

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	9
2.1.3	Monotonicity and Orderings	11
2.1.4	Maps	22
2.2	Transfer between Domains	22
2.3	General Domain Theory	28
2.3.1	General Order Theory Tools	28
2.3.2	Flat Ordering	29
2.4	Generic Recursion Combinator for Complete Lattice Structured Domains	35
2.4.1	Transfer	42
2.5	Assert and Assume	43
2.6	Basic Concepts	46
2.6.1	Nondeterministic Result Lattice and Monad	46
2.6.2	VCG Setup	56
2.6.3	Data Refinement	58
2.6.4	Derived Program Constructs	61
2.6.5	Proof Rules	62
2.6.6	Relators	74
2.6.7	Autoref Setup	74
2.6.8	Convenience Rules	77
2.7	Less-Equal or Fail	85
2.8	Data Refinement Heuristics	90
2.8.1	Type Based Heuristics	90
2.8.2	Patterns	92
2.8.3	Refinement Relations	92
2.9	More Combinators	94
2.10	Generic While-Combinator	95

2.11	While-Loops	102
2.11.1	Data Refinement Rules	103
2.11.2	Autoref Setup	109
2.11.3	Invariants	111
2.11.4	Convenience	130
2.12	Deterministic Monad	131
2.12.1	Deterministic Result Lattice	131
2.13	Partial Function Package Setup	139
2.13.1	Nondeterministic Result Monad	140
2.13.2	Deterministic Result Monad	140
2.14	Transfer Setup	141
2.14.1	Transfer to Deterministic Result Lattice	142
2.14.2	Transfer to Plain Function	144
2.14.3	Total correctness in deterministic monad	144
2.14.4	Relator-Based Transfer	145
2.14.5	Post-Simplification Setup	146
2.15	Foreach Loops	146
2.15.1	Auxilliary Lemmas	146
2.15.2	Definition	146
2.15.3	Proof Rules	147
2.15.4	FOREACH with empty sets	165
2.15.5	Monotonicity	165
2.15.6	Nres-Fold with Interruption (nfoldli)	166
2.15.7	LIST FOREACH combinator	174
2.15.8	FOREACH with duplicates	179
2.15.9	Miscellaneous Utility Lemmas	181
2.16	More Automation	184
2.17	Autoref for the Refinement Monad	196
2.18	Refinement Framework	197
2.18.1	Convenience Constructs	197
2.18.2	Syntax Sugar	198
3	Examples	201
3.1	Breadth First Search	201
3.1.1	Distances in a Graph	201
3.1.2	Invariants	203
3.1.3	Algorithm	205
3.1.4	Verification Tasks	206
3.2	Machine Words	209
3.2.1	Setup	210
3.2.2	Example	210
4	Conclusion and Future Work	213

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-file  $\langle$ refine-mono-prover.ML $\rangle$ 

  setup Refine-Mono-Prover.setup
  declaration Refine-Mono-Prover.decl-setup

  method-setup refine-mono =
     $\langle$ Scan.succeed (fn ctxt  $\Rightarrow$  SIMPLE-METHOD' (
      Refine-Mono-Prover.untriggered-mono-tac ctxt
    )) $\rangle$ 
    Refinement framework: Monotonicity prover

  locale mono-setup-loc =
    — Locale to set up monotonicity prover for given ordering operator
    fixes le :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
    assumes refl: le x x
  begin
    lemma monoI:  $(\bigwedge f g x. (\bigwedge x. le (f x) (g x)) \Longrightarrow le (B f x) (B g x))$ 
       $\Longrightarrow$  monotone (fun-ord le) (fun-ord le) B
    unfolding monotone-def fun-ord-def by blast

    lemma mono-if:  $\llbracket le t t'; le e e' \rrbracket \Longrightarrow le (If b t e) (If b t' e')$  by auto
    lemma mono-let:  $(\bigwedge x. le (f x) (f' x)) \Longrightarrow le (Let x f) (Let x f')$  by auto

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

    declaration  $\langle$ Refine-Mono-Prover.declare-mono-triggers  $\@$ {thms monoI} $\rangle$ 

  end

  interpretation order-mono-setup: mono-setup-loc ( $\leq$ ) :: 'a::preorder  $\Rightarrow$  -
    by standard auto
```

declaration $\langle \text{Refine-Mono-Prover.declare-mono-triggers} @\{\text{thms Orderings.monoI}\} \rangle$

lemmas [*refine-mono*] =
 $\text{lfp-mono}[OF \text{le-funI}, THEN \text{le-funD}]$
 $\text{gfp-mono}[OF \text{le-funI}, THEN \text{le-funD}]$

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*] = $\text{monoI monotoneI}[of (\leq) (\leq)]$

lemmas [*refine-mono*] = $\text{TrueI le-funI order-refl}$

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

$\llbracket \bigwedge a b. p=(a,b) \implies f a b \leq f' a b \rrbracket \implies \text{case-prod } f p \leq \text{case-prod } f' p$
by (*auto split: prod.split*)

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

assumes $fn \leq fn'$
assumes $\bigwedge v. x=\text{Some } v \implies fs v \leq fs' v$
shows $\text{case-option } fn fs x \leq \text{case-option } fn' fs' x$
using *assms by (auto split: option.split)*

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 $\text{case-list } fn fc l \leq \text{case-list } fn' fc' l$
using *assms by (auto split: list.split)*

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

assumes $b \implies m1 \leq m1'$
assumes $\neg b \implies m2 \leq m2'$
shows $(\text{if } b \text{ then } m1 \text{ else } m2) \leq (\text{if } b \text{ then } m1' \text{ else } m2')$
using *assms by auto*

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

$f x \leq f' x' \implies \text{Let } x f \leq \text{Let } x' f' \text{ by auto}$

2.1.1 Uncategorized Lemmas

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

$\implies \forall i. P i$
by (*metis linorder-neq-iff*)

2.1.2 Well-Foundedness

lemma *wf-no-infinite-down-chainI*:
assumes $\bigwedge f. [\bigwedge i. (f (Suc i), f i) \in r] \implies False$
shows *wf r*
by (*metis assms wf-iff-no-infinite-down-chain*)

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*: $\bigwedge s s' t. [(s, s') \in R; (s, t) \in S; (x0', s') \in S'^*]$
 $\implies \exists t'. (s', t') \in S' \wedge (t, t') \in R$
assumes *CLOSED*: $Domain S \subseteq S^* \cdot \{x0\}$
shows $wf (S^{-1})$
proof (*rule wf-no-infinite-down-chainI, simp*)

Informal proof: Assume there is an infinite chain in S . Due to the closedness property of S , it can be extended to start at $x0$. Now, we inductively construct an infinite chain in S' , such that each element of the new chain is in relation with the corresponding element of the original chain: The first element is $x0'$. For any element $i+1$, the simulation property yields the next element. This chain contradicts well-foundedness of S' .

fix *f*
assume *CHAIN*: $\bigwedge i. (f i, f (Suc i)) \in S$

Extend to start with $x0$

obtain *f'* **where** *CHAIN'*: $\bigwedge i. (f' i, f' (Suc i)) \in S$ **and** [*simp*]: $f' 0 = x0$
proof –

{
fix *x*
assume *S*: $x = f 0$
from *CHAIN* **have** $f 0 \in Domain S$ **by** *auto*
with *CLOSED* **have** $(x0, x) \in S^*$ **by** (*auto simp: S*)
then obtain *g k* **where** *G0*: $g 0 = x0$ **and** *X*: $x = g k$
and *CH*: $(\forall i < k. (g i, g (Suc i)) \in S)$
proof *induct*
case *base* **thus** *?case* **by** *force*
next
case (*step y z*) **show** *?case* **proof** (*rule step.hyps(3)*)
fix *g k*
assume $g 0 = x0$ **and** $y = g k$
and $\forall i < k. (g i, g (Suc i)) \in S$
thus *?case* **using** $\langle y, z \rangle \in S$
by (*rule-tac step.premis[where g=g(Suc k := z) and k=Suc k]*)
auto

```

    qed
  qed
  define f' where f' i = (if i < k then g i else f (i - k)) for i
  have  $\exists f'. f' 0 = x0 \wedge (\forall i. (f' i, f' (Suc i)) \in S)$ 
    apply (rule-tac x=f' in exI)
    apply (unfold f'-def)
    apply (rule conjI)
    using S X G0 apply (auto) []
    apply (rule-tac k=k in all-nat-split-at)
    apply (auto)
    apply (simp add: CH)
    apply (subgoal-tac k = Suc i)
    apply (simp add: S[symmetric] CH X)
    apply simp
    apply (simp add: CHAIN)
    apply (subgoal-tac Suc i - k = Suc (i - k))
    using CHAIN apply simp
    apply simp
  done
}
then obtain f' where  $\forall i. (f' i, f' (Suc i)) \in S$  and  $f' 0 = x0$  by blast
thus ?thesis by (blast intro!: that)
qed

```

Construct chain in S'

```

define g' where g' = rec-nat x0' ( $\lambda i x. SOME x'$ 
  ( $x, x' \in S' \wedge (f' (Suc i), x') \in R \wedge (x0', x') \in S'^*$  )
{
  fix i
  note [simp] = g'-def
  have  $(g' i, g' (Suc i)) \in S' \wedge (f' (Suc i), g' (Suc i)) \in R$ 
     $\wedge (x0', g' (Suc i)) \in S'^*$ 
  proof (induct i)
    case 0
    from SIM[OF STARTR] CHAIN'[of 0] obtain t' where
       $(x0', t') \in S'$  and  $(f' (Suc 0), t') \in R$  by auto
    moreover hence  $(x0', t') \in S'^*$  by auto
    ultimately show ?case
      by (auto intro: someI2 simp: STARTR)
  next
  case (Suc i)
  with SIM[OF - CHAIN'[of Suc i]]
  obtain t' where LS:  $(g' (Suc i), t') \in S'$  and  $(f' (Suc (Suc i)), t') \in R$ 
    by auto
  moreover from LS Suc have  $(x0', t') \in S'^*$  by auto
  ultimately show ?case
    apply simp
    apply (rule-tac a=t' in someI2)
    apply auto

```

```

  done
  qed
} hence S'CHAIN:  $\forall i. (g' i, g'(Suc i)) \in S'$  by simp

```

This contradicts well-foundedness

```

with WF show False
  by (erule-tac wf-no-infinite-down-chainE[where f=g']) simp
qed

```

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)
  unfolding finite-psupset-def by (blast intro: wf-bounded-supset)

```

```

definition less-than-bool  $\equiv \{ (a, b). a < (b::bool) \}$ 
lemma wf-less-than-bool[simp, intro!]: wf (less-than-bool)
  unfolding less-than-bool-def
  by (auto simp: wf-def)
lemma less-than-bool-iff[simp]:
   $(x, y) \in \text{less-than-bool} \iff x = \text{False} \wedge y = \text{True}$ 
  by (auto simp: less-than-bool-def)

```

```

definition greater-bounded N  $\equiv \text{inv-image less-than } (\lambda x. N - x)$ 
lemma wf-greater-bounded[simp, intro!]: wf (greater-bounded N) by (auto simp:
greater-bounded-def)

```

```

lemma greater-bounded-Suc-iff[simp]:  $(\text{Suc } x, x) \in \text{greater-bounded } N \iff \text{Suc } x \leq N$ 
by (auto simp: greater-bounded-def)

```

2.1.3 Monotonicity and Orderings

```

lemma mono-const[simp, intro!]: mono  $(\lambda \cdot. c)$  by (auto intro: monoI)
lemma mono-if:  $\llbracket \text{mono } S1; \text{mono } S2 \rrbracket \implies$ 
  mono  $(\lambda F s. \text{if } b \text{ } s \text{ then } S1 \text{ } F \text{ } s \text{ else } S2 \text{ } F \text{ } s)$ 
  apply rule
  apply (rule le-funI)
  apply (auto dest: monoD[THEN le-funD])
  done

```

```

lemma mono-infI: mono f  $\implies$  mono g  $\implies$  mono (inf f g)
  unfolding inf-fun-def
  apply (rule monoI)
  apply (metis inf-mono monoD)
  done

```

```

lemma mono-infI':
  mono f  $\implies$  mono g  $\implies$  mono  $(\lambda x. \text{inf } (f \ x) \ (g \ x) :: 'b::\text{lattice})$ 
  by (rule mono-infI[unfolded inf-fun-def])

```

```

lemma mono-infArg:
  fixes  $f :: 'a::lattice \Rightarrow 'b::order$ 
  shows  $mono\ f \Longrightarrow mono\ (\lambda x. f\ (inf\ x\ X))$ 
  apply (rule monoI)
  apply (erule monoD)
  apply (metis inf-mono order-refl)
  done

lemma mono-Sup:
  fixes  $f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice$ 
  shows  $mono\ f \Longrightarrow Sup\ (f'S) \leq f\ (Sup\ S)$ 
  apply (rule Sup-least)
  apply (erule imageE)
  apply simp
  apply (erule monoD)
  apply (erule Sup-upper)
  done

lemma mono-SupI:
  fixes  $f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice$ 
  assumes  $mono\ f$ 
  assumes  $S' \subseteq f'S$ 
  shows  $Sup\ S' \leq f\ (Sup\ S)$ 
  by (metis Sup-subset-mono assms mono-Sup order-trans)

lemma mono-Inf:
  fixes  $f :: 'a::complete-lattice \Rightarrow 'b::complete-lattice$ 
  shows  $mono\ f \Longrightarrow f\ (Inf\ S) \leq Inf\ (f'S)$ 
  apply (rule Inf-greatest)
  apply (erule imageE)
  apply simp
  apply (erule monoD)
  apply (erule Inf-lower)
  done

lemma mono-funpow:  $mono\ (f::'a::order \Rightarrow 'a) \Longrightarrow mono\ (f^{\sim}i)$ 
  apply (induct i)
  apply (auto intro!: monoI)
  apply (auto dest: monoD)
  done

lemma mono-id[simp, intro!]:
   $mono\ id$ 
   $mono\ (\lambda x. x)$ 
  by (auto intro: monoI)

declare SUP-insert[simp]

```

```

lemma (in semilattice-inf) le-infD1:
  a ≤ inf b c ⇒ a ≤ b by (rule le-infE)
lemma (in semilattice-inf) le-infD2:
  a ≤ inf b c ⇒ a ≤ c by (rule le-infE)
lemma (in semilattice-inf) inf-leI:
  [ [ x ≤ a; x ≤ b ] ⇒ x ≤ c ] ⇒ inf a b ≤ c
  by (metis inf-le1 inf-le2)

lemma top-Sup: (top::'a::complete-lattice) ∈ A ⇒ Sup A = top
  by (metis Sup-upper top-le)
lemma bot-Inf: (bot::'a::complete-lattice) ∈ A ⇒ Inf A = bot
  by (metis Inf-lower le-bot)

lemma mono-compD: mono f ⇒ x ≤ y ⇒ f o x ≤ f o y
  apply (rule le-funI)
  apply (auto dest: monoD le-funD)
  done

```

Galois Connections

```

locale galois-connection =
  fixes a::'a::complete-lattice ⇒ 'b::complete-lattice and γ
  assumes galois: c ≤ γ(a) ↔ α(c) ≤ a
begin
  lemma αγ-defl: α(γ(x)) ≤ x
  proof -
    have γ x ≤ γ x by simp
    with galois show α(γ(x)) ≤ x by blast
  qed
  lemma γα-infl: x ≤ γ(α(x))
  proof -
    have α x ≤ α x by simp
    with galois show x ≤ γ(α(x)) by blast
  qed

  lemma α-mono: mono α
  proof
    fix x::'a and y::'a
    assume x ≤ y
    also note γα-infl[of y]
    finally show α x ≤ α y by (simp add: galois)
  qed

  lemma γ-mono: mono γ
  by rule (metis αγ-defl galois inf-absorb1 le-infE)

  lemma dist-γ[simp]:
    γ (inf a b) = inf (γ a) (γ b)
  apply (rule order-antisym)

```

```

apply (rule mono-inf[OF  $\gamma$ -mono])
apply (simp add: galois)
apply (simp add: galois[symmetric])
done

```

```

lemma dist- $\alpha$ [simp]:
   $\alpha$  (sup a b) = sup ( $\alpha$  a) ( $\alpha$  b)
by (metis (no-types)  $\alpha$ -mono galois mono-sup order-antisym
      sup-ge1 sup-ge2 sup-least)

```

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 lfp f = a
  apply (rule antisym)
  apply (rule lfp-lowerbound)
  apply (rule FIXP)
  by (metis LEAST MONO lfp-unfold)

```

```

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 gfp f = a
  apply (rule antisym)
  apply (metis GREATEST MONO gfp-unfold)
  apply (rule gfp-upperbound)
  apply (rule FIXP)
  done

```

```

lemma lfp-le-gfp': mono f  $\implies$  lfp f x  $\leq$  gfp f x
  by (rule le-funD[OF lfp-le-gfp])

```

```

lemma lfp-induct':
  assumes M: mono f
  assumes IS:  $\bigwedge m. \llbracket m \leq \text{lfp } f; m \leq P \rrbracket \implies f m \leq P$ 
  shows lfp f  $\leq$  P
  apply (rule lfp-induct[OF M])
  apply (rule IS)
  by simp-all

```

```

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  $lfp\ (\lambda s. sup\ m0\ (f\ s)) \leq P$ 
apply (rule lfp-induct')
apply (meson  $MONO'$  monoI order-mono-setup.refl sup-mono)
apply (rule sup-least)
apply (rule  $I0$ )
apply (rule  $IS$ , assumption+)
apply (subst lfp-unfold)
apply (meson  $MONO'$  monoI order-mono-setup.refl sup-mono)
apply (rule le-supI2)
apply (rule  $monoD[OF\ M]$ )
.

```

Connecting Complete Lattices and Chain-Complete Partial Orders

```

lemma (in complete-lattice) is-ccpo: class.ccpo Sup ( $\leq$ ) ( $<$ )
  apply unfold-locales
  apply (erule Sup-upper)
  apply (erule Sup-least)
  done

```

```

lemma (in complete-lattice) is-dual-ccpo: class.ccpo Inf ( $\geq$ ) ( $>$ )
  apply unfold-locales
  apply (rule less-le-not-le)
  apply (rule order-refl)
  apply (erule (1) order-trans)
  apply (erule (1) order.antisym)
  apply (erule Inf-lower)
  apply (erule Inf-greatest)
  done

```

```

lemma ccpo-mono-simp: monotone ( $\leq$ ) ( $\leq$ )  $f \longleftrightarrow$  mono  $f$ 
  unfolding monotone-def mono-def by simp

```

```

lemma ccpo-monoI: mono  $f \implies$  monotone ( $\leq$ ) ( $\leq$ )  $f$ 
  by (simp add: ccpo-mono-simp)

```

```

lemma ccpo-monoD: monotone ( $\leq$ ) ( $\leq$ )  $f \implies$  mono  $f$ 
  by (simp add: ccpo-mono-simp)

```

```

lemma dual-ccpo-mono-simp: monotone ( $\geq$ ) ( $\geq$ )  $f \longleftrightarrow$  mono  $f$ 
  unfolding monotone-def mono-def by auto

```

lemma *dual-ccpo-monoI*: $\text{mono } f \implies \text{monotone } (\geq) (\geq) f$
by (*simp add: dual-ccpo-mono-simp*)

lemma *dual-ccpo-monoD*: $\text{monotone } (\geq) (\geq) f \implies \text{mono } f$
by (*simp add: dual-ccpo-mono-simp*)

lemma *ccpo-lfp-simp*: $\bigwedge f. \text{mono } f \implies \text{ccpo.fixp } \text{Sup } (\leq) f = \text{lfp } f$
apply (*rule antisym*)
defer
apply (*rule lfp-lowerbound*)
apply (*drule ccpo.fixp-unfold[OF is-ccpo ccpo-monoI, symmetric]*)
apply *simp*

apply (*rule ccpo.fixp-lowerbound[OF is-ccpo ccpo-monoI], assumption*)
apply (*simp add: lfp-unfold[symmetric]*)
done

lemma *ccpo-gfp-simp*: $\bigwedge f. \text{mono } f \implies \text{ccpo.fixp } \text{Inf } (\geq) f = \text{gfp } f$
apply (*rule antisym*)
apply (*rule gfp-upperbound*)
apply (*drule ccpo.fixp-unfold[OF is-dual-ccpo dual-ccpo-monoI, symmetric]*)
apply *simp*

apply (*rule ccpo.fixp-lowerbound[OF is-dual-ccpo dual-ccpo-monoI], assumption*)
apply (*simp add: gfp-unfold[symmetric]*)
done

abbreviation *chain-admissible* $P \equiv \text{ccpo.admissible } \text{Sup } (\leq) P$
abbreviation *is-chain* $\equiv \text{Complete-Partial-Order.chain } (\leq)$
lemmas *chain-admissibleI*[*intro?*] = *ccpo.admissibleI*[**where** *lub=Sup* **and** *ord=(\leq)*]

abbreviation *dual-chain-admissible* $P \equiv \text{ccpo.admissible } \text{Inf } (\lambda x y. y \leq x) P$
abbreviation *is-dual-chain* $\equiv \text{Complete-Partial-Order.chain } (\lambda x y. y \leq x)$
lemmas *dual-chain-admissibleI*[*intro?*] =
ccpo.admissibleI[**where** *lub=Inf* **and** *ord=($\lambda x y. y \leq x$)*]

lemma *dual-chain-iff*[*simp*]: $\text{is-dual-chain } C = \text{is-chain } C$
unfolding *chain-def*
apply *auto*
done

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

lemma *is-chain-empty*[*simp, intro!*]: $\text{is-chain } \{\}$
by (*rule chainI*) *auto*

lemma *is-dual-chain-empty*[*simp, intro!*]: $\text{is-dual-chain } \{\}$
by (*rule dual-chainI*) *auto*

lemma *point-chainI*: $is-chain\ M \implies is-chain\ ((\lambda f. f\ x)\ 'M)$
by (*auto intro: chainI le-funI dest: chainD le-funD*)

We transfer the admissible induction lemmas to complete lattices.

lemma *lfp-cadm-induct*:
 $\llbracket chain-admissible\ P; P\ (Sup\ \{\}) ; mono\ f; \bigwedge x. P\ x \implies P\ (f\ x) \rrbracket \implies P\ (lfp\ f)$
by (*simp only: ccpo-mono-simp[symmetric] ccpo-lfp-simp[symmetric]*)
(rule ccpo.fixp-induct[OF is-ccpo])

lemma *gfp-cadm-induct*:
 $\llbracket dual-chain-admissible\ P; P\ (Inf\ \{\}) ; mono\ f; \bigwedge x. P\ x \implies P\ (f\ x) \rrbracket \implies P\ (gfp\ f)$
by (*simp only: dual-ccpo-mono-simp[symmetric] ccpo-gfp-simp[symmetric]*)
(rule ccpo.fixp-induct[OF is-dual-ccpo])

Continuity and Kleene Fixed Point Theorem

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

definition *strict f* $\equiv f\ bot = bot$

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

lemma *contI[intro?]*: $\llbracket \bigwedge C. C \neq \{\} \implies f\ (Sup\ C) = Sup\ (f\ 'C) \rrbracket \implies cont\ f$
unfolding *cont-def* **by** *auto*

lemma *contD*: $cont\ f \implies C \neq \{\} \implies f\ (Sup\ C) = Sup\ (f\ 'C)$
unfolding *cont-def* **by** *auto*

lemma *contD'*: $cont\ f \implies C \neq \{\} \implies f\ (Sup\ C) = Sup\ (f\ 'C)$
by (*fact contD*)

lemma *strictD[dest]*: $strict\ f \implies f\ bot = bot$
unfolding *strict-def* **by** *auto*

— We only add this lemma to the simpset for functions on the same type. Otherwise, the simplifier tries it much too often.

lemma *strictD-simp[simp]*: $strict\ f \implies f\ (bot::'a::bot) = (bot::'a)$
unfolding *strict-def* **by** *auto*

lemma *strictI[intro?]*: $f\ bot = bot \implies strict\ f$
unfolding *strict-def* **by** *auto*

lemma *inf-distribD[simp]*:
 $inf-distrib\ f \implies strict\ f$
 $inf-distrib\ f \implies cont\ f$
unfolding *inf-distrib-def* **by** *auto*

lemma *inf-distribI[intro?]*: $\llbracket strict\ f; cont\ f \rrbracket \implies inf-distrib\ f$
unfolding *inf-distrib-def* **by** *auto*

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

```

apply (cases C={})
apply (auto dest: inf-distribD contD')
done

```

```

lemma inf-distribI':
  fixes f :: 'a::complete-lattice  $\Rightarrow$  'b::complete-lattice
  assumes B:  $\bigwedge C. f (\text{Sup } C) = \text{Sup } (f' C)$ 
  shows inf-distrib f
  apply (rule)
  apply (rule)
  apply (rule B[of {}, simplified])
  apply (rule)
  apply (rule B)
  done

```

```

lemma cont-is-mono[simp]:
  fixes f :: 'a::complete-lattice  $\Rightarrow$  'b::complete-lattice
  shows cont f  $\Longrightarrow$  mono f
  apply (rule monoI)
  apply (drule-tac C={x,y} in contD)
  apply (auto simp: le-iff-sup)
  done

```

```

lemma inf-distrib-is-mono[simp]:
  fixes f :: 'a::complete-lattice  $\Rightarrow$  'b::complete-lattice
  shows inf-distrib f  $\Longrightarrow$  mono f
  by simp

```

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

```

theorem gen-kleene-lfp:
  fixes f :: 'a::complete-lattice  $\Rightarrow$  'a
  assumes CONT: cont f
  shows lfp ( $\lambda x. \text{sup } m (f x)$ ) = ( $\text{SUP } i. (f \sim i) m$ )
proof (rule antisym)
  let ?f = ( $\lambda x. \text{sup } m (f x)$ )
  let ?K = { ( $f \sim i$ ) m | i . True }
  note MONO = cont-is-mono[OF CONT]
  note MONO'[refine-mono] = monoD[OF MONO]
  {
    fix i
    have ( $f \sim i$ ) m  $\leq$  lfp ?f
    apply (induct i)
    apply simp
    apply (subst lfp-unfold)
    apply (meson MONO' monoI order-mono-setup.refl sup-mono)
    apply simp

    apply (subst lfp-unfold)
    apply (meson MONO' monoI order-mono-setup.refl sup-mono)
  }

```

```

    apply simp
    by (metis MONO' le-supI2)
  } thus (SUP i. (f~i) m) ≤ lfp ?f by (blast intro: SUP-least)
next
let ?f = (λx. sup m (f x))
show lfp ?f ≤ (SUP i. (f~i) m)
  apply (rule lfp-lowerbound)
  apply (rule sup-least)
  apply (rule order-trans[OF - SUP-upper[where i=0]], simp-all) []
  apply (simp add: contD [OF CONT])
  apply (rule Sup-subset-mono)
  apply (auto)
  apply (rule-tac x=Suc i in range-eqI)
  apply simp
done

```

qed

theorem kleene-lfp:

```

fixes f:: 'a::complete-lattice ⇒ 'a
assumes CONT: cont f
shows lfp f = (SUP i. (f~i) bot)
using gen-kleene-lfp[OF CONT,where m=bot] by simp

```

theorem

```

fixes f:: 'a::complete-lattice ⇒ 'a
assumes CONT: cont f
shows lfp f = (SUP i. (f~i) bot)

```

proof (rule antisym)

```

let ?K={ (f~i) bot | i . True}
note MONO=cont-is-mono[OF CONT]

```

```
{
```

```
  fix i
```

```
  have (f~i) bot ≤ lfp f
```

```
    apply (induct i)
```

```
    apply simp
```

```
    apply simp
```

```
    by (metis MONO lfp-unfold monoD)
```

```
} thus (SUP i. (f~i) bot) ≤ lfp f by (blast intro: SUP-least)
```

next

```
show lfp f ≤ (SUP i. (f~i) bot)
```

```
  apply (rule lfp-lowerbound)
```

```
  apply (simp add: contD [OF CONT])
```

```
  apply (rule Sup-subset-mono)
```

```
  apply auto
```

```
  apply (rule-tac x=Suc i in range-eqI)
```

```
  apply auto
```

```
done
```

qed

```

lemma SUP-funpow-contracting:
  fixes f :: 'a ⇒ ('a::complete-lattice)
  assumes C: cont f
  shows f (SUP i. (f~i) m) ≤ (SUP i. (f~i) m)
proof -
  have 1: ∧i x. f ((f~i) x) = (f~(Suc i)) x
    by simp

  have f (SUP i. (f~i) m) = (SUP i. f ((f~i) m))
    by (subst contD[OF C]) (simp-all add: image-comp)
  also have ... ≤ (SUP i. (f~i) m)
    apply (rule SUP-least)
    apply (simp, subst 1)
    apply (rule SUP-upper)
  ..
  finally show ?thesis .
qed

lemma gen-kleene-chain-conv:
  fixes f :: 'a::complete-lattice ⇒ 'a
  assumes C: cont f
  shows (SUP i. (f~i) m) = (SUP i. ((λx. sup m (f x))~i) bot)
proof -
  let ?f' = λx. sup m (f x)
  show ?thesis
  proof (intro antisym SUP-least)
    from C have C': cont ?f'
      unfolding cont-def
      by (simp add: SUP-sup-distrib[symmetric])

    fix i
    show (f~i) m ≤ (SUP i. (?f'~i) bot)
  proof (induction i)
    case 0 show ?case
      by (rule order-trans[OF - SUP-upper[where i=1]]) auto
    next
      case (Suc i)
      from cont-is-mono[OF C, THEN monoD, OF Suc]
      have (f~(Suc i)) m ≤ f (SUP i. ((λx. sup m (f x))~i) bot)
        by simp
      also have ... ≤ sup m ... by simp
      also note SUP-funpow-contracting[OF C]
      finally show ?case .
    qed
  next
    fix i
    show (?f'~i) bot ≤ (SUP i. (f~i) m)
  proof (induction i)

```

```

    case 0 thus ?case by simp
  next
    case (Suc i)
    from monoD[OF cont-is-mono[OF C] Suc]
    have (?f' ~ Suc i) bot ≤ sup m (f (SUP i. (f ~ i) m))
      by (simp add: le-supI2)
    also have ... ≤ (SUP i. (f ~ i) m)
      apply (rule sup-least)
      apply (rule order-trans[OF - SUP-upper[where i=0]], simp-all) []
      apply (rule SUP-funpow-contracting[OF C])
    done
  finally show ?case .
qed
qed
qed

```

```

theorem
  assumes C: cont f
  shows lfp (λx. sup m (f x)) = (SUP i. (f ~ i) m)
  apply (subst gen-kleene-chain-conv[OF C])
  apply (rule kleene-lfp)
  using C
  unfolding cont-def
  apply (simp add: SUP-sup-distrib[symmetric])
done

```

```

lemma (in galois-connection) dual-inf-dist-γ: γ (Inf C) = Inf (γ' C)
  apply (rule antisym)
  apply (rule Inf-greatest)
  apply clarify
  apply (rule monoD[OF γ-mono])
  apply (rule Inf-lower)
  apply simp

```

```

  apply (subst galois)
  apply (rule Inf-greatest)
  apply (subst galois[symmetric])
  apply (rule Inf-lower)
  apply simp
done

```

```

lemma (in galois-connection) inf-dist-α: inf-distrib α
  apply (rule inf-distribI')
  apply (rule antisym)

  apply (subst galois[symmetric])
  apply (rule Sup-least)

```

```

apply (subst galois)
apply (rule Sup-upper)
apply simp

apply (rule Sup-least)
apply clarify
apply (rule monoD[OF  $\alpha$ -mono])
apply (rule Sup-upper)
apply simp
done

```

2.1.4 Maps

Key-Value Set

```

lemma map-to-set-simps[simp]:
  map-to-set Map.empty = {}
  map-to-set [a→b] = {(a,b)}
  map-to-set (m | K) = map-to-set m ∩ K × UNIV
  map-to-set (m(x:=None)) = map-to-set m - {x} × UNIV
  map-to-set (m(x→v)) = map-to-set m - {x} × UNIV ∪ {(x,v)}
  map-to-set m ∩ dom m × UNIV = map-to-set m
  m k = Some v ⇒ (k,v) ∈ map-to-set m
  single-valued (map-to-set m)
apply (simp-all)
by (auto simp: map-to-set-def restrict-map-def split: if-split-asm
      intro: single-valuedI)

lemma map-to-set-inj:
  (k,v) ∈ map-to-set m ⇒ (k,v') ∈ map-to-set m ⇒ v = v'
by (auto simp: map-to-set-def)

```

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 ≡ x=y

```

```

definition [simp]: REFINEG-TRANSFER-ALIGN x y == True

```

```

lemma REFINEG-TRANSFER-ALIGN1: REFINEG-TRANSFER-ALIGN x y by
simp

```

```

lemma START-REFINEG-TRANSFER:

```

```

assumes REFINEG-TRANSFER-ALIGN  $d\ c$ 
assumes  $c \leq a$ 
assumes REFINEG-TRANSFER-POST-SIMP  $c\ d$ 
shows  $d \leq a$ 
using assms
by (simp add: REFINEG-TRANSFER-POST-SIMP-def)

```

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

```

unfolding REFINEG-TRANSFER-POST-SIMP-def ..

```

ML ‹

```

structure RefineG-Transfer = struct

```

```

  structure Post-Processors = Theory-Data (
    type T = (Proof.context -> tactic') Symtab.table
    val empty = Symtab.empty
    val extend = I
    val merge = Symtab.join (K snd)
  )

```

```

  fun add-post-processor name tac =
    Post-Processors.map (Symtab.update-new (name,tac))
  fun delete-post-processor name =
    Post-Processors.map (Symtab.delete name)
  val get-post-processors = Post-Processors.get #> Symtab.dest

```

```

  fun post-process-tac ctxt = let
    val tacs = get-post-processors (Proof-Context.theory-of ctxt)
    |> map (fn (-,tac) => tac ctxt)

```

```

    val tac = REPEAT-DETERM' (CHANGED o EVERY' (map (fn t => TRY
o t) tacs))
  in
    tac
  end

```

```

  structure Post-Simp = Generic-Data (
    type T = simpset
    val empty = HOL-basic-ss
    val extend = I
    val merge = Raw-Simplifier.merge-ss
  )

```

```

  fun post-simps-op f a context = let
    val ctxt = Context.proof-of context
    fun do-it ss = simpset-of (f (put-simpset ss ctxt, a))
  in
    Post-Simp.map do-it context

```

```

end

val add-post-simps = post-simps-op (op addsimps)
val del-post-simps = post-simps-op (op delsimps)

fun get-post-ss ctxt = let
  val ss = Post-Simp.get (Context.Proof ctxt)
  val ctxt = put-simpset ss ctxt
in
  ctxt
end

structure post-subst = Named-Thms
  ( val name = @{binding refine-transfer-post-subst}
    val description = Refinement Framework: ^
      Transfer postprocessing substitutions );

fun post-subst-tac ctxt = let
  val s-thms = post-subst.get ctxt
  val dis-tac = (ALLGOALS (Tagged-Solver.solve-tac ctxt))
  val cnv = Cond-Rewr-Conv.cond-rewrs-conv dis-tac s-thms
  val ts-conv = Conv.top-sweep-conv cnv ctxt
  val ss = get-post-ss ctxt
in
  REPEAT o CHANGED o
  (Simplifier.simp-tac ss THEN' CONVERSION ts-conv)
end

structure transfer = Named-Thms
  ( val name = @{binding refine-transfer}
    val description = Refinement Framework: ^
      Transfer rules );

fun transfer-tac thms ctxt i st = let
  val thms = thms @ transfer.get ctxt;
  val ss = put-simpset HOL-basic-ss ctxt addsimps @{thms nested-case-prod-simp}
in
  REPEAT-DETERM1 (
    COND (has-fewer-prems (Thm.nprems-of st)) no-tac (
      FIRST [
        Method.assm-tac ctxt i,
        resolve-tac ctxt thms i,
        Tagged-Solver.solve-tac ctxt i,
        CHANGED-PROP (simp-tac ss i)]
    )) st
end

(* Adjust right term to have same structure as left one *)

```



```

fun align-tac ctxt = IF-EXGOAL (fn i => fn st =>
  case Logic.concl-of-goal (Thm.prop-of st) i of
    @{\mpat Trueprop (REFINEG-TRANSFER-ALIGN ?c -)} => let
      val c = Thm.cterm-of ctxt c
      val cT = Thm.ctyp-of-cterm c

      val rl = @{\thm REFINEG-TRANSFER-ALIGN}
        |> Thm.incr-indexes (Thm.maxidx-of st + 1)
        |> Thm.instantiate' [NONE,SOME cT] [NONE,SOME c]
        (*val - = tracing (@{\make-string} rl)*)
      in
        resolve-tac ctxt [rl] i st
      end
    | - => Seq.empty
  )

fun post-transfer-tac thms ctxt = let open Autoref-Tacticals in
  resolve-tac ctxt @{\thms START-REFINEG-TRANSFER}
  THEN' align-tac ctxt
  THEN' IF-SOLVED (transfer-tac thms ctxt)
  (post-process-tac ctxt THEN' resolve-tac ctxt @{\thms STOP-REFINEG-TRANSFER})
  (K all-tac)

end

fun get-post-simp-rules context = Context.proof-of context
  |> get-post-ss
  |> simpset-of
  |> Raw-Simplifier.dest-ss
  |> #simps |> map snd

local
  val add-ps = Thm.declaration-attribute (add-post-simps o single)
  val del-ps = Thm.declaration-attribute (del-post-simps o single)
in
  val setup = I
    #> add-post-processor RefineG-Transfer.post-subst post-subst-tac
    #> post-subst.setup
    #> transfer.setup
    #> Attrib.setup @{\binding refine-transfer-post-simp}
      (Attrib.add-del add-ps del-ps)
      (declaration of transfer post simplification rules)
    #> Global-Theory.add-thms-dynamic (
      @{\binding refine-transfer-post-simps}, get-post-simp-rules)
end
end
>

```

```

setup ‹RefineG-Transfer.setup›
method-setup refine-transfer =
  ‹Scan.lift (Args.mode post) -- Attrib.thms
  >> (fn (post,thms) => fn ctxt => SIMPLE-METHOD'
      ( if post then RefineG-Transfer.post-transfer-tac thms ctxt
        else RefineG-Transfer.transfer-tac thms ctxt))
  › Invoke transfer rules

```

```

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  $\alpha \ (\text{if } b \text{ then } s1 \text{ else } s2) \leq (\text{if } b \text{ then } S1 \text{ else } S2)$ 
  using assms by auto

```

```

lemma transfer-prod[refine-transfer]:
  assumes  $\bigwedge a \ b. \alpha \ (f \ a \ b) \leq F \ a \ b$ 
  shows  $\alpha \ (\text{case-prod } f \ x) \leq (\text{case-prod } F \ x)$ 
  using assms by (auto split: prod.split)

```

```

lemma transfer-Let[refine-transfer]:
  assumes  $\bigwedge x. \alpha \ (f \ x) \leq F \ x$ 
  shows  $\alpha \ (\text{Let } x \ f) \leq \text{Let } x \ F$ 
  using assms by auto

```

```

lemma transfer-option[refine-transfer]:
  assumes  $\alpha \ fa \leq Fa$ 
  assumes  $\bigwedge x. \alpha \ (fb \ x) \leq Fb \ x$ 
  shows  $\alpha \ (\text{case-option } fa \ fb \ x) \leq \text{case-option } Fa \ Fb \ x$ 
  using assms by (auto split: option.split)

```

```

lemma transfer-sum[refine-transfer]:
  assumes  $\bigwedge l. \alpha \ (fl \ l) \leq Fl \ l$ 
  assumes  $\bigwedge r. \alpha \ (fr \ r) \leq Fr \ r$ 
  shows  $\alpha \ (\text{case-sum } fl \ fr \ x) \leq (\text{case-sum } Fl \ Fr \ x)$ 
  using assms by (auto split: sum.split)

```

```

lemma transfer-list[refine-transfer]:
  assumes  $\alpha \ fn \leq Fn$ 
  assumes  $\bigwedge x \ xs. \alpha \ (fc \ x \ xs) \leq Fc \ x \ xs$ 
  shows  $\alpha \ (\text{case-list } fn \ fc \ l) \leq \text{case-list } Fn \ Fc \ l$ 
  using assms by (auto split: list.split)

```

```

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  $\alpha (rec\text{-list}\ fn\ fc\ l\ s) \leq rec\text{-list}\ fn'\ fc'\ l\ s$ 
  apply (induct l arbitrary: s)
  apply (simp add: FN)
  apply (simp add: FC)
  done

```

```

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  $\alpha (rec\text{-nat}\ fn\ fs\ n\ s) \leq rec\text{-nat}\ fn'\ fs'\ n\ s$ 
  apply (induct n arbitrary: s)
  apply (simp add: FN)
  apply (simp add: FC)
  done

```

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  $\alpha$ 
  apply rule
  apply (subgoal-tac is-chain {x,y})
  apply (drule  $\alpha\text{-dist}$ )
  apply (auto simp: le-iff-sup)  $\square$ 
  apply (rule chainI)
  apply auto
  done

```

```

  lemma  $\alpha\text{-strict}$ [simp]:  $\alpha\ bot = bot$ 
  using  $\alpha\text{-dist}$ [of {}] by simp

```

end

Transfer into ccpo

```

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

```

Transfer into ccpo with distributive transfer function.

