development version of Isabelle/HOL

express a heap. Hence, a direction of future research is to add a heap to our nondeterminism monad. Argumentation about the heap could be done with a separation logic [19] formalism, like the one that we already developed for Imperative/HOL [15].

Bibliography

- [1] R.-J. Back. *On the correctness of refinement steps in program development*. PhD thesis, Department of Computer Science, University of Helsinki, 1978. Report A-1978-4.
- [2] R.-J. Back and J. von Wright. *Refinement Calculus — A Systematic Introduction*. Springer, 1998.
- [3] R. J. R. Back and J. von Wright. Refinement concepts formalized in higher order logic. *Formal Aspects of Computing*, 2, 1990.
- [4] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with isabelle/hol. In *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2008.
- [5] D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In O. A. Mohamed, C. M. noz, and S. Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*, volume 5170 of *Lecture Notes in Computer Science*, pages 167–182. Springer-Verlag, 2008.
- [6] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
- [7] F. Haftmann. *Code Generation from Specifications in Higher Order Logic*. PhD thesis, Technische Universität München, 2009.
- [8] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *Functional and Logic Programming (FLOPS 2010)*, LNCS. Springer, 2010.
- [9] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972. 10.1007/BF00289507.
- [10] A. Krauss. Recursive definitions of monadic functions. In A. Bove, E. Komendantskaya, and M. Niqui, editors, *Workshop on Partiality*

- and Recursion in Interactive Theorem Proving (PAR 2010)*, volume 43, pages 1–13, 2010.
- [11] P. Lammich. Collections framework. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. <http://isa-afp.org/entries/collections.shtml>, Dec. 2009. Formal proof development.
 - [12] P. Lammich. Tree automata. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. <http://isa-afp.org/entries/Tree-Automata.shtml>, Dec. 2009. Formal proof development.
 - [13] P. Lammich and A. Lochbihler. The Isabelle collections framework. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010.
 - [14] T. Langbacka, R. Ruksenas, and J. von Wright. Tkwinhol: A tool for doing window inference in hol. In *In Proc. 1995 International Workshop on Higher Order Logic Theorem Proving and its Applications, Lecture*, pages 245–260. Springer-Verlag, 1995.
 - [15] R. Meis. Integration von separation logic in das imperative hol-framework, 2011.
 - [16] M. Müller-Olm. *Modular Compiler Verification — A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *LNCS*. Springer, 1997.
 - [17] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
 - [18] V. Preoteasa. *Program Variables — The Core of Mechanical Reasoning about Imperative Programs*. PhD thesis, Turku Centre for Computer Science, 2006.
 - [19] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002.
 - [20] R. Ruksenas and J. von Wright. A tool for data refinement. Technical Report TUCS Technical Report No 119, Turku Centre for Computer Science, 1997.
 - [21] M. Schwenke and B. Mahony. The essence of expression refinement. In *Proc. of International Refinement Workshop and Formal Methods*, pages 324–333, 1998.
 - [22] M. Staples. *A Mechanised Theory of Refinement*. PhD thesis, University of Cambridge, 1999. 2nd edition.