```

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

  lemma  $\alpha\text{-mono}[\text{simp}, \text{intro!}]: \text{mono } \alpha$ 
  proof
    fix  $x\ y :: 'c$ 
    assume  $LE: x \leq y$ 
    hence  $C[\text{simp}, \text{intro!}]: \text{is-chain } \{x, y\}$  by (auto intro: chainI)
    from  $LE$  have  $\alpha\ x \leq \text{sup } (\alpha\ x) (\alpha\ y)$  by simp
    also have  $\dots = \text{Sup } (\alpha \{x, y\})$  by simp
    also have  $\dots = \alpha (\text{Sup } \{x, y\})$ 
      by (rule  $\alpha\text{-dist}[\text{symmetric}]$ ) simp
    also have  $\text{Sup } \{x, y\} = y$ 
      apply (rule antisym)
      apply (rule ccpo-Sup-least[OF C]) using  $LE$  apply auto []
      apply (rule ccpo-Sup-upper[OF C]) by auto
    finally show  $\alpha\ x \leq \alpha\ y$  .
  qed

  lemma  $\alpha\text{-strict}[\text{simp}]: \alpha (\text{Sup } \{\}) = \text{bot}$ 
    using  $\alpha\text{-dist}[\text{of } \{\}]$  by simp
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)  $F$ 
   $\Longrightarrow$  Complete-Partial-Order.chain le  $\{y . \exists f \in F. y = f\ x\}$ 
  unfolding Complete-Partial-Order.chain-def
  by (auto simp: fun-ord-def)

lemma ccpo-lift:
  assumes class.ccpo lub le lt
  shows class.ccpo (fun-lub lub) (fun-ord le) (mk-less (fun-ord le))
proof –
  interpret ccpo: ccpo lub le lt by fact
  show ?thesis
    apply unfold-locales
    apply (simp-all only: mk-less-def fun-ord-def fun-lub-def)

```

```

apply simp
using ccpo.order-trans apply blast
using ccpo.order.antisym apply blast
using ccpo.ccpo-Sup-upper apply (blast intro: chain-f-apply)
using ccpo.ccpo-Sup-least apply (blast intro: chain-f-apply)
done
qed

```

```

lemma fun-lub-simps[simp]:
  fun-lub lub {} = (λx. lub {})
  fun-lub lub {f} = (λx. lub {f x})
  unfolding fun-lub-def by auto

```

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 where x≠b and C={x}
  | x where x≠b and C={b,x}
proof -
  have ∃ m1 m2. C ⊆ {m1,m2} ∧ (m1 = b ∨ m2 = b)
  apply (rule ccontr)
  proof clarsimp
  assume ∀ m1 m2. C ⊆ {m1, m2} → m1≠b ∧ m2≠b
  then obtain m1 m2 where m1∈C m2∈C
    m1≠m2 m1≠b m2≠b
  by blast
  with A show False
  unfolding Complete-Partial-Order.chain-def flat-ord-def
  by auto
  qed
  then obtain m where C ⊆ {m,b} by blast
  thus ?thesis
  apply (cases m=b)
  using that
  apply blast+
  done
qed

```

```

lemma flat-lub-simps[simp]:
  flat-lub b {} = b
  flat-lub b {x} = x
  flat-lub b (insert b X) = flat-lub b X
  unfolding flat-lub-def
  by auto

```

lemma *flat-ord-simps*[*simp*]:
flat-ord b b x
by (*simp add: flat-ord-def*)

interpretation *flat-ord*: *ccpo flat-lub b flat-ord b mk-less (flat-ord b)*
apply *unfold-locales*
apply (*simp-all only: mk-less-def flat-ord-def*) **apply** *auto* [4]
apply (*erule flat-ord-chain-cases, auto*) []
apply (*erule flat-ord-chain-cases, auto*) []
done

interpretation *flat-le-mono-setup*: *mono-setup-loc flat-ord b*
by *standard auto*

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)*
apply (*rule ccpo-lift*)
apply *unfold-locales*
done

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* ≡ *flatf-ord.fixp b*

lemma *flatf-fp-mono*[*refine-mono*]:
— The fixed point combinator is monotonic
assumes *flatf-mono b f*
and *flatf-mono b g*
and $\bigwedge Z x. \text{flat-ord } b (f Z x) (g Z x)$
shows *flat-ord b (flatf-fp b f x) (flatf-fp b g x)*
proof —
have *flatf-ord b (flatf-fp b f) (flatf-fp b g)*
apply (*rule flatf-ord.fixp-mono[OF assms(1,2)]*)
using *assms(3)* **by** (*simp add: fun-ord-def*)
thus *?thesis* **unfolding** *fun-ord-def* **by** *blast*
qed

lemma *flatf-admissible-pointwise*:
 $(\bigwedge x. P x b) \implies$
ccpo.admissible (flatf-lub b) (flatf-ord b) ($\lambda g. \forall x. P x (g x)$)
apply (*intro ccpo.admissibleI allI impI*)

```

apply (drule-tac  $x=x$  in chain-f-apply)
apply (erule flat-ord-chain-cases)
apply (auto simp add: fun-lub-def) [2]
apply (force simp add: fun-lub-def) []
apply (auto simp add: fun-lub-def) []
done

```

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*: *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 $\text{post } a \ x \ (\text{flatf-fp } b \ B \ x)$

proof –

define ub **where** $ub = \text{flatf-fp } b \ B$

have $(\forall x \ a. \text{pre } a \ x \longrightarrow \text{post } a \ x \ (\text{flatf-fp } b \ B \ x))$

$\wedge \text{flatf-ord } b \ (\text{flatf-fp } b \ B) \ ub$

apply (*rule* *flatf-ord.fixp-induct[OF - MONO]*)

apply (*rule* *admissible-conj*)

apply (*rule* *flatf-admissible-pointwise*)

apply (*blast intro: cond-bot*)

apply (*rule* *ccpo.admissibleI*)

apply (*blast intro: flatf-ord.ccpo-Sup-least*)

apply (*auto intro: cond-bot simp: fun-ord-def flat-ord-def*) []

apply (*rule* *conjI*)

unfolding *ub-def*

apply (*blast intro: STEP*)

apply (*subst* *flatf-ord.fixp-unfold[OF MONO]*)

apply (*blast intro: monotoneD[OF MONO]*)

done

with *PRE0* **show** *?thesis* **by** *blast*

qed

The next rule covers transfer between fixed points. It allows to lift a point-wise transfer condition $P x y \longrightarrow tr (f x) (f y)$ 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*]: $\bigwedge x'. tr b x'$

assumes *MONO*: *flatf-mono* $b B$

assumes *FP'*: $fp' = B' fp'$

assumes *R0*: $P x x'$

assumes *RS*: $\bigwedge f f' x x'.$

$\llbracket \bigwedge x x'. P x x' \implies tr (f x) (f' x'); P x x'; fp' = f \rrbracket$

$\implies tr (B f x) (B' f' x')$

shows $tr (flatf-fp b B x) (fp' x')$

apply (*rule flatf-fp-induct-pointwise*[**where** $pre = \lambda x y. P y x, OF - MONO$])

apply *simp*

apply (*rule R0*)

apply (*subst FP'*)

apply (*blast intro: RS*)

done

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* $\equiv flat-ord bot$

abbreviation *flat-ge* $\equiv flat-ord top$

abbreviation *flatf-le* $\equiv flatf-ord bot$

abbreviation *flatf-ge* $\equiv flatf-ord top$

The flat ordering implies the lattice ordering

lemma *flat-ord-compat*:

fixes $x y :: 'a :: complete-lattice$

shows

flat-le $x y \implies x \leq y$

flat-ge $x y \implies x \geq y$

unfolding *flat-ord-def* **by** *auto*

lemma *flatf-ord-compat*:

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

shows

flatf-le $x y \implies x \leq y$

flatf-ge $x y \implies x \geq y$

by (*auto simp: flat-ord-compat le-fun-def fun-ord-def*)

abbreviation *flat-mono-le* $\equiv flat-mono bot$

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

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

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

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

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

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

lemma $\textit{lfp-eq-flatf-lfp}$:

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

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

proof –

from $\textit{lfp-unfold}[OF M, \textit{symmetric}]$ **have** $B (\textit{lfp } B) = \textit{lfp } B$.

hence $\textit{flatf-le } (B (\textit{lfp } B)) (\textit{lfp } B)$ **by** \textit{simp}

with $\textit{flatf-ord.fixp-lowerbound}[OF FM]$ **have** $\textit{flatf-le } (\textit{flatf-lfp } B) (\textit{lfp } B)$.

with $\textit{flatf-ord-compat}$ **have** $\textit{flatf-lfp } B \leq \textit{lfp } B$ **by** \textit{blast}

also

from $\textit{flatf-ord.fixp-unfold}[OF FM, \textit{symmetric}]$ **have** $B (\textit{flatf-lfp } B) = \textit{flatf-lfp}$

B .

hence $B (\textit{flatf-lfp } B) \leq \textit{flatf-lfp } B$ **by** \textit{simp}

with $\textit{lfp-lowerbound}[\textit{where } A=\textit{flatf-lfp } B]$ **have** $\textit{lfp } B \leq \textit{flatf-lfp } B$.

finally show $\textit{lfp } B = \textit{flatf-lfp } B$..

qed

lemma $\textit{gfp-eq-flatf-gfp}$:

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

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

proof –

from $\textit{gfp-unfold}[OF M, \textit{symmetric}]$ **have** $B (\textit{gfp } B) = \textit{gfp } B$.

hence $\textit{flatf-ge } (B (\textit{gfp } B)) (\textit{gfp } B)$ **by** \textit{simp}

with $\textit{flatf-ord.fixp-lowerbound}[OF FM]$ **have** $\textit{flatf-ge } (\textit{flatf-gfp } B) (\textit{gfp } B)$.

with $\textit{flatf-ord-compat}$ **have** $\textit{gfp } B \leq \textit{flatf-gfp } B$ **by** \textit{blast}

also

from $\textit{flatf-ord.fixp-unfold}[OF FM, \textit{symmetric}]$ **have** $B (\textit{flatf-gfp } B) = \textit{flatf-gfp}$

B .

hence $\textit{flatf-gfp } B \leq B (\textit{flatf-gfp } B)$ **by** \textit{simp}

with $\textit{gfp-upperbound}[\textit{where } X=\textit{flatf-gfp } B]$ **have** $\textit{flatf-gfp } B \leq \textit{gfp } B$.

finally show $\textit{gfp } B = \textit{flatf-gfp } B$.

qed

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

lemma $\textit{wf-fixp-induct}$:

– Well-Founded induction for arbitrary fixed points

fixes $a :: 'a$

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

assumes $WF: \textit{wf } V$

```

assumes P0: pre a x
assumes STEP:  $\bigwedge f a x. \llbracket$ 
   $\bigwedge a' x'. \llbracket$  pre a' x';  $(x', x) \in V$   $\rrbracket \implies$  post a' x' (f x'); fp B = f; pre a x
 $\rrbracket \implies$  post a x (B f x)
shows post a x (fp B x)
proof –
  have  $\forall a. \text{pre } a \ x \longrightarrow \text{post } a \ x \ (\text{fp } B \ x)$ 
    using WF
    apply (induct x rule: wf-induct-rule)
    apply (intro allI impI)
    apply (subst fixp-unfold)
    apply (rule STEP)
    apply (simp)
    apply (simp)
    apply (simp)
    done
  with P0 show ?thesis by blast
qed

```

lemma flatf-lfp-transfer:

— Transfer rule for least fixed points

```

fixes B::(-  $\Rightarrow$  'a::order-bot)  $\Rightarrow$  -
assumes TR-BOT[simp]:  $\bigwedge x. \text{tr bot } x$ 
assumes MONO: flatf-mono-le B
assumes MONO': flatf-mono-le B'
assumes R0: P x x'
assumes RS:  $\bigwedge f f' x x'.$ 
   $\llbracket \bigwedge x x'. P \ x \ x' \implies \text{tr } (f \ x) \ (f' \ x'); P \ x \ x'; \text{flatf-lfp } B' = f \rrbracket$ 
   $\implies \text{tr } (B \ f \ x) \ (B' \ f' \ x')$ 
shows tr (flatf-lfp B x) (flatf-lfp B' x')
apply (rule flatf-fixp-transfer[where tr=tr and fp'=flatf-lfp B' and P=P])
apply (fact|rule flatf-ord.fixp-unfold[OF MONO'])+
done

```

lemma flatf-gfp-transfer:

— Transfer rule for greatest fixed points

```

fixes B::(-  $\Rightarrow$  'a::order-top)  $\Rightarrow$  -
assumes TR-TOP[simp]:  $\bigwedge x. \text{tr } x \ \text{top}$ 
assumes MONO: flatf-mono-ge B
assumes MONO': flatf-mono-ge B'
assumes R0: P x x'
assumes RS:  $\bigwedge f f' x x'.$ 
   $\llbracket \bigwedge x x'. P \ x \ x' \implies \text{tr } (f \ x) \ (f' \ x'); P \ x \ x'; \text{flatf-gfp } B = f \rrbracket$ 
   $\implies \text{tr } (B \ f \ x) \ (B' \ f' \ x')$ 
shows tr (flatf-gfp B x) (flatf-gfp B' x')
apply (rule flatf-fixp-transfer[where
  tr= $\lambda x \ y. \text{tr } y \ x$  and fp'=flatf-gfp B and P= $\lambda x \ y. P \ y \ x$ ])
apply (fact|assumption|rule flatf-ord.fixp-unfold[OF MONO] RS)+

```

2.4. GENERIC RECURSION COMBINATOR FOR COMPLETE LATTICE STRUCTURED DOMAINS

done

lemma *meta-le-everything-if-top*: $(m=top) \implies (\bigwedge x. x \leq (m::'a::order-top))$
by *auto*

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*]: $\bigwedge a\ a'::'a::complete-lattice.$
 $flat-ge\ a\ a' \implies flat-ge\ b\ b' \implies flat-ge\ (sup\ a\ b)\ (sup\ a'\ b')$
 by (*auto simp: flat-ord-def*)

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

proof —

note *UNF1* = *flatf-ord.fixp-unfold*[*OF M1*, *symmetric*]

note *UNF2* = *flatf-ord.fixp-unfold*[*OF M2*, *symmetric*]

from *UNF1* **have** *1*: *flatf-ord* *b2* (*B* (*flatf-fp* *b1 B*)) (*flatf-fp* *b1 B*) **by** *simp*

from *UNF2* **have** *2*: *flatf-ord* *b1* (*B* (*flatf-fp* *b2 B*)) (*flatf-fp* *b2 B*) **by** *simp*

from *flatf-ord.fixp-lowerbound*[*OF M2 1*] *flatf-ord.fixp-lowerbound*[*OF M1 2*]

show *?thesis* **unfolding** *fun-ord-def flat-ord-def* **by** *auto*

qed

lemma *flat-ord-top*[*simp*]: *flat-ord* *b b x* **by** (*simp add: flat-ord-def*)

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*

∨ (*flatf-lfp* *B x* = *bot* ∧ *flatf-gfp* *B x* = *top*)

using *fp-compare*[*OF MLE MGE*] .

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

where *trimono* *B* ≡ ~~*flatf-lfp* *B bot* = *flatf-gfp* *B bot*~~ *flatf-mono-ge* *B* ∧ *mono* *B*

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

[[*flatf-mono-ge* *B*; *mono* *B*]] ⇒ *trimono* *B*

unfolding *trimono-def* **by** *auto*

lemma *trimono-trigger*: *trimono* *B* ⇒ *trimono* *B* .

declaration <*Refine-Mono-Prover.declare-mono-triggers* @{*thms trimono-trigger*}>

lemma *trimonoD-flatf-ge*: *trimono* *B* ⇒ *flatf-mono-ge* *B*

unfolding *trimono-def* **by** *auto*

lemma *trimonoD-mono*: *trimono* *B* ⇒ *mono* *B*

unfolding *trimono-def* **by** *auto*

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

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

lemma *trimono-alt*:

trimono *B* ⇔ (∀ *ord* ∈ *fun-ord* *triords*. *monotone* *ord* *ord* *B*)

unfolding *trimono-def*

by (*auto simp: ccpo-mono-simp*[*symmetric*] *triords-def*

fun-ord-def[*abs-def*] *le-fun-def*[*abs-def*])

lemma *trimonoI'*:

assumes ∧*ord*. *ord* ∈ *triords* ⇒ *monotone* (*fun-ord* *ord*) (*fun-ord* *ord*) *B*

shows *trimono* *B*

unfolding *trimono-alt* **using** *assms* **by** *blast*

2.4. GENERIC RECURSION COMBINATOR FOR COMPLETE LATTICE STRUCTURED DOMAINS

definition *REC* **where** $REC\ B\ x \equiv$
if (*trimono* *B*) *then* (*lfp* *B* *x*) *else* (*top*::'*a*::*complete-lattice*)

definition *RECT* (*RECT_T*) **where** $RECT\ B\ x \equiv$
if (*trimono* *B*) *then* (*flatf-gfp* *B* *x*) *else* (*top*::'*a*::*complete-lattice*)

lemma *RECT-gfp-def*: $RECT\ B\ x =$
(if (*trimono* *B*) *then* (*gfp* *B* *x*) *else* (*top*::'*a*::*complete-lattice*))
unfolding *RECT-def*
by (*auto simp: gfp-eq-flatf-gfp[OF trimonoD-flatf-ge trimonoD-mono]*)

lemma *REC-unfold*: $trimono\ B \implies REC\ B = B\ (REC\ B)$
unfolding *REC-def* [*abs-def*]
by (*simp add: lfp-unfold[OF trimonoD-mono, symmetric]*)

lemma *RECT-unfold*: $\llbracket trimono\ B \rrbracket \implies RECT\ B = B\ (RECT\ B)$
unfolding *RECT-def* [*abs-def*]
by (*simp add: flatf-ord.fixp-unfold[OF trimonoD-flatf-ge, symmetric]*)

lemma *REC-mono*[*refine-mono*]:
assumes [*simp*]: *trimono* *B*
assumes *LE*: $\bigwedge F\ x. (B\ F\ x) \leq (B'\ F\ x)$
shows $(REC\ B\ x) \leq (REC\ B'\ x)$
unfolding *REC-def*
apply *clarsimp*
apply (*rule lfp-mono[THEN le-funD]*)
apply (*rule LE[THEN le-funI]*)
done

lemma *RECT-mono*[*refine-mono*]:
assumes [*simp*]: *trimono* *B'*
assumes *LE*: $\bigwedge F\ x. flat-ge\ (B\ F\ x)\ (B'\ F\ x)$
shows $flat-ge\ (RECT\ B\ x)\ (RECT\ B'\ x)$
unfolding *RECT-def*
apply *clarsimp*
apply (*rule flatf-fp-mono, (simp-all add: trimonoD) [2]*)
apply (*rule LE*)
done

lemma *REC-le-RECT*: $REC\ body\ x \leq RECT\ body\ x$
unfolding *REC-def* *RECT-gfp-def*
apply (*cases trimono body*)
apply *clarsimp*
apply (*rule lfp-le-gfp[THEN le-funD]*)
apply (*simp add: trimonoD*)
apply *simp*
done

```

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:
     $\bigwedge f a x. \llbracket \bigwedge a' x'. \text{pre } a' x' \Longrightarrow \text{post } a' x' (f x'); \text{pre } a x; f \leq (\text{lfp } B) \rrbracket \Longrightarrow \text{post } a x (B f x)$ 
  shows post a x (lfp B x)
proof –
  define u where u = lfp B

  have [simp]:  $\bigwedge f. f \leq \text{lfp } B \Longrightarrow B f \leq \text{lfp } B$ 
    by (metis (poly-guards-query) MONO lfp-unfold monoD)

  have ( $\forall a x. \text{pre } a x \longrightarrow \text{post } a x (\text{lfp } B x) \wedge \text{lfp } B \leq u$ )
    apply (rule lfp-cadm-induct[where f=B])
    apply (rule admissible-conj)
    apply (rule ADM1)
    apply (rule)
    apply (blast intro: Sup-least)
    apply (simp add: le-fun-def ADM2) []
    apply fact
    apply (intro conjI allI impI)
  unfolding u-def
  apply (blast intro: IS)
  apply simp
  done
  with P0 show ?thesis by blast
qed

```

```

lemma REC-rule-arb:
  fixes x::'x and arb::'arb
  assumes M: trimono body
  assumes I0: pre arb x
  assumes IS:  $\bigwedge f \text{ arb } x. \llbracket \bigwedge \text{arb}' x. \text{pre } \text{arb}' x \Longrightarrow f x \leq M \text{arb}' x; \text{pre } \text{arb } x; f \leq \text{REC } \text{body} \rrbracket \Longrightarrow \text{body } f x \leq M \text{arb } x$ 
  shows REC body x  $\leq M \text{arb } x$ 
  unfolding REC-def
  apply (clarsimp simp: M)
  apply (rule lfp-induct-pointwise[where pre=pre])
  apply (auto intro!: chain-admissibleI SUP-least) [2]
  apply (simp add: trimonoD[OF M])

```

2.4. GENERIC RECURSION COMBINATOR FOR COMPLETE LATTICE STRUCTURED DOMAINS

```

apply (rule I0)
apply (rule IS, assumption+)
apply (auto simp: REC-def[abs-def] intro!: le-funI dest: le-funD) []
done

```

lemma *RECT-rule-arb*:

```

assumes M: trimono body
assumes WF: wf (V::('x×'x) set)
assumes I0: pre (arb::'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 RECT body  $x \leq M \text{ arb } x$ 
apply (rule wf-fixp-induct[where fp=RECT and pre=pre and B=body])
apply (rule RECT-unfold)
apply (simp-all add: M) [2]
apply (rule WF)
apply fact
apply (rule IS)
apply assumption
apply assumption
apply assumption
done

```

lemma *REC-rule*:

```

fixes x::'x
assumes M: trimono body
assumes I0: 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 REC body  $x \leq M x$ 
by (rule REC-rule-arb[where pre= $\lambda-. \text{pre}$  and M= $\lambda-. M$ , OF assms])

```

lemma *RECT-rule*:

```

assumes M: trimono body
assumes WF: wf (V::('x×'x) set)
assumes I0: 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 RECT body  $x \leq M x$ 
by (rule RECT-rule-arb[where pre= $\lambda-. \text{pre}$  and M= $\lambda-. M$ , OF assms])

```

lemma *REC-rule-arb2*:

assumes *M*: trimono body
assumes *I0*: pre (arb::*'arb*) (arc::*'arc*) (*x*::*'x*)
assumes *IS*: $\bigwedge f \text{ arb arc } x. \llbracket$
 $\bigwedge \text{arb}' \text{ arc}' x'. \llbracket \text{pre arb}' \text{ arc}' x' \rrbracket \implies f x' \leq M \text{ arb}' \text{ arc}' x';$
 $\text{pre arb arc } x$
 $\rrbracket \implies \text{body } f x \leq M \text{ arb arc } x$
shows *REC* body $x \leq M \text{ arb arc } x$
apply (rule order-trans)
apply (rule *REC-rule-arb*)
where pre=case-prod pre **and** M=case-prod M **and** arb=(arb, arc),
OF M])
by (auto intro: assms)

lemma *REC-rule-arb3*:

assumes *M*: trimono body
assumes *I0*: pre (arb::*'arb*) (arc::*'arc*) (ard::*'ard*) (*x*::*'x*)
assumes *IS*: $\bigwedge f \text{ arb arc ard } x. \llbracket$
 $\bigwedge \text{arb}' \text{ arc}' \text{ ard}' x'. \llbracket \text{pre arb}' \text{ arc}' \text{ ard}' x' \rrbracket \implies f x' \leq M \text{ arb}' \text{ arc}' \text{ ard}' x';$
 $\text{pre arb arc ard } x$
 $\rrbracket \implies \text{body } f x \leq M \text{ arb arc ard } x$
shows *REC* body $x \leq M \text{ arb arc ard } x$
apply (rule order-trans)
apply (rule *REC-rule-arb2*)
where pre=case-prod pre **and** M=case-prod M **and** arb=(arb, arc) **and**
arc=ard,
OF M])
by (auto intro: assms)

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*: $\bigwedge f \text{ arb arc } x. \llbracket$
 $\bigwedge \text{arb}' \text{ arc}' x'. \llbracket \text{pre arb}' \text{ arc}' x'; (x',x) \in V \rrbracket \implies f x' \leq M \text{ arb}' \text{ arc}' x';$
 $\text{pre arb arc } x;$
 $f \leq \text{RECT body}$
 $\rrbracket \implies \text{body } f x \leq M \text{ arb arc } x$
shows *RECT* body $x \leq M \text{ arb arc } x$
apply (rule order-trans)
apply (rule *RECT-rule-arb*)
where pre=case-prod pre **and** M=case-prod M **and** arb=(arb, arc),
OF M WF])
by (auto intro: assms)

lemma *RECT-rule-arb3*:

assumes *M*: trimono body
assumes *WF*: wf (*V*::*'x* rel)

2.4. GENERIC RECURSION COMBINATOR FOR COMPLETE LATTICE STRUCTURED DOMAINS

```

assumes I0: pre (arb::arb) (arc::arc) (ard::ard) (x::x)
assumes IS:  $\bigwedge f \text{ arb arc ard } x. \llbracket$ 
   $\bigwedge \text{arb}' \text{ arc}' \text{ ard}' x'. \llbracket \text{pre } \text{arb}' \text{ arc}' \text{ ard}' x'; (x',x) \in V \rrbracket \implies f x' \leq M \text{ arb}' \text{ arc}'$ 
   $\text{ard}' x'$ ;
  pre arb arc ard x;
   $f \leq \text{RECT body}$ 
 $\rrbracket \implies \text{body } f x \leq M \text{ arb arc ard } x$ 
shows RECT body  $x \leq M \text{ arb arc ard } x$ 
apply (rule order-trans)
apply (rule RECT-rule-arb2)
  where pre=case-prod pre and M=case-prod M and arb=(arb, arc) and
  arc=ard,
  (OF M WF])
by (auto intro: assms)

```

lemma RECT-*eq-REC*:

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

assumes NT: RECT body $x \neq \text{top}$

shows RECT body $x = \text{REC body } x$

proof (*cases trimono body*)

case M: True

show ?*thesis*

using NT M

unfolding RECT-*def* REC-*def*

proof *clarsimp*

from *lfp-unfold*[OF *trimonoD-mono*[OF M], *symmetric*]

have flatf-*ge* (body (lfp body)) (lfp body) **by** *simp*

note flatf-*ord.fixp-lowerbound*[

OF *trimonoD-flatf-ge*[OF M], of lfp body, OF *this*]

moreover assume flatf-*gfp* body $x \neq \text{top}$

ultimately show flatf-*gfp* body $x = \text{lfp body } x$

by (auto *simp add: fun-ord-def flat-ord-def*)

qed

next

case False **thus** ?*thesis* **unfolding** RECT-*def* REC-*def* **by** *auto*

qed

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: $\bigwedge f \text{ arb } x.$

$$\llbracket \bigwedge arb' x'. \llbracket pre arb' x'; (x', x) \in V \rrbracket \implies f x' \leq M arb' x';$$

$$pre arb x; REC_T body = f \rrbracket$$

$$\implies body f x \leq M arb x$$
assumes *NT*: $M a x \neq top$
shows $RECT body x = REC body x \wedge RECT body x \leq M a x$
proof
show $RECT body x \leq M a x$
by (*rule* *RECT-rule-arb*[*OF* *M WF*, **where** *pre=pre*, *OF* *I0 IS*])

with *NT* **have** $RECT body x \neq top$ **by** (*metis* *top.extremum-unique*)
thus $RECT body x = REC body x$ **by** (*rule* *RECT-eq-REC*)
qed

2.4.1 Transfer

lemma (*in transfer*) *transfer-RECT'*[*refine-transfer*]:
assumes *REC-EQ*: $\bigwedge x. fr x = b fr x$
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 $\alpha (fr x) \leq RECT B x$
unfolding *RECT-def*
proof *clarsimp*
assume *MONO*: *trimono B*
show $\alpha (fr x) \leq flatf-gfp B x$
apply (*rule* *flatf-fixp-transfer*[**where** $B=B$ **and** $fp'=fr$ **and** $P=(=)$,
OF - *trimonoD-flatf-ge*[*OF* *MONO*])]
apply *simp*
apply (*rule* *ext*, *fact*)
apply (*simp*)
apply (*simp*, *rule* *REF*, *blast*)
done
qed

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 $\alpha (RECT b x) \leq RECT B x$
apply (*rule* *transfer-RECT'*)
apply (*rule* *RECT-unfold*[*OF* *M*, *THEN* *fun-cong*])
by *fact*

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 $\alpha (REC b x) \leq REC B x$
unfolding *REC-def*

apply (*clarsimp* *simp*: *M*)
apply (*rule* *lfp-induct-pointwise*[**where** $B=b$ **and** $pre=(=)$])
apply (*rule*)

```

apply clarsimp
apply (subst  $\alpha$ -dist)
apply (auto simp add: chain-def le-fun-def) []
apply (rule Sup-least)
apply auto []
apply simp
apply (simp add: trimonoD[OF M])
apply (rule refl)
apply (subst lfp-unfold)
apply (simp add: trimonoD)
apply (rule REF)
apply blast
done

```

lemma *RECT-transfer-rel*:

```

assumes [simp]: trimono F trimono F'
assumes TR-top[simp]:  $\bigwedge x. tr\ x\ top$ 
assumes P-start[simp]:  $P\ x\ x'$ 
assumes IS:  $\bigwedge D\ D'\ x\ x'. [\bigwedge x\ x'. P\ x\ x' \implies tr\ (D\ x)\ (D'\ x'); P\ x\ x'; RECT\ F$ 
 $=\ D] \implies tr\ (F\ D\ x)\ (F'\ D'\ x')$ 
shows  $tr\ (RECT\ F\ x)\ (RECT\ F'\ x')$ 
unfolding RECT-def
apply auto
apply (rule flatf-gfp-transfer[where tr=tr and P=P])
apply (auto simp: trimonoD-flatf-ge)
apply (rule IS)
apply (auto simp: RECT-def)
done

```

lemma *RECT-transfer-rel'*:

```

assumes [simp]: trimono F trimono F'
assumes TR-top[simp]:  $\bigwedge x. tr\ x\ top$ 
assumes P-start[simp]:  $P\ x\ x'$ 
assumes IS:  $\bigwedge D\ D'\ x\ x'. [\bigwedge x\ x'. P\ x\ x' \implies tr\ (D\ x)\ (D'\ x'); P\ x\ x'] \implies tr$ 
 $(F\ D\ x)\ (F'\ D'\ x')$ 
shows  $tr\ (RECT\ F\ x)\ (RECT\ F'\ x')$ 
using RECT-transfer-rel[where tr=tr and P=P,OF assms(1,2,3,4)] IS by
blast

```

end

2.5 Assert and Assume

```

theory RefineG-Assert
imports RefineG-Transfer
begin

```

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

definition *iASSUME* return $\Phi \equiv$ if Φ 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
    unfolding ASSERT-eq iASSERT-def by auto

  lemma ASSUME-simps[simp,code]:
    ASSUME True = return ()
    ASSUME False = bot
    unfolding ASSUME-eq iASSUME-def by auto

  lemma le-ASSERTI:  $\llbracket \Phi \Longrightarrow M \leq M' \rrbracket \Longrightarrow M \leq \text{bind } (\text{ASSERT } \Phi) (\lambda-. M')$ 
    apply (cases  $\Phi$ ) by (auto simp: ibind-strict imonad1)

  lemma le-ASSERTI-pres:  $\llbracket \Phi \Longrightarrow M \leq \text{bind } (\text{ASSERT } \Phi) (\lambda-. M') \rrbracket$ 
     $\Longrightarrow M \leq \text{bind } (\text{ASSERT } \Phi) (\lambda-. M')$ 
    apply (cases  $\Phi$ ) by (auto simp: ibind-strict imonad1)

  lemma ASSERT-leI:  $\llbracket \Phi; \Phi \Longrightarrow M \leq M' \rrbracket \Longrightarrow \text{bind } (\text{ASSERT } \Phi) (\lambda-. M) \leq$ 
 $M'$ 
    apply (cases  $\Phi$ ) by (auto simp: ibind-strict imonad1)

  lemma ASSUME-leI:  $\llbracket \Phi \Longrightarrow M \leq M' \rrbracket \Longrightarrow \text{bind } (\text{ASSUME } \Phi) (\lambda-. M) \leq M'$ 
    apply (cases  $\Phi$ ) by (auto simp: ibind-strict imonad1)
  lemma ASSUME-leI-pres:  $\llbracket \Phi \Longrightarrow \text{bind } (\text{ASSUME } \Phi) (\lambda-. M) \leq M' \rrbracket$ 
     $\Longrightarrow \text{bind } (\text{ASSUME } \Phi) (\lambda-. M) \leq M'$ 
    apply (cases  $\Phi$ ) by (auto simp: ibind-strict imonad1)
  lemma le-ASSUMEI:  $\llbracket \Phi; \Phi \Longrightarrow M \leq M' \rrbracket \Longrightarrow M \leq \text{bind } (\text{ASSUME } \Phi) (\lambda-.$ 
 $M')$ 
    apply (cases  $\Phi$ ) by (auto simp: ibind-strict imonad1)

```

The order of these declarations does matter!

lemmas [intro?] = ASSERT-leI le-ASSUMEI

lemmas [*intro?*] = *le-ASSERTI ASSUME-leI*

lemma *ASSERT-le-iff*:

$bind\ (ASSERT\ \Phi)\ (\lambda\cdot.\ S) \leq S' \iff (S' \neq top \implies \Phi) \wedge S \leq S'$
by (*cases* Φ) (*auto simp: ibind-strict imonad1 simp: top-unique*)

lemma *ASSUME-le-iff*:

$bind\ (ASSUME\ \Phi)\ (\lambda\cdot.\ S) \leq S' \iff (\Phi \implies S \leq S')$
by (*cases* Φ) (*auto simp: ibind-strict imonad1*)

lemma *le-ASSERT-iff*:

$S \leq bind\ (ASSERT\ \Phi)\ (\lambda\cdot.\ S') \iff (\Phi \implies S \leq S')$
by (*cases* Φ) (*auto simp: ibind-strict imonad1*)

lemma *le-ASSUME-iff*:

$S \leq bind\ (ASSUME\ \Phi)\ (\lambda\cdot.\ S') \iff (S \neq bot \implies \Phi) \wedge S \leq S'$
by (*cases* Φ) (*auto simp: ibind-strict imonad1 simp: bot-unique*)

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)

$\implies (unit \implies 'mac) \implies ('mac::complete-lattice)$

and *creturn* :: *unit* \implies '*muc* **and** *cASSERT* **and** *cASSUME*

and *abind* :: ('*mua*::complete-lattice)

$\implies (unit \implies 'maa) \implies ('maa::complete-lattice)$

and *areturn* :: *unit* \implies '*mua* **and** *aASSERT* **and** *aASSUME*

and α :: '*mac* \implies '*maa*

begin

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

$\llbracket \Phi \implies \alpha\ M \leq M' \rrbracket$

$\implies \alpha\ (cbind\ (cASSERT\ \Phi)\ (\lambda\cdot.\ M)) \leq (abind\ (aASSERT\ \Phi)\ (\lambda\cdot.\ M'))$

apply (*cases* Φ)

apply (*auto simp: c.ibind-strict a.ibind-strict c.imonad1 a.imonad1*)

done

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

$\llbracket \Phi; \Phi \implies \alpha\ M \leq M' \rrbracket$

$\implies \alpha\ (cbind\ (cASSUME\ \Phi)\ (\lambda\cdot.\ M)) \leq (abind\ (aASSUME\ \Phi)\ (\lambda\cdot.\ M'))$

apply (*auto simp: c.ibind-strict a.ibind-strict c.imonad1 a.imonad1*)

done

end

```

locale transfer-generic-Assert-remove =
  a: generic-Assert abind areturn aASSERT aASSUME +
  transfer  $\alpha$ 
  for abind :: ('mua::complete-lattice)
     $\Rightarrow$  (unit $\Rightarrow$ 'maa)  $\Rightarrow$  ('maa::complete-lattice)
  and areturn :: unit  $\Rightarrow$  'mua and aASSERT and aASSUME
  and  $\alpha$  :: 'mac  $\Rightarrow$  'maa
begin
  lemma transfer-ASSERT-remove[refine-transfer]:
     $\llbracket \Phi \Longrightarrow \alpha M \leq M' \rrbracket \Longrightarrow \alpha M \leq \text{abind } (a\text{ASSERT } \Phi) (\lambda\cdot. M')$ 
    by (rule a.le-ASSERTI)

  lemma transfer-ASSUME-remove[refine-transfer]:
     $\llbracket \Phi; \Phi \Longrightarrow \alpha M \leq M' \rrbracket \Longrightarrow \alpha M \leq \text{abind } (a\text{ASSUME } \Phi) (\lambda\cdot. M')$ 
    by (rule a.le-ASSUMEI)
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 $M \leq \text{SPEC } \Phi$ 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 $nres :: (type) \text{ complete-lattice}$

begin

fun *less-eq-nres* **where**

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

fun *less-nres* **where**

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

fun *sup-nres* **where**

$sup\ -\ FAILi = FAILi \mid$
 $sup\ FAILi\ - = FAILi \mid$
 $sup\ (RES\ a)\ (RES\ b) = RES\ (a \cup b)$

fun *inf-nres* **where**

$inf\ x\ FAILi = x \mid$
 $inf\ FAILi\ x = x \mid$
 $inf\ (RES\ a)\ (RES\ b) = RES\ (a \cap b)$

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

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

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

definition $top \equiv FAILi$

instance

apply (*intro-classes*)

unfolding *Sup-nres-def Inf-nres-def bot-nres-def top-nres-def*

apply (*case-tac* x , *case-tac* $[!]$ y , *auto*) \square

apply (*case-tac* x , *auto*) \square

apply (*case-tac* x , *case-tac* $[!]$ y , *case-tac* $[!]$ z , *auto*) \square

apply (*case-tac* x , (*case-tac* $[!]$ y)?, *auto*) \square

apply (*case-tac* x , (*case-tac* $[!]$ y)?, *simp-all*) \square

apply (*case-tac* x , (*case-tac* $[!]$ y)?, *auto*) \square

apply (*case-tac* x , *case-tac* $[!]$ y , *case-tac* $[!]$ z , *auto*) \square

apply (*case-tac* x , (*case-tac* $[!]$ y)?, *auto*) \square

apply (*case-tac* x , (*case-tac* $[!]$ y)?, *auto*) \square

apply (*case-tac* x , *case-tac* $[!]$ y , *case-tac* $[!]$ z , *auto*) \square

apply (*case-tac* x , *auto*) \square

apply (*case-tac* z , *fastforce+*) \square

apply (*case-tac* x , *auto*) \square

apply (*case-tac* z , *fastforce+*) \square

apply *auto* \square

apply *auto* []
done

end

abbreviation *FAIL* \equiv *top::'a nres*
abbreviation *SUCCEED* \equiv *bot::'a nres*
abbreviation *SPEC* $\Phi \equiv$ *RES (Collect Φ)*
definition *RETURN* $x \equiv$ *RES {x}*

We try to hide the original *FAIL**i*-element as well as possible.

lemma *nres-cases*[*case-names FAIL RES, cases type*]:
obtains $M=FAIL \mid X$ **where** $M=RES X$
apply (*cases M, fold top-nres-def*) **by** *auto*

lemma *nres-simp-internals*:
 $RES \{\} = SUCCEED$
 $FAILi = FAIL$
unfolding *top-nres-def bot-nres-def* **by** *simp-all*

lemma *nres-inequalities*[*simp*]:
 $FAIL \neq RES X$
 $FAIL \neq SUCCEED$
 $FAIL \neq RETURN x$
 $SUCCEED \neq FAIL$
 $SUCCEED \neq RETURN x$
 $RES X \neq FAIL$
 $RETURN x \neq FAIL$
 $RETURN x \neq SUCCEED$
unfolding *top-nres-def bot-nres-def RETURN-def*
by *auto*

lemma *nres-more-simps*[*simp*]:
 $SUCCEED = RES X \longleftrightarrow X=\{\}$
 $RES X = SUCCEED \longleftrightarrow X=\{\}$
 $RES X = RETURN x \longleftrightarrow X=\{x\}$
 $RES X = RES Y \longleftrightarrow X=Y$
 $RETURN x = RES X \longleftrightarrow \{x\}=X$
 $RETURN x = RETURN y \longleftrightarrow x=y$
unfolding *top-nres-def bot-nres-def RETURN-def* **by** *auto*

lemma *nres-order-simps*[*simp*]:
 $\bigwedge M. SUCCEED \leq M$
 $\bigwedge M. M \leq SUCCEED \longleftrightarrow M=SUCCEED$
 $\bigwedge M. M \leq FAIL$
 $\bigwedge M. FAIL \leq M \longleftrightarrow M=FAIL$
 $\bigwedge X Y. RES X \leq RES Y \longleftrightarrow X \leq Y$
 $\bigwedge X. Sup X = FAIL \longleftrightarrow FAIL \in X$
 $\bigwedge X f. Sup (f ' X) = FAIL \longleftrightarrow FAIL \in f ' X$


```

 $\bigwedge X. FAIL = Sup X \longleftrightarrow FAIL \in X$ 
 $\bigwedge X f. FAIL = Sup (f ' X) \longleftrightarrow FAIL \in f ' X$ 
 $\bigwedge X. FAIL \in X \implies Sup X = FAIL$ 
 $\bigwedge X. FAIL \in f ' X \implies Sup (f ' X) = FAIL$ 
 $\bigwedge A. Sup (RES ' A) = RES (Sup A)$ 
 $\bigwedge A. Sup (RES ' A) = RES (Sup A)$ 
 $\bigwedge A. A \neq \{\} \implies Inf (RES ' A) = RES (Inf A)$ 
 $\bigwedge A. A \neq \{\} \implies Inf (RES ' A) = RES (Inf A)$ 
 $Inf \{\} = FAIL$ 
 $Inf UNIV = SUCCEED$ 
 $Sup \{\} = SUCCEED$ 
 $Sup UNIV = FAIL$ 
 $\bigwedge x y. RETURN x \leq RETURN y \longleftrightarrow x = y$ 
 $\bigwedge x Y. RETURN x \leq RES Y \longleftrightarrow x \in Y$ 
 $\bigwedge X y. RES X \leq RETURN y \longleftrightarrow X \subseteq \{y\}$ 
unfolding Sup-nres-def Inf-nres-def RETURN-def
by (auto simp add: bot-unique top-unique nres-simp-internals)

```

lemma *Sup-eq-RESE*:

assumes $Sup A = RES B$

obtains C **where** $A = RES ' C$ **and** $B = Sup C$

proof –

show *?thesis*

using *assms* **unfolding** *Sup-nres-def*

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

apply (*rule-tac* $C = \{X. RES X \in A\}$ **in** *that*)

apply *auto* []

apply (*case-tac* x , *auto simp: nres-simp-internals*) []

apply (*auto simp: nres-simp-internals*) []

done

qed

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

Pointwise Reasoning

ML <

structure *refine-pw-simps* = *Named-Thms*

(*val* *name* = @{*binding* *refine-pw-simps*}

val *description* = *Refinement Framework: ^*
Simplifier rules for pointwise reasoning)

>

setup <*refine-pw-simps.setup*>

definition *nofail* $S \equiv S \neq FAIL$

definition *inres* $S x \equiv RETURN x \leq S$

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

nofail $FAIL \longleftrightarrow False$

```

nofail (RES X)  $\longleftrightarrow$  True
nofail (RETURN x)  $\longleftrightarrow$  True
nofail SUCCEED  $\longleftrightarrow$  True
unfolding nofail-def
by (simp-all add: RETURN-def)

```

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

```

inres FAIL = ( $\lambda$ -. True)
inres (RES X) = ( $\lambda$ x.  $x \in X$ )
inres (RETURN x) = ( $\lambda$ y.  $x=y$ )
inres SUCCEED = ( $\lambda$ -. False)
unfolding inres-def [abs-def]
by (auto simp add: RETURN-def)

```

lemma *not-nofail-iff*:

```

 $\neg$ nofail S  $\longleftrightarrow$  S=FAIL by (cases S) auto

```

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

```

 $\neg$ nofail S  $\implies$  inres S x
apply (cases S) by auto

```

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

```

S  $\neq$  FAIL  $\longleftrightarrow$  nofail S
FAIL  $\neq$  S  $\longleftrightarrow$  nofail S
by (cases S, simp-all)+

```

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

lemma *pw-le-iff*:

```

S  $\leq$  S'  $\longleftrightarrow$  (nofail S'  $\longrightarrow$  (nofail S  $\wedge$  ( $\forall$ x. inres S x  $\longrightarrow$  inres S' x)))
apply (cases S, simp-all)
apply (case-tac [!] S', auto)
done

```

lemma *pw-eq-iff*:

```

S=S'  $\longleftrightarrow$  (nofail S = nofail S'  $\wedge$  ( $\forall$ x. inres S x  $\longleftrightarrow$  inres S' x))
apply (rule iffI)
apply simp
apply (rule antisym)
apply (simp-all add: pw-le-iff)
done

```

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

```

( $\exists$ x. inres S x)  $\longrightarrow$  (nofail S  $\longleftrightarrow$  nofail S')  $\wedge$  ( $\forall$ x. inres S x  $\longleftrightarrow$  inres S' x)
by (auto simp : flat-ord-def pw-eq-iff)

```

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

```

(nofail S)  $\longrightarrow$  nofail S'  $\wedge$  ( $\forall$ x. inres S x  $\longleftrightarrow$  inres S' x)
apply (simp add: flat-ord-def pw-eq-iff) apply safe

```

```

apply simp
apply simp
apply simp
apply (rule ccontr)
apply simp
done

```

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

lemma *pw-leI*:
 $(\text{nofail } S' \longrightarrow (\text{nofail } S \wedge (\forall x. \text{inres } S \ x \longrightarrow \text{inres } S' \ x))) \Longrightarrow S \leq S'$
by (*simp add: pw-le-iff*)

lemma *pw-leI'*:
assumes $\text{nofail } S' \Longrightarrow \text{nofail } S$
assumes $\bigwedge x. \llbracket \text{nofail } S'; \text{inres } S \ x \rrbracket \Longrightarrow \text{inres } S' \ x$
shows $S \leq S'$
using *assms*
by (*simp add: pw-le-iff*)

lemma *pw-eqI*:
assumes $\text{nofail } S = \text{nofail } S'$
assumes $\bigwedge x. \text{inres } S \ x \longleftrightarrow \text{inres } S' \ x$
shows $S = S'$
using *assms* **by** (*simp add: pw-eq-iff*)

lemma *pwD1*:
assumes $S \leq S' \quad \text{nofail } S'$
shows $\text{nofail } S$
using *assms* **by** (*simp add: pw-le-iff*)

lemma *pwD2*:
assumes $S \leq S' \quad \text{inres } S \ x$
shows $\text{inres } S' \ x$
using *assms*
by (*auto simp add: pw-le-iff*)

lemmas *pwD* = *pwD1* *pwD2*

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

lemma *le-nofailI*: $\llbracket \text{nofail } M' \Longrightarrow M \leq M' \rrbracket \Longrightarrow M \leq M'$
by (*cases M'*) *auto*

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]*:
 $\text{nofail } (\text{sup } a \ b) \longleftrightarrow \text{nofail } a \wedge \text{nofail } b$
apply (*cases a, simp*)

apply (*cases b, simp-all*)
done

lemma *pw-sup-inres*[*refine-pw-simps*]:
 $inres (sup a b) x \longleftrightarrow inres a x \vee inres b x$
apply (*cases a, simp*)
apply (*cases b, simp*)
apply (*simp*)
done

lemma *pw-Sup-inres*[*refine-pw-simps*]: $inres (Sup X) r \longleftrightarrow (\exists M \in X. inres M r)$
apply (*cases Sup X*)
apply (*simp*)
apply (*erule beXI[rotated]*)
apply *simp*
apply (*erule Sup-eq-RESE*)
apply (*simp*)
done

lemma *pw-SUP-inres* [*refine-pw-simps*]: $inres (Sup (f ' X)) r \longleftrightarrow (\exists M \in X. inres (f M) r)$
using *pw-Sup-inres [of f ' X]* **by** *simp*

lemma *pw-Sup-nofail*[*refine-pw-simps*]: $nofail (Sup X) \longleftrightarrow (\forall x \in X. nofail x)$
apply (*cases Sup X*)
apply *force*
apply *simp*
apply (*erule Sup-eq-RESE*)
apply *auto*
done

lemma *pw-SUP-nofail* [*refine-pw-simps*]: $nofail (Sup (f ' X)) \longleftrightarrow (\forall x \in X. nofail (f x))$
using *pw-Sup-nofail [of f ' X]* **by** *simp*

lemma *pw-inf-nofail*[*refine-pw-simps*]:
 $nofail (inf a b) \longleftrightarrow nofail a \vee nofail b$
apply (*cases a, simp*)
apply (*cases b, simp-all*)
done

lemma *pw-inf-inres*[*refine-pw-simps*]:
 $inres (inf a b) x \longleftrightarrow inres a x \wedge inres b x$
apply (*cases a, simp*)
apply (*cases b, simp*)
apply (*simp*)
done

lemma *pw-Inf-nofail*[*refine-pw-simps*]: $nofail (Inf C) \longleftrightarrow (\exists x \in C. nofail x)$

```

apply (cases C={})
apply simp
apply (cases Inf C)
apply (subgoal-tac C={FAIL})
apply simp
apply auto []
apply (subgoal-tac C≠{FAIL})
apply (auto simp: not-nofail-iff) []
apply auto []
done

```

lemma *pw-INF-nofail* [*refine-pw-simps*]: $\text{nofail } (\text{Inf } (f \text{ ' } C)) \longleftrightarrow (\exists x \in C. \text{nofail } (f \ x))$
using *pw-Inf-nofail* [*of f ' C*] **by** *simp*

lemma *pw-Inf-inres* [*refine-pw-simps*]: $\text{inres } (\text{Inf } C) \ r \longleftrightarrow (\forall M \in C. \text{inres } M \ r)$
apply (*unfold Inf-nres-def*)
apply *auto*
apply (*case-tac M*)
apply *force*
apply *force*
apply (*case-tac M*)
apply *force*
apply *force*
done

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

lemma *nofail-RES-conv*: $\text{nofail } m \longleftrightarrow (\exists M. m = \text{RES } M)$ **by** (*cases m*) *auto*

primrec *the-RES* **where** *the-RES* (*RES X*) = *X*

lemma *the-RES-inv* [*simp*]: $\text{nofail } m \implies \text{RES } (\text{the-RES } m) = m$
by (*cases m*) *auto*

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

lemma *nf-inres-RES* [*simp*]: $\text{nf-inres } (\text{RES } X) \ x \longleftrightarrow x \in X$
by (*simp add: refine-pw-simps*)

lemma *nf-inres-SPEC* [*simp*]: $\text{nf-inres } (\text{SPEC } \Phi) \ x \longleftrightarrow \Phi \ x$
by (*simp add: refine-pw-simps*)

lemma *nofail-antimono-fun*: $f \leq g \implies (\text{nofail } (g \ x) \longrightarrow \text{nofail } (f \ x))$
by (*auto simp: pw-le-iff dest: le-funD*)

Monad Operators

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

lemma *bind-FAIL[simp]*: $\text{bind } \text{FAIL } f = \text{FAIL}$
unfolding *bind-def* **by** (*auto split: nres.split*)

lemma *bind-SUCCEED[simp]*: $\text{bind } \text{SUCCEED } f = \text{SUCCEED}$
unfolding *bind-def* **by** (*auto split: nres.split*)

lemma *bind-RES*: $\text{bind } (\text{RES } X) f = \text{Sup } (f'X)$ **unfolding** *bind-def*
by (*auto*)

adhoc-overloading

Monad-Syntax.bind Refine-Basic.bind

lemma *pw-bind-nofail[refine-pw-simps]*:
 $\text{nofail } (\text{bind } M f) \longleftrightarrow (\text{nofail } M \wedge (\forall x. \text{inres } M x \longrightarrow \text{nofail } (f x)))$
apply (*cases M*)
by (*auto simp: bind-RES refine-pw-simps*)

lemma *pw-bind-inres[refine-pw-simps]*:
 $\text{inres } (\text{bind } M f) = (\lambda x. \text{nofail } M \longrightarrow (\exists y. (\text{inres } M y \wedge \text{inres } (f y) x)))$
apply (*rule ext*)
apply (*cases M*)
apply (*auto simp add: bind-RES refine-pw-simps*)
done

lemma *pw-bind-le-iff*:
 $\text{bind } M f \leq S \longleftrightarrow (\text{nofail } S \longrightarrow \text{nofail } M) \wedge$
 $(\forall x. \text{nofail } M \wedge \text{inres } M x \longrightarrow f x \leq S)$
by (*auto simp: pw-le-iff refine-pw-simps*)

lemma *pw-bind-leI*: \llbracket
 $\text{nofail } S \implies \text{nofail } M; \bigwedge x. \llbracket \text{nofail } M; \text{inres } M x \rrbracket \implies f x \leq S \rrbracket$
 $\implies \text{bind } M f \leq S$
by (*simp add: pw-bind-le-iff*)

lemma *nres-monad1[simp]*: $\text{bind } (\text{RETURN } x) f = f x$
by (*rule pw-eqI*) (*auto simp: refine-pw-simps*)

lemma *nres-monad2[simp]*: $\text{bind } M \text{ RETURN} = M$
by (*rule pw-eqI*) (*auto simp: refine-pw-simps*)

lemma *nres-monad3[simp]*: $\text{bind } (\text{bind } M f) g = \text{bind } M (\lambda x. \text{bind } (f x) g)$
by (*rule pw-eqI*) (*auto simp: refine-pw-simps*)

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

lemma *bind-cong*:
assumes $m=m'$
assumes $\bigwedge x. \text{RETURN } x \leq m' \implies f x = f' x$
shows $\text{bind } m f = \text{bind } m' f'$
using *assms*
by (*auto simp: refine-pw-simps pw-eq-iff pw-le-iff*)

lemma *bind-mono[refine-mono]*:
 $\llbracket M \leq M'; \bigwedge x. \text{RETURN } x \leq M \implies f x \leq f' x \rrbracket \implies \text{bind } M f \leq \text{bind } M' f'$
 $\llbracket \text{flat-ge } M M'; \bigwedge x. \text{flat-ge } (f x) (f' x) \rrbracket \implies \text{flat-ge } (\text{bind } M f) (\text{bind } M' f')$
apply (*auto simp: refine-pw-simps pw-ords-iff*) \square
apply (*auto simp: refine-pw-simps pw-ords-iff*) \square
done

lemma *bind-mono1[simp, intro!]*: *mono* ($\lambda M. \text{bind } M f$)
apply (*rule monoI*)
apply (*rule bind-mono*)
by *auto*

lemma *bind-mono1'[simp, intro!]*: *mono* *bind*
apply (*rule monoI*)
apply (*rule le-funI*)
apply (*rule bind-mono*)
by *auto*

lemma *bind-mono2'[simp, intro!]*: *mono* (*bind* *M*)
apply (*rule monoI*)
apply (*rule bind-mono*)
by (*auto dest: le-funD*)

lemma *bind-distrib-sup1*: $\text{bind } (\text{sup } M N) f = \text{sup } (\text{bind } M f) (\text{bind } N f)$
by (*auto simp add: pw-eq-iff refine-pw-simps*)

lemma *bind-distrib-sup2*: $\text{bind } m (\lambda x. \text{sup } (f x) (g x)) = \text{sup } (\text{bind } m f) (\text{bind } m g)$
by (*auto simp: pw-eq-iff refine-pw-simps*)

lemma *bind-distrib-Sup1*: $\text{bind } (\text{Sup } M) f = (\text{SUP } m \in M. \text{bind } m f)$
by (*auto simp: pw-eq-iff refine-pw-simps*)

lemma *bind-distrib-Sup2*: $F \neq \{\}$ $\implies \text{bind } m (\text{Sup } F) = (\text{SUP } f \in F. \text{bind } m f)$
by (*auto simp: pw-eq-iff refine-pw-simps*)

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

by (rule pw-eqI) (auto simp add: refine-pw-simps)

2.6.2 VCG Setup

lemma SPEC-cons-rule:

assumes $m \leq \text{SPEC } \Phi$

assumes $\bigwedge x. \Phi x \implies \Psi x$

shows $m \leq \text{SPEC } \Psi$

using assms by (auto simp: pw-le-iff)

lemmas SPEC-trans = order-trans[where z=SPEC Postcond for Postcond, zero-var-indices]

ML ‹

structure Refine = struct

structure vcg = Named-Thms

(val name = @{binding refine-vcg}

val description = Refinement Framework: $\hat{\quad}$

Verification condition generation rules (intro))

structure vcg-cons = Named-Thms

(val name = @{binding refine-vcg-cons}

val description = Refinement Framework: $\hat{\quad}$

Consequence rules tried by VCG)

structure refine0 = Named-Thms

(val name = @{binding refine0}

val description = Refinement Framework: $\hat{\quad}$

Refinement rules applied first (intro))

structure refine = Named-Thms

(val name = @{binding refine}

val description = Refinement Framework: Refinement rules (intro))

structure refine2 = Named-Thms

(val name = @{binding refine2}

val description = Refinement Framework: $\hat{\quad}$

Refinement rules 2nd stage (intro))

(* If set to true, the product splitter of refine-rcg is disabled. *)

val no-prod-split =

Attrib.setup-config-bool @{binding refine-no-prod-split} (K false);

fun rcg-tac add-thms ctxt =

let

val cons-thms = vcg-cons.get ctxt

val ref-thms = (refine0.get ctxt

@ add-thms @ refine.get ctxt @ refine2.get ctxt);

val prod-ss = (Splitter.add-split @{thm prod.split}


```

    (put-simpset HOL-basic-ss ctxt));
  val prod-simp-tac =
    if Config.get ctxt no-prod-split then
      K no-tac
    else
      (simp-tac prod-ss THEN'
        REPEAT-ALL-NEW (resolve-tac ctxt @ {thms impI all}));
  in
    REPEAT-ALL-NEW-FWD (DETERM o FIRST' [
      resolve-tac ctxt ref-thms,
      resolve-tac ctxt cons-thms THEN' resolve-tac ctxt ref-thms,
      prod-simp-tac
    ])
  end;

  fun post-tac ctxt = REPEAT-ALL-NEW-FWD (FIRST' [
    eq-assume-tac,
    (*match-tac ctxt thms,*)
    SOLVED' (Tagged-Solver.solve-tac ctxt)])

end;
>
setup <Refine.vcg.setup>
setup <Refine.vcg-cons.setup>
setup <Refine.refine0.setup>
setup <Refine.refine.setup>
setup <Refine.refine2.setup>

method-setup refine-rcg =
  <Attrib.thms >> (fn add-thms => fn ctxt => SIMPLE-METHOD' (
    Refine.rcg-tac add-thms ctxt THEN-ALL-NEW-FWD (TRY o Refine.post-tac
    ctxt)
  ))>
  Refinement framework: Generate refinement conditions

method-setup refine-vcg =
  <Attrib.thms >> (fn add-thms => fn ctxt => SIMPLE-METHOD' (
    Refine.rcg-tac (add-thms @ Refine.vcg.get ctxt) ctxt THEN-ALL-NEW-FWD
    (TRY o Refine.post-tac ctxt)
  ))>
  Refinement framework: Generate refinement and verification conditions

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**

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

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

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

lemma

$$\begin{aligned} \text{conc-fun-FAIL}[\text{simp}]: & \Downarrow R \ \text{FAIL} = \text{FAIL} \ \mathbf{and} \\ \text{conc-fun-RES}: & \Downarrow R \ (\text{RES } X) = \text{RES } (R^{-1} \text{``} X) \\ \mathbf{unfolding} & \ \text{conc-fun-def} \ \mathbf{by} \ (\text{auto split: nres.split}) \end{aligned}$$

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

$$\begin{aligned} \Uparrow R \ \text{FAIL} &= \text{FAIL} \\ X \subseteq \text{Domain } R &\Longrightarrow \Uparrow R \ (\text{RES } X) = \text{RES } (R \text{``} X) \\ \neg(X \subseteq \text{Domain } R) &\Longrightarrow \Uparrow R \ (\text{RES } X) = \text{FAIL} \\ \mathbf{unfolding} & \ \text{abs-fun-def} \ \mathbf{by} \ (\text{auto split: nres.split}) \end{aligned}$$

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

lemma *conc-abs-swap*: $m' \leq \Downarrow R m \longleftrightarrow \Uparrow R m' \leq m$

unfolding *conc-fun-def* *abs-fun-def* **using** SV

by (*auto split: nres.split*)

(*metis ImageE converseD single-valuedD subsetD*)

lemma *ac-galois*: *galois-connection* ($\Uparrow R$) ($\Downarrow R$)

apply (*unfold-locales*)

by (*rule conc-abs-swap*)

end

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

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

apply (*cases* M)

apply *simp*
apply (*auto simp: abs-fun-simps abs-fun-def*)
done

lemma *pw-abs-inres[refine-pw-simps]*:
 $inres (\uparrow R M) a \longleftrightarrow (nofail (\uparrow R M) \longrightarrow (\exists c. inres M c \wedge (c,a) \in R))$
apply (*cases M*)
apply *simp*
apply (*auto simp: abs-fun-def*)
done

lemma *pw-conc-nofail[refine-pw-simps]*:
 $nofail (\Downarrow R S) = nofail S$
by (*cases S*) (*auto simp: conc-fun-RES*)

lemma *pw-conc-inres[refine-pw-simps]*:
 $inres (\Downarrow R S') = (\lambda s. nofail S' \longrightarrow (\exists s'. (s,s') \in R \wedge inres S' s'))$
apply (*rule ext*)
apply (*cases S'*)
apply (*auto simp: conc-fun-RES*)
done

lemma *abs-fun-strict[simp]*:
 $\uparrow R SUCCEED = SUCCEED$
unfolding *abs-fun-def* **by** (*auto split: nres.split*)

lemma *conc-fun-strict[simp]*:
 $\Downarrow R SUCCEED = SUCCEED$
unfolding *conc-fun-def* **by** (*auto split: nres.split*)

lemma *conc-fun-mono[simp, intro!]*: $mono (\Downarrow R)$
by *rule* (*auto simp: pw-le-iff refine-pw-simps*)

lemma *abs-fun-mono[simp, intro!]*: $mono (\uparrow R)$
by *rule* (*auto simp: pw-le-iff refine-pw-simps*)

lemma *conc-fun-R-mono*:
assumes $R \subseteq R'$
shows $\Downarrow R M \leq \Downarrow R' M$
using *assms*
by (*auto simp: pw-le-iff refine-pw-simps*)

lemma *conc-fun-chain*: $\Downarrow R (\Downarrow S M) = \Downarrow (R \circ S) M$
unfolding *conc-fun-def*
by (*auto split: nres.split*)

lemma *conc-Id[simp]*: $\Downarrow Id = id$
unfolding *conc-fun-def [abs-def]* **by** (*auto split: nres.split*)

lemma *abs-Id*[*simp*]: $\uparrow Id = id$
unfolding *abs-fun-def* [*abs-def*] **by** (*auto split: nres.split*)

lemma *conc-fun-fail-iff*[*simp*]:
 $\Downarrow R S = FAIL \longleftrightarrow S = FAIL$
 $FAIL = \Downarrow R S \longleftrightarrow S = FAIL$
by (*auto simp add: pw-eq-iff refine-pw-simps*)

lemma *conc-trans*[*trans*]:
assumes $A: C \leq \Downarrow R B$ **and** $B: B \leq \Downarrow R' A$
shows $C \leq \Downarrow R (\Downarrow R' A)$
using *assms* **by** (*fastforce simp: pw-le-iff refine-pw-simps*)

lemma *abs-trans*[*trans*]:
assumes $A: \uparrow R C \leq B$ **and** $B: \uparrow R' B \leq A$
shows $\uparrow R' (\uparrow R C) \leq A$
using *assms* **by** (*fastforce simp: pw-le-iff refine-pw-simps*)

Transitivity Reasoner Setup

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

lemma *conc-trans-additional*[*trans*]:
 $\bigwedge A B C. A \leq \Downarrow R B \implies B \leq C \implies A \leq \Downarrow R C$
 $\bigwedge A B C. A \leq \Downarrow Id B \implies B \leq \Downarrow R C \implies A \leq \Downarrow R C$
 $\bigwedge A B C. A \leq \Downarrow R B \implies B \leq \Downarrow Id C \implies A \leq \Downarrow R C$
 $\bigwedge A B C. A \leq \Downarrow Id B \implies B \leq \Downarrow Id C \implies A \leq C$
 $\bigwedge A B C. A \leq \Downarrow Id B \implies B \leq C \implies A \leq C$
 $\bigwedge A B C. A \leq B \implies B \leq \Downarrow Id C \implies A \leq C$
using *conc-trans*[**where** $R=R$ **and** $R'=Id$]
by (*auto intro: order-trans*)

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

lemma *abs-trans-additional*[*trans*]:
 $\bigwedge A B C. \llbracket A \leq B; \uparrow R B \leq C \rrbracket \implies \uparrow R A \leq C$
 $\bigwedge A B C. \llbracket \uparrow Id A \leq B; \uparrow R B \leq C \rrbracket \implies \uparrow R A \leq C$
 $\bigwedge A B C. \llbracket \uparrow R A \leq B; \uparrow Id B \leq C \rrbracket \implies \uparrow R A \leq C$
 $\bigwedge A B C. \llbracket \uparrow Id A \leq B; \uparrow Id B \leq C \rrbracket \implies A \leq C$
 $\bigwedge A B C. \llbracket \uparrow Id A \leq B; B \leq C \rrbracket \implies A \leq C$
 $\bigwedge A B C. \llbracket A \leq B; \uparrow Id B \leq C \rrbracket \implies A \leq C$
apply (*auto simp: refine-pw-simps pw-le-iff*)
apply *fastforce+*
done

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*

apply *unfold-locales*

by (*simp-all add: ASSERT-def ASSUME-def*)

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 Φ

inres (*ASSERT* Φ) x

by (*cases* Φ , *simp-all*)⁺

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

nofail (*ASSUME* Φ)

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

by (*cases* Φ , *simp-all*)⁺

Recursion

lemma *pw-REC-nofail*:

shows *nofail* (*REC* $B\ x$) \longleftrightarrow *trimono* $B \wedge$

($\exists F. (\forall x.$

nofail ($F\ x$) \longrightarrow *nofail* ($B\ F\ x$)

$\wedge (\forall x'. \textit{inres}$ ($B\ F\ x$) $x' \longrightarrow$ *inres* ($F\ x$) x')

) \wedge *nofail* ($F\ x$)

proof –

have *nofail* (*REC* $B\ x$) \longleftrightarrow *trimono* $B \wedge$

($\exists F. (\forall x. B\ F\ x \leq F\ x) \wedge$ *nofail* ($F\ x$))

unfolding *REC-def lfp-def*

apply (*auto simp: refine-pw-simps intro: le-funI dest: le-funD*)

done

thus *?thesis*

unfolding *pw-le-iff* .

qed

lemma *pw-REC-inres*:

$$\begin{aligned} \text{inres } (\text{REC } B \ x) \ x' &= (\text{trimono } B \longrightarrow \\ &(\forall F. (\forall x''. \\ &\text{nofail } (F \ x'') \longrightarrow \text{nofail } (B \ F \ x'') \\ &\wedge (\forall x. \text{inres } (B \ F \ x'') \ x \longrightarrow \text{inres } (F \ x'') \ x)) \\ &\longrightarrow \text{inres } (F \ x) \ x')) \end{aligned}$$

proof –

$$\begin{aligned} \text{have } \text{inres } (\text{REC } B \ x) \ x' \\ \longleftrightarrow (\text{trimono } B \longrightarrow (\forall F. (\forall x''. B \ F \ x'' \leq F \ x'') \longrightarrow \text{inres } (F \ x) \ x')) \\ \text{unfolding } \text{REC-def } \text{lfp-def} \\ \text{by } (\text{auto simp: refine-pw-simps intro: le-funI dest: le-funD}) \\ \text{thus ?thesis unfolding pw-le-iff .} \end{aligned}$$

qed

lemmas *pw-REC = pw-REC-inres pw-REC-nofail*

lemma *pw-RECT-nofail*:

$$\begin{aligned} \text{shows } \text{nofail } (\text{RECT } B \ x) \longleftrightarrow \text{trimono } B \wedge \\ (\forall F. (\forall y. \text{nofail } (B \ F \ y) \longrightarrow \\ \text{nofail } (F \ y) \wedge (\forall x. \text{inres } (F \ y) \ x \longrightarrow \text{inres } (B \ F \ y) \ x)) \longrightarrow \\ \text{nofail } (F \ x)) \end{aligned}$$

proof –

$$\begin{aligned} \text{have } \text{nofail } (\text{RECT } B \ x) \longleftrightarrow (\text{trimono } B \wedge (\forall F. (\forall y. F \ y \leq B \ F \ y) \longrightarrow \text{nofail} \\ (F \ x))) \\ \text{unfolding } \text{RECT-gfp-def } \text{gfp-def} \\ \text{by } (\text{auto simp: refine-pw-simps intro: le-funI dest: le-funD}) \\ \text{thus ?thesis} \\ \text{unfolding pw-le-iff .} \end{aligned}$$

qed

lemma *pw-RECT-inres*:

$$\begin{aligned} \text{shows } \text{inres } (\text{RECT } B \ x) \ x' &= (\text{trimono } B \longrightarrow \\ &(\exists M. (\forall y. \text{nofail } (B \ M \ y) \longrightarrow \\ &\text{nofail } (M \ y) \wedge (\forall x. \text{inres } (M \ y) \ x \longrightarrow \text{inres } (B \ M \ y) \ x)) \wedge \\ &\text{inres } (M \ x) \ x')) \end{aligned}$$

proof –

$$\begin{aligned} \text{have } \text{inres } (\text{RECT } B \ x) \ x' \longleftrightarrow \text{trimono } B \longrightarrow (\exists M. (\forall y. M \ y \leq B \ M \ y) \wedge \\ \text{inres } (M \ x) \ x') \\ \text{unfolding } \text{RECT-gfp-def } \text{gfp-def} \\ \text{by } (\text{auto simp: refine-pw-simps intro: le-funI dest: le-funD}) \\ \text{thus ?thesis unfolding pw-le-iff .} \end{aligned}$$

qed

lemmas *pw-RECT = pw-RECT-inres 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*]:

$m \leq \text{SPEC } (\lambda\cdot. \text{True}) \implies m \leq \text{RES UNIV } \mathbf{by} \text{ auto}$

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

by (*auto simp: RETURN-def*)

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

by *auto*

lemma *SUCCEED-rule* [*refine-vcg*]: $\text{SUCCEED} \leq \text{SPEC } \Phi$ **by** *auto*

lemma *FAIL-rule*: $\text{False} \implies \text{FAIL} \leq \text{SPEC } \Phi$ **by** *auto*

lemma *SPEC-rule* [*refine-vcg*]: $\llbracket \bigwedge x. \Phi x \implies \Phi' x \rrbracket \implies \text{SPEC } \Phi \leq \text{SPEC } \Phi'$ **by** *auto*

lemma *RETURN-to-SPEC-rule* [*refine-vcg*]: $m \leq \text{SPEC } ((=) v) \implies m \leq \text{RETURN } v$

by (*simp add: pw-le-iff refine-pw-simps*)

lemma *Sup-img-rule-complete*:

$(\forall x. x \in S \longrightarrow f x \leq \text{SPEC } \Phi) \longleftrightarrow \text{Sup } (f'S) \leq \text{SPEC } \Phi$

apply *rule*

apply (*rule pw-leI*)

apply (*auto simp: pw-le-iff refine-pw-simps*) \square

apply (*intro allI impI*)

apply (*rule pw-leI*)

apply (*auto simp: pw-le-iff refine-pw-simps*) \square

done

lemma *SUP-img-rule-complete*:

$(\forall x. x \in S \longrightarrow f x \leq \text{SPEC } \Phi) \longleftrightarrow \text{Sup } (f'S) \leq \text{SPEC } \Phi$

using *Sup-img-rule-complete* [*of S f*] **by** *simp*

lemma *Sup-img-rule* [*refine-vcg*]:

$\llbracket \bigwedge x. x \in S \implies f x \leq \text{SPEC } \Phi \rrbracket \implies \text{Sup}(f'S) \leq \text{SPEC } \Phi$

by (*auto simp: SUP-img-rule-complete[symmetric]*)

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

lemma *bind-rule-complete*: $\text{bind } M f \leq \text{SPEC } \Phi \longleftrightarrow M \leq \text{SPEC } (\lambda x. f x \leq \text{SPEC } \Phi)$

by (*auto simp: pw-le-iff refine-pw-simps*)

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

$\llbracket M \leq \text{SPEC } (\lambda x. f x \leq \text{SPEC } \Phi) \rrbracket \implies \text{bind } M (\lambda x. f x) \leq \text{SPEC } \Phi$

— Note: η -expanded version helps Isabelle's unification to keep meaningful variable names from the program

by (*auto simp: bind-rule-complete*)

lemma *ASSUME-rule* [*refine-vcg*]: $\llbracket \Phi \implies \Psi () \rrbracket \implies \text{ASSUME } \Phi \leq \text{SPEC } \Psi$

by (*cases* Φ) *auto*

lemma *ASSERT-rule[refine-vcg]*: $\llbracket \Phi; \Phi \implies \Psi \ () \rrbracket \implies \text{ASSERT } \Phi \leq \text{SPEC } \Psi$ **by** *auto*

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

$\llbracket \bigwedge a \ b. \ p=(a,b) \implies S \ a \ b \leq \text{SPEC } \Phi \rrbracket \implies \text{case-prod } S \ p \leq \text{SPEC } \Phi$
by (*auto split: prod.split*)

lemma *prod2-rule[refine-vcg]*:

assumes $\bigwedge a \ b \ c \ d. \llbracket ab=(a,b); \ cd=(c,d) \rrbracket \implies f \ a \ b \ c \ d \leq \text{SPEC } \Phi$
shows $(\lambda(a,b) \ (c,d). \ f \ a \ b \ c \ d) \ ab \ cd \leq \text{SPEC } \Phi$
using *assms*
by (*auto split: prod.split*)

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

$\llbracket b \implies S1 \leq \text{SPEC } \Phi; \neg b \implies S2 \leq \text{SPEC } \Phi \rrbracket$
 $\implies (\text{if } b \ \text{then } S1 \ \text{else } S2) \leq \text{SPEC } \Phi$
by (*auto*)

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

$\llbracket v=\text{None} \implies S1 \leq \text{SPEC } \Phi; \bigwedge x. \ v=\text{Some } x \implies f2 \ x \leq \text{SPEC } \Phi \rrbracket$
 $\implies \text{case-option } S1 \ f2 \ v \leq \text{SPEC } \Phi$
by (*auto split: option.split*)

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

$f \ x \leq \text{SPEC } \Phi \implies \text{Let } x \ f \leq \text{SPEC } \Phi$ **by** *auto*

lemma *Let-rule'*:

assumes $\bigwedge x. \ x=v \implies f \ x \leq \text{SPEC } \Phi$
shows $\text{Let } v \ (\lambda x. \ f \ x) \leq \text{SPEC } \Phi$
using *assms* **by** *simp*

lemma *REC-le-rule*:

assumes *M: trimono body*
assumes *I0: (x,x')∈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 *REC body x ≤ M x'*
by (*rule REC-rule-arb[OF M, where pre= $\lambda x' \ x. \ (x,x') \in R$, OF I0 IS]*)

Proving Monotonicity

lemma *nr-mono-bind*:


```

assumes  $MA: \text{mono } A$  and  $MB: \bigwedge s. \text{mono } (B s)$ 
shows  $\text{mono } (\lambda F s. \text{bind } (A F s) (\lambda s'. B s F s'))$ 
apply (rule monoI)
apply (rule le-funI)
apply (rule bind-mono)
apply (auto dest: monoD[OF MA, THEN le-funD]) []
apply (auto dest: monoD[OF MB, THEN le-funD]) []
done

```

```

lemma nr-mono-bind':  $\text{mono } (\lambda F s. \text{bind } (f s) F)$ 
apply rule
apply (rule le-funI)
apply (rule bind-mono)
apply (auto dest: le-funD)
done

```

lemmas *nr-mono = 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*]: $S \leq \Downarrow Id S$ **by** *auto*

lemma *RES-refine*:
 $\llbracket \bigwedge s. s \in S \implies \exists s' \in S'. (s, s') \in R \rrbracket \implies RES S \leq \Downarrow R (RES S')$
by (*auto simp: conc-fun-RES*)

lemma *SPEC-refine*:
assumes $S \leq SPEC (\lambda x. \exists x'. (x, x') \in R \wedge \Phi x')$
shows $S \leq \Downarrow R (SPEC \Phi)$
using *assms*
by (*force simp: pw-le-iff refine-pw-simps*)

lemma *Id-SPEC-refine*[*refine*]:
 $S \leq SPEC \Phi \implies S \leq \Downarrow Id (SPEC \Phi)$ **by** *simp*

lemma *RETURN-refine*[*refine*]:
assumes $(x, x') \in R$
shows $RETURN x \leq \Downarrow R (RETURN x')$
using *assms*
by (*auto simp: RETURN-def conc-fun-RES*)

lemma *RETURN-SPEC-refine*:
assumes $\exists x'. (x, x') \in R \wedge \Phi x'$
shows *RETURN* $x \leq \Downarrow R$ (*SPEC* Φ)
using *assms*
by (*auto simp: pw-le-iff refine-pw-simps*)

lemma *FAIL-refine*[*refine*]: $X \leq \Downarrow R$ *FAIL* **by** *auto*
lemma *SUCCEED-refine*[*refine*]: $SUCCEED \leq \Downarrow R$ X' **by** *auto*

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* $a b$)
using *assms* **by** (*auto simp: pw-le-iff refine-pw-simps*)

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

lemma *bind-refine'*:
fixes $R' :: ('a \times 'b)$ *set* **and** $R :: ('c \times 'd)$ *set*
assumes $R1: M \leq \Downarrow R' M'$
assumes $R2: \bigwedge x x'. \llbracket (x, x') \in R'; \text{inres } M x; \text{inres } M' x';$
 $\text{nofail } M; \text{nofail } M' \rrbracket$
 $\llbracket \implies f x \leq \Downarrow R (f' x') \rrbracket$
shows *bind* $M (\lambda x. f x) \leq \Downarrow R$ (*bind* $M' (\lambda x'. f' x')$)
using *assms*
apply (*simp add: pw-le-iff refine-pw-simps*)
apply *fast*
done

lemma *bind-refine*[*refine*]:
fixes $R' :: ('a \times 'b)$ *set* **and** $R :: ('c \times 'd)$ *set*
assumes $R1: M \leq \Downarrow R' M'$
assumes $R2: \bigwedge x x'. \llbracket (x, x') \in R' \rrbracket$
 $\implies f x \leq \Downarrow R (f' x')$
shows *bind* $M (\lambda x. f x) \leq \Downarrow R$ (*bind* $M' (\lambda x'. f' x')$)
apply (*rule bind-refine'*) **using** *assms* **by** *auto*

lemma *bind-refine-abs'*:
fixes $R' :: ('a \times 'b)$ *set* **and** $R :: ('c \times 'd)$ *set*
assumes $R1: M \leq \Downarrow R' M'$
assumes $R2: \bigwedge x x'. \llbracket (x, x') \in R'; \text{nf-inres } M' x' \rrbracket$
 $\llbracket \implies f x \leq \Downarrow R (f' x') \rrbracket$
shows *bind* $M (\lambda x. f x) \leq \Downarrow R$ (*bind* $M' (\lambda x'. f' x')$)
using *assms*
apply (*simp add: pw-le-iff refine-pw-simps*)
apply *blast*
done

Special cases for refinement of binding to *RES* statements

lemma *bind-refine-RES*:

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

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

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

by (*auto intro!*: *bind-refine'*)

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

declare *bind-refine-RES(3)[refine]*

lemma *ASSERT-refine[refine]*:

$$\begin{aligned} & \llbracket \Phi' \implies \Phi \rrbracket \implies ASSERT \Phi \leq \Downarrow Id (ASSERT \Phi') \\ & \text{by } (cases \Phi') \text{ auto} \end{aligned}$$

lemma *ASSUME-refine[refine]*:

$$\begin{aligned} & \llbracket \Phi \implies \Phi' \rrbracket \implies ASSUME \Phi \leq \Downarrow Id (ASSUME \Phi') \\ & \text{by } (cases \Phi) \text{ auto} \end{aligned}$$

Assertions and assumptions are treated specially in bindings

lemma *ASSERT-refine-right*:

$$\begin{aligned} & \text{assumes } \Phi \implies S \leq \Downarrow R S' \\ & \text{shows } S \leq \Downarrow R (do \{ASSERT \Phi; S'\}) \\ & \text{using } \textit{assms} \text{ by } (cases \Phi) \text{ auto} \end{aligned}$$

lemma *ASSERT-refine-right-pres*:

$$\begin{aligned} & \text{assumes } \Phi \implies S \leq \Downarrow R (do \{ASSERT \Phi; S'\}) \\ & \text{shows } S \leq \Downarrow R (do \{ASSERT \Phi; S'\}) \\ & \text{using } \textit{assms} \text{ by } (cases \Phi) \text{ auto} \end{aligned}$$

lemma *ASSERT-refine-left*:

$$\begin{aligned} & \text{assumes } \Phi \\ & \text{assumes } \Phi \implies S \leq \Downarrow R S' \\ & \text{shows } do\{ASSERT \Phi; S\} \leq \Downarrow R S' \\ & \text{using } \textit{assms} \text{ by } (cases \Phi) \text{ auto} \end{aligned}$$

lemma *ASSUME-refine-right*:

$$\begin{aligned} & \text{assumes } \Phi \\ & \text{assumes } \Phi \implies S \leq \Downarrow R S' \\ & \text{shows } S \leq \Downarrow R (do \{ASSUME \Phi; S'\}) \\ & \text{using } \textit{assms} \text{ by } (cases \Phi) \text{ auto} \end{aligned}$$

lemma *ASSUME-refine-left*:
assumes $\Phi \implies S \leq \Downarrow R S'$
shows $do \{ASSUME \Phi; S\} \leq \Downarrow R S'$
using *assms* **by** (*cases* Φ) *auto*

lemma *ASSUME-refine-left-pres*:
assumes $\Phi \implies do \{ASSUME \Phi; S\} \leq \Downarrow R S'$
shows $do \{ASSUME \Phi; S\} \leq \Downarrow R S'$
using *assms* **by** (*cases* Φ) *auto*

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-rcg*.

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* $R \Phi s \equiv \exists s'. (s, s') \in R \wedge \Phi s'$

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

unfolding *lift-assn-def* **by** *auto*

lemma *REC-refine[refine]*:
assumes M : *trimono body*
assumes $R0$: $(x, x') \in R$
assumes RS : $\bigwedge f f' x x'. \llbracket \bigwedge x x'. (x, x') \in R \implies f x \leq \Downarrow S (f' x'); (x, x') \in R; REC \text{ body}' = f' \rrbracket$
 $\implies \text{body } f x \leq \Downarrow S (\text{body}' f' x')$
shows $REC (\lambda f x. \text{body } f x) x \leq \Downarrow S (REC (\lambda f' x'. \text{body}' f' x') x')$
unfolding *REC-def*
apply (*clarsimp simp add: M*)
apply (*rule lfp-induct-pointwise* [**where** $\text{pre} = \lambda x' x. (x, x') \in R$ **and** $B = \text{body}$])
apply *rule*
apply *clarsimp*
apply (*blast intro: SUP-least*)

apply *simp*

apply (*simp add: trimonoD[OF M]*)

apply (*rule R0*)

apply (*subst lfp-unfold, simp add: trimonoD*)

apply (*rule RS*)

apply *blast*

apply *blast*

apply (*simp add: REC-def[abs-def]*)

done

lemma *RECT-refine[refine]*:

assumes *M: trimono body*

assumes *R0: (x,x')∈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')*

unfolding *RECT-def*

apply (*clarsimp simp add: M*)

apply (*rule flatf-fix-transfer[where*

fp'=flatf-gfp body

and *B'=body*

and *P= $\lambda x x'. (x',x) \in R$,*

OF - - flatf-ord.fix-unfold[OF M[THEN trimonoD-flatf-ge]] R0])

apply *simp*

apply (*simp add: trimonoD*)

by (*rule RS*)

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 (*if b then S1 else S2*) $\leq \Downarrow R$ (*if b' then S1' else S2'*)

using *assms by auto*

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

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

shows *Let x f $\leq \Downarrow R$ (Let x' f')*

using *assms by auto*

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')∈R'*

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

shows *Let m ($\lambda x. f x$) $\leq \Downarrow R$ (Let m' ($\lambda x'. f' x')$)*

using *assms* by *auto*

lemma *Let-refine'*:
assumes $(m, m') \in R$
assumes $(m, m') \in R \implies f m \leq \Downarrow S (f' m')$
shows $\text{Let } m f \leq \Downarrow S (\text{Let } m' f')$
using *assms* by *simp*

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 $\text{case-option } n f v \leq \Downarrow Rb (\text{case-option } n' f' v')$
using *assms*
by (*auto split: option.split simp: option-rel-def*)

lemma *list-case-refine[refine]*:
assumes $(li, l) \in \langle S \rangle \text{list-rel}$
assumes $fni \leq \Downarrow R fn$
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 $(\text{case } li \text{ of } \llbracket \implies fni \mid xi \# xsi \implies fci xi xsi \leq \Downarrow R (\text{case } l \text{ of } \llbracket \implies fn \mid x \# xs$
 $\implies fc x xs)$
using *assms* by (*auto split: list.split*)

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 $M \leq \Downarrow R (\text{Let } x f)$ **using** *assms* by *auto*

lemma *intro-Let-refine[refine2]*:
assumes $f x \leq \Downarrow R M'$
shows $\text{Let } x f \leq \Downarrow R M'$ **using** *assms* by *auto*

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 $\text{Let } x f \leq \Downarrow R (\text{bind } M' (\lambda x'. f' x'))$
using *assms*
apply (*simp add: pw-le-iff refine-pw-simps*)
apply *fast*
done

lemma *bind-Let-refine2*[*refine2*]: \llbracket
 $m' \leq \Downarrow R' (\text{RETURN } x);$
 $\bigwedge x'. \llbracket \text{inres } m' x'; (x', x) \in R \rrbracket \implies f' x' \leq \Downarrow R (f x)$
 $\rrbracket \implies m' \gg (\lambda x'. f' x') \leq \Downarrow R (\text{Let } x (\lambda x. f x))$
apply (*simp add: pw-le-iff refine-pw-simps*)
apply *blast*
done

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 $\text{RETURN } (\text{Let } x f) \leq \Downarrow R (\text{bind } M' (\lambda x'. f' x'))$
using *assms*
apply (*simp add: pw-le-iff refine-pw-simps*)
apply *fast*
done

lemma *RETURN-as-SPEC-refine*[*refine2*]:
assumes $M \leq \text{SPEC } (\lambda c. (c, a) \in R)$
shows $M \leq \Downarrow R (\text{RETURN } a)$
using *assms*
by (*simp add: pw-le-iff refine-pw-simps*)

lemma *RETURN-as-SPEC-refine-old*:
 $\bigwedge M R. M \leq \Downarrow R (\text{SPEC } (\lambda x. x = v)) \implies M \leq \Downarrow R (\text{RETURN } v)$
by (*simp add: RETURN-def*)

lemma *if-RETURN-refine* [*refine2*]:
assumes $b \longleftrightarrow b'$
assumes $\llbracket b; b \rrbracket \implies \text{RETURN } S1 \leq \Downarrow R S1'$
assumes $\llbracket \neg b; \neg b \rrbracket \implies \text{RETURN } S2 \leq \Downarrow R S2'$
shows $\text{RETURN } (\text{if } b \text{ then } S1 \text{ else } S2) \leq \Downarrow R (\text{if } b' \text{ then } S1' \text{ else } S2')$

using *assms*
by (*simp add: pw-le-iff refine-pw-simps*)

lemma *RES-sng-as-SPEC-refine*[*refine2*]:
assumes $M \leq \text{SPEC } (\lambda c. (c, a) \in R)$
shows $M \leq \Downarrow R (\text{RES } \{a\})$
using *assms*
by (*simp add: pw-le-iff refine-pw-simps*)

lemma *intro-spec-refine-iff*:
 $(\text{bind } (\text{RES } X) f \leq \Downarrow R M) \longleftrightarrow (\forall x \in X. f x \leq \Downarrow R M)$
apply (*simp add: pw-le-iff refine-pw-simps*)
apply *blast*
done

lemma *intro-spec-refine*[*refine2*]:
assumes $\bigwedge x. x \in X \implies f x \leq \Downarrow R M$
shows $\text{bind } (RES X) (\lambda x. f x) \leq \Downarrow R M$
using *assms*
by (*simp add: intro-spec-refine-iff*)

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 $m \leq \Downarrow R' (RETURN m')$
assumes $\bigwedge x x'. (x, x') \in R' \implies f x \leq \Downarrow R (f' x')$
shows $\text{bind } m (\lambda x. f x) \leq \Downarrow R (\text{Let } m' (\lambda x'. f' x'))$
using *assms*
apply (*simp add: pw-le-iff refine-pw-simps*)
apply *blast*
done

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

lemma *intro-bind-refine*:
assumes $m \leq \Downarrow R' (RETURN m')$
assumes $\bigwedge x. (x, m') \in R' \implies f x \leq \Downarrow R m''$
shows $\text{bind } m (\lambda x. f x) \leq \Downarrow R m''$
using *assms*
apply (*simp add: pw-le-iff refine-pw-simps*)
apply *blast*
done

lemma *intro-bind-refine-id*:
assumes $m \leq (SPEC ((=) m'))$
assumes $f m' \leq \Downarrow R m''$
shows $\text{bind } m f \leq \Downarrow R m''$
using *assms*
apply (*simp add: pw-le-iff refine-pw-simps*)
apply *blast*
done

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*:
 $\llbracket b \implies t \leq m; \neg b \implies e \leq m \rrbracket \implies \text{If } b \text{ t } e \leq m \text{ by } \textit{simp}$

lemma *lhs-step-RES*:
 $\llbracket \bigwedge x. x \in X \implies RETURN x \leq m \rrbracket \implies RES X \leq m$
by (*simp add: pw-le-iff*)

lemma *lhs-step-SPEC*:

$\llbracket \bigwedge x. \Phi x \implies \text{RETURN } x \leq m \rrbracket \implies \text{SPEC } (\lambda x. \Phi x) \leq m$
by (*simp add: pw-le-iff*)

lemma *lhs-step-bind*:

fixes $m :: 'a \text{ nres}$ **and** $f :: 'a \Rightarrow 'b \text{ nres}$
assumes $\text{nofail } m' \implies \text{nofail } m$
assumes $\bigwedge x. \text{nf-inres } m x \implies f x \leq m'$
shows $\text{do } \{x \leftarrow m; f x\} \leq m'$
using *assms*
by (*simp add: pw-le-iff refine-pw-simps*) *blast*

lemma *rhs-step-bind*:

assumes $m \leq \Downarrow R m' \quad \text{inres } m x \quad \bigwedge x'. (x, x') \in R \implies \text{lhs} \leq \Downarrow S (f' x')$
shows $\text{lhs} \leq \Downarrow S (m' \ggg f')$
using *assms*
by (*simp add: pw-le-iff refine-pw-simps*) *blast*

lemma *rhs-step-bind-RES*:

assumes $x' \in X'$
assumes $m \leq \Downarrow R (f' x')$
shows $m \leq \Downarrow R (\text{RES } X' \ggg f')$
using *assms* **by** (*simp add: pw-le-iff refine-pw-simps*) *blast*

lemma *rhs-step-bind-SPEC*:

assumes $\Phi x'$
assumes $m \leq \Downarrow R (f' x')$
shows $m \leq \Downarrow R (\text{SPEC } \Phi \ggg f')$
using *assms* **by** (*simp add: pw-le-iff refine-pw-simps*) *blast*

lemma *RES-bind-choose*:

assumes $x \in X$
assumes $m \leq f x$
shows $m \leq \text{RES } X \ggg f$
using *assms* **by** (*auto simp: pw-le-iff refine-pw-simps*)

lemma *pw-RES-bind-choose*:

$\text{nofail } (\text{RES } X \ggg f) \longleftrightarrow (\forall x \in X. \text{nofail } (f x))$
 $\text{inres } (\text{RES } X \ggg f) y \longleftrightarrow (\exists x \in X. \text{inres } (f x) y)$
by (*auto simp: refine-pw-simps*)

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 $(\text{case } p' \text{ of } (x1', x2') \Rightarrow f' x1' x2') \leq \Downarrow R (\text{case } p \text{ of } (x1, x2) \Rightarrow f x1 x2)$
using *assms* **by** (*auto split: prod.split*)

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

by (*simp add: nres-rel-def-internal relAPP-def*)

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

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

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

by (*auto simp: nres-rel-def conc-fun-chain[symmetric] conc-trans*)

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

by (*simp add: pw-le-iff nres-rel-def*)

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

by (*auto simp: nres-rel-def*)

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

by (*auto simp: nres-rel-def*)

lemma *param-RES[param]*:

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

unfolding *set-rel-def nres-rel-def*

by (*fastforce intro: RES-refine*)

lemma *param-RETURN[param]*:

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

by (*auto simp: nres-rel-def RETURN-refine*)

lemma *param-bind[param]*:

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

by (*auto simp: nres-rel-def intro: bind-refine dest: fun-relD*)

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

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

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

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

by (*auto intro: nres-relI*)

2.6.7 Autoref Setup

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

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

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

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

using *assms*

by (*auto simp: pw-le-iff refine-pw-simps nres-rel-def*)

context begin interpretation *autoref-syn* .

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

$do\ \{ASSERT\ \Phi;\ m\} \equiv OP\ (op\text{-}nres\text{-}ASSERT\text{-}bnd\ \Phi)\m

by *simp*

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

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

$do\ \{ASSUME\ \Phi;\ m\} \equiv OP\ (op\text{-}nres\text{-}ASSUME\text{-}bnd\ \Phi)\m

by *simp*

end

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

by (*auto simp: nres-rel-def*)

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

by (*auto simp: nres-rel-def*)

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

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

by (*auto simp: nres-rel-def RETURN-refine*)

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

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

apply (*intro fun-relI*)

apply (*rule nres-relI*)

apply (*rule bind-refine*)

apply (*erule nres-relD*)

apply (*erule (1) fun-relD[THEN nres-relD]*)

done

context begin interpretation *autoref-syn* .

lemma *autoref-ASSERT*[*autoref-rules*]:
assumes $\Phi \implies (m', m) \in \langle R \rangle \text{nres-rel}$
shows (
 m' ,
 $(OP \text{ (op-nres-ASSERT-bnd } \Phi) \text{ ::: } \langle R \rangle \text{nres-rel} \rightarrow \langle R \rangle \text{nres-rel}) \$ m) \in \langle R \rangle \text{nres-rel}$
using *assms unfolding nres-rel-def*
by (*simp add: ASSERT-refine-right*)

lemma *autoref-ASSUME*[*autoref-rules*]:
assumes *SIDE-PRECOND* Φ
assumes $\Phi \implies (m', m) \in \langle R \rangle \text{nres-rel}$
shows (
 m' ,
 $(OP \text{ (op-nres-ASSUME-bnd } \Phi) \text{ ::: } \langle R \rangle \text{nres-rel} \rightarrow \langle R \rangle \text{nres-rel}) \$ m) \in \langle R \rangle \text{nres-rel}$
using *assms unfolding nres-rel-def*
by (*simp add: ASSUME-refine-right*)

lemma *autoref-REC*[*autoref-rules*]:
assumes $(B, B') \in (Ra \rightarrow \langle Rr \rangle \text{nres-rel}) \rightarrow Ra \rightarrow \langle Rr \rangle \text{nres-rel}$
assumes *DEFER trimono B*
shows (*REC B*,
 $(OP \text{ REC}$
 $\text{ ::: } ((Ra \rightarrow \langle Rr \rangle \text{nres-rel}) \rightarrow Ra \rightarrow \langle Rr \rangle \text{nres-rel}) \rightarrow Ra \rightarrow \langle Rr \rangle \text{nres-rel}) \$ B'$
 $) \in Ra \rightarrow \langle Rr \rangle \text{nres-rel}$
apply (*intro fun-relI*)
using *assms*
apply (*auto simp: nres-rel-def intro!: REC-refine*)
apply (*simp add: fun-rel-def*)
apply *blast*
done

theorem *param-RECT*[*param*]:
assumes $(B, B') \in (Ra \rightarrow \langle Rr \rangle \text{nres-rel}) \rightarrow Ra \rightarrow \langle Rr \rangle \text{nres-rel}$
and *trimono B*
shows $(RECT_T B, RECT_T B') \in Ra \rightarrow \langle Rr \rangle \text{nres-rel}$
apply (*intro fun-relI*)
using *assms*
apply (*auto simp: nres-rel-def intro!: RECT-refine*)
apply (*simp add: fun-rel-def*)
apply *blast*
done

lemma *autoref-RECT*[*autoref-rules*]:
assumes $(B, B') \in (Ra \rightarrow \langle Rr \rangle \text{nres-rel}) \rightarrow Ra \rightarrow \langle Rr \rangle \text{nres-rel}$
assumes *DEFER trimono B*
shows (*RECT B*,
 $(OP \text{ RECT}$
 $\text{ ::: } ((Ra \rightarrow \langle Rr \rangle \text{nres-rel}) \rightarrow Ra \rightarrow \langle Rr \rangle \text{nres-rel}) \rightarrow Ra \rightarrow \langle Rr \rangle \text{nres-rel}) \$ B'$
 $) \in Ra \rightarrow \langle Rr \rangle \text{nres-rel}$

```

using assms
unfolding autoref-tag-defs
by (rule param-RECT)

```

end

2.6.8 Convenience Rules

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

```

lemma ref-two-step:  $A \leq \Downarrow R B \implies B \leq C \implies A \leq \Downarrow R C$ 
by (rule conc-trans-additional)

```

```

lemma pw-ref-iff:
shows  $S \leq \Downarrow R S'$ 
 $\longleftrightarrow$  (nofail  $S'$ 
 $\longrightarrow$  nofail  $S \wedge (\forall x. \text{inres } S x \longrightarrow (\exists s'. (x, s') \in R \wedge \text{inres } S' s'))$ )
by (simp add: pw-le-iff refine-pw-simps)

```

```

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  $S \leq \Downarrow R S'$ 
using assms
by (simp add: pw-ref-iff)

```

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

```

lemma introR:  $(a, a') \in R \implies (a, a') \in R$  .

```

```

lemma intro-prgR:  $c \leq \Downarrow R a \implies c \leq \Downarrow R a$  by auto

```

```

lemma refine-IdI:  $m \leq m' \implies m \leq \Downarrow \text{Id } m'$  by simp

```

```

lemma le-ASSERTI-pres:
assumes  $\Phi \implies S \leq \text{do } \{ \text{ASSERT } \Phi; S' \}$ 
shows  $S \leq \text{do } \{ \text{ASSERT } \Phi; S' \}$ 
using assms by (auto intro: le-ASSERTI)

```

```

lemma RETURN-ref-SPECD:
assumes  $\text{RETURN } c \leq \Downarrow R (\text{SPEC } \Phi)$ 
obtains  $a$  where  $(c, a) \in R \quad \Phi a$ 
using assms
by (auto simp: pw-le-iff refine-pw-simps)

```

```

lemma RETURN-ref-RETURN:
assumes  $\text{RETURN } c \leq \Downarrow R (\text{RETURN } a)$ 
shows  $(c, a) \in R$ 
using assms

```

apply (*auto simp: pw-le-iff refine-pw-simps*)
done

lemma *return-refine-prop-return*:
assumes *nofail m*
assumes $RETURN\ x \leq \Downarrow R\ m$
obtains x' **where** $(x, x') \in R$ $RETURN\ x' \leq m$
using *assms*
by (*auto simp: refine-pw-simps pw-le-iff*)

lemma *ignore-snd-refine-conv*:
 $(m \leq \Downarrow (R \times_r UNIV)\ m') \longleftrightarrow m \ggg (RETURN\ o\ fst) \leq \Downarrow R\ (m' \ggg (RETURN\ o\ fst))$
by (*auto simp: pw-le-iff refine-pw-simps*)

lemma *ret-le-down-conv*:
 $nofail\ m \implies RETURN\ c \leq \Downarrow R\ m \longleftrightarrow (\exists a. (c, a) \in R \wedge RETURN\ a \leq m)$
by (*auto simp: pw-le-iff refine-pw-simps*)

lemma *SPEC-eq-is-RETURN*:
 $SPEC\ ((=)\ x) = RETURN\ x$
 $SPEC\ (\lambda x. x=y) = RETURN\ y$
by (*auto simp: RETURN-def*)

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

lemma *refine2spec-aux*:
 $a \leq \Downarrow R\ b \longleftrightarrow (nofail\ b \longrightarrow a \leq SPEC\ (\lambda r. (\exists x. inres\ b\ x \wedge (r, x) \in R)))$
by (*auto simp: pw-le-iff refine-pw-simps*)

lemma *refine2specI*:
assumes $nofail\ b \implies a \leq SPEC\ (\lambda r. (\exists x. inres\ b\ x \wedge (r, x) \in R))$
shows $a \leq \Downarrow R\ b$
using *assms* **by** (*simp add: refine2spec-aux*)

lemma *specify-left*:
assumes $m \leq SPEC\ \Phi$
assumes $\bigwedge x. \Phi\ x \implies f\ x \leq M$
shows $do\ \{ x \leftarrow m; f\ x \} \leq M$
using *assms* **by** (*auto simp: pw-le-iff refine-pw-simps*)

lemma *build-rel-SPEC*:
 $M \leq SPEC\ (\lambda x. \Phi\ (\alpha\ x) \wedge I\ x) \implies M \leq \Downarrow (build-rel\ \alpha\ I)\ (SPEC\ \Phi)$
by (*auto simp: pw-le-iff refine-pw-simps build-rel-def*)

lemma *build-rel-SPEC-conv*: $\Downarrow (br\ \alpha\ I)\ (SPEC\ \Phi) = SPEC\ (\lambda x. I\ x \wedge \Phi\ (\alpha\ x))$
by (*auto simp: br-def pw-eq-iff refine-pw-simps*)

lemma *refine-IdD*: $c \leq \Downarrow Id a \implies c \leq a$ **by** *simp*

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.

using *assms*

by (*auto simp: pw-le-iff refine-pw-simps*)

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

— 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.

by (*auto simp: pw-le-iff refine-pw-simps*)

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

by (*auto simp: pw-eq-iff refine-pw-simps*)

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

by *auto*

lemma *le-RES-nofailI*:

assumes $a \leq RES\ x$

shows *nofail* a

using *assms*

by (*metis nofail-simps(2) pwD1*)

lemma *add-invar-refineI*:

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

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

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

using *assms*

by (*simp add: pw-le-iff refine-pw-simps sv-add-invar*)

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

by (*simp add: pw-eq-iff refine-pw-simps*)

blast

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

apply (*simp add: pw-eq-iff refine-pw-simps*)

apply *blast*
done

lemma *le-SPEC-bindI*:
assumes Φx
assumes $m \leq f x$
shows $m \leq \text{SPEC } \Phi \gg= f$
using *assms* **by** (*auto simp add: pw-le-iff refine-pw-simps*)

lemma *bind-assert-refine*:
assumes $m1 \leq \text{SPEC } \Phi$
assumes $\bigwedge x. \Phi x \implies m2 x \leq m'$
shows $do \{x \leftarrow m1; \text{ASSERT } (\Phi x); m2 x\} \leq m'$
using *assms*
by (*simp add: pw-le-iff refine-pw-simps*) *blast*

lemma *RETURN-refine-iff[simp]*: $\text{RETURN } x \leq \Downarrow R (\text{RETURN } y) \longleftrightarrow (x, y) \in R$
by (*auto simp: pw-le-iff refine-pw-simps*)

lemma *RETURN-RES-refine-iff*:
 $\text{RETURN } x \leq \Downarrow R (\text{RES } Y) \longleftrightarrow (\exists y \in Y. (x, y) \in R)$
by (*auto simp: pw-le-iff refine-pw-simps*)

lemma *RETURN-RES-refine*:
assumes $\exists x'. (x, x') \in R \wedge x' \in X$
shows $\text{RETURN } x \leq \Downarrow R (\text{RES } X)$
using *assms*
by (*auto simp: pw-le-iff refine-pw-simps*)

lemma *in-nres-rel-iff*: $(a, b) \in \langle R \rangle \text{nres-rel} \longleftrightarrow a \leq \Downarrow R b$
by (*auto simp: nres-rel-def*)

lemma *inf-RETURN-RES*:
 $\text{inf } (\text{RETURN } x) (\text{RES } X) = (\text{if } x \in X \text{ then } \text{RETURN } x \text{ else } \text{SUCCEED})$
 $\text{inf } (\text{RES } X) (\text{RETURN } x) = (\text{if } x \in X \text{ then } \text{RETURN } x \text{ else } \text{SUCCEED})$
by (*auto simp: pw-eq-iff refine-pw-simps*)

lemma *inf-RETURN-SPEC[simp]*:
 $\text{inf } (\text{RETURN } x) (\text{SPEC } (\lambda y. \Phi y)) = \text{SPEC } (\lambda y. y = x \wedge \Phi x)$
 $\text{inf } (\text{SPEC } (\lambda y. \Phi y)) (\text{RETURN } x) = \text{SPEC } (\lambda y. y = x \wedge \Phi x)$
by (*auto simp: pw-eq-iff refine-pw-simps*)

lemma *RES-sng-eq-RETURN*: $\text{RES } \{x\} = \text{RETURN } x$
by *simp*

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

by (*auto simp: pw-eq-iff refine-pw-simps*)

lemma *conc-fun-SPEC*:

$\Downarrow R$ (*SPEC* ($\lambda x. \Phi x$)) = *SPEC* ($\lambda y. \exists x. (y,x) \in R \wedge \Phi x$)

by (*auto simp: pw-eq-iff refine-pw-simps*)

lemma *conc-fun-RETURN*:

$\Downarrow R$ (*RETURN* x) = *SPEC* ($\lambda y. (y,x) \in R$)

by (*auto simp: pw-eq-iff refine-pw-simps*)

lemma *use-spec-rule*:

assumes $m \leq \text{SPEC } \Psi$

assumes $m \leq \text{SPEC } (\lambda s. \Psi s \longrightarrow \Phi s)$

shows $m \leq \text{SPEC } \Phi$

using *assms*

by (*auto simp: pw-le-iff refine-pw-simps*)

lemma *strengthen-SPEC*: $m \leq \text{SPEC } \Phi \implies m \leq \text{SPEC}(\lambda s. \text{inres } m s \wedge \text{nofail } m \wedge \Phi s)$

— Strengthen SPEC by adding trivial upper bound for result

by (*auto simp: pw-le-iff refine-pw-simps*)

lemma *weaken-SPEC*:

$m \leq \text{SPEC } \Phi \implies (\bigwedge x. \Phi x \implies \Psi x) \implies m \leq \text{SPEC } \Psi$

by (*force elim!: order-trans*)

lemma *bind-le-nofailI*:

assumes *nofail* m

assumes $\bigwedge x. \text{RETURN } x \leq m \implies f x \leq m'$

shows $m \gg=f \leq m'$

using *assms*

by (*simp add: refine-pw-simps pw-le-iff*) *blast*

lemma *bind-le-shift*:

bind $m f \leq m'$

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

by (*auto simp: pw-le-iff refine-pw-simps*)

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

fixes $t e :: 'a \text{ nres}$

shows (*If* $b t e \gg= (\lambda x. f x)$) = (*If* $b (t \gg= (\lambda x. f x)) (e \gg= (\lambda x. f x))$)

by *simp*

lemma *unused-bind-conv*:

assumes *NO-MATCH* (*ASSERT* Φ) m

assumes *NO-MATCH* (*ASSUME* Φ) m

shows $(m \gg (\lambda x. c)) = (\text{ASSERT } (\text{nofail } m) \gg (\lambda x. \text{ASSUME } (\exists x. \text{inres } m x) \gg (\lambda x. c)))$
by $(\text{auto simp: pw-eq-iff refine-pw-simps})$

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

lemma *let-to-bind-conv*:
Let $x f = \text{RETURN } x \gg f$
by *simp*

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

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

lemma *push-in-let-conv*:
Let $x (\lambda x. \text{RETURN } (f x)) = \text{RETURN } (\text{Let } x f)$
Let $x (\text{RETURN } o f) = \text{RETURN } (\text{Let } x f)$
by *simp-all*

lemma *pull-out-RETURN-case-option*:
case-option $(\text{RETURN } a) (\lambda v. \text{RETURN } (f v)) x = \text{RETURN } (\text{case-option } a f x)$
by $(\text{auto split: option.splits})$

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 $\text{do } \{ b \leftarrow ci; \text{if } b \text{ then } ti \text{ else } ei \} \leq \Downarrow R (\text{if } b \text{ then } t \text{ else } e)$
using *assms*
by $(\text{auto simp add: refine-pw-simps pw-le-iff})$

lemma *intro-RETURN-Let-refine*:
assumes $\text{RETURN } (f x) \leq \Downarrow R M'$
shows $\text{RETURN } (\text{Let } x f) \leq \Downarrow R M'$
using *assms* **by** *auto*

lemma *ife-FAIL-to-ASSERT-conv*:
 $(\text{if } \Phi \text{ then } m \text{ else } \text{FAIL}) = \text{op-nres-ASSERT-bnd } \Phi m$
by $(\text{cases } \Phi, \text{auto})$

lemma *nres-bind-let-law*: $(\text{do } \{ x \leftarrow \text{do } \{ \text{let } y=v; f y \}; g x \} :: \text{- nres})$
 $= \text{do } \{ \text{let } y=v; x \leftarrow f y; g x \}$ **by** *auto*

lemma *unused-bind-RES-ne*[*simp*]: $X \neq \{ \} \implies \text{do } \{ - \leftarrow \text{RES } X; m \} = m$
by $(\text{auto simp: pw-eq-iff refine-pw-simps})$

lemma *le-ASSERT-defI1*:

assumes $c \equiv do \{ASSERT \Phi; m\}$
assumes $\Phi \implies m' \leq c$
shows $m' \leq c$
using *assms*
by (*simp add: le-ASSERTI*)

lemma *refine-ASSERT-defI1*:

assumes $c \equiv do \{ASSERT \Phi; m\}$
assumes $\Phi \implies m' \leq \Downarrow R c$
shows $m' \leq \Downarrow R c$
using *assms*
by (*simp, refine-vcg*)

lemma *le-ASSERT-defI2*:

assumes $c \equiv do \{ASSERT \Phi; ASSERT \Psi; m\}$
assumes $\llbracket \Phi; \Psi \rrbracket \implies m' \leq c$
shows $m' \leq c$
using *assms*
by (*simp add: le-ASSERTI*)

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 $m' \leq \Downarrow R c$
using *assms*
by (*simp, refine-vcg*)

lemma *ASSERT-le-defI*:

assumes $c \equiv do \{ASSERT \Phi; m\}$
assumes Φ
assumes $\Phi \implies m' \leq m$
shows $c \leq m$
using *assms* **by** (*auto*)

lemma *ASSERT-same-eq-conv*: $(ASSERT \Phi \gg m) = (ASSERT \Phi \gg n) \longleftrightarrow (\Phi \longrightarrow m=n)$

by *auto*

lemma *case-prod-bind-simp[simp]*:

$(\lambda x. (case\ x\ of\ (a, b) \Rightarrow f\ a\ b) \leq SPEC\ \Phi) = (\lambda(a,b). f\ a\ b \leq SPEC\ \Phi)$
by *auto*

lemma *RECT-eq-REC'*: $nofail\ (RECT\ B\ x) \implies RECT\ B\ x = REC\ B\ x$

by (*subst RECT-eq-REC; simp-all add: nofail-def*)

lemma *rel2p-nres-RETURN[rel2p]*: $rel2p\ (\langle A \rangle nres-rel)\ (RETURN\ x)\ (RETURN$

$y) = \text{rel2p } A \ x \ y$
by (*auto simp: rel2p-def dest: nres-relD intro: nres-relI*)

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 $P \leq \text{SPEC } (\lambda s. Q \ s \longleftrightarrow R \ s)$
using *assms[THEN pw-le-iff[THEN iffD1]]*
by (*auto intro!: pw-leI*)

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 –
have $A \leq \text{inf } (\text{SPEC } P) (\text{SPEC } Q)$ **using** *assms* **by** (*rule-tac inf-greatest*)
assumption
thus *?thesis* **by** (*auto simp add:Collect-conj-eq*)
qed

lemma *SPEC-rule-conjunct1*:
assumes $A \leq \text{SPEC } (\lambda v. P \ v \wedge Q \ v)$
shows $A \leq \text{SPEC } P$
proof –
note *assms*
also have $\dots \leq \text{SPEC } P$ **by** (*rule SPEC-rule*) *auto*
finally show *?thesis* .
qed

lemma *SPEC-rule-conjunct2*:
assumes $A \leq \text{SPEC } (\lambda v. P \ v \wedge Q \ v)$
shows $A \leq \text{SPEC } Q$
proof –
note *assms*
also have $\dots \leq \text{SPEC } Q$ **by** (*rule SPEC-rule*) *auto*
finally show *?thesis* .
qed

Pointwise Reasoning

lemma *inres-if*:
 $\llbracket \text{inres } (\text{if } P \ \text{then } Q \ \text{else } R) \ x; \llbracket P; \text{inres } Q \ x \rrbracket \Longrightarrow S; \llbracket \neg P; \text{inres } R \ x \rrbracket \Longrightarrow S \rrbracket$
 $\Longrightarrow S$
by (*metis (full-types)*)

lemma *inres-SPEC*:
 $\text{inres } M \ x \Longrightarrow M \leq \text{SPEC } \Phi \Longrightarrow \Phi \ x$
by (*auto dest: pwD2*)

lemma *SPEC-nofail*:

$X \leq \text{SPEC } \Phi \implies \text{nofail } X$

by (*auto dest: pwD1*)

lemma *nofail-SPEC*: $\text{nofail } m \implies m \leq \text{SPEC } (\lambda-. \text{True})$

by (*simp add: pw-le-iff*)

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

by (*simp add: pw-le-iff*)

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

by (*simp add: pw-le-iff*)

end

2.7 Less-Equal or Fail

theory *Refine-Leaf*

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 *leafI*[*intro?*]:

assumes $\text{nofail } m \implies m \leq m'$ **shows** $m \leq_n m'$

using *assms unfolding le-or-fail-def* **by** *auto*

lemma *leafD*:

assumes $\text{nofail } m$

assumes $m \leq_n m'$

shows $m \leq m'$

using *assms unfolding le-or-fail-def* **by** *blast*

lemma *pw-leaf-iff*:

$m \leq_n m' \longleftrightarrow (\text{nofail } m \longrightarrow (\forall x. \text{inres } m x \longrightarrow \text{inres } m' x))$

unfolding *le-or-fail-def* **by** (*auto simp add: pw-le-iff refine-pw-simps*)

lemma *le-by-leafI*: $\llbracket \text{nofail } m' \implies \text{nofail } m; m \leq_n m' \rrbracket \implies m \leq m'$

by (*auto simp: pw-le-iff pw-leaf-iff*)

lemma *inres-leaf-mono*: $m \leq_n m' \implies \text{nofail } m \implies \text{inres } m x \implies \text{inres } m' x$

by (*auto simp: pw-leaf-iff*)

lemma *leaf-trans[trans]*: $\llbracket a \leq_n RES X; RES X \leq_n c \rrbracket \implies a \leq_n c$
by (*auto simp: pw-leaf-iff*)

lemma *leaf-trans-nofail*: $\llbracket a \leq_n b; nofail b; b \leq_n c \rrbracket \implies a \leq_n c$
by (*auto simp: pw-leaf-iff*)

lemma *leaf-refl[simp]*: $a \leq_n a$
by (*auto simp: pw-leaf-iff*)

lemma *leaf-RES-UNIV[simp, intro!]*: $m \leq_n RES UNIV$
by (*auto simp: pw-leaf-iff*)

lemma *leaf-FAIL[simp, intro!]*: $m \leq_n FAIL$ **by** (*auto simp: pw-leaf-iff*)

lemma *FAIL-leaf[simp, intro!]*: $FAIL \leq_n m$
by (*auto simp: le-or-fail-def*)

lemma *leaf-lift*:
 $m \leq F \implies m \leq_n F$
by (*auto simp add: pw-leaf-iff pw-le-iff*)

lemma *leaf-RETURN-rule[refine-vcg]*:
 $\Phi m \implies RETURN m \leq_n SPEC \Phi$ **by** (*simp add: pw-leaf-iff*)

lemma *leaf-bind-rule[refine-vcg]*:
 $\llbracket m \leq_n SPEC (\lambda x. f x \leq_n SPEC \Phi) \rrbracket \implies m \gg f \leq_n SPEC \Phi$
by (*auto simp add: pw-leaf-iff refine-pw-simps*)

lemma *RETURN-leaf-RES-iff[simp]*: $RETURN x \leq_n RES Y \longleftrightarrow x \in Y$
by (*auto simp add: pw-leaf-iff refine-pw-simps*)

lemma *RES-leaf-RES-iff[simp]*: $RES X \leq_n RES Y \longleftrightarrow X \subseteq Y$
by (*auto simp add: pw-leaf-iff refine-pw-simps*)

lemma *leaf-Let-rule[refine-vcg]*: $f x \leq_n SPEC \Phi \implies Let x f \leq_n SPEC \Phi$
by *simp*

lemma *leaf-If-rule[refine-vcg]*:
 $\llbracket c \implies t \leq_n SPEC \Phi; \neg c \implies e \leq_n SPEC \Phi \rrbracket \implies If c t e \leq_n SPEC \Phi$
by *simp*

lemma *leaf-RES-rule[refine-vcg]*:
 $\llbracket \bigwedge x. \Psi x \implies \Phi x \rrbracket \implies SPEC \Psi \leq_n SPEC \Phi$
 $\llbracket \bigwedge x. x \in X \implies \Phi x \rrbracket \implies RES X \leq_n SPEC \Phi$
by *auto*

lemma *leaf-True-rule*: $\llbracket \bigwedge x. \Phi x \rrbracket \implies m \leq_n SPEC \Phi$
by (*auto simp add: pw-leaf-iff refine-pw-simps*)

lemma *sup-leaf-iff*: $(sup a b \leq_n m) \longleftrightarrow (nofail a \wedge nofail b \longrightarrow a \leq_n m \wedge b \leq_n m)$

by (*auto simp: pw-leof-iff refine-pw-simps*)

lemma *sup-leof-rule*[*refine-vcg*]:

assumes $\llbracket \text{nofail } a; \text{nofail } b \rrbracket \Longrightarrow a \leq_n m$

assumes $\llbracket \text{nofail } a; \text{nofail } b \rrbracket \Longrightarrow b \leq_n m$

shows $\text{sup } a \ b \leq_n m$

using *assms* **by** (*auto simp: pw-leof-iff refine-pw-simps*)

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

$\llbracket v = \text{None} \Longrightarrow S1 \leq_n \text{SPEC } \Phi; \bigwedge x. v = \text{Some } x \Longrightarrow f2 \ x \leq_n \text{SPEC } \Phi \rrbracket$

$\Longrightarrow (\text{case } v \text{ of } \text{None} \Rightarrow S1 \mid \text{Some } x \Rightarrow f2 \ x) \leq_n \text{SPEC } \Phi$

by (*cases v auto*)

lemma *ASSERT-leof-rule*[*refine-vcg*]:

assumes $\Phi \Longrightarrow m \leq_n m'$

shows $\text{do } \{ \text{ASSERT } \Phi; m \} \leq_n m'$

using *assms*

by (*cases Φ , auto simp: pw-leof-iff*)

lemma *leof-ASSERT-rule*[*refine-vcg*]: $\llbracket \Phi \Longrightarrow m \leq_n m' \rrbracket \Longrightarrow m \leq_n \text{ASSERT } \Phi$
 $\gg m'$

by (*auto simp: pw-leof-iff refine-pw-simps*)

lemma *leof-ASSERT-refine-rule*[*refine*]: $\llbracket \Phi \Longrightarrow m \leq_n \Downarrow R \ m' \rrbracket \Longrightarrow m \leq_n \Downarrow R$
 $(\text{ASSERT } \Phi \gg m')$

by (*auto simp: pw-leof-iff refine-pw-simps*)

lemma *ASSUME-leof-iff*: $(\text{ASSUME } \Phi \leq_n \text{SPEC } \Psi) \longleftrightarrow (\Phi \longrightarrow \Psi \ ())$

by (*auto simp: pw-leof-iff*)

lemma *ASSUME-leof-rule*[*refine-vcg*]:

assumes $\Phi \Longrightarrow \Psi \ ()$

shows $\text{ASSUME } \Phi \leq_n \text{SPEC } \Psi$

using *assms*

by (*auto simp: ASSUME-leof-iff*)

lemma *SPEC-rule-conj-leofI1*:

assumes $m \leq \text{SPEC } \Phi$

assumes $m \leq_n \text{SPEC } \Psi$

shows $m \leq \text{SPEC } (\lambda s. \Phi \ s \wedge \Psi \ s)$

using *assms* **by** (*auto simp: pw-le-iff pw-leof-iff*)

lemma *SPEC-rule-conj-leofI2*:

assumes $m \leq_n \text{SPEC } \Phi$

assumes $m \leq \text{SPEC } \Psi$

shows $m \leq \text{SPEC } (\lambda s. \Phi \ s \wedge \Psi \ s)$

using *assms* **by** (*auto simp: pw-le-iff pw-leof-iff*)

lemma *SPEC-rule-leof-conjI*:

assumes $m \leq_n \text{SPEC } \Phi \quad m \leq_n \text{SPEC } \Psi$
shows $m \leq_n \text{SPEC } (\lambda x. \Phi x \wedge \Psi x)$
using *assms* **by** (*auto simp: pw-leof-iff*)

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 $m \leq_n \text{SPEC } \Phi$
using *assms* **by** (*auto simp: pw-leof-iff refine-pw-simps*)

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 $m \leq \text{SPEC } \Phi$
using *assms* **by** (*auto simp: pw-leof-iff pw-le-iff refine-pw-simps*)

lemma *leof-strengthen-SPEC*:

$m \leq_n \text{SPEC } \Phi \implies m \leq_n \text{SPEC } (\lambda x. \text{inres } m x \wedge \Phi x)$
by (*auto simp: pw-leof-iff*)

lemma *build-rel-SPEC-leof*:

assumes $m \leq_n \text{SPEC } (\lambda x. I x \wedge \Phi (\alpha x))$
shows $m \leq_n \Downarrow(\text{br } \alpha I) (\text{SPEC } \Phi)$
using *assms* **by** (*auto simp: build-rel-SPEC-conv*)

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

assumes $M \leq_n \text{SPEC } (\lambda c. (c, a) \in R)$
shows $M \leq_n \Downarrow R (\text{RETURN } a)$
using *assms*
by (*simp add: pw-leof-iff refine-pw-simps*)

lemma *ASSERT-leof-defI*:

assumes $c \equiv \text{do } \{ \text{ASSERT } \Phi; m' \}$
assumes $\Phi \implies m' \leq_n m$
shows $c \leq_n m$
using *assms* **by** (*auto simp: pw-leof-iff refine-pw-simps*)

lemma *leof-fun-conv-le*:

$(f x \leq_n M x) \longleftrightarrow (f x \leq (\text{if } \text{nofail } (f x) \text{ then } M x \text{ else } \text{FAIL}))$
by (*auto simp: pw-le-iff pw-leof-iff*)

lemma *leof-add-nofailI*: $\llbracket \text{nofail } m \implies m \leq_n m' \rrbracket \implies m \leq_n m'$

by (*auto simp: pw-le-iff pw-leof-iff*)

lemma *leof-cons-rule*[*refine-vcg-cons*]:

assumes $m \leq_n \text{SPEC } Q$
assumes $\bigwedge x. Q x \implies P x$

shows $m \leq_n SPEC P$
using *assms*
by (*auto simp: pw-le-iff pw-leof-iff*)

lemma *RECT-rule-leof*:

assumes *WF*: $wf (V::('x \times 'x) \text{ set})$
assumes *I0*: $pre (x::'x)$
assumes *IS*: $\bigwedge f x. [\bigwedge x'. [pre x'; (x',x) \in V] \implies f x' \leq_n M x'; pre x;$
 $RECT \text{ body} = f$
 $]] \implies \text{body } f x \leq_n M x$
shows $RECT \text{ body } x \leq_n M x$
apply (*cases \neg trimono body*)
apply (*simp add: RECT-def*)
using *assms*
unfolding *leof-fun-conv-le*
apply $-$
apply (*rule RECT-rule*[**where** $pre=pre$ **and** $V=V$])
apply *clarsimp-all*

proof $-$

fix $xa :: 'x$
assume *a1*: $\bigwedge x'. [pre x'; (x', xa) \in V] \implies RECT_T \text{ body } x' \leq (\text{if } \text{nofail } (RECT_T \text{ body } x') \text{ then } M x' \text{ else } FAIL)$
assume *a2*: $\bigwedge x f. [\bigwedge x'. [pre x'; (x', x) \in V] \implies f x' \leq (\text{if } \text{nofail } (f x') \text{ then } M x' \text{ else } FAIL); pre x; RECT_T \text{ body} = f] \implies \text{body } f x \leq (\text{if } \text{nofail } (\text{body } f x) \text{ then } M x \text{ else } FAIL)$
assume *a3*: $pre xa$
assume *a4*: $\text{nofail } (RECT_T \text{ body } xa)$
assume *a5*: trimono body
have *f6*: $\forall x. \neg pre x \vee (x, xa) \notin V \vee (\text{if } \text{nofail } (RECT_T \text{ body } x) \text{ then } RECT_T \text{ body } x \leq M x \text{ else } RECT_T \text{ body } x \leq FAIL)$
using *a1* **by** *presburger*
have *f7*: $\forall x f. ((\exists xa. (pre xa \wedge (xa, x) \in V) \wedge \neg f xa \leq (\text{if } \text{nofail } (f xa) \text{ then } M xa \text{ else } FAIL)) \vee \neg pre x \vee RECT_T \text{ body } \neq f) \vee \text{body } f x \leq (\text{if } \text{nofail } (\text{body } f x) \text{ then } M x \text{ else } FAIL)$
using *a2* **by** *blast*
obtain $xx :: ('x \Rightarrow 'a \text{ nres}) \Rightarrow 'x \Rightarrow 'x$ **where**
 $f8: \forall x0 x1. (\exists v2. (pre v2 \wedge (v2, x1) \in V) \wedge \neg x0 v2 \leq (\text{if } \text{nofail } (x0 v2) \text{ then } M v2 \text{ else } FAIL)) = ((pre (xx x0 x1) \wedge (xx x0 x1, x1) \in V) \wedge \neg x0 (xx x0 x1) \leq (\text{if } \text{nofail } (x0 (xx x0 x1)) \text{ then } M (xx x0 x1) \text{ else } FAIL))$
by *moura*
have *f9*: $\forall x0 x1. (x0 (xx x0 x1) \leq (\text{if } \text{nofail } (x0 (xx x0 x1)) \text{ then } M (xx x0 x1) \text{ else } FAIL)) = (\text{if } \text{nofail } (x0 (xx x0 x1)) \text{ then } x0 (xx x0 x1) \leq M (xx x0 x1) \text{ else } x0 (xx x0 x1) \leq FAIL)$
by *presburger*
have $\text{nofail } (\text{body } (RECT_T \text{ body } xa))$
using *a5 a4* **by** (*metis (no-types) RECT-unfold*)
then show $\text{body } (RECT_T \text{ body } xa) \leq M xa$
using *f9 f8 f7 f6 a3* **by** *fastforce*

qed

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.

```
ML <
  structure refine-heuristics-simps = Named-Thms
    ( val name = @{binding refine-hsimp}
      val description = Refinement Framework: ^
        Data refinement heuristics simp rules );
  >
```

```
setup <refine-heuristics-simps.setup>
```

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×'b) set ⇒ bool **where** *RELATES* R ≡ True
lemma *RELATESI*: *RELATES* R **by** (simp add: *RELATES-def*)

```
ML <
structure Refine-dref-type = struct
  structure pattern-rules = Named-Thms
    ( val name = @{binding refine-dref-pattern}
      val description = Refinement Framework: ^
        Pattern rules to recognize refinement goal );

  structure RELATES-rules = Named-Thms (
```

```

    val name = @{binding refine-dref-RELATES}
    val description = Refinement Framework:  $\hat{\quad}$ 
      Type based heuristics introduction rules
  );

  val tracing =
    Attrib.setup-config-bool @{binding refine-dref-tracing} (K false);

  (* Check whether term contains schematic variable *)
  fun
    has-schematic (Var _) = true |
    has-schematic (Abs (_,-,t)) = has-schematic t |
    has-schematic (t1$t2) = has-schematic t1 orelse has-schematic t2 |
    has-schematic - = false;

  (* Match proof states where the conclusion of some goal has the specified
     shape *)
  fun match-goal-shape-tac (shape:term $\rightarrow$ bool) (ctxt:Proof.context) i thm =
    if Thm.nprems-of thm  $\geq$  i then
      let
        val t = HOLogic.dest-Trueprop (Logic.concl-of-goal (Thm.prop-of thm) i);
      in
        (if shape t then all-tac thm else no-tac thm)
      end
    else
      no-tac thm;

  fun output-failed-msg ctxt failed-t = let
    val failed-t-str = Pretty.string-of
      (Syntax.pretty-term (Config.put show-types true ctxt) failed-t);
    val msg = Failed to resolve refinement goal  $\backslash n$   $\hat{\quad}$  failed-t-str;
    val - = if Config.get ctxt tracing then Output.tracing msg else ();
  in () end;

  (* Try to apply patternI-rules, ensure that produced first subgoal
     contains a schematic variable, and then solve it using
     refine-dref-RELATES-rules. *)
  fun type-tac ctxt =
    ALL-GOALS-FWD (TRY o (
      resolve-tac ctxt (pattern-rules.get ctxt) THEN'
      match-goal-shape-tac has-schematic ctxt THEN'
      (SOLVED' (REPEAT-ALL-NEW (resolve-tac ctxt (RELATES-rules.get ctxt)))
        ORELSE' (fn i => fn st => let
          val failed-t =
            HOLogic.dest-Trueprop (Logic.concl-of-goal (Thm.prop-of st) i);
          val - = output-failed-msg ctxt failed-t;
          in no-tac st end)
        )
    )

```

```

    ));

end;
>

setup <Refine-dref-type.RELATES-rules.setup>
setup <Refine-dref-type.pattern-rules.setup>

method-setup refine-dref-type =
  <Scan.lift (Args.mode trace -- Args.mode nopost)
  >> (fn (tracing,nopost) =>
    fn ctxt => (let
      val ctxt =
        if tracing then Config.put Refine-dref-type.tracing true ctxt else ctxt;
    in
      SIMPLE-METHOD (CHANGED (
        Refine-dref-type.type-tac ctxt
        THEN (if nopost then all-tac else ALLGOALS (TRY o Refine.post-tac
          ctxt))))
      end))
  >
  Use type-based heuristics to instantiate data refinement relations

```

2.8.2 Patterns

This section defines the patterns that are recognized as data refinement goals.

lemma *RELATESI-memb*[*refine-dref-pattern*]:

$$RELATES R \implies (a,b) \in R \implies (a,b) \in R .$$

lemma *RELATESI-refspec*[*refine-dref-pattern*]:

$$RELATES R \implies S \leq \downarrow R S' \implies S \leq \downarrow R S' .$$

Allows refine-rules to add *RELATES* goals if they introduce hidden relations

lemma *RELATES-pattern*[*refine-dref-pattern*]: *RELATES* *R* \implies *RELATES* *R* .

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* **by** (*simp add: RELATES-def*)

lemma *prod-rel-RELATES*[*refine-dref-RELATES*]:

$$RELATES Ra \implies RELATES Rb \implies RELATES (\langle Ra, Rb \rangle \text{prod-rel})$$

by (*simp add: RELATES-def prod-rel-def*)

declare *prod-rel-sv*[*refine-hsimp*]
lemma *prod-rel-iff*[*refine-hsimp*]:
 $((a,b),(a',b')) \in \langle A,B \rangle \text{prod-rel} \iff (a,a') \in A \wedge (b,b') \in B$
by (*auto simp: prod-rel-def*)

lemmas [*refine-hsimp*] = *prod-rel-id-simp*

lemma *option-rel-RELATES*[*refine-dref-RELATES*]:
 $RELATES Ra \implies RELATES (\langle Ra \rangle \text{option-rel})$
by (*simp add: RELATES-def option-rel-def*)

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*]:
 $RELATES R \implies RELATES (\langle R \rangle \text{set-rel})$ **by** (*simp add: RELATES-def*)

lemma *set-rel-empty-eq*: $(S,S') \in \langle X \rangle \text{set-rel} \implies S = \{\} \iff S' = \{\}$
by (*auto simp: set-rel-def*)

lemma *set-rel-sngD*: $(\{a\},\{b\}) \in \langle R \rangle \text{set-rel} \implies (a,b) \in R$
by (*auto simp: set-rel-def*)

lemma *Image-insert*[*refine-hsimp*]:
 $(a,b) \in R \implies \text{single-valued } R \implies R \text{ ``insert } a \ A = \text{insert } b \ (R \text{ ``} A)$
by (*auto dest: single-valuedD*)

lemmas [*refine-hsimp*] = *Image-Un*

lemma *Image-Diff*[*refine-hsimp*]:
 $\text{single-valued } (\text{converse } R) \implies R \text{ ``}(A - B) = R \text{ ``} A - R \text{ ``} B$
by (*auto dest: single-valuedD*)

lemma *Image-Inter*[*refine-hsimp*]:
 $\text{single-valued } (\text{converse } R) \implies R \text{ ``}(A \cap B) = R \text{ ``} A \cap R \text{ ``} B$
by (*auto dest: single-valuedD*)

lemma *list-rel-RELATES*[*refine-dref-RELATES*]:
 $RELATES R \implies RELATES (\langle R \rangle \text{list-rel})$ **by** (*simp add: RELATES-def*)

lemmas [*refine-hsimp*] = *list-rel-sv-iff list-rel-simp*

lemma *RELATES-nres-rel*[*refine-dref-RELATES*]: *RELATES* $R \implies \text{RELATES} (\langle R \rangle_{\text{nres-rel}})$
by (*simp add: RELATES-def*)

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

OBTAIN $P \equiv \text{ASSERT } (\exists x. P x) \gg \text{SPEC } P$

lemma *OBTAIN-nofail*[*refine-pw-simps*]: *nofail* (*OBTAIN* P) $\longleftrightarrow (\exists x. P x)$

unfolding *OBTAIN-def*

by (*auto simp: refine-pw-simps*)

lemma *OBTAIN-inres*[*refine-pw-simps*]: *inres* (*OBTAIN* P) $x \longleftrightarrow (\forall x. \neg P x) \vee P x$

unfolding *OBTAIN-def*

by (*auto simp: refine-pw-simps*)

lemma *OBTAIN-rule*[*refine-vcg*]: $\llbracket \exists x. P x; \bigwedge x. P x \implies Q x \rrbracket \implies \text{OBTAIN } P \leq \text{SPEC } Q$

unfolding *OBTAIN-def*

by *auto*

lemma *OBTAIN-refine-iff*: *OBTAIN* $P \leq \Downarrow R$ (*OBTAIN* Q) $\longleftrightarrow (Ex Q \longrightarrow Ex P \wedge \text{Collect } P \subseteq R^{-1} \text{Collect } Q)$

unfolding *OBTAIN-def* **by** (*auto simp: pw-le-iff refine-pw-simps*)

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 *OBTAIN* $P \leq \Downarrow R$ (*OBTAIN* Q)

using *assms* **unfolding** *OBTAIN-refine-iff*

by *blast*

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 $SELECT\ P \equiv \text{if } \exists x. P\ x \text{ then } RES\ \{Some\ x \mid x. P\ x\} \text{ else } RETURN\ None$

lemma $SELECT\text{-rule}[refine\text{-vcg}]$:
assumes $\bigwedge x. P\ x \implies Q\ (Some\ x)$
assumes $\forall x. \neg P\ x \implies Q\ None$
shows $SELECT\ P \leq SPEC\ Q$
unfolding $SELECT\text{-def}$
using $assms$
by $auto$

lemma $SELECT\text{-refine-iff}$: $(SELECT\ P \leq \Downarrow((R)\text{option-rel}) (SELECT\ P'))$
 \longleftrightarrow (
 $(\exists x. P' \longrightarrow \exists x. P) \wedge$
 $(\forall x. P\ x \longrightarrow (\exists x'. (x, x') \in R \wedge P' x'))$
 $)$
by $(force\ simp: SELECT\text{-def}\ pw\text{-le-iff}\ refine\text{-pw-simps})$

lemma $SELECT\text{-refine}[refine]$:
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 $SELECT\ P \leq \Downarrow((R)\text{option-rel}) (SELECT\ P')$
unfolding $SELECT\text{-refine-iff}$ **using** $assms$ **by** $blast$

lemma $SELECT\text{-as-SPEC}$: $SELECT\ P = SPEC\ (\lambda None \implies \forall x. \neg P\ x \mid Some\ x \implies P\ x)$
unfolding $SELECT\text{-def}$ **by** $(auto\ simp: pw\text{-eq-iff}\ split: option.split)$

lemma $SELECT\text{-pw}[refine\text{-pw-simps}]$:
 $nofail\ (SELECT\ P)$
 $inres\ (SELECT\ P)\ r \longleftrightarrow (r = None \longrightarrow (\forall x. \neg P\ x)) \wedge (\forall x. r = Some\ x \longrightarrow P\ x)$
unfolding $SELECT\text{-def}$
by $auto$

lemma $SELECT\text{-pw-simps}[simp]$:
 $nofail\ (SELECT\ P)$
 $inres\ (SELECT\ P)\ None \longleftrightarrow (\forall x. \neg P\ x)$
 $inres\ (SELECT\ P)\ (Some\ x) \longleftrightarrow P\ x$
by $(auto\ simp: refine\text{-pw-simps})$

end

2.10 Generic While-Combinator

theory $RefineG\text{-While}$
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

iWHILEIT bind return I b f s0 \equiv *RECT (WHILEI-body bind return I b f) s0*

definition *iWHILE bind return* \equiv *iWHILEI bind return* $(\lambda-. \text{True})$

definition *iWHILET bind return* \equiv *iWHILEIT bind return* $(\lambda-. \text{True})$

lemma *mono-prover-monoI*[*refine-mono*]:

monotone (fun-ord (\leq)) (fun-ord (\leq)) B \implies mono B

apply (*rule ccpo-monoD*)

apply (*simp add: le-fun-def[abs-def] fun-ord-def[abs-def]*)

done

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 ($\lambda x. \text{bind } (f x) g$)*

assumes *ibind-mono-ge*: $\llbracket \text{flat-ge } m m'; \bigwedge x. \text{flat-ge } (f x) (f' x) \rrbracket$
 $\implies \text{flat-ge } (\text{bind } m f) (\text{bind } m' f')$

assumes *ibind-mono*: $\llbracket (\leq) m m'; \bigwedge x. (\leq) (f x) (f' x) \rrbracket$
 $\implies (\leq) (\text{bind } m f) (\text{bind } m' f')$

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*)
unfolding *WHILEI-body-def*
by *refine-mono*

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)
unfolding *WHILEI-def*
apply (*subst REC-unfold*[*OF WHILEI-body-trimono*])
unfolding *WHILEI-body-def*
apply (*rule refl*)
done

lemma *REC-mono-ref*[*refine-mono*]:
 $\llbracket \text{trimono } B; \bigwedge D x. B D x \leq B' D x \rrbracket \implies \text{REC } B x \leq \text{REC } B' x$
unfolding *REC-def*
apply *clarsimp*
apply (*rule lfp-mono*[*THEN le-funD*])
by (*rule le-funI*)

lemma *RECT-mono-ref*[*refine-mono*]:
 $\llbracket \text{trimono } B; \bigwedge D x. B D x \leq B' D x \rrbracket \implies \text{RECT } B x \leq \text{RECT } B' x$
unfolding *RECT-gfp-def*
apply *clarsimp*
apply (*rule gfp-mono*[*THEN le-funD*])
by (*rule le-funI*)

lemma *WHILEI-weaken*:
assumes *IW*: $\bigwedge x. I x \implies I' x$
shows *WHILEI I' b f x* \leq *WHILEI I b f x*
unfolding *WHILEI-def*
apply (*rule REC-mono-ref*[*OF WHILEI-body-trimono*])
apply (*auto simp add: WHILEI-body-def dest: IW*)
done

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)
unfolding *WHILEIT-def*
apply (*subst RECT-unfold*[*OF WHILEI-body-trimono*])
unfolding *WHILEI-body-def*

apply (*rule refl*)
done

lemma *WHILEIT-weaken*:
assumes *IW*: $\bigwedge x. I x \implies I' x$
shows *WHILEIT* $I' b f x \leq \text{WHILEIT } I b f x$
unfolding *WHILEIT-def*
apply (*rule RECT-mono-ref*[*OF WHILEI-body-trimono*])
apply (*auto simp add: WHILEI-body-def dest: IW*)
done

lemma *WHILEI-le-WHILEIT*: *WHILEI* $I b f s \leq \text{WHILEIT } I b f s$
unfolding *WHILEI-def WHILEIT-def*
by (*rule REC-le-RECT*)

While without Annotated Invariant

lemma *WHILE-unfold*:
 $\text{WHILE } b f s = (\text{if } b s \text{ then bind } (f s) (\text{WHILE } b f) \text{ else return } s)$
unfolding *WHILE-def*
apply (*subst WHILEI-unfold*)
apply *simp*
done

lemma *WHILET-unfold*:
 $\text{WHILET } b f s = (\text{if } b s \text{ then bind } (f s) (\text{WHILET } b f) \text{ else return } s)$
unfolding *WHILET-def*
apply (*subst WHILEIT-unfold*)
apply *simp*
done

lemma *transfer-WHILEIT-esc*[*refine-transfer*]:
assumes *REF*: $\bigwedge x. \text{return } (f x) \leq F x$
shows $\text{return } (\text{while } b f x) \leq \text{WHILEIT } I b F x$

proof –

interpret *transfer return* .
show *?thesis*
unfolding *WHILEIT-def*
apply (*rule transfer-RECT'*[**where** *fr=while b f*])
apply (*rule while-unfold*)
unfolding *WHILEI-body-def*
apply (*split if-split, intro allI impI conjI*)
apply *simp-all*

apply (*rule order-trans*[*OF - ibind-mono*[*OF REF order-refl*]])
apply (*simp add: imonad-laws*)
done

qed

lemma *transfer-WHILET-esc*[*refine-transfer*]:
assumes *REF*: $\bigwedge x. \text{return } (f x) \leq F x$
shows $\text{return } (\text{while } b f x) \leq \text{WHILET } b F x$
unfolding *WHILET-def*
using *assms* **by** (*rule transfer-WHILEIT-esc*)

lemma *WHILE-mono-prover-rule*[*refine-mono*]:
 $\llbracket \bigwedge x. f x \leq f' x \rrbracket \implies \text{WHILE } b f s0 \leq \text{WHILE } b f' s0$
 $\llbracket \bigwedge x. f x \leq f' x \rrbracket \implies \text{WHILEI } I b f s0 \leq \text{WHILEI } I b f' s0$
 $\llbracket \bigwedge x. f x \leq f' x \rrbracket \implies \text{WHILET } b f s0 \leq \text{WHILET } b f' s0$
 $\llbracket \bigwedge x. f x \leq f' x \rrbracket \implies \text{WHILEIT } I b f s0 \leq \text{WHILEIT } I b f' s0$
 $\llbracket \bigwedge x. \text{flat-ge } (f x) (f' x) \rrbracket \implies \text{flat-ge } (\text{WHILET } b f s0) (\text{WHILET } b f' s0)$
 $\llbracket \bigwedge x. \text{flat-ge } (f x) (f' x) \rrbracket \implies \text{flat-ge } (\text{WHILEIT } I b f s0) (\text{WHILEIT } I b f' s0)$
unfolding *WHILE-def* *WHILEI-def* *WHILEI-body-def*
WHILET-def *WHILEIT-def*
by *refine-mono+*

end

locale *transfer-WHILE* =
c: *generic-WHILE* *cbind* *creturn* *cWHILEIT* *cWHILEI* *cWHILET* *cWHILE* +
a: *generic-WHILE* *abind* *areturn* *aWHILEIT* *aWHILEI* *aWHILET* *aWHILE* +
dist-transfer α
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*: $\llbracket \alpha m \leq M; \bigwedge x. \alpha (f x) \leq F x \rrbracket$
 $\implies \alpha (\text{cbind } m f) \leq \text{abind } M F$
assumes *transfer-return*: $\alpha (\text{creturn } x) \leq \text{areturn } x$
begin

lemma *transfer-WHILEIT*[*refine-transfer*]:
assumes *REF*: $\bigwedge x. \alpha (f x) \leq F x$
shows $\alpha (\text{cWHILEIT } I b f x) \leq \text{aWHILEIT } I b F x$
unfolding *c.WHILEIT-def* *a.WHILEIT-def*
apply (*rule transfer-RECT*[*OF* - *c.WHILEI-body-trimono*])
unfolding *WHILEI-body-def*
apply *auto*
apply (*rule transfer-bind*)
apply (*rule REF*)
apply *assumption*
apply (*rule transfer-return*)
done

lemma *transfer-WHILEI*[*refine-transfer*]:
assumes *REF*: $\bigwedge x. \alpha (f x) \leq F x$
shows $\alpha (c \text{WHILEI } I b f x) \leq a \text{WHILEI } I b F x$
unfolding *c.WHILEI-def a.WHILEI-def*
apply (*rule transfer-REC*[*OF - c.WHILEI-body-trimono*])
unfolding *WHILEI-body-def*
apply *auto*
apply (*rule transfer-bind*)
apply (*rule REF*)
apply *assumption*
apply (*rule transfer-return*)
done

lemma *transfer-WHILE*[*refine-transfer*]:
assumes *REF*: $\bigwedge x. \alpha (f x) \leq F x$
shows $\alpha (c \text{WHILE } b f x) \leq a \text{WHILE } b F x$
unfolding *c.WHILE-def a.WHILE-def*
using *assms by (rule transfer-WHILEI)*

lemma *transfer-WHILET*[*refine-transfer*]:
assumes *REF*: $\bigwedge x. \alpha (f x) \leq F x$
shows $\alpha (c \text{WHILET } b f x) \leq a \text{WHILET } b F x$
unfolding *c.WHILET-def a.WHILET-def*
using *assms by (rule transfer-WHILEIT)*

end

locale *generic-WHILE-rules* =
generic-WHILE bind return WHILEIT WHILEI WHILET WHILE
for *bind return SPEC WHILEIT WHILEI WHILET WHILE +*
assumes *iSPEC-eq*: $\text{SPEC } \Phi = \text{Sup } \{ \text{return } x \mid x. \Phi x \}$
assumes *ibind-rule*: $\llbracket M \leq \text{SPEC } (\lambda x. f x \leq \text{SPEC } \Phi) \rrbracket \implies \text{bind } M f \leq \text{SPEC } \Phi$
begin

lemma *ireturn-eq*: $\text{return } x = \text{SPEC } ((=) x)$
unfolding *iSPEC-eq by auto*

lemma *iSPEC-rule*: $(\bigwedge x. \Phi x \implies \Psi x) \implies \text{SPEC } \Phi \leq \text{SPEC } \Psi$
unfolding *iSPEC-eq*
by (*auto intro: Sup-mono*)

lemma *ireturn-rule*: $\Phi x \implies \text{return } x \leq \text{SPEC } \Phi$
unfolding *ireturn-eq*
by (*auto intro: iSPEC-rule*)

lemma *WHILEI-rule*:
assumes *I0*: $I s$
assumes *ISTEP*: $\bigwedge s. \llbracket I s; b s \rrbracket \implies f s \leq \text{SPEC } I$

assumes $CONS: \bigwedge s. \llbracket I\ s; \neg\ b\ s \rrbracket \implies \Phi\ s$
shows $WHILEI\ I\ b\ f\ s \leq SPEC\ \Phi$
apply (*rule order-trans*[**where** $y=SPEC\ (\lambda s. I\ s \wedge \neg\ b\ s)$])
apply (*unfold WHILEI-def*)
apply (*rule REC-rule*[*OF WHILEI-body-trimono*])
apply (*rule I0*)

unfolding *WHILEI-body-def*
apply (*split if-split*)
apply (*intro impI conjI*)
apply *simp-all*
apply (*rule ibind-rule*)
apply (*erule (1) order-trans*[*OF ISTEP*])
apply (*rule iSPEC-rule, assumption*)

apply (*rule ireturn-rule*)
apply *simp*

apply (*rule iSPEC-rule*)
apply (*simp add: CONS*)

done

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 $WHILEIT\ I\ b\ f\ s \leq SPEC\ \Phi$

unfolding *WHILEIT-def*
apply (*rule RECT-rule*[*OF WHILEI-body-trimono WF, where pre=I, OF I0*])
unfolding *WHILEI-body-def*
apply (*split if-split*)
apply (*intro impI conjI*)
apply *simp-all*

apply (*rule ibind-rule*)
apply (*rule order-trans*[*OF IS*], *assumption+*)
apply (*rule iSPEC-rule*)
apply *simp*

apply (*rule ireturn-rule*)
apply (*simp add: PHI*)
done

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 ≤ SPEC Φ
unfolding WHILE-def
apply (rule order-trans[OF WHILEI-weaken WHILEI-rule])
apply (rule TrueI)
apply (rule I0)
using assms by auto

```

```

lemma WHILET-rule:
  assumes WF: wf R
  assumes I0: I s
  assumes IS:  $\bigwedge s. \llbracket I s; b s \rrbracket \implies f s \leq \text{SPEC } (\lambda s'. I s' \wedge (s', s) \in R)$ 
  assumes PHI:  $\bigwedge s. \llbracket I s; \neg b s \rrbracket \implies \Phi s$ 
  shows WHILET b f s ≤ SPEC Φ
  unfolding WHILET-def
  apply (rule order-trans[OF WHILEIT-weaken WHILEIT-rule])
  apply (rule TrueI)
  apply (rule WF)
  apply (rule I0)
  using assms by auto

```

end

end

2.11 While-Loops

theory Refine-While

imports

Refine-Basic Refine-Leaf Generic/RefineG-While

begin

definition WHILEIT ($WHILE_T^-$) **where**

$WHILEIT \equiv iWHILEIT \text{ bind } RETURN$

definition WHILEI ($WHILE^-$) **where** $WHILEI \equiv iWHILEI \text{ bind } RETURN$

definition WHILET ($WHILE_T$) **where** $WHILET \equiv iWHILET \text{ bind } RETURN$

definition WHILE **where** $WHILE \equiv iWHILE \text{ bind } RETURN$

interpretation while?: generic-WHILE-rules bind RETURN SPEC

WHILEIT WHILEI WHILET WHILE

apply unfold-locales

apply (simp-all add: WHILEIT-def WHILEI-def WHILET-def WHILE-def)

apply refine-mono

apply refine-mono

apply (subst RES-Sup-RETURN[symmetric])

apply (rule-tac f=Sup in arg-cong) **apply** auto []

apply (erule bind-rule)

done

lemmas [refine-vcg] = WHILEI-rule
 lemmas [refine-vcg] = WHILEIT-rule

2.11.1 Data Refinement Rules

lemma *ref-WHILEI-invarI*:
 assumes $I s \implies M \leq \Downarrow R (WHILEI I b f s)$
 shows $M \leq \Downarrow R (WHILEI I b f s)$
 apply (subst WHILEI-unfold)
 apply (cases I s)
 apply (subst WHILEI-unfold[symmetric])
 apply (erule assms)
 apply simp
 done

lemma *ref-WHILEIT-invarI*:
 assumes $I s \implies M \leq \Downarrow R (WHILEIT I b f s)$
 shows $M \leq \Downarrow R (WHILEIT I b f s)$
 apply (subst WHILEIT-unfold)
 apply (cases I s)
 apply (subst WHILEIT-unfold[symmetric])
 apply (erule assms)
 apply simp
 done

lemma *WHILEI-le-rule*:
 assumes $R0: (s, s') \in R$
 assumes $RS: \bigwedge W s s'. \llbracket \bigwedge s s'. (s, s') \in R \implies W s \leq M s'; (s, s') \in R \rrbracket \implies$
 $do \{ASSERT (I s); \text{if } b s \text{ then bind } (f s) W \text{ else RETURN } s\} \leq M s'$
 shows $WHILEI I b f s \leq M s'$
 unfolding WHILEI-def
 apply (rule REC-le-rule[where M=M])
 apply (simp add: WHILEI-body-def, refine-mono)
 apply (rule R0)
 apply (erule (1) order-trans[rotated, OF RS])
 unfolding WHILEI-body-def
 apply auto
 done

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 $RS: \bigwedge s s'. \llbracket (s,s') \in R'; I' s'; I s; b s \rrbracket$
 $\implies f s \leq \text{sup} (\Downarrow R' (do \{ASSUME (b' s'); f' s'\})) (\Downarrow R' (RETURN s'))$
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 $WHILEI I b f s \leq \Downarrow R (WHILEI I' b' f' s')$
apply (rule *ref-WHILEI-invarI*)
apply (rule *WHILEI-le-rule[where R=R']*)
apply (erule *R0*)
apply (rule *ref-WHILEI-invarI*)
apply (frule (1) *RI*)
apply (case-tac $b s = \text{False}$)
apply (subst *WHILEI-unfold*)
apply (auto dest: *RB intro: RETURN-refine R-REF*) []

apply *simp*
apply (rule *order-trans[OF monoD[OF bind-mono1 RS], assumption+]*)
apply (*simp only: bind-distrib-sup1*)
apply (rule *sup-least*)
apply (subst *WHILEI-unfold*)
apply (*simp add: RB, intro impI*)
apply (rule *bind-refine*)
apply (rule *order-refl*)
apply *simp*

apply (*simp add: pw-le-iff refine-pw-simps, blast*)
done

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 $WHILEI I b f x \leq \Downarrow R (WHILEI I' b' f' x')$
apply (rule *WHILEI-invisible-refine-genR[OF R0]*)
apply *assumption*
apply (erule (1) *IREF*)
apply (*simp add: COND-REF*)
apply (rule *le-supI1*)
apply (*simp add: COND-REF STEP-REF*)
apply (rule *R-REF, assumption+*)
done

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 $\text{WHILE } b f s \leq \Downarrow R (\text{WHILE } b' f' s')$
unfolding WHILE-def
apply (rule $\text{WHILEI-invisible-refine-genR}$)
apply (rule $\text{assms, (assumption+)?}$)
done

lemma WHILE-refine-genR :

assumes $R0: (x, x') \in R'$
assumes $\text{COND-REF}: \bigwedge x x'. \llbracket (x, x') \in R' \rrbracket \implies b x = b' x'$
assumes STEP-REF :
 $\bigwedge x x'. \llbracket (x, x') \in R'; b x; b' x' \rrbracket \implies f x \leq \Downarrow R' (f' x')$
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 $\text{WHILE } b f x \leq \Downarrow R (\text{WHILE } b' f' x')$
unfolding WHILE-def
apply (rule $\text{WHILEI-refine-genR}$)
apply (rule $\text{assms, (assumption+)?}$)
done

lemma $\text{WHILE-refine-genR}'$:

assumes $R0: (x, x') \in R'$
assumes $\text{COND-REF}: \bigwedge x x'. \llbracket (x, x') \in R'; I' x' \rrbracket \implies b x = b' x'$
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 $R\text{-REF}$:
 $\bigwedge x x'. \llbracket (x, x') \in R'; \neg b x; \neg b' x' \rrbracket \implies (x, x') \in R$
shows $\text{WHILE } b f x \leq \Downarrow R (\text{WHILEI } I' b' f' x')$
unfolding WHILE-def
apply (rule $\text{WHILEI-refine-genR}$)
apply (rule $\text{assms TrueI, (assumption+)?}$)
done

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 $\text{WHILEI-invisible-refine}$:

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 $\text{WHILEI } I b f s \leq \Downarrow R (\text{WHILEI } I' b' f' s')$
apply (rule $\text{WHILEI-invisible-refine-genR}[\text{where } R'=R]$)
apply (rule $\text{assms, (assumption+)?}$)
done

lemma *WHILEI-refine*[*refine*]:
assumes *R0*: $I' x' \Longrightarrow (x, x') \in R$
assumes *IREF*: $\bigwedge x x'. \llbracket (x, x') \in R; I' x' \rrbracket \Longrightarrow I x$
assumes *COND-REF*: $\bigwedge x x'. \llbracket (x, x') \in R; I x; I' x' \rrbracket \Longrightarrow b x = b' x'$
assumes *STEP-REF*:
 $\bigwedge x x'. \llbracket (x, x') \in R; b x; b' x'; I x; I' x' \rrbracket \Longrightarrow f x \leq \Downarrow R (f' x')$
shows *WHILEI* $I b f x \leq \Downarrow R (WHILEI I' b' f' x')$
apply (*rule* *WHILEI-invisible-refine*[*OF R0*])
apply *assumption*
apply (*erule* (1) *IREF*)
apply (*simp add*: *COND-REF*)
apply (*rule le-supI1*)
apply (*simp add*: *COND-REF STEP-REF*)
done

lemma *WHILE-invisible-refine*:
assumes *R0*: $(s, s') \in R$
assumes *RB*: $\bigwedge s s'. \llbracket (s, s') \in R; b' s' \rrbracket \Longrightarrow b s$
assumes *RS*: $\bigwedge s s'. \llbracket (s, s') \in R; b s \rrbracket$
 $\Longrightarrow f s \leq \sup (\Downarrow R (do \{ASSUME (b' s'); f' s'\})) (\Downarrow R (RETURN s'))$
shows *WHILE* $b f s \leq \Downarrow R (WHILE b' f' s')$
unfolding *WHILE-def*
apply (*rule* *WHILEI-invisible-refine*)
using *assms by auto*

lemma *WHILE-le-rule*:
assumes *R0*: $(s, s') \in R$
assumes *RS*: $\bigwedge W s s'. \llbracket \bigwedge s s'. (s, s') \in R \Longrightarrow W s \leq M s'; (s, s') \in R \rrbracket \Longrightarrow$
 $do \{if b s then bind (f s) W else RETURN s\} \leq M s'$
shows *WHILE* $b f s \leq M s'$
unfolding *WHILE-def*
apply (*rule* *WHILEI-le-rule*[*OF R0*])
apply (*simp add*: *RS*)
done

lemma *WHILE-refine*[*refine*]:
assumes *R0*: $(x, x') \in R$
assumes *COND-REF*: $\bigwedge x x'. \llbracket (x, x') \in R \rrbracket \Longrightarrow b x = b' x'$
assumes *STEP-REF*:
 $\bigwedge x x'. \llbracket (x, x') \in R; b x; b' x' \rrbracket \Longrightarrow f x \leq \Downarrow R (f' x')$
shows *WHILE* $b f x \leq \Downarrow R (WHILE b' f' x')$
unfolding *WHILE-def*
apply (*rule* *WHILEI-refine*)
using *assms by auto*

lemma *WHILE-refine'*[*refine*]:
assumes *R0*: $I' x' \Longrightarrow (x, x') \in R$
assumes *COND-REF*: $\bigwedge x x'. \llbracket (x, x') \in R; I' x' \rrbracket \Longrightarrow b x = b' x'$

assumes *STEP-REF*:

$\bigwedge x x'. \llbracket (x,x') \in R; b x; b' x'; I' x' \rrbracket \implies f x \leq \Downarrow R (f' x')$

shows *WHILE* $b f x \leq \Downarrow R (WHILEI I' b' f' x')$

unfolding *WHILE-def*

apply (*rule WHILEI-refine*)

using *assms by auto*

lemma *AIF-leI*: $\llbracket \Phi; \Phi \implies S \leq S' \rrbracket \implies (if \ \Phi \ then \ S \ else \ FAIL) \leq S'$

by *auto*

lemma *ref-AIFI*: $\llbracket \Phi \implies S \leq \Downarrow R S' \rrbracket \implies S \leq \Downarrow R (if \ \Phi \ then \ S' \ else \ FAIL)$

by (*cases* Φ) *auto*

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* $I b f x \leq \Downarrow R (WHILEIT I' b' f' x')$

apply (*rule ref-WHILEIT-invarI*)

unfolding *WHILEIT-def*

apply (*rule RECT-refine[OF WHILEI-body-trimono R0]*)

apply *assumption*

unfolding *WHILEI-body-def*

apply (*rule ref-AIFI*)

apply (*rule AIF-leI*)

apply (*blast intro: IREF*)

apply (*rule if-refine*)

apply (*simp add: COND-REF*)

apply (*rule bind-refine*)

apply (*rule STEP-REF, assumption+*) \llbracket

apply *assumption*

apply (*rule RETURN-refine*)

apply (*simp add: R-REF*)

done

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 $WHILET\ b\ f\ x \leq \Downarrow R$ ($WHILET\ b'\ f'\ x'$)
unfolding $WHILET-def$
apply (*rule* $WHILEIT-refine-genR[OF\ R0]$)
apply (*rule* $TrueI$)
apply (*rule* $assms, assumption+$)
done

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 $WHILET\ b\ f\ x \leq \Downarrow R$ ($WHILEIT\ I'\ b'\ f'\ x'$)
unfolding $WHILET-def$
apply (*rule* $WHILEIT-refine-genR[OF\ R0]$)
apply *assumption*
apply (*rule* $TrueI$)
apply (*rule* $assms, assumption+$)
done

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 $WHILEIT\ I\ b\ f\ x \leq \Downarrow R$ ($WHILEIT\ I'\ b'\ f'\ x'$)
using $WHILEIT-refine-genR[where\ R=R\ and\ R'=R, OF\ assms]$.

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 $WHILET\ b\ f\ x \leq \Downarrow R$ ($WHILET\ b'\ f'\ x'$)
unfolding $WHILET-def$
apply (*rule* $WHILEIT-refine$)
using *assms by auto*

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 $WHILET\ b\ f\ x \leq \Downarrow R$ ($WHILEIT\ I'\ b'\ f'\ x'$)
unfolding $WHILET-def$
apply (*rule* $WHILEIT-refine$)

using *assms* by *auto*

lemma *WHILEI-refine-new-invar*:

assumes *R0*: $I' x' \implies (x, x') \in R$

assumes *INV0*: $\llbracket I' x'; (x, x') \in R \rrbracket \implies I x$

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 *STEP-INV*:

$\bigwedge x x'. \llbracket (x, x') \in R; b x; b' x'; I x; I' x'; \text{nofail} (f x) \rrbracket \implies f x \leq \text{SPEC } I$

shows *WHILEI* $I b f x \leq \Downarrow R (\text{WHILEI } I' b' f' x')$

apply (rule *WHILEI-refine-genR*[**where**

$I=I$ and $I'=I'$ and $x'=x'$ and $x=x$ and $R=R$ and $b=b$ and $b'=b'$ and $f'=f'$

and $f=f$

and $R'=\{ (c, a). (c, a) \in R \wedge I c \}$

])

using *assms*

by (*auto intro: add-invar-refineI*)

lemma *WHILEIT-refine-new-invar*:

assumes *R0*: $I' x' \implies (x, x') \in R$

assumes *INV0*: $\llbracket I' x'; (x, x') \in R \rrbracket \implies I x$

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 *STEP-INV*:

$\bigwedge x x'. \llbracket \text{nofail} (f x); (x, x') \in R; b x; b' x'; I x; I' x' \rrbracket \implies f x \leq \text{SPEC } I$

shows *WHILEIT* $I b f x \leq \Downarrow R (\text{WHILEIT } I' b' f' x')$

apply (rule *WHILEIT-refine-genR*[**where**

$I=I$ and $I'=I'$ and $x'=x'$ and $x=x$ and $R=R$ and $b=b$ and $b'=b'$ and $f'=f'$

and $f=f$

and $R'=\{ (c, a). (c, a) \in R \wedge I c \}$

])

using *assms*

by (*auto intro: add-invar-refineI*)

2.11.2 Autoref Setup

context begin interpretation *autoref-syn* .

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

WHILEIT $I \equiv \text{OP} (\text{WHILEIT } I)$

WHILEI $I \equiv \text{OP} (\text{WHILEI } I)$

by *auto*

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

assumes $\bigwedge x x'. \llbracket (x, x') \in R \rrbracket \implies (c x, c' x') \in \text{Id}$

assumes $\bigwedge x x'. \llbracket \text{REMOVE-INTERNAL } c' x'; (x, x') \in R \rrbracket$

$\implies (f\ x, f'\$x') \in \langle R \rangle nres\text{-}rel$
assumes $(s, s') \in R$
shows (*WHILET* $c\ f\ s$,
 $(OP\ WHILET ::: (R \rightarrow Id) \rightarrow (R \rightarrow \langle R \rangle nres\text{-}rel) \rightarrow R \rightarrow \langle R \rangle nres\text{-}rel)\$c'\$f'\s')
 $\in \langle R \rangle nres\text{-}rel$
using *assms*
by (*auto simp add: nres-rel-def fun-rel-def intro!: WHILET-refine*)

lemma *autoref-WHILEIT*[*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 (*WHILET* $c\ f\ s$,
 $(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'\s')
 $\in \langle R \rangle nres\text{-}rel$
using *assms*
by (*auto simp add: nres-rel-def fun-rel-def intro!: WHILET-refine'*)

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 (*WHILET* $c\ f$,
 $(OP\ (WHILEIT\ I) ::: (R \rightarrow Id) \rightarrow (R \rightarrow \langle R \rangle nres\text{-}rel) \rightarrow R \rightarrow \langle R \rangle nres\text{-}rel)\$c'\$f'$)
 $\in R \rightarrow \langle R \rangle nres\text{-}rel$
unfolding *autoref-tag-defs*
by (*parametricity*
add: autoref-WHILEIT[unfolded autoref-tag-defs]
assms[unfolded autoref-tag-defs])

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')
 $\in \langle R \rangle nres\text{-}rel$
using *assms*
by (*auto simp add: nres-rel-def fun-rel-def intro!: WHILE-refine*)

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* $c\ f$,
 $(OP\ WHILE ::: (R \rightarrow Id) \rightarrow (R \rightarrow \langle R \rangle nres\text{-}rel) \rightarrow R \rightarrow \langle R \rangle nres\text{-}rel)\$c'\$f'$)
 $\in R \rightarrow \langle R \rangle nres\text{-}rel$

using *assms*
by (*auto simp add: nres-rel-def fun-rel-def intro!: WHILE-refine*)

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 $\bigwedge x x'. \llbracket REMOVE-INTERNAL\ c'\ x'; I x'; (x,x') \in R \rrbracket \implies (f\ x, f'\$x') \in \langle R \rangle nres-rel$

assumes $I\ s' \implies (s, s') \in R$

shows (*WHILE* *c f s*,

(*OP* (*WHILEI* *I*) $::: (R \rightarrow Id) \rightarrow (R \rightarrow \langle R \rangle nres-rel) \rightarrow R \rightarrow \langle R \rangle nres-rel$) $\$c'\$f'\$s'$
 $\in \langle R \rangle nres-rel$)

using *assms*

by (*auto simp add: nres-rel-def fun-rel-def intro!: WHILE-refine'*)

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 $\bigwedge x x'. \llbracket REMOVE-INTERNAL\ c'\ x'; (x,x') \in R; I x' \rrbracket \implies (f\ x, f'\$x') \in \langle R \rangle nres-rel$

shows (*WHILE* *c f*,

(*OP* (*WHILEI* *I*) $::: (R \rightarrow Id) \rightarrow (R \rightarrow \langle R \rangle nres-rel) \rightarrow R \rightarrow \langle R \rangle nres-rel$) $\$c'\f'
 $\in R \rightarrow \langle R \rangle nres-rel$)

unfolding *autoref-tag-defs*

by (*parametricity*

add: autoref-WHILEI[*unfolded autoref-tag-defs*]

assms[*unfolded autoref-tag-defs*])

end

2.11.3 Invariants

Tail Recursion

context *begin*

private lemma *tailrec-transform-aux1*:

assumes *RETURN* *s* $\leq m$

shows *REC* ($\lambda D\ s. \sup (a\ s \gg D) (b\ s)$) $s \leq \text{lfp } (\lambda x. \sup m (x \gg a)) \gg b$

(**is** *REC* $?F\ s \leq \text{lfp } ?f \gg b$)

proof (*rule* *REC-rule*[**where** *pre* = $\lambda s. \text{RETURN } s \leq \text{lfp } ?f$], *refine-mono*)

show *RETURN* *s* $\leq \text{lfp } (\lambda x. \sup m (x \gg a))$

apply (*subst* *lfp-unfold*, *tagged-solver*)

using *assms* **apply** (*simp* *add: le-supI1*)

done

next

fix *f x*

assume *IH*: $\bigwedge x. \text{RETURN } x \leq \text{lfp } ?f \implies$

$f\ x \leq \text{lfp } ?f \gg b$

```

and PRE: RETURN  $x \leq \text{lf}p \ ?f$ 

show  $\text{sup } (a \ x \ggg f) (b \ x) \leq \text{lf}p \ ?f \ggg b$ 
proof (rule sup-least)
  show  $b \ x \leq \text{lf}p \ ?f \ggg b$ 
  using PRE
  by (simp add: pw-le-iff refine-pw-simps) blast
next
from PRE have  $a \ x \leq \text{lf}p \ ?f \ggg a$ 
  by (simp add: pw-le-iff refine-pw-simps) blast
also have  $\dots \leq \text{lf}p \ ?f$ 
  apply (subst (2) lf-p-unfold, tagged-solver)
  apply (simp add: le-supI2)
  done
finally show  $a \ x \ggg f \leq \text{lf}p \ ?f \ggg b$ 
  using IH
  by (simp add: pw-le-iff refine-pw-simps) blast
qed
qed

private corollary tailrec-transform1:
  fixes  $m :: 'a \ \text{nres}$ 
  shows  $m \ggg \text{REC } (\lambda D \ s. \ \text{sup } (a \ s \ggg D) (b \ s)) \leq \text{lf}p \ (\lambda x. \ \text{sup } m \ (x \ggg a)) \ggg$ 
  b
  apply (cases nofail m)
  apply (erule bind-le-nofailI)
  apply (erule tailrec-transform-aux1)
  apply (simp add: not-nofail-iff lf-p-const)
  done

private lemma tailrec-transform-aux2:
  fixes  $m :: 'a \ \text{nres}$ 
  shows  $\text{lf}p \ (\lambda x. \ \text{sup } m \ (x \ggg a))$ 
   $\leq m \ggg \text{REC } (\lambda D \ s. \ \text{sup } (a \ s \ggg D) (\text{RETURN } s))$ 
  (is  $\text{lf}p \ ?f \leq m \ggg \text{REC } ?F$ )
  apply (subst gen-kleene-lfp)
  apply (simp add: cont-def pw-eq-iff refine-pw-simps, blast)
  apply (rule SUP-least, simp)
proof -
  fix  $i$ 
  show  $((\lambda x. \ x \ggg a) \ \sim\sim i) \ m \leq m \ggg \text{REC } (\lambda D \ s. \ \text{sup } (a \ s \ggg D) (\text{RETURN } s))$ 
  apply (induction i arbitrary: m)
  apply simp
  apply (subst REC-unfold, refine-mono)
  apply (simp add: pw-le-iff refine-pw-simps, blast)

  apply (subst funpow-Suc-right)

```



```

apply simp
apply (rule order-trans)
apply rprems
apply simp

apply (subst (2) REC-unfold, refine-mono)
apply (simp add: bind-distrib-sup2)
done
qed

```

```

private lemma tailrec-transform-aux3: REC ( $\lambda D s. \text{sup } (a s \ggg D)$ ) (RETURN
s)  $s \ggg b$ 
 $\leq \text{REC } (\lambda D s. \text{sup } (a s \ggg D)) (b s)$  s
apply (subst bind-le-shift)
apply (rule REC-rule, refine-mono)
apply (rule TrueI)
apply auto
apply (subst (asm) (4) REC-unfold, refine-mono)
apply (rule order-trans[OF bind-mono(1)[OF order-refl]])
apply rprems
apply (subst (3) REC-unfold, refine-mono)
apply (simp add: refine-pw-simps pw-le-iff, blast)

apply (subst REC-unfold, refine-mono, simp)
done

```

```

private lemma tailrec-transform2:
 $\text{lfp } (\lambda x. \text{sup } m (\text{bind } x a)) \ggg b \leq m \ggg \text{REC } (\lambda D s. \text{sup } (a s \ggg D)) (b s)$ 
proof –
from bind-mono(1)[OF tailrec-transform-aux2 order-refl]
have  $\text{lfp } (\lambda x. \text{sup } m (\text{bind } x a)) \ggg b$ 
 $\leq m \ggg (\lambda x. \text{REC } (\lambda D s. \text{sup } (a s \ggg D)) (\text{RETURN } s)) x \ggg b$ 
by simp
also from bind-mono(1)[OF order-refl tailrec-transform-aux3]
have  $\dots \leq m \ggg \text{REC } (\lambda D s. \text{sup } (a s \ggg D)) (b s)$  .
finally show ?thesis .
qed

```

definition *tailrec-body* $a b \equiv (\lambda D s. \text{sup } (\text{bind } (a s) D)) (b s)$

```

theorem tailrec-transform:
 $m \ggg \text{REC } (\lambda D s. \text{sup } (a s \ggg D)) (b s) = \text{lfp } (\lambda x. \text{sup } m (\text{bind } x a)) \ggg b$ 
apply (rule antisym)
apply (rule tailrec-transform1)
apply (rule tailrec-transform2)
done

```

theorem *tailrec-transform'*:

$m \gg= REC (tailrec-body a b) = lfp (\lambda x. sup m (bind x a)) \gg= b$
unfolding *tailrec-body-def*
by (*rule tailrec-transform*)

lemma *WHILE* $c f =$
 $REC (tailrec-body$
 $(\lambda s. do \{ASSUME (c s); f s\}$
 $(\lambda s. do \{ASSUME (\neg c s); RETURN s\}$
 $)$
unfolding *WHILE-def WHILEI-def WHILEI-body-def tailrec-body-def*
apply (*fo-rule fun-cong arg-cong*)+ **apply** (*intro ext*)
apply *auto*
done

lemma *WHILEI-tailrec-conv*: *WHILEI* $I c f =$
 $REC (tailrec-body$
 $(\lambda s. do \{ASSERT (I s); ASSUME (c s); f s\}$
 $(\lambda s. do \{ASSERT (I s); ASSUME (\neg c s); RETURN s\}$
 $)$
unfolding *WHILEI-def WHILEI-body-def tailrec-body-def*
apply (*fo-rule fun-cong arg-cong*)+ **apply** (*intro ext*)
apply *auto*
done

lemma *WHILEIT-tailrec-conv*: *WHILEIT* $I c f =$
 $RECT (tailrec-body$
 $(\lambda s. do \{ASSERT (I s); ASSUME (c s); f s\}$
 $(\lambda s. do \{ASSERT (I s); ASSUME (\neg c s); RETURN s\}$
 $)$
unfolding *WHILEIT-def WHILEI-body-def tailrec-body-def*
apply (*fo-rule fun-cong arg-cong*)+ **apply** (*intro ext*)
apply *auto*
done

definition *WHILEI-lfp-body* $I m c f \equiv$
 $(\lambda x. sup m (do \{$
 $s \leftarrow x;$
 $- \leftarrow ASSERT (I s);$
 $- \leftarrow ASSUME (c s);$
 $f s$
 $\}))$

lemma *WHILEI-lfp-conv*: $m \gg= WHILEI I c f =$
 $do \{$
 $s \leftarrow lfp (WHILEI-lfp-body I m c f);$
 $ASSERT (I s);$
 $ASSUME (\neg c s);$
 $RETURN s$
 $\}$

```

}
unfolding WHILEI-lfp-body-def
apply (subst WHILEI-tailrec-conv)
apply (subst tailrec-transform')
..
end

```

Most Specific Invariant

definition *msii* — Most specific invariant for WHILE-loop
where $msii\ I\ m\ c\ f \equiv lfp\ (WHILEI-lfp-body\ I\ m\ c\ f)$

lemma [*simp*, *intro!*]: *mono* ($WHILEI-lfp-body\ I\ m\ c\ f$)
unfolding *WHILEI-lfp-body-def* **by** *tagged-solver*

definition *filter-ASSUME* $c\ m \equiv do\ \{x \leftarrow m; ASSUME\ (c\ x); RETURN\ x\}$

definition *filter-ASSERT* $c\ m \equiv do\ \{x \leftarrow m; ASSERT\ (c\ x); RETURN\ x\}$

lemma [*refine-pw-simps*]: *nofail* ($filter-ASSUME\ c\ m$) \longleftrightarrow *nofail* m
unfolding *filter-ASSUME-def*
by (*simp add: refine-pw-simps*)

lemma [*refine-pw-simps*]: *inres* ($filter-ASSUME\ c\ m$) x
 \longleftrightarrow (*nofail* $m \longrightarrow inres\ m\ x \wedge c\ x$)
unfolding *filter-ASSUME-def*
by (*simp add: refine-pw-simps*)

lemma *msii-is-invar*:
 $m \leq msii\ I\ m\ c\ f$
 $m \leq msii\ I\ m\ c\ f \implies bind\ (filter-ASSUME\ c\ (filter-ASSERT\ I\ m))\ f \leq msii\ I\ m\ c\ f$
unfolding *msii-def*
apply –
apply (*subst lfp-unfold, simp*)
apply (*simp add: WHILEI-lfp-body-def*)

unfolding *filter-ASSUME-def filter-ASSERT-def*
apply (*subst lfp-unfold, simp*)
apply (*simp add: WHILEI-lfp-body-def*)
apply (*simp only: refine-pw-simps pw-le-iff*) **apply** *blast*
done

lemma *WHILE-msii-conv*: $m \gg= WHILEI\ I\ c\ f$
 $= filter-ASSUME\ (Not\ o\ c)\ (filter-ASSERT\ I\ (msii\ I\ m\ c\ f))$
unfolding *WHILEI-lfp-conv filter-ASSERT-def filter-ASSUME-def msii-def*
by *simp*

lemma *msii-induct*:

```

assumes  $I0: m0 \leq P$ 
assumes  $IS: \bigwedge m. \llbracket m \leq msii\ I\ m0\ c\ f; m \leq P; \endpre>$ 
```

```

  filter-ASSUME  $c$  (filter-ASSERT  $I\ m$ )  $\ggg f \leq msii\ I\ m0\ c\ f$ 
   $\rrbracket \implies filter-ASSUME\ c$  (filter-ASSERT  $I\ m$ )  $\ggg f \leq P$ 

```

```

shows  $msii\ I\ m0\ c\ f \leq P$ 

```

```

unfolding  $msii-def\ WHILEI-lfp-body-def$ 

```

```

apply (rule  $lfp-gen-induct$ , tagged-solver)

```

```

unfolding  $msii-def[symmetric]\ WHILEI-lfp-body-def[symmetric]$ 

```

```

apply (rule  $I0$ )

```

```

apply (drule  $IS$ , assumption)

```

```

unfolding  $filter-ASSERT-def\ filter-ASSUME-def$ 

```

```

apply  $simp-all$ 

```

```

done

```

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  $I\ s$ 

```

```

using  $assms(\mathcal{I})$ 

```

```

apply  $induct$ 

```

```

using  $I$  apply  $auto$   $\square$ 

```

```

using  $S$  apply  $fastforce$   $\square$ 

```

```

done

```

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  $is-rwof-invar\ m0\ cond\ step\ I$ 

```

```

using  $assms$  unfolding  $is-rwof-invar-def$  by  $blast$ 

```

lemma *rwof-cons*: $\llbracket is\text{-}rwof\text{-}invar\ m0\ cond\ step\ I; rwof\ m0\ cond\ step\ s \rrbracket \implies I\ s$
unfolding *is-rwof-invar-def*
using *establish-rwof-invar*[of *m0 I cond step s*]
by *blast*

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 $m0 \ggg WHILE\ cond\ step \leq SPEC\ (\lambda s. rwof\ m0\ cond\ step\ s \wedge \neg cond\ s \wedge I\ s)$
proof –
from *I0* **obtain** *M0* **where** [*simp*]: $m0 = RES\ M0$ **and** $M0 \subseteq Collect\ I$
by (*cases m0*) *auto*

show *?thesis*
apply *simp*
apply *refine-vcg*
apply (*rule WHILE-rule*[**where** $I = \lambda s. I\ s \wedge rwof\ m0\ cond\ step\ s$])
proof –
fix *s* **assume** $s \in M0$ **thus** $I\ s \wedge rwof\ m0\ cond\ step\ s$
using *I0* **by** (*auto intro: rwof.init*) \square

next
fix *s*
assume *A*: $I\ s \wedge rwof\ m0\ cond\ step\ s$ **and** *C*: $cond\ s$
hence $step\ s \leq SPEC\ I$ **by** – (*rule S, simp-all*)
also then obtain *S'* **where** $step\ s = RES\ S'$ **by** (*cases step s*) *auto*
from *rwof.step*[*OF conjunct2*[*OF A*] *C this*] *this*
have $step\ s \leq SPEC\ (rwof\ m0\ cond\ step)$ **by** *auto*
finally (*SPEC-rule-conjI*)
show $step\ s \leq SPEC\ (\lambda s. I\ s \wedge rwof\ m0\ cond\ step\ s)$.
qed *auto*
qed

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* $\longleftrightarrow (nofail\ I \longrightarrow inres\ I\ x \wedge \neg b\ x)$
unfolding *filter-nb-def*
by (*simp-all add: refine-pw-simps*)

lemma *filter-nb-mono*: $m \leq m' \implies filter\text{-}nb\ cond\ m \leq filter\text{-}nb\ cond\ m'$

unfolding *filter-nb-def*
by *refine-mono*

lemma *filter-nb-cont*:
 $filter\text{-}nb\ cond\ (Sup\ M) = Sup\ \{filter\text{-}nb\ cond\ m\ |\ m.\ m \in M\}$
apply (*rule antisym*)
apply (*simp add: pw-le-iff refine-pw-simps*)
apply (*auto intro: not-nofail-inres simp: refine-pw-simps*) []

apply (*simp add: pw-le-iff refine-pw-simps*)
apply (*auto intro: not-nofail-inres simp: refine-pw-simps*) []
done

lemma *filter-nb-FAIL[simp]*: $filter\text{-}nb\ cond\ FAIL = FAIL$
by (*simp add: filter-nb-def*)

lemma *filter-nb-RES[simp]*: $filter\text{-}nb\ cond\ (RES\ X) = RES\ \{x \in X.\ \neg cond\ x\}$
by (*simp add: pw-eq-iff refine-pw-simps*)

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 $m0 \gg\gg WHILE\ b\ f \leq filter\text{-}nb\ b\ I$
apply (*unfold WHILE-def WHILEI-def*)
apply (*refine-rcg order-trans[OF I0] refine-vcg pw-bind-leI*)
using $I0$ **apply** (*simp add: pw-le-iff refine-pw-simps*)

apply (*rule REC-rule[OF WHILEI-body-trimono, where pre= $\lambda s.$ RETURN*
 $s \leq I$)
using $I0$ **apply** (*simp add: pw-le-iff refine-pw-simps*)

unfolding *WHILEI-body-def*
apply (*split if-split*)
apply (*intro impI conjI*)
apply (*simp-all*)
using $ISTEP$
apply (*simp (no-asm-use) only: pw-le-iff refine-pw-simps*) **apply** *blast*
apply (*simp only: pw-le-iff refine-pw-simps*) **apply** *metis*
done

primrec *bounded-WHILE'*
 $:: nat \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a\ nres) \Rightarrow 'a\ nres \Rightarrow 'a\ nres$
where
 $bounded\text{-}WHILE'\ 0\ cond\ step\ m = m$
 $| bounded\text{-}WHILE'\ (Suc\ n)\ cond\ step\ m = do\ \{$
 $\ x \leftarrow m;$

```

    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 (*induction n arbitrary: m*)

case 0 **thus** ?*case* **by** (*simp cong: if-cong*)

next

case (*Suc n*)

have *aux1*: do {

```

  x ← m;
  if cond x then do {
    x ← bounded-WHILE n cond step (step x);
    if cond x then step x else RETURN x
  }
  else RETURN x
} = do {
  x ← do {
    x ← m;
    if cond x then bounded-WHILE n cond step (step x) else RETURN x
  };
  if cond x then step x else RETURN x
}
by (simp add: pw-eq-iff refine-pw-simps)

```

show ?*case*

apply (*simp cong: if-cong*)

apply (*subst aux1*)

apply (*subst Suc.IH*)

apply (*simp add: pw-eq-iff refine-pw-simps*)

done

qed

lemma *bounded-WHILE'-eq*:

bounded-WHILE' n cond step m = bounded-WHILE n cond step m
apply (*induct n arbitrary: m*)
apply (*auto cong: if-cong simp: bounded-WHILE-shift*)
done

lemma *mWHILE-unfold*: $m \gg= \text{WHILE cond step} = \text{do } \{$

$x \leftarrow m;$
 $\text{if cond } x \text{ then step } x \gg= \text{WHILE cond step}$
 $\text{else RETURN } x$
 $\}$
by (*subst WHILE-unfold[abs-def]*) (*rule refl*)

lemma *WHILE-bounded-aux1*:

filter-nb cond (bounded-WHILE n cond step m) $\leq m \gg= \text{WHILE cond step}$
unfolding *bounded-WHILE'-eq[symmetric]*
apply (*induct n arbitrary: m*)
apply *simp*
apply (*subst mWHILE-unfold*)
apply (*simp add: pw-le-iff refine-pw-simps, blast*)

apply *simp*
apply (*subst mWHILE-unfold*)
apply (*auto simp add: pw-le-iff refine-pw-simps*)
done

lemma *WHILE-bounded-aux2*:

$m \gg= \text{WHILE cond step}$
 $\leq \text{filter-nb cond } (\text{Sup } \{\text{bounded-WHILE } n \text{ cond step } m \mid n. \text{True}\})$
apply (*rule WHILE-rule-gen-le*)
apply (*metis (mono-tags, lifting) Sup-upper bounded-WHILE.simps(1) mem-Collect-eq*)

proof –

fix s
assume $\text{RETURN } s \leq \text{Sup } \{\text{bounded-WHILE } n \text{ cond step } m \mid n. \text{True}\}$
then obtain n **where** $\text{RETURN } s \leq \text{bounded-WHILE } n \text{ cond step } m$
by (*fold inres-def*) (*auto simp: refine-pw-simps*)
moreover assume $\text{cond } s$
ultimately have $\text{step } s \leq \text{bounded-WHILE } (\text{Suc } n) \text{ cond step } m$
by (*simp add: pw-le-iff refine-pw-simps, blast*)
thus $\text{step } s \leq \text{Sup } \{\text{bounded-WHILE } n \text{ cond step } m \mid n. \text{True}\}$
by (*metis (mono-tags, lifting) Sup-upper2 mem-Collect-eq*)

qed

lemma *WHILE-bounded*:

$m \gg= \text{WHILE cond step}$


```

= filter-nb cond (Sup {bounded-WHILE n cond step m | n. True})
apply (rule antisym)
apply (rule WHILE-bounded-aux2)

apply (simp add: filter-nb-cont)
apply (rule Sup-least)
apply (auto simp: WHILE-bounded-aux1)
done

```

Relation to rwof

lemma *rwof-in-bounded-WHILE*:

```

assumes rwof m0 cond step s
shows  $\exists n. \text{RETURN } s \leq (\text{bounded-WHILE } n \text{ cond step } m0)$ 
using assms
apply induction
apply (rule-tac  $x=0$  in exI)
apply simp

apply clarsimp
apply (rule-tac  $x=\text{Suc } n$  in exI)
apply (auto simp add: pw-le-iff refine-pw-simps) []
done

```

lemma *bounded-WHILE-FAIL-rwof*:

```

assumes bounded-WHILE n cond step m0 = FAIL
assumes M0:  $m0 \neq \text{FAIL}$ 
shows  $\exists n' < n. \exists x X.$ 
  bounded-WHILE  $n'$  cond step  $m0 = \text{RES } X$ 
   $\wedge x \in X \wedge \text{cond } x \wedge \text{step } x = \text{FAIL}$ 
using assms
proof (induction n)
  case 0 thus ?case by simp
next
  case (Suc n)
  assume bounded-WHILE (Suc n) cond step  $m0 = \text{FAIL}$ 
  hence bounded-WHILE n cond step  $m0 = \text{FAIL}$ 
   $\vee (\exists X x. \text{bounded-WHILE } n \text{ cond step } m0 = \text{RES } X$ 
     $\wedge x \in X \wedge \text{cond } x \wedge \text{step } x = \text{FAIL})$ 
  (is ?C1  $\vee$  ?C2)
  apply (cases bounded-WHILE n cond step  $m0$ )
  apply simp
  apply (simp add: pw-eq-iff refine-pw-simps split: if-split-asm)
  apply (auto intro: not-nofail-inres simp: refine-pw-simps)
  done
moreover {
  assume ?C1
  from Suc.IH[OF this M0] obtain  $n' x X$  where  $n' < n$  and
    1: bounded-WHILE  $n'$  cond step  $m0 = \text{RES } X \wedge x \in X \wedge \text{cond } x \wedge \text{step } x =$ 

```

FAIL

```

  by blast
  hence 2:  $n' < \text{Suc } n$  by simp
  from 1 2 have ?case by blast
} moreover {
  assume ?C2
  hence ?case by blast
} ultimately show ?case by blast
qed

```

lemma *bounded-WHILE-RES-rwof*:

```

  assumes bounded-WHILE  $n$  cond step  $m0 = \text{RES } X$ 
  assumes  $x \in X$ 
  shows rwof  $m0$  cond step  $x$ 
  using assms
proof (induction  $n$  arbitrary:  $x X$ )
  case 0 thus ?case by (simp add: rwof.init)
next
  case (Suc  $n$ )

```

from *Suc.prem*s(1) **obtain** Xh **where**

```

  BWN: bounded-WHILE  $n$  cond step  $m0 = \text{RES } Xh$ 
  and  $\forall x \in Xh. \text{cond } x \longrightarrow \text{nofail } (\text{step } x)$ 
  and  $X = \{x'. \exists x \in Xh. \text{cond } x \wedge \text{inres } (\text{step } x) x'\} \cup \{x. x \in Xh \wedge \neg \text{cond } x\}$ 
apply (cases bounded-WHILE  $n$  cond step  $m0$ )
apply simp
apply (rule that, assumption)
apply (force simp: refine-pw-simps pw-eq-iff) []
apply (auto simp add: refine-pw-simps pw-eq-iff split: if-split-asm)
done

```

with *Suc.prem*s(2) **obtain** $xh X'$ **where**

```

   $xh \in Xh$  and
  C:  $\text{cond } xh \longrightarrow \text{step } xh = \text{RES } X' \wedge x \in X'$  and NC:  $\neg \text{cond } xh \longrightarrow x = xh$ 
by (auto simp: nofail-RES-conv)

```

show ?case

```

apply (cases cond  $xh$ )
apply (rule rwof.step[where  $Y = X'$ ])
apply (rule Suc.IH[OF BWN  $\langle xh \in Xh \rangle$ ])
apply assumption
apply (simp-all add: C) [2]

apply (rule Suc.IH[OF BWN])
apply (simp add: NC  $\langle xh \in Xh \rangle$ )
done
qed

```

lemma *rwof-FAIL-imp-WHILE-FAIL*:

assumes RW : $\text{rwof } m0 \text{ cond step } s$
and C : $\text{cond } s$
and S : $\text{step } s = \text{FAIL}$
shows $m0 \ggg \text{WHILE cond step} = \text{FAIL}$
proof –
from $\text{rwof-in-bounded-WHILE}[OF RW]$ **obtain** n **where**
 $\text{RETURN } s \leq \text{bounded-WHILE } n \text{ cond step } m0 \dots$
with C **have** $\text{step } s \leq \text{bounded-WHILE } (Suc\ n) \text{ cond step } m0$
by ($\text{auto simp add: pw-le-iff refine-pw-simps}$)
with S **have** $\text{bounded-WHILE } (Suc\ n) \text{ cond step } m0 = \text{FAIL}$ **by** simp
with $\text{WHILE-bounded-aux1}[of\ cond\ Suc\ n\ step\ m0]$ **show** $?thesis$
by ($\text{simp del: bounded-WHILE.simps}$)
qed

lemma $\text{pw-bounded-WHILE-RES-rwof}$: $\llbracket \text{nofail } (\text{bounded-WHILE } n \text{ cond step } m0);$
 $\text{inres } (\text{bounded-WHILE } n \text{ cond step } m0) \ x \rrbracket \implies \text{rwof } m0 \text{ cond step } x$
using $\text{bounded-WHILE-RES-rwof}[of\ n\ cond\ step\ m0 - x]$
by ($\text{auto simp add: pw-eq-iff}$)

corollary $\text{WHILE-nofail-imp-rwof-nofail}$:
assumes $\text{nofail } (m0 \ggg \text{WHILE cond step})$
assumes RW : $\text{rwof } m0 \text{ cond step } s$
assumes C : $\text{cond } s$
shows $\text{nofail } (\text{step } s)$
apply (rule ccontr) **apply** ($\text{simp add: nofail-def}$)
using $\text{assms rwof-FAIL-imp-WHILE-FAIL}[OF RW C]$
by auto

lemma WHILE-le-WHILEI : $\text{WHILE } b\ f\ s \leq \text{WHILEI } I\ b\ f\ s$
unfolding WHILE-def
by ($\text{rule WHILEI-weaken}$) simp

corollary $\text{WHILEI-nofail-imp-rwof-nofail}$:
assumes NF : $\text{nofail } (m0 \ggg \text{WHILEI } I \text{ cond step})$
assumes RW : $\text{rwof } m0 \text{ cond step } s$
assumes C : $\text{cond } s$
shows $\text{nofail } (\text{step } s)$
proof –
from NF **have** $\text{nofail } (m0 \ggg \text{WHILE cond step})$
using WHILE-le-WHILEI
by ($\text{fastforce simp: pw-le-iff refine-pw-simps}$)
from $\text{WHILE-nofail-imp-rwof-nofail}[OF\ this\ RW\ C]$ **show** $?thesis$.
qed

corollary $\text{WHILET-nofail-imp-rwof-nofail}$:
assumes NF : $\text{nofail } (m0 \ggg \text{WHILET cond step})$
assumes RW : $\text{rwof } m0 \text{ cond step } s$

```

  assumes  $C: cond\ s$ 
  shows  $nofail\ (step\ s)$ 
proof -
  from  $NF$  have  $nofail\ (m0 \gg\ WHILE\ cond\ step)$ 
    using  $WHILEI-le-WHILEIT$  unfolding  $WHILE-def\ WHILET-def$ 
    by ( $fastforce\ simp: pw-le-iff\ refine-pw-simps$ )
  from  $WHILE-nofail-imp-rwof-nofail$ [ $OF\ this\ RW\ C$ ] show  $?thesis$  .
qed

```

```

corollary  $WHILEIT-nofail-imp-rwof-nofail$ :
  assumes  $NF: nofail\ (m0 \gg\ WHILEIT\ I\ cond\ step)$ 
  assumes  $RW: rwof\ m0\ cond\ step\ s$ 
  assumes  $C: cond\ s$ 
  shows  $nofail\ (step\ s)$ 
proof -
  from  $NF$  have  $nofail\ (m0 \gg\ WHILE\ cond\ step)$ 
    using  $WHILE-le-WHILEI\ WHILEI-le-WHILEIT$  unfolding  $WHILE-def$ 
    by ( $fastforce\ simp: pw-le-iff\ refine-pw-simps$ )
  from  $WHILE-nofail-imp-rwof-nofail$ [ $OF\ this\ RW\ C$ ] show  $?thesis$  .
qed

```

```

lemma  $pw-rwof-in-bounded-WHILE$ :
   $rwof\ m0\ cond\ step\ x \implies \exists n. inres\ (bounded-WHILE\ n\ cond\ step\ m0)\ x$ 
  using  $rwof-in-bounded-WHILE$ [ $of\ m0\ cond\ step\ x$ ]
  by ( $auto\ simp\ add: pw-le-iff\ intro: not-nofail-inres$ )

```

WHILE-loops in the nofail-case

```

lemma  $nofail-WHILE-eq-rwof$ :
  assumes  $NF: nofail\ (m0 \gg\ WHILE\ cond\ step)$ 
  shows  $m0 \gg\ WHILE\ cond\ step = SPEC\ (\lambda s. rwof\ m0\ cond\ step\ s \wedge \neg cond\ s)$ 
proof -
  from  $NF$   $WHILE-bounded$ [ $of\ m0\ cond\ step$ ] have  $NF'$ :
     $nofail\ (Sup\ \{filter-nb\ cond\ m\ | m. \exists n. m = bounded-WHILE\ n\ cond\ step\ m0\})$ 
    by ( $auto\ simp: filter-nb-cont$ )

```

```

  show  $?thesis$ 
    unfolding  $WHILE-bounded$ [ $of\ m0\ cond\ step$ ]  $filter-nb-cont$ 
    apply  $simp$ 
  proof ( $rule\ antisym$ )
    show  $Sup\ \{filter-nb\ cond\ m\ | m. \exists n. m = bounded-WHILE\ n\ cond\ step\ m0\}$ 
       $\leq SPEC\ (\lambda s. rwof\ m0\ cond\ step\ s \wedge \neg cond\ s)$ 
      using  $NF'$   $pw-bounded-WHILE-RES-rwof$ [ $of\ -\ cond\ step\ m0$ ]
      by ( $fastforce\ simp: pw-le-iff\ refine-pw-simps$ )
    next
      show  $SPEC\ (\lambda s. rwof\ m0\ cond\ step\ s \wedge \neg cond\ s)$ 

```

```

    < Sup {filter-nb cond m |m. ∃ n. m = bounded-WHILE n cond step m0}
    using NF' pw-rwof-in-bounded-WHILE[of m0 cond step]
    by (fastforce simp: pw-le-iff refine-pw-simps)
  qed
qed

```

```

lemma WHILE-refine-rwof:
  assumes nofail (m ≫ WHILE c f) ⇒ mi ≤ SPEC (λs. rwof m c f s ∧ ¬c
s)
  shows mi ≤ m ≫ WHILE c f
  apply (cases nofail (m ≫ WHILE c f))
  apply (subst nofail-WHILE-eq-rwof, simp, fact)
  apply (simp add: pw-le-iff)
  done

```

```

lemma pw-rwof-init:
  assumes NF: nofail (m0 ≫ WHILE cond step)
  shows inres m0 s ⇒ rwof m0 cond step s and nofail m0
  apply -
  using NF apply (cases m0, simp)
  apply (rule rwof.init, assumption)
  apply auto []

  using NF apply (simp add: refine-pw-simps)
  done

```

```

lemma rwof-init:
  assumes NF: nofail (m0 ≫ WHILE cond step)
  shows m0 ≤ SPEC (rwof m0 cond step)
  using pw-rwof-init[OF NF]
  by (simp add: pw-le-iff refine-pw-simps)

```

```

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' ⇒ rwof m0 cond step s'
  using NF
  apply (clarsimp simp: nofail-RES-conv)
  apply (rule rwof.step[OF R C])
  apply (assumption)
  by simp

```

```

lemma rwof-step':
  [[ nofail (step s); rwof m0 cond step s; cond s ]]
  ⇒ step s ≤ SPEC (rwof m0 cond step)
  using pw-rwof-step'[of step s m0 cond]
  by (simp add: pw-le-iff refine-pw-simps)

```

```

lemma pw-rwof-step:
  assumes NF: nofail ( $m0 \gg \gg \text{WHILE } \text{cond } \text{step}$ )
  assumes R: rwof  $m0 \text{ cond } \text{step } s$ 
  assumes C: cond  $s$ 
  shows inres ( $\text{step } s$ )  $s' \implies \text{rwof } m0 \text{ cond } \text{step } s'$ 
    and nofail ( $\text{step } s$ )
  using WHILE-nofail-imp-rwof-nofail[OF NF R C] pw-rwof-step'[OF - assms(2-)]
  by simp-all

```

```

lemma rwof-step:
   $\llbracket \text{nofail } (m0 \gg \gg \text{WHILE } \text{cond } \text{step}); \text{rwof } m0 \text{ cond } \text{step } s; \text{cond } s \rrbracket$ 
   $\implies \text{step } s \leq \text{SPEC } (\text{rwof } m0 \text{ cond } \text{step})$ 
  using pw-rwof-step[of  $m0 \text{ cond } \text{step } s$ ]
  by (simp add: pw-le-iff refine-pw-simps)

```

```

lemma (in  $-$ ) rwof-leof-init:  $m \leq_n \text{SPEC } (\text{rwof } m \text{ c } f)$ 
  apply rule
  using rwof.init
  apply (fastforce simp: nofail-RES-conv)
  done

```

```

lemma (in  $-$ ) rwof-leof-step:  $\llbracket \text{rwof } m \text{ c } f \text{ s}; \text{c } s \rrbracket \implies f \text{ s} \leq_n \text{SPEC } (\text{rwof } m \text{ c } f)$ 
  apply rule
  using rwof.step
  apply (fastforce simp: nofail-RES-conv)
  done

```

```

lemma (in  $-$ ) rwof-step-refine:
  assumes NF: nofail ( $m0 \gg \gg \text{WHILE } \text{cond } \text{step}$ )
  assumes A: rwof  $m0 \text{ cond } \text{step}' \text{ s}$ 
  assumes FR:  $\bigwedge s. \llbracket \text{rwof } m0 \text{ cond } \text{step } s; \text{cond } s \rrbracket \implies \text{step}' \text{ s} \leq_n \text{step } s$ 
  shows rwof  $m0 \text{ cond } \text{step } s$ 
  apply (rule establish-rwof-invar[OF - - A])
  apply (rule rwof-leof-init)

  apply (rule leof-trans-nofail[OF FR], assumption+)
  apply (rule WHILE-nofail-imp-rwof-nofail[OF NF], assumption+)

  apply (simp add: rwof-leof-step)
  done

```

Adding Invariants

```

lemma WHILE-eq-I-rwof:  $m \gg \gg \text{WHILE } \text{c } f = m \gg \gg \text{WHILEI } (\text{rwof } m \text{ c } f) \text{ c}$ 
  proof (rule antisym)
  have  $m \gg \gg \text{WHILEI } (\text{rwof } m \text{ c } f) \text{ c } f$ 

```

```

≤ ↓↓{(s,s) | s. rwof m c f s}
  (m ≫= WHILE c f)
unfolding WHILE-def
apply (rule bind-refine)
apply (rule intro-prgR[where R={ (s,s) | s. rwof m c f s}])
apply (auto simp: pw-le-iff refine-pw-simps) []
apply (cases m, simp, rule rwof.init, simp-all) []

apply (rule WHILEI-refine)
apply (auto simp: pw-le-iff refine-pw-simps pw-rwof-step')
done
thus m ≫= WHILEI (rwof m c f) c f ≤ m ≫= WHILE c f
  by (simp add: pw-le-iff refine-pw-simps)
next
show m ≫= WHILE c f ≤ m ≫= WHILEI (rwof m c f) c f
  unfolding WHILE-def
  apply (rule bind-mono)
  apply (rule order-refl)
  apply (rule WHILEI-weaken)
  ..
qed

lemma WHILET-eq-I-rwof: m ≫= WHILET c f = m ≫= WHILEIT (rwof m c
f) c f
proof (rule antisym)
  have m ≫= WHILEIT (rwof m c f) c f
  ≤ ↓↓{(s,s) | s. rwof m c f s}
    (m ≫= WHILET c f)
  unfolding WHILET-def
  apply (rule bind-refine)
  apply (rule intro-prgR[where R={ (s,s) | s. rwof m c f s}])
  apply (auto simp: pw-le-iff refine-pw-simps) []
  apply (cases m, simp, rule rwof.init, simp-all) []

  apply (rule WHILEIT-refine)
  apply (auto simp: pw-le-iff refine-pw-simps pw-rwof-step')
  done
thus m ≫= WHILEIT (rwof m c f) c f ≤ m ≫= WHILET c f
  by (simp add: pw-le-iff refine-pw-simps)
next
show m ≫= WHILET c f ≤ m ≫= WHILEIT (rwof m c f) c f
  unfolding WHILET-def
  apply (rule bind-mono)
  apply (rule order-refl)
  apply (rule WHILEIT-weaken)
  ..
qed

```

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*: $\bigwedge s s'. \llbracket (s, s') \in R; \text{rwof } m \text{ c f s}; \text{rwof } m' \text{ c' f' s'} \rrbracket \implies c \text{ s} = c' \text{ s}'$

assumes *S*: $\bigwedge s s'. \llbracket (s, s') \in R; \text{rwof } m \text{ c f s}; \text{rwof } m' \text{ c' f' s}'; c \text{ s}; c' \text{ s}' \rrbracket \implies f \text{ s} \leq_n \Downarrow R (f' \text{ s}')$

shows $\exists s'. (s, s') \in R \wedge \text{rwof } m' \text{ c' f' s}'$

using *RW*

apply (*induction rule: establish-rwof-invar[rotated -1, consumes 1]*)

using *M rwof-init[OF NF]*

apply (*simp only: pw-leof-iff pw-le-iff refine-pw-simps, blast*) []

using *C S rwof-step[OF NF]*

apply (*simp only: pw-leof-iff pw-le-iff refine-pw-simps, blast*) []

done

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*: $\llbracket \text{rwof init cond step } s; \text{cond } s \rrbracket$

$\implies \text{step } s \leq_n \text{SPEC } (\lambda s'. (s, s') \in \text{rwof-rel init cond step})$

unfolding *rwof-rel-def*

by (*auto simp: pw-leof-iff refine-pw-simps*)

lemma *rwof-reachable*:

assumes *rwof init cond step s*

shows $\exists s0. \text{inres init } s0 \wedge (s0, s) \in (\text{rwof-rel init cond step})^*$

using *assms*

apply (*induction*)

unfolding *rwof-rel-def*

apply (*auto intro: rwof.intros*) []

apply *clarsimp*

apply (*intro exI conjI, assumption*)

apply (*rule rtrancl-into-rtrancl, assumption*)

apply (*auto intro: rwof.intros*) []

done

theorem *nofail-WHILEIT-wf-rel*:

assumes *NF*: *nofail (init >> WHILEIT I cond step)*


```

shows wf ((rwof-rel init cond step)-1)
proof (rule ccontr)
  assume ¬wf ((rwof-rel init cond step)-1)
  then obtain f where IP:  $\bigwedge i. (f\ i, f\ (Suc\ i)) \in rwof\text{-rel}\ \text{init}\ \text{cond}\ \text{step}$ 
    unfolding wf-iff-no-infinite-down-chain by auto
  hence rwof init cond step (f 0) by (auto simp: rwof-rel-def)
  then obtain s0 sn where (s0, sn)  $\in (rwof\text{-rel}\ \text{init}\ \text{cond}\ \text{step})^*$    inres init s0
sn = f 0
  using rwof-reachable by metis
  then obtain f where P:  $\bigwedge i. (f\ i, f\ (Suc\ i)) \in rwof\text{-rel}\ \text{init}\ \text{cond}\ \text{step}$  and I:
inres init (f 0)
  using IP
proof (induct arbitrary: f)
  case (step sk sn)
  let ?f = case-nat sk f
  have sk = ?f 0  $\bigwedge i. (?f\ i, ?f\ (Suc\ i)) \in rwof\text{-rel}\ \text{init}\ \text{cond}\ \text{step}$ 
    using step by (auto split: nat.splits)
  then show ?case using step by blast
qed auto

from P have [simp]:  $\bigwedge i. \text{cond}\ (f\ i)$ 
  unfolding rwof-rel-def by auto

from P have SIR:  $\bigwedge i. \text{inres}\ (\text{step}\ (f\ i))\ (f\ (Suc\ i))$ 
  unfolding rwof-rel-def by auto

define F where F = (WHILEI-body ( $\gg$ ) RETURN I cond step)

{
  assume M: trimono F
  define f' where f' x = (if x $\in$ range f then FAIL else gfp F x) for x

  have f'  $\leq$  F f'
    unfolding f'-def
    apply (rule le-funI)
    apply (case-tac x $\in$ range f)
    apply simp-all
    defer
    apply (subst gfp-unfold)
    using M apply (simp add: trimono-def)
    apply (unfold F-def WHILEI-body-def) []
    apply (auto simp: pw-le-iff refine-pw-simps) []

    apply (unfold F-def WHILEI-body-def not-nofail-iff[symmetric]) []
    using SIR
    apply (auto simp: pw-le-iff refine-pw-simps) []
    done
from gfp-upperbound[of f' F, OF this] have  $\bigwedge x. f'\ x \leq \text{gfp}\ F\ x$ 
  by (auto dest: le-funD)

```

```

from this[of f 0] have gfp F (f 0) = FAIL
  unfolding f'-def
  by auto

} note aux=this

have FAIL ≤ WHILEIT I cond step (f 0)
  unfolding WHILET-def WHILEIT-def RECT-def
  apply clarsimp
  apply (subst gfp-eq-flatf-gfp[symmetric])
  apply (simp-all add: trimono-def) [2]
  apply (unfold F-def[symmetric])
  by (rule aux)
with NF I show False by (auto simp: refine-pw-simps)
qed

```

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*: $\bigwedge s i X. \llbracket l \leq i; i < u \rrbracket$

$\implies f(i, (F X s)) \leq SPEC(\lambda(i', r). i' = i + 1 \wedge r = F(\text{insert}(list!i) X) s)$

shows *WHILET* $c f (l, s0)$

$\leq SPEC(\lambda(-, r). r = F \{ list!i \mid i. l \leq i \wedge i < u \} s0)$

apply (*cases* $\neg(l < u)$)

apply (*subst WHILET-unfold*)

using *F0* **apply** (*simp add: CEQ*)

apply (*rule subst, assumption*)

apply (*(fo-rule cong refl)+, auto*) []

apply (*simp*)

apply (*rule WHILET-rule* [

where $R = \text{measure}(\lambda(i, -). u - i)$

and $I = \lambda(i, s). l \leq i \wedge i < u \wedge s = F \{ list!j \mid j. l \leq j \wedge j < i \} s0$

])

apply *rule*

apply *simp*

apply (*subst F0[symmetric]*)

apply (*(fo-rule cong refl)+, auto*) []

apply (*clarsimp simp: CEQ*)

apply (*rule order-trans[OF Fs], simp-all*) []

apply (*auto*

intro!: *fun-cong*[*OF arg-cong*[*of - - F*]]

elim: *less-SucE*) []

```

apply (auto simp: CEQ) []
done

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*

apply (*case-tac* [!] *x*)

apply *simp-all*

done

fun *inf-dres* **where**

inf dFAILi x = *x* |

inf x dFAILi = *x* |

inf (dRETURN a) (dRETURN b) = (*if a=b then dRETURN b else dSUCCEEDi*) |

$inf\ dSUCCEEDi - = dSUCCEEDi \mid$
 $inf - dSUCCEEDi = dSUCCEEDi$

lemma *inf-dres-addsimps*[simp]:
 $inf\ x\ dSUCCEEDi = dSUCCEEDi$
 $inf\ dSUCCEEDi\ x = dSUCCEEDi$
 $inf\ x\ dFAILi = x$
 $inf\ (dRETURN\ v)\ x \neq dFAILi$
apply (*case-tac* [!] x)
apply *simp-all*
done

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**
 $less-eq-dres\ dSUCCEEDi - \longleftrightarrow True \mid$
 $less-eq-dres - dFAILi \longleftrightarrow True \mid$
 $less-eq-dres\ (dRETURN\ (a::'a))\ (dRETURN\ b) \longleftrightarrow a=b \mid$
 $less-eq-dres - - \longleftrightarrow False$

definition *less-dres* **where** $less-dres\ (a::'a\ dres)\ b \longleftrightarrow a \leq b \wedge \neg b \leq a$

lemma *less-eq-dres-split-conv*:
 $a \leq b \longleftrightarrow (case\ (a,b)\ of$
 $\quad (dSUCCEEDi,-) \Rightarrow True$
 $\quad \mid (-,dFAILi) \Rightarrow True$
 $\quad \mid (dRETURN\ (a::'a),\ dRETURN\ b) \Rightarrow a=b$
 $\quad \mid - \Rightarrow False$
 $)$
by (*auto split: dres.split*)

lemma *inf-dres-split-conv*:
 $inf\ a\ b = (case\ (a,b)\ of$
 $\quad (dFAILi,x) \Rightarrow x$
 $\quad \mid (x,dFAILi) \Rightarrow x$
 $\quad \mid (dRETURN\ a,\ dRETURN\ b) \Rightarrow (if\ a=b\ then\ dRETURN\ b\ else\ dSUCCEEDi)$
 $\quad \mid - \Rightarrow dSUCCEEDi)$
by (*auto split: dres.split*)

lemma *sup-dres-split-conv*:

```

sup a b = (case (a,b) of
  (dSUCCEEDi,x) ⇒ x
| (x,dSUCCEEDi) ⇒ x
| (dRETURN a, dRETURN b) ⇒ (if a=b then dRETURN b else dFAILi)
| - ⇒ dFAILi)
by (auto split: dres.split)

```

instance

apply *intro-classes*

supply *less-eq-dres-split-conv[simp] less-dres-def[simp] dres.splits[split]*

supply *inf-dres-split-conv[simp] sup-dres-split-conv[simp] if-splits[split]*

subgoal **by** *auto*

subgoal **by** *auto*

subgoal **by** *auto*

subgoal **by** *auto*

subgoal **by** *auto*

subgoal **by** *auto*

subgoal **by** *auto*

subgoal **by** *auto*

subgoal **by** *auto*

subgoal **by** *auto*

subgoal **by** (*auto simp: Inf-dres-def*)

subgoal **for** *A z*

apply (*clarsimp simp: Inf-dres-def; safe*)

subgoal **by** *force*

subgoal **by** *force*

subgoal **premises** *prems*

using *prems(2-)* **apply** (*drule-tac prems(1)*) **apply** (*drule-tac prems(1)*)

apply (*auto*)

done

subgoal **premises** *prems*

using *prems(2-)* **apply** (*frule-tac prems(1)*)

by (*auto; metis the-equality*)

done

subgoal **by** (*auto simp: Sup-dres-def; metis the-equality*)

subgoal

apply (*clarsimp simp: Sup-dres-def; safe*)

apply *force*

apply *force*

subgoal **premises** *prems*

using *prems(2-)*

apply (*drule-tac prems(1)*)

apply (*drule-tac prems(1)*)

apply (*drule-tac prems(1)*)

apply (*auto*)

done

apply *force*

subgoal **premises** *prems*

```

    using prems(2-) apply (frule-tac prems(1))
    by (auto; metis the-equality)
  done
  subgoal by (auto simp: Inf-dres-def top-dres-def)
  subgoal by (auto simp: Sup-dres-def bot-dres-def)
  done

```

end

abbreviation $dSUCCEED \equiv (bot::'a \text{ dres})$

abbreviation $dFAIL \equiv (top::'a \text{ dres})$

lemma $dres\text{-cases}[cases \text{ type}, case\text{-names } dSUCCEED \text{ dRETURN } dFAIL]:$

```

  obtains  $x=dSUCCEED \mid r$  where  $x=dRETURN \ r \mid \ x=dFAIL$ 
  unfolding bot-dres-def top-dres-def by (cases  $x$ ) auto

```

lemmas $[simp] = dres.case(1,2)[folded \text{ top-dres-def } \text{ bot-dres-def}]$

lemma $dres\text{-order-simps}[simp]:$

```

 $x \leq dSUCCEED \longleftrightarrow x = dSUCCEED$ 
 $dFAIL \leq x \longleftrightarrow x = dFAIL$ 
 $dRETURN \ r \neq dFAIL$ 
 $dRETURN \ r \neq dSUCCEED$ 
 $dFAIL \neq dRETURN \ r$ 
 $dSUCCEED \neq dRETURN \ r$ 
 $dFAIL \neq dSUCCEED$ 
 $dSUCCEED \neq dFAIL$ 
 $x=y \implies \text{inf } (dRETURN \ x) \ (dRETURN \ y) = dRETURN \ y$ 
 $x \neq y \implies \text{inf } (dRETURN \ x) \ (dRETURN \ y) = dSUCCEED$ 
 $x=y \implies \text{sup } (dRETURN \ x) \ (dRETURN \ y) = dRETURN \ y$ 
 $x \neq y \implies \text{sup } (dRETURN \ x) \ (dRETURN \ y) = dFAIL$ 
apply (simp-all add: bot-unique top-unique)
apply (simp-all add: bot-dres-def top-dres-def)
done

```

lemma $dres\text{-Sup-cases}:$

```

obtains  $S \subseteq \{dSUCCEED\}$  and  $\text{Sup } S = dSUCCEED$ 
   $\mid dFAIL \in S$  and  $\text{Sup } S = dFAIL$ 
   $\mid a \ b$  where  $a \neq b \quad dRETURN \ a \in S \quad dRETURN \ b \in S \quad dFAIL \notin S \quad \text{Sup } S =$ 
 $dFAIL$ 
   $\mid a$  where  $S \subseteq \{dSUCCEED, dRETURN \ a\} \quad dRETURN \ a \in S \quad \text{Sup } S =$ 
 $dRETURN \ a$ 

```

proof –

```

  show ?thesis
  apply (cases  $S \subseteq \{dSUCCEED\}$ )
  apply (rule that(1), assumption)
  apply (simp add: Sup-dres-def bot-dres-def)

  apply (cases  $dFAIL \in S$ )

```

```

apply (rule that(2), assumption)
apply (simp add: Sup-dres-def bot-dres-def top-dres-def)

apply (cases  $\exists a b. a \neq b \wedge dRETURN\ a \in S \wedge dRETURN\ b \in S$ )
apply (elim exE conjE)
apply (rule that(3), assumption+)
apply (auto simp add: Sup-dres-def bot-dres-def top-dres-def) []

apply simp
apply (cases  $\exists a. dRETURN\ a \in S$ )
apply (elim exE)
apply (rule-tac a=a in that(4)) []
apply auto [] apply (case-tac xa, auto) []
apply auto []
apply (auto simp add: Sup-dres-def bot-dres-def top-dres-def) []

apply auto apply (case-tac x, auto) []
done

```

qed

lemma *dres-Inf-cases*:

```

obtains  $S \subseteq \{dFAIL\}$  and  $Inf\ S = dFAIL$ 
|  $dSUCCEED \in S$  and  $Inf\ S = dSUCCEED$ 
|  $a\ b$  where  $a \neq b$   $dRETURN\ a \in S$   $dRETURN\ b \in S$   $dSUCCEED \notin S$   $Inf$ 
 $S = dSUCCEED$ 
|  $a$  where  $S \subseteq \{dFAIL, dRETURN\ a\}$   $dRETURN\ a \in S$   $Inf\ S = dRETURN$ 
 $a$ 

```

proof –

```

show ?thesis
apply (cases  $S \subseteq \{dFAIL\}$ )
apply (rule that(1), assumption)
apply (simp add: Inf-dres-def top-dres-def)

apply (cases  $dSUCCEED \in S$ )
apply (rule that(2), assumption)
apply (simp add: Inf-dres-def bot-dres-def top-dres-def)

apply (cases  $\exists a b. a \neq b \wedge dRETURN\ a \in S \wedge dRETURN\ b \in S$ )
apply (elim exE conjE)
apply (rule that(3), assumption+)
apply (auto simp add: Inf-dres-def bot-dres-def top-dres-def) []

apply simp
apply (cases  $\exists a. dRETURN\ a \in S$ )
apply (elim exE)
apply (rule-tac a=a in that(4)) []
apply auto [] apply (case-tac xa, auto) []
apply auto []
apply (auto simp add: Inf-dres-def bot-dres-def top-dres-def) []

```

```

    apply auto apply (case-tac x, auto) []
  done
qed

```

lemma *dres-chain-eq-res*:

```

  is-chain M  $\implies$ 
    dRETURN r  $\in$  M  $\implies$  dRETURN s  $\in$  M  $\implies$  r=s
  by (metis chainD less-eq-dres.simps(4))

```

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
  apply (rule dres-Sup-cases[of M])
  using dres-chain-eq-res[OF CHAIN]
  by auto

```

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
  apply (rule dres-Inf-cases[of M])
  using dres-chain-eq-res[OF CHAIN]
  by auto

```

lemma *dres-internal-simps[simp]*:

```

  dSUCCEEDi = dSUCCEED
  dFAILi = dFAIL
  unfolding top-dres-def bot-dres-def by auto

```

Monad Operations

function *dbind where*

```

  dbind dFAIL - = dFAIL
  | dbind dSUCCEED - = dSUCCEED
  | dbind (dRETURN x) f = f x
  unfolding bot-dres-def top-dres-def
  by pat-completeness auto
  termination by lexicographic-order

```

ad hoc-overloading

```

  Monad-Syntax.bind dbind

```

lemma [code]:


```

dbind (dRETURN x) f = f x
dbind (dSUCCEEDi) f = dSUCCEEDi
dbind (dFAILi) f = dFAILi
by simp-all

```

```

lemma dres-monad1[simp]: dbind (dRETURN x) f = f x
  by (simp)

```

```

lemma dres-monad2[simp]: dbind M dRETURN = M
  apply (cases M)
  apply (auto)
  done

```

```

lemma dres-monad3[simp]: dbind (dbind M f) g = dbind M ( $\lambda x. \text{dbind } (f x) g$ )
  apply (cases M)
  apply auto
  done

```

```

lemmas dres-monad-laws = dres-monad1 dres-monad2 dres-monad3

```

```

lemma dbind-mono[refine-mono]:
   $\llbracket M \leq M'; \bigwedge x. \text{dRETURN } x \leq M \implies f x \leq f' x \rrbracket \implies \text{dbind } M f \leq \text{dbind } M' f'$ 
   $\llbracket \text{flat-ge } M M'; \bigwedge x. \text{flat-ge } (f x) (f' x) \rrbracket \implies \text{flat-ge } (\text{dbind } M f) (\text{dbind } M' f')$ 
  apply (cases M, simp-all)
  apply (cases M', simp-all)
  apply (cases M, simp-all add: flat-ord-def)
  apply (cases M', simp-all)
  done

```

```

lemma dbind-mono1[simp, intro!]: mono dbind
  apply (rule monoI)
  apply (rule le-funI)
  apply (rule dbind-mono)
  by auto

```

```

lemma dbind-mono2[simp, intro!]: mono (dbind M)
  apply (rule monoI)
  apply (rule dbind-mono)
  by (auto dest: le-funD)

```

```

lemma dr-mono-bind:
  assumes MA: mono A and MB:  $\bigwedge s. \text{mono } (B s)$ 
  shows mono ( $\lambda F s. \text{dbind } (A F s) (\lambda s'. B s F s')$ )
  apply (rule monoI)
  apply (rule le-funI)
  apply (rule dbind-mono)
  apply (auto dest: monoD[OF MA, THEN le-funD])  $\square$ 
  apply (auto dest: monoD[OF MB, THEN le-funD])  $\square$ 
  done

```

lemma *dr-mono-bind'*: $\text{mono } (\lambda F s. \text{dbind } (f s) F)$
apply *rule*
apply (*rule le-funI*)
apply (*rule dbind-mono*)
apply (*auto dest: le-funD*)
done

lemmas *dr-mono* = *mono-if dr-mono-bind dr-mono-bind' mono-const mono-id*

lemma [*refine-mono*]:
 $\text{dbind } dSUCCEED f = dSUCCEED$
 $\text{dbind } dFAIL f = dFAIL$
by (*simp-all*)

definition *dASSERT* $\equiv iASSERT dRETURN$

definition *dASSUME* $\equiv iASSUME dRETURN$

interpretation *dres-assert*: *generic-Assert dbind dRETURN dASSERT dASSUME*
apply *unfold-locales*

by (*auto simp: dASSERT-def dASSUME-def*)

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*

$dWHILEIT dWHILEI dWHILET dWHILE$

apply *unfold-locales*

apply (*auto simp: dWHILEIT-def dWHILEI-def dWHILET-def dWHILE-def*)

apply *refine-mono+*

done

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*]:

$dFAIL \neq dSUCCEED$

$\bigwedge x. dRETURN x \neq dSUCCEED$

$\bigwedge m f. [\![m \neq dSUCCEED; \bigwedge x. f x \neq dSUCCEED]\!] \implies \text{dbind } m f \neq dSUCCEED$

$\bigwedge \Phi. dASSERT \Phi \neq dSUCCEED$

$\bigwedge b m1 m2. [\![m1 \neq dSUCCEED; m2 \neq dSUCCEED]\!] \implies \text{If } b m1 m2 \neq dSUCCEED$

$\bigwedge x f. [\![\bigwedge x. f x \neq dSUCCEED]\!] \implies \text{Let } x f \neq dSUCCEED$

```

 $\bigwedge g p. \llbracket \bigwedge x1 x2. g x1 x2 \neq dSUCCEED \rrbracket \implies case\text{-}prod\ g\ p \neq dSUCCEED$ 
 $\bigwedge fn\ fs\ x.$ 
 $\llbracket fn \neq dSUCCEED; \bigwedge v. fs\ v \neq dSUCCEED \rrbracket \implies case\text{-}option\ fn\ fs\ x \neq dSUCCEED$ 
 $\bigwedge fn\ fc\ x.$ 
 $\llbracket fn \neq dSUCCEED; \bigwedge x\ xs. fc\ x\ xs \neq dSUCCEED \rrbracket \implies case\text{-}list\ fn\ fc\ x \neq dSUCCEED$ 
apply (auto split: prod.split option.split list.split)
apply (case-tac m, auto) []
apply (case-tac  $\Phi$ , auto) []
done

```

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  $\neq dSUCCEED$ 
unfolding RECT-def
apply (split if-split)
apply (intro impI conjI)
apply simp-all

```

```

apply (rule flatf.fp-induct-pointwise[where pre= $\lambda-$  -. True and B=B and b=top
and post= $\lambda-$  - m. m  $\neq dSUCCEED$ ])
apply (simp-all add: trimonoD A)
done

```

lemma *dres-ne-bot-dWHILEIT*[refine-transfer]:

```

assumes  $\bigwedge x. f\ x \neq dSUCCEED$ 
shows dWHILEIT I b f s  $\neq dSUCCEED$  using assms
unfolding dWHILEIT-def iWHILEIT-def WHILEI-body-def
apply refine-transfer
done

```

lemma *dres-ne-bot-dWHILET*[refine-transfer]:

```

assumes  $\bigwedge x. f\ x \neq dSUCCEED$ 
shows dWHILET b f s  $\neq dSUCCEED$  using assms
unfolding dWHILET-def iWHILET-def iWHILEIT-def WHILEI-body-def
apply refine-transfer
done

```

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
by *unfold-locales* (*auto simp add: Sup-upper Sup-least*)

lemma *nrec-admissible*: *nrec.admissible* ($\lambda(f::'a \Rightarrow 'b \text{ nres}).$
($\forall x0. f\ x0 \leq \text{SPEC } (P\ x0)$))
apply (*rule cppo.admissibleI*)
apply (*unfold fun-lub-def*)
apply (*intro allI impI*)
apply (*rule Sup-least*)
apply *auto*
done

declaration \langle *Partial-Function.init nrec* @{*term nrec.fixp-fun*}
@{*term nrec.mono-body*} @{*thm nrec.fixp-rule-uc*} @{*thm nrec.fixp-induct-uc*}
(**SOME* @{*thm fixp-induct-nrec*}*) (*NONE*) \rangle

lemma *bind-mono-pfun*[*partial-function-mono*]:
fixes *C* :: 'a \Rightarrow ('b \Rightarrow 'c nres) \Rightarrow ('d nres)
shows
[[*monotone* (*fun-ord* (\leq)) (\leq) *B*;
 $\bigwedge y. \text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda f. C\ y\ f)$]] \Longrightarrow
monotone (*fun-ord* (\leq)) (\leq) ($\lambda f. \text{bind } (B\ f) (\lambda y. C\ y\ f)$)
apply *rule*
apply (*rule Refine-Basic.bind-mono*)
apply (*blast dest: monotoneD*)
done

2.13.2 Deterministic Result Monad

interpretation *drec*:

partial-function-definitions (\leq) *Sup*::'a dres set \Rightarrow 'a dres
by *unfold-locales* (*auto simp add: Sup-upper Sup-least*)

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 –
have [*simp*]: *fun-ord* ((\leq) :: 'b dres \Rightarrow - \Rightarrow -) = (\leq)
apply (*intro ext*)

```

unfolding fun-ord-def le-fun-def
by (rule refl)

have [simp]:  $\bigwedge A x. \{y. \exists f \in A. y = f x\} = (\lambda f. f x) 'A$  by auto

show ?thesis
apply (rule ccpo.admissibleI)
apply (unfold fun-lub-def)
apply clarsimp
apply (drule-tac x=x in point-chainI)
apply (erule dres-Sup-chain-cases)
apply auto
apply (metis (poly-guards-query) SUP-bot-conv(1))
apply (metis (poly-guards-query) SUP-bot-conv(1))
apply metis
done
qed

declaration  $\langle$ Partial-Function.init drec @{term drec.fixp-fun}
  @{term drec.mono-body} @{thm drec.fixp-rule-uc} @{thm drec.fixp-induct-uc}
  NONE $\rangle$ 

lemma drec-bind-mono-pfun[partial-function-mono]:
fixes C :: 'a  $\Rightarrow$  ('b  $\Rightarrow$  'c dres)  $\Rightarrow$  ('d dres)
shows
 $\llbracket$  monotone (fun-ord ( $\leq$ )) ( $\leq$ ) B;
 $\bigwedge y. \text{monotone (fun-ord ( $\leq$ )) ( $\leq$ ) } (\lambda f. C y f) \rrbracket \Longrightarrow$ 
 $\text{monotone (fun-ord ( $\leq$ )) ( $\leq$ ) } (\lambda f. \text{dbind (B f) } (\lambda y. C y f))$ 
apply rule
apply (rule dbind-mono)
apply (blast dest: monotoneD)+
done

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* $r \equiv \text{case } r \text{ of}$

$dSUCCEEDi \Rightarrow SUCCEED$

| $dFAILi \Rightarrow FAIL$

| $dRETURN x \Rightarrow RETURN x$

lemma *nres-of-simps*[*simp*]:

nres-of $dSUCCEED = SUCCEED$

nres-of $dFAIL = FAIL$

nres-of $(dRETURN x) = RETURN x$

apply –

unfolding *nres-of-def bot-dres-def top-dres-def*

by (*auto simp del: dres-internal-simps*)

lemma *nres-of-mono: mono nres-of*

apply (*rule*)

apply (*case-tac x, simp-all, case-tac y, simp-all*)

done

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 $A \Longrightarrow nres-of (Inf A) = Inf (nres-of' A)$

apply *simp-all*

apply (*case-tac a, simp-all, case-tac [!] b, simp-all*) [*1*]

apply (*simp add: less-le*)

apply (*case-tac a, simp-all, case-tac [!] b, simp-all*) [*1*]

apply (*erule dres-Sup-chain-cases*)

apply (*cases A={}*)

apply *auto* []

apply (*subgoal-tac A={dSUCCEED}, auto*) []

apply (*case-tac A={dRETURN r}*)

apply *auto* []

apply (*subgoal-tac A={dSUCCEED, dRETURN r}, auto*) []

apply (*drule imageI[where f=nres-of]*)

```

apply auto []

apply (erule dres-Inf-chain-cases)
apply (cases A={})
apply auto []
apply (subgoal-tac A={dFAIL}, auto) []

apply (case-tac A={dRETURN r})
apply auto []
apply (subgoal-tac A={dFAIL,dRETURN r}, auto) []

apply (drule imageI[where f=nres-of])
apply (auto intro: bot-Inf [symmetric]) []
done

```

```

lemma nres-correctD:
assumes nres-of S ≤ SPEC Φ
shows
S=dRETURN x ⇒ Φ x
S≠dFAIL
using assms apply –
apply (cases S, simp-all)+
done

```

Transfer Theorems Setup

```

interpretation dres: dist-transfer nres-of
apply unfold-locales
apply (simp add: nres-transfer)
done

```

```

lemma nres-of-transfer[refine-transfer]: nres-of x ≤ nres-of x by simp

```

```

lemma det-FAIL[refine-transfer]: nres-of (dFAIL) ≤ FAIL by auto

```

```

lemma det-SUCCEED[refine-transfer]: nres-of (dSUCCEED) ≤ SUCCEED by auto

```

```

lemma det-SPEC: Φ x ⇒ nres-of (dRETURN x) ≤ SPEC Φ by simp

```

```

lemma det-RETURN[refine-transfer]:

```

```

nres-of (dRETURN x) ≤ RETURN x by simp

```

```

lemma det-bind[refine-transfer]:

```

```

assumes nres-of m ≤ M

```

```

assumes  $\bigwedge x. nres-of (f x) \leq F x$ 

```

```

shows nres-of (dbind m f) ≤ bind M F

```

```

using assms

```

```

apply (cases m)

```

```

apply (auto simp: pw-le-iff refine-pw-simps)

```

```

done

```

```

interpretation det-assert: transfer-generic-Assert-remove

```

bind RETURN ASSERT ASSUME
nres-of
by *unfold-locales*

interpretation *det-while: transfer-WHILE*
dbind dRETURN dWHILEIT dWHILEI dWHILET dWHILE
bind RETURN WHILEIT WHILEI WHILET WHILE nres-of
apply *unfold-locales*
apply (*auto intro: det-bind*)
done

2.14.2 Transfer to Plain Function

interpretation *plain: transfer RETURN .*

lemma *plain-RETURN[refine-transfer]: RETURN a ≤ RETURN a* **by** *simp*

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

$\llbracket \text{RETURN } x \leq M; \bigwedge x. \text{RETURN } (f x) \leq F x \rrbracket \implies \text{RETURN } (\text{Let } x f) \leq \text{bind } M F$

apply (*erule order-trans[rotated,OF bind-mono(1)]*)
apply *assumption*
apply *simp*
done

interpretation *plain-assert: transfer-generic-Assert-remove*
bind RETURN ASSERT ASSUME
RETURN
by *unfold-locales*

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 $\text{RETURN } (\text{the-res } S) \leq S'$. Here, *the-res* extracts the result from a deterministic monad. As *the-res* is executable, the above shows that $(\text{the-res } S)$ 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 ≤ S'*
assumes *S ≠ dSUCCEED*
shows *RETURN (the-res S) ≤ S'*

using *assms*
by (*cases S, simp-all*)

The following rule sets up a refinement goal, a transfer goal, and a final optimization goal.

definition *detTAG* $x \equiv x$

lemma *detTAGI*: $x = \text{detTAG } x$ **unfolding** *detTAG-def* **by** *simp*

lemma *autoref-detI*:

assumes $(b,a) \in \langle R \rangle \text{nres-rel}$

assumes $\text{RETURN } c \leq b$

assumes $c = \text{detTAG } d$

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

using *assms*

unfolding *nres-rel-def detTAG-def*

by *simp*

2.14.4 Relator-Based Transfer

definition *dres-nres-rel-internal-def*:

$\text{dres-nres-rel } R \equiv \{(c,a). \text{nres-of } c \leq \Downarrow R a\}$

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

by (*simp add: dres-nres-rel-internal-def relAPP-def*)

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

by (*simp add: dres-nres-rel-def*)

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

by (*simp add: dres-nres-rel-def*)

lemma *dres-nres-rel-as-br-conv*:

$\langle R \rangle \text{dres-nres-rel} = \text{br nres-of } (\lambda-. \text{True}) \text{ } O \langle R \rangle \text{nres-rel}$

unfolding *dres-nres-rel-def br-def nres-rel-def* **by** *auto*

definition *plain-nres-rel-internal-def*:

$\text{plain-nres-rel } R \equiv \{(c,a). \text{RETURN } c \leq \Downarrow R a\}$

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

by (*simp add: plain-nres-rel-internal-def relAPP-def*)

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

by (*simp add: plain-nres-rel-def*)

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

by (*simp add: plain-nres-rel-def*)

lemma *plain-nres-rel-as-br-conv*:

$\langle R \rangle \text{plain-nres-rel} = \text{br RETURN } (\lambda-. \text{True}) \text{ } O \langle R \rangle \text{nres-rel}$

unfolding *plain-nres-rel-def br-def nres-rel-def* **by** *auto*

2.14.5 Post-Simplification Setup

lemma *dres-unit-simps[refine-transfer-post-simp]*:
 $dbind (dRETURN (u::unit)) f = f ()$
by *auto*

lemma *Let-dRETURN-simp[refine-transfer-post-simp]*:
 $Let\ m\ dRETURN = dRETURN\ m$ **by** *auto*

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*:
 $it \subseteq S \implies x \in it \implies S - (it - \{x\}) = insert\ x\ (S - it)$ **by** *auto*

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* $f \equiv \lambda(xs, \sigma). \text{do } \{$
 $\text{let } x = \text{hd } xs; \sigma' \leftarrow f \ x \ \sigma; \text{RETURN } (\text{tl } xs, \sigma')$
 $\}$

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** $\text{FOREACHoci } R \ \Phi \ S \ c \ f \ \sigma \theta$
 $\equiv \text{do } \{$
 $\text{ASSERT } (\text{finite } S);$
 $xs \leftarrow \text{SPEC } (\lambda xs. \text{distinct } xs \wedge S = \text{set } xs \wedge \text{sorted-wrt } R \ xs);$
 $(-, \sigma) \leftarrow \text{WHILEIT}$
 $(\lambda(it, \sigma). \exists xs'. xs = xs' @ it \wedge \Phi (\text{set } it) \ \sigma) (\text{FOREACH-cond } c) (\text{FOREACH-body}$
 $f) (xs, \sigma \theta);$
 $\text{RETURN } \sigma \}$

Foreach with continuation condition and annotated invariant:

definition *FOREACHci* (FOREACH_C^-) **where** $\text{FOREACHci} \equiv \text{FOREACHoci}$
 $(\lambda-. \text{True})$

Foreach with continuation condition:

definition *FOREACHc* (FOREACH_C) **where** $\text{FOREACHc} \equiv \text{FOREACHci}$ ($\lambda-$
 $-. \text{True})$

Foreach with annotated invariant:

definition *FOREACHi* (FOREACH^-) **where**
 $\text{FOREACHi } \Phi \ S \equiv \text{FOREACHci } \Phi \ S \ (\lambda-. \text{True})$

Foreach with annotated invariant and order:

definition *FOREACHoi* (FOREACH_O^-) **where**
 $\text{FOREACHoi } R \ \Phi \ S \equiv \text{FOREACHoci } R \ \Phi \ S \ (\lambda-. \text{True})$

Basic foreach

definition *FOREACH* $S \equiv \text{FOREACHc } S \ (\lambda-. \text{True})$

lemmas *FOREACH-to-oci-unfold*

$= \text{FOREACHci-def } \text{FOREACHc-def } \text{FOREACHi-def } \text{FOREACHoi-def } \text{FOREACH-def}$

2.15.3 Proof Rules

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

assumes $FIN: \text{finite } S$

assumes $I0: I \ S \ \sigma \theta$

assumes *IP*:

$\bigwedge x \text{ it } \sigma. \llbracket c \ \sigma; x \in \text{it}; \text{it} \subseteq S; I \ \text{it} \ \sigma; \forall y \in \text{it} - \{x\}. R \ x \ y;$
 $\forall y \in S - \text{it}. R \ y \ x \rrbracket \implies f \ x \ \sigma \leq \text{SPEC} \ (I \ (\text{it} - \{x\}))$

assumes *III1*: $\bigwedge \sigma. \llbracket I \ \{\} \ \sigma \rrbracket \implies P \ \sigma$

assumes *III2*: $\bigwedge \text{it} \ \sigma. \llbracket \text{it} \neq \{\}; \text{it} \subseteq S; I \ \text{it} \ \sigma; \neg c \ \sigma;$
 $\forall x \in \text{it}. \forall y \in S - \text{it}. R \ y \ x \rrbracket \implies P \ \sigma$

shows *FOREACHoci* $R \ I \ S \ c \ f \ \sigma \ 0 \leq \text{SPEC} \ P$

unfolding *FOREACHoci-def*

apply (*intro refine-vcg*)

apply (*rule FIN*)

apply (*subgoal-tac wf (measure (\lambda(xs, -). length xs))*)

apply *assumption*

apply *simp*

apply (*insert I0, simp add: I0*) []

unfolding *FOREACH-body-def FOREACH-cond-def*

apply (*rule refine-vcg*)⁺

apply (*(simp, elim conjE exE)*)⁺ []

apply (*rename-tac xs'' s xs' \sigma xs*)

defer

apply (*(simp, elim conjE exE)*)⁺

apply (*rename-tac x s xs' \sigma xs*)

defer

proof –

fix *xs' \sigma xs*

assume *I-xs'*: $I \ (\text{set } xs') \ \sigma$

and *sorted-xs-xs'*: *sorted-wrt* $R \ (xs \ @ \ xs')$

and *dist*: *distinct* $xs \ \ \text{distinct } xs' \ \ \text{set } xs \ \cap \ \text{set } xs' = \{\}$

and *S-eq*: $S = \text{set } xs \ \cup \ \text{set } xs'$

from *S-eq* **have** $\text{set } xs' \subseteq S$ **by** *simp*

from *dist S-eq* **have** *S-diff*: $S - \text{set } xs' = \text{set } xs$ **by** *blast*

{ **assume** $xs' \neq []$ $c \ \sigma$

from $\langle xs' \neq [] \rangle$ **obtain** $x \ xs''$ **where** *xs'-eq*: $xs' = x \ \# \ xs''$ **by** (*cases xs', auto*)

have *x-in-xs'*: $x \in \text{set } xs'$ **and** *x-nin-xs''*: $x \notin \text{set } xs''$

using $\langle \text{distinct } xs' \rangle$ **unfolding** *xs'-eq* **by** *simp-all*

from *IP*[*of* $\sigma \ x \ \text{set } xs'$, *OF* $\langle c \ \sigma \rangle \ x\text{-in-}xs' \ \langle \text{set } xs' \subseteq S \rangle \ \langle I \ (\text{set } xs') \ \sigma \rangle]$ *x-nin-xs''*
sorted-xs-xs' S-diff

show $f \ (\text{hd } xs') \ \sigma \leq \text{SPEC}$

$(\lambda x. (\exists xs'a. xs \ @ \ xs' = xs'a \ @ \ \text{tl } xs') \wedge$

$I \ (\text{set } (\text{tl } xs')) \ x)$

apply (*simp add: xs'-eq*)

apply (*simp add: sorted-wrt-append*)

```

done
}

{ assume  $xs' = [] \vee \neg(c \sigma)$ 
  show  $P \sigma$ 
  proof (cases  $xs' = []$ )
    case True thus  $P \sigma$  using  $\langle I \text{ (set } xs') \sigma \rangle$  by (simp add: III)
  next
    case False note  $xs'\text{-neq-nil} = \text{this}$ 
    with  $\langle xs' = [] \vee \neg c \sigma \rangle$  have  $\neg c \sigma$  by simp

    from II2 [of set  $xs' \sigma$ ] S-diff sorted- $xs$ - $xs'$ 
    show  $P \sigma$ 
    apply (simp add:  $xs'\text{-neq-nil}$  S-eq  $\langle \neg c \sigma \rangle$  I- $xs'$ )
    apply (simp add: sorted-wrt-append)
  done
qed
}
qed

```

lemma FOREACHoi-rule[refine-vcg]:

```

assumes FIN: finite S
assumes I0: I S  $\sigma$  0
assumes IP:
   $\bigwedge x \text{ it } \sigma. \llbracket x \in \text{it}; \text{it} \subseteq S; I \text{ it } \sigma; \forall y \in \text{it} - \{x\}. R \ x \ y; \forall y \in S - \text{it}. R \ y \ x \rrbracket \implies f \ x \ \sigma \leq \text{SPEC } (I \ (\text{it} - \{x\}))$ 
assumes III:  $\bigwedge \sigma. \llbracket I \ \{\} \ \sigma \rrbracket \implies P \ \sigma$ 
shows FOREACHoi R I S f  $\sigma$  0  $\leq$  SPEC P
unfolding FOREACHoi-def
by (rule FOREACHoci-rule) (simp-all add: assms)

```

lemma FOREACHci-rule[refine-vcg]:

```

assumes FIN: finite S
assumes I0: I S  $\sigma$  0
assumes IP:
   $\bigwedge x \text{ it } \sigma. \llbracket x \in \text{it}; \text{it} \subseteq S; I \ \text{it } \sigma; c \ \sigma \rrbracket \implies f \ x \ \sigma \leq \text{SPEC } (I \ (\text{it} - \{x\}))$ 
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 FOREACHci I S c f  $\sigma$  0  $\leq$  SPEC P
unfolding FOREACHci-def
by (rule FOREACHoci-rule) (simp-all add: assms)

```

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 $\sigma\theta :: '\sigma$
fixes $\sigma\theta' :: '\sigma a$
fixes $R :: (('S\ set \times '\sigma) \times ('Sa\ set \times '\sigma a))\ set$
assumes INJ : $inj\text{-on } \alpha\ S$
assumes $REFS[simp]$: $S' = \alpha'S$
assumes $RR\text{-}OK$: $\bigwedge x\ y. \llbracket x \in S; y \in S; RR\ x\ y \rrbracket \implies RR'(\alpha\ x)(\alpha\ y)$
assumes $REF0$: $((S, \sigma\theta), (\alpha'S, \sigma\theta')) \in R$
assumes $REFC$: $\bigwedge it\ \sigma\ it'\ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi' it'\ \sigma'; \Phi it\ \sigma;$
 $\forall x \in S - it. \forall y \in it. RR\ x\ y; \forall x \in S' - it'. \forall y \in it'. RR'\ x\ y;$
 $it' = \alpha' it; ((it, \sigma), (it', \sigma')) \in R$
 $\rrbracket \implies c\ \sigma \longleftrightarrow c'\ \sigma'$
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 $REF\text{-}R\text{-}DONE$: $\bigwedge \sigma\ \sigma'. \llbracket \Phi \{\sigma\}; \Phi' \{\sigma'\}; ((\{\sigma\}, \sigma), (\{\sigma'\}, \sigma')) \in R \rrbracket$
 $\implies (\sigma, \sigma') \in R'$
assumes $REF\text{-}R\text{-}BRK$: $\bigwedge it\ \sigma\ it'\ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi it\ \sigma; \Phi' it'\ \sigma';$
 $\forall x \in S - it. \forall y \in it. RR\ x\ y; \forall x \in S' - it'. \forall y \in it'. RR'\ x\ y;$
 $it' = \alpha' it; ((it, \sigma), (it', \sigma')) \in R;$
 $it \neq \{\}; it' \neq \{\};$
 $\neg c\ \sigma; \neg c'\ \sigma'$
 $\rrbracket \implies (\sigma, \sigma') \in R'$
shows $FOREACHoci\ RR\ \Phi\ S\ c\ f\ \sigma\ \theta \leq \Downarrow R' (FOREACHoci\ RR'\ \Phi'\ S'\ c'\ f'\ \sigma\ \theta')$

supply $[[simproc\ del:\ defined\ all]]$
unfolding $FOREACHoci\ def$
apply ($refine\text{-}rcg\ WHILEIT\text{-}refine\text{-}genR$ **where**
 $R' = \{((xs, \sigma), (xs', \sigma')) .$
 $xs' = map\ \alpha\ xs \wedge$
 $set\ xs \subseteq S \wedge set\ xs' \subseteq S' \wedge$
 $(\forall x \in S - set\ xs. \forall y \in set\ xs. RR\ x\ y) \wedge$
 $(\forall x \in S' - set\ xs'. \forall y \in set\ xs'. RR'\ x\ y) \wedge$
 $((set\ xs, \sigma), (set\ xs', \sigma')) \in R\}$
 \rrbracket

```

using REFS INJ apply (auto dest: finite-imageD) []
apply (rule intro-prgR[where R={ $(xs, xs')$  .  $xs' = \text{map } \alpha \text{ } xs$  }])
apply (rule SPEC-refine)
using INJ RR-OK
apply (auto
  simp add: distinct-map sorted-wrt-map
  intro: sorted-wrt-mono-rel[of - RR]) []
using REF0 apply auto []

apply simp apply (rule conjI)
using INJ apply clarsimp
apply (erule map-eq-appendE)
apply clarsimp
apply (rule-tac  $x=l$  in  $exI$ )
apply simp
apply (subst inj-on-map-eq-map[where  $f=\alpha, \text{symmetric}$ ])
apply (rule subset-inj-on, assumption, blast)
apply assumption

apply (simp split: prod.split-asm, elim conjE)
apply (rule REFPHI, auto) []

apply (simp add: FOREACH-cond-def split: prod.split prod.split-asm,
  intro allI impI conj-cong) []
apply auto []
apply (rule REFC, auto) []

unfolding FOREACH-body-def
apply refine-rcg
apply (rule REFSTEP) []
prefer 3 apply auto []
prefer 3 apply auto []
apply simp-all[13]
apply auto []
apply (rename-tac  $a \ b \ d \ e \ f \ g \ h \ i$ )
apply (case-tac  $h$ , auto simp: FOREACH-cond-def) []
apply auto []
apply (auto simp: FOREACH-cond-def) []
apply (clarsimp simp: FOREACH-cond-def)
apply (rule ccontr)
apply (rename-tac  $a \ b \ d \ e \ f$ )
apply (case-tac  $b$ )
apply (auto simp: sorted-wrt-append) [2]

apply (auto simp: FOREACH-cond-def) []
apply (rename-tac  $a \ b \ d \ e$ )
apply (case-tac  $b$ )
apply (auto) [2]

```

```

apply (clarsimp simp: FOREACH-cond-def)
apply (rule ccontr)
apply (rename-tac a b d e f)
apply (case-tac b)
apply (auto simp: sorted-wrt-append) [2]

apply (clarsimp simp: FOREACH-cond-def)
apply (clarsimp simp: FOREACH-cond-def)

apply (clarsimp simp: map-ll)
apply (intro conjI)

apply (rename-tac a b d e f g)
apply (case-tac b, auto) []
apply (rename-tac a b d e f g)
apply (case-tac b, auto) []
apply (rename-tac a b d e f g)
apply (case-tac b, auto simp: sorted-wrt-append) []
apply (rename-tac a b d e f g)
apply (case-tac b, auto simp: sorted-wrt-append) []
apply (rename-tac a b d e f g)
apply (case-tac b, auto) []

apply (rule introR[where R={{(xs,σ),(xs',σ')}.
  xs'=map α xs ∧ Φ (set xs) σ ∧ Φ' (set xs') σ' ∧
  set xs ⊆ S ∧ set xs' ⊆ S' ∧
  (∀ x∈S - set xs. ∀ y∈set xs. RR x y) ∧
  (∀ x∈S' - set xs'. ∀ y∈set xs'. RR' x y) ∧
  ((set xs,σ),(set xs',σ')) ∈ R ∧
  ¬ FOREACH-cond c (xs,σ) ∧ ¬ FOREACH-cond c' (xs',σ')
  }
  ]))
apply auto []
apply (simp add: FOREACH-cond-def, elim conjE)
apply (elim disjE1, simp-all) []
using REF-R-DONE apply auto []
using REF-R-BRK apply auto []
done

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:  $\bigwedge it \ \sigma \ it' \ \sigma'. \llbracket$ 

```


$it' = \alpha it; it \subseteq S; it' \subseteq S'; \Phi' it' \sigma'; \Phi'' it \sigma it' \sigma'; \Phi it \sigma; (\sigma, \sigma') \in R$
 $\Downarrow \Rightarrow c \sigma \longleftrightarrow c' \sigma'$
assumes *REFPHI*: $\bigwedge it \sigma it' \sigma'. \Downarrow$
 $it' = \alpha it; it \subseteq S; it' \subseteq S'; \Phi' it' \sigma'; \Phi'' it \sigma it' \sigma'; (\sigma, \sigma') \in R$
 $\Downarrow \Rightarrow \Phi it \sigma$
assumes *REFSTEP*: $\bigwedge x it \sigma x' it' \sigma'. \Downarrow \forall y \in it - \{x\}. RR x y;$
 $x' = \alpha x; x \in it; x' \in it'; it' = \alpha it; it \subseteq S; it' \subseteq S';$
 $\Phi it \sigma; \Phi' it' \sigma'; \Phi'' it \sigma it' \sigma'; c \sigma; c' \sigma';$
 $(\sigma, \sigma') \in R$
 $\Downarrow \Rightarrow f x \sigma$
 $\leq \Downarrow (\{(\sigma, \sigma'). (\sigma, \sigma') \in R \wedge \Phi'' (it - \{x\}) \sigma (it' - \{x'\}) \sigma'\}) (f' x' \sigma')$
shows *FOREACHoci* $RR \Phi S c f \sigma \theta \leq \Downarrow R (FOREACHoci RR' \Phi' S' c' f' \sigma \theta')$

apply (rule *FOREACHoci-refine-genR*
 where $R = \{((it, \sigma), (it', \sigma')). (\sigma, \sigma') \in R \wedge \Phi'' it \sigma it' \sigma'\}$
)
apply *fact*
apply *fact*
apply *fact*
using *REF0* *REFPHI0* **apply** *blast*
using *REFC* **apply** *auto* \Downarrow
using *REFPHI* **apply** *auto* \Downarrow
using *REFSTEP* **apply** *auto* \Downarrow
apply *auto* \Downarrow
apply *auto* \Downarrow
done

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*: $\bigwedge x y. \Downarrow [x \in S; y \in S; RR x y] \Rightarrow RR' (\alpha x) (\alpha y)$
assumes *REFC*: $\bigwedge it \sigma it' \sigma'. \Downarrow$
 $it' = \alpha it; it \subseteq S; it' \subseteq S'; \Phi' it' \sigma'; \Phi it \sigma; (\sigma, \sigma') \in R$
 $\Downarrow \Rightarrow c \sigma \longleftrightarrow c' \sigma'$
assumes *REFPHI*: $\bigwedge it \sigma it' \sigma'. \Downarrow$
 $it' = \alpha it; it \subseteq S; it' \subseteq S'; \Phi' it' \sigma'; (\sigma, \sigma') \in R$
 $\Downarrow \Rightarrow \Phi it \sigma$
assumes *REFSTEP*: $\bigwedge x it \sigma x' it' \sigma'. \Downarrow \forall y \in it - \{x\}. RR x y;$
 $x' = \alpha x; x \in it; x' \in it'; it' = \alpha it; it \subseteq S; it' \subseteq S';$
 $\Phi it \sigma; \Phi' it' \sigma'; c \sigma; c' \sigma';$
 $(\sigma, \sigma') \in R$
 $\Downarrow \Rightarrow f x \sigma \leq \Downarrow R (f' x' \sigma')$
shows *FOREACHoci* $RR \Phi S c f \sigma \theta \leq \Downarrow R (FOREACHoci RR' \Phi' S' c' f' \sigma \theta')$
apply (rule *FOREACHoci-refine*[**where** $\Phi'' = \lambda \cdot \cdot \cdot \cdot \text{True}$])
apply (rule *assms*) $+$

using *assms* by *simp-all*

lemma *FOREACHoci-weaken*:

assumes *IREF*: $\bigwedge it \sigma. it \subseteq S \implies I it \sigma \implies I' it \sigma$

shows *FOREACHoci* $RR I' S c f \sigma 0 \leq FOREACHoci RR I S c f \sigma 0$

apply (*rule* *FOREACHoci-refine-rcg*[**where** $\alpha=id$ **and** $R=Id$, *simplified*])

apply (*auto intro*: *IREF*)

done

lemma *FOREACHoci-weaken-order*:

assumes *RRREF*: $\bigwedge x y. x \in S \implies y \in S \implies RR x y \implies RR' x y$

shows *FOREACHoci* $RR I S c f \sigma 0 \leq FOREACHoci RR' I S c f \sigma 0$

apply (*rule* *FOREACHoci-refine-rcg*[**where** $\alpha=id$ **and** $R=Id$, *simplified*])

apply (*auto intro*: *RRREF*)

done

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 $\sigma 0 :: 'S$

fixes $\sigma 0' :: 'Sa$

fixes $R :: (('S \text{ set} \times 'S) \times ('Sa \text{ set} \times 'Sa)) \text{ set}$

assumes *INJ*: *inj-on* αS

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* $RR \Phi S f \sigma 0 \leq \Downarrow R' (FOREACHoi RR' \Phi' S' f' \sigma 0')$

unfolding *FOREACHoi-def*

apply (*rule* *FOREACHoci-refine-genR*)

apply (*fact* | *simp*)⁺

using *REFSTEP* **apply** *auto* \square

apply (*fact* | *simp*)+
done

lemma *FOREACHoi-refine*:

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes *INJ*: *inj-on* α S
assumes *REFS*: $S' = \alpha 'S$
assumes *REF0*: $(\sigma 0, \sigma 0') \in R$
assumes *RR-OK*: $\bigwedge x y. \llbracket x \in S; y \in S; RR \ x \ y \rrbracket \Longrightarrow RR' (\alpha \ x) (\alpha \ y)$
assumes *REFPHI0*: $\Phi'' S \ \sigma 0 \ (\alpha 'S) \ \sigma 0'$
assumes *REFPHI*: $\bigwedge it \ \sigma \ it' \ \sigma'. \llbracket$
 $it' = \alpha 'it; it \subseteq S; it' \subseteq S'; \Phi' \ it' \ \sigma'; \Phi'' \ it \ \sigma \ it' \ \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \Longrightarrow \Phi \ it \ \sigma$
assumes *REFSTEP*: $\bigwedge x \ it \ \sigma \ x' \ it' \ \sigma'. \llbracket \forall y \in it - \{x\}. RR \ x \ y;$
 $x' = \alpha \ x; x \in it; x' \in it'; it' = \alpha 'it; it \subseteq S; it' \subseteq S';$
 $\Phi \ it \ \sigma; \Phi' \ it' \ \sigma'; \Phi'' \ it \ \sigma \ it' \ \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \Longrightarrow f \ x \ \sigma$
 $\leq \Downarrow (\{(\sigma, \sigma'). (\sigma, \sigma') \in R \wedge \Phi'' (it - \{x\}) \ \sigma (it' - \{x'\}) \ \sigma'\}) (f' \ x' \ \sigma')$
shows *FOREACHoi* $RR \ \Phi \ S \ f \ \sigma 0 \leq \Downarrow R (FOREACHoi \ RR' \ \Phi' \ S' \ f' \ \sigma 0')$
unfolding *FOREACHoi-def*
apply (*rule* *FOREACHoi-refine* [*of* α - - - - - Φ'])
apply (*simp-all add: assms*)
done

lemma *FOREACHoi-refine-rcg[refine]*:

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes *INJ*: *inj-on* α S
assumes *REFS*: $S' = \alpha 'S$
assumes *REF0*: $(\sigma 0, \sigma 0') \in R$
assumes *RR-OK*: $\bigwedge x y. \llbracket x \in S; y \in S; RR \ x \ y \rrbracket \Longrightarrow RR' (\alpha \ x) (\alpha \ y)$
assumes *REFPHI*: $\bigwedge it \ \sigma \ it' \ \sigma'. \llbracket$
 $it' = \alpha 'it; it \subseteq S; it' \subseteq S'; \Phi' \ it' \ \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \Longrightarrow \Phi \ it \ \sigma$
assumes *REFSTEP*: $\bigwedge x \ it \ \sigma \ x' \ it' \ \sigma'. \llbracket \forall y \in it - \{x\}. RR \ x \ y;$
 $x' = \alpha \ x; x \in it; x' \in it'; it' = \alpha 'it; it \subseteq S; it' \subseteq S';$
 $\Phi \ it \ \sigma; \Phi' \ it' \ \sigma'; (\sigma, \sigma') \in R$
 $\rrbracket \Longrightarrow f \ x \ \sigma \leq \Downarrow R (f' \ x' \ \sigma')$
shows *FOREACHoi* $RR \ \Phi \ S \ f \ \sigma 0 \leq \Downarrow R (FOREACHoi \ RR' \ \Phi' \ S' \ f' \ \sigma 0')$
apply (*rule* *FOREACHoi-refine* [**where** $\Phi'' = \lambda$ - - - . *True*])
apply (*rule assms*)+
using *assms* **by** *simp-all*

lemma *FOREACHci-refine-genR*:

fixes $\alpha :: 'S \Rightarrow 'Sa$ — Abstraction mapping of elements
fixes $S :: 'S \text{ set}$ — Concrete set

```

fixes  $S' :: 'Sa\ set$  — Abstract set
fixes  $\sigma 0 :: 'σ$ 
fixes  $\sigma 0' :: 'σa$ 
fixes  $R :: (('S\ set \times 'σ) \times ('Sa\ set \times 'σa))\ set$ 
assumes INJ: inj-on  $\alpha\ S$ 
assumes REFS[simp]:  $S' = \alpha\ 'S$ 
assumes REF0:  $((S, \sigma 0), (\alpha\ 'S, \sigma 0')) \in R$ 
assumes REFC:  $\bigwedge it\ \sigma\ it'\ \sigma'. \llbracket$ 
   $it \subseteq S; it' \subseteq S'; \Phi\ it'\ \sigma'; \Phi\ it\ \sigma;$ 
   $it' = \alpha\ it; ((it, \sigma), (it', \sigma')) \in R$ 
]]  $\implies c\ \sigma \longleftrightarrow c'\ \sigma'$ 
assumes REFPHI:  $\bigwedge it\ \sigma\ it'\ \sigma'. \llbracket$ 
   $it \subseteq S; it' \subseteq S'; \Phi\ it'\ \sigma';$ 
   $it' = \alpha\ it; ((it, \sigma), (it', \sigma')) \in R$ 
]]  $\implies \Phi\ it\ \sigma$ 
assumes REFSTEP:  $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'. \llbracket$ 
   $it \subseteq S; it' \subseteq S'; \Phi\ it\ \sigma; \Phi'\ it'\ \sigma';$ 
   $x' = \alpha\ x; it' = \alpha\ it; ((it, \sigma), (it', \sigma')) \in R;$ 
   $x \in it; x' \in it';$ 
   $c\ \sigma; c'\ \sigma'$ 
]]  $\implies f\ x\ \sigma$ 
   $\leq \Downarrow (\{(\sigma, \sigma'). ((it - \{x\}, \sigma), (it' - \{x'\}, \sigma')) \in R\}) (f'\ x'\ \sigma')$ 
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:  $\bigwedge it\ \sigma\ it'\ \sigma'. \llbracket$ 
   $it \subseteq S; it' \subseteq S'; \Phi\ it\ \sigma; \Phi'\ it'\ \sigma';$ 
   $it' = \alpha\ it; ((it, \sigma), (it', \sigma')) \in R;$ 
   $it \neq \{\}; it' \neq \{\};$ 
   $\neg c\ \sigma; \neg c'\ \sigma'$ 
]]  $\implies (\sigma, \sigma') \in R'$ 
shows FOREACHci  $\Phi\ S\ c\ f\ \sigma 0 \leq \Downarrow R' (FOREACHci\ \Phi'\ S'\ c'\ f'\ \sigma 0')$ 
unfolding FOREACHci-def
apply (rule FOREACHoci-refine-genR)
apply (fact[simp])+
using REFC apply auto  $\llbracket$ 
using REFPHI apply auto  $\llbracket$ 
using REFSTEP apply auto  $\llbracket$ 
apply (fact[simp])+
using REF-R-BRK apply auto  $\llbracket$ 
done

```

lemma *FOREACHci-refine*:

```

fixes  $\alpha :: 'S \Rightarrow 'Sa$ 
fixes  $S :: 'S\ set$ 
fixes  $S' :: 'Sa\ set$ 
assumes INJ: 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 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  $\Phi \ S \ c \ f \ \sigma \ 0 \leq \Downarrow R (FOREACHci \ \Phi' \ S' \ c' \ f' \ \sigma \ 0')$ 
unfolding FOREACHci-def
apply (rule FOREACHoci-refine [of  $\alpha$  - - - - -  $\Phi'$ ])
apply (simp-all add: asms)
done

```

lemma FOREACHci-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:  $(\sigma \ 0, \sigma \ 0') \in R$ 
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  $\Phi \ S \ c \ f \ \sigma \ 0 \leq \Downarrow R (FOREACHci \ \Phi' \ S' \ c' \ f' \ \sigma \ 0')$ 
apply (rule FOREACHci-refine[where  $\Phi'' = \lambda$  - - - . True])
apply (rule asms)+
using asms by auto

```

lemma FOREACHci-weaken:

```

assumes IREF:  $\bigwedge it \ \sigma. it \subseteq S \implies I \ it \ \sigma \implies I' \ it \ \sigma$ 
shows FOREACHci  $I' \ S \ c \ f \ \sigma \ 0 \leq FOREACHci \ I \ S \ c \ f \ \sigma \ 0$ 
apply (rule FOREACHci-refine-rcg[where  $\alpha = id$  and  $R = Id$ , simplified])
apply (auto intro: IREF)
done

```

lemma FOREACHi-rule[refine-vcg]:

assumes *FIN*: *finite S*
assumes *I0*: $I\ S\ \sigma\ 0$
assumes *IP*:
 $\bigwedge x\ it\ \sigma. \llbracket x \in it; it \subseteq S; I\ it\ \sigma \rrbracket \implies f\ x\ \sigma \leq SPEC\ (I\ (it - \{x\}))$
assumes *II*: $\bigwedge \sigma. \llbracket I\ \{\} \ \sigma \rrbracket \implies P\ \sigma$
shows *FOREACHi* $I\ S\ f\ \sigma\ 0 \leq SPEC\ P$
unfolding *FOREACHi-def*
apply (*rule FOREACHci-rule*[*of S I*])
using *assms by auto*

lemma *FOREACHc-rule*:

assumes *FIN*: *finite S*
assumes *I0*: $I\ S\ \sigma\ 0$
assumes *IP*:
 $\bigwedge x\ it\ \sigma. \llbracket x \in it; it \subseteq S; I\ it\ \sigma; c\ \sigma \rrbracket \implies f\ x\ \sigma \leq SPEC\ (I\ (it - \{x\}))$
assumes *III*: $\bigwedge \sigma. \llbracket I\ \{\} \ \sigma \rrbracket \implies P\ \sigma$
assumes *II2*: $\bigwedge it\ \sigma. \llbracket it \neq \{\}; it \subseteq S; I\ it\ \sigma; \neg c\ \sigma \rrbracket \implies P\ \sigma$
shows *FOREACHc* $S\ c\ f\ \sigma\ 0 \leq SPEC\ P$
unfolding *FOREACHc-def*
apply (*rule order-trans*[*OF FOREACHci-weaken*], *rule TrueI*)
apply (*rule FOREACHci-rule*[**where** $I=I$])
using *assms by auto*

lemma *FOREACH-rule*:

assumes *FIN*: *finite S*
assumes *I0*: $I\ S\ \sigma\ 0$
assumes *IP*:
 $\bigwedge x\ it\ \sigma. \llbracket x \in it; it \subseteq S; I\ it\ \sigma \rrbracket \implies f\ x\ \sigma \leq SPEC\ (I\ (it - \{x\}))$
assumes *II*: $\bigwedge \sigma. \llbracket I\ \{\} \ \sigma \rrbracket \implies P\ \sigma$
shows *FOREACH* $S\ f\ \sigma\ 0 \leq SPEC\ P$
unfolding *FOREACH-def FOREACHc-def*
apply (*rule order-trans*[*OF FOREACHci-weaken*], *rule TrueI*)
apply (*rule FOREACHci-rule*[**where** $I=I$])
using *assms by auto*

lemma *FOREACHc-refine-genR*:

fixes $\alpha :: 'S \Rightarrow 'Sa$ — Abstraction mapping of elements
fixes $S :: 'S\ set$ — Concrete set
fixes $S' :: 'Sa\ set$ — Abstract set
fixes $\sigma\ 0 :: ' \sigma$
fixes $\sigma\ 0' :: ' \sigma a$
fixes $R :: (('S\ set \times ' \sigma) \times ('Sa\ set \times ' \sigma a))\ set$
assumes *INJ*: *inj-on* $\alpha\ S$
assumes *REFS*[*simp*]: $S' = \alpha\ S$
assumes *REF0*: $((S, \sigma\ 0), (\alpha\ S, \sigma\ 0')) \in R$
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S';$
 $it' = \alpha\ it; ((it, \sigma), (it', \sigma')) \in R$

```

]]  $\Rightarrow c \sigma \longleftrightarrow c' \sigma'$ 
assumes REFSTEP:  $\bigwedge x \text{ it } \sigma \ x' \ \text{it}' \ \sigma'. \ [$ 
   $\text{it} \subseteq S; \ \text{it}' \subseteq S';$ 
   $x' = \alpha \ x; \ \text{it}' = \alpha \ \text{it}; \ ((\text{it}, \sigma), (\text{it}', \sigma')) \in R;$ 
   $x \in \text{it}; \ x' \in \text{it}';$ 
   $c \ \sigma; \ c' \ \sigma'$ 
]]  $\Rightarrow f \ x \ \sigma$ 
 $\leq \Downarrow (\{(\sigma, \sigma'). ((\text{it} - \{x\}, \sigma), (\text{it}' - \{x'\}, \sigma')) \in R\}) (f' \ x' \ \sigma')$ 
assumes REF-R-DONE:  $\bigwedge \sigma \ \sigma'. \ [ (\{\}, \sigma), (\{\}, \sigma') \in R ]$ 
 $\Rightarrow (\sigma, \sigma') \in R'$ 
assumes REF-R-BRK:  $\bigwedge \text{it } \sigma \ \text{it}' \ \sigma'. \ [$ 
   $\text{it} \subseteq S; \ \text{it}' \subseteq S';$ 
   $\text{it}' = \alpha \ \text{it}; \ ((\text{it}, \sigma), (\text{it}', \sigma')) \in R;$ 
   $\text{it} \neq \{\}; \ \text{it}' \neq \{\};$ 
   $\neg c \ \sigma; \ \neg c' \ \sigma'$ 
]]  $\Rightarrow (\sigma, \sigma') \in R'$ 
shows FOREACHc S c f  $\sigma \theta \leq \Downarrow R' (FOREACHc S' c' f' \sigma \theta')$ 
unfolding FOREACHc-def
apply (rule FOREACHci-refine-genR)
apply simp-all
apply (fact|simp)+
using REFC apply auto []
using REFSTEP apply auto []
using REF-R-DONE apply auto []
using REF-R-BRK apply auto []
done

```

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:  $\bigwedge \text{it } \sigma \ \text{it}' \ \sigma'. \ [$ 
   $\text{it}' = \alpha \ \text{it}; \ \text{it} \subseteq S; \ \text{it}' \subseteq S'; \ \Phi'' \ \text{it } \sigma \ \text{it}' \ \sigma'; \ (\sigma, \sigma') \in R$ 
]]  $\Rightarrow c \ \sigma \longleftrightarrow c' \ \sigma'$ 
assumes REFSTEP:  $\bigwedge x \ \text{it } \sigma \ x' \ \text{it}' \ \sigma'. \ [$ 
   $x' = \alpha \ x; \ x \in \text{it}; \ x' \in \text{it}'; \ \text{it}' = \alpha \ \text{it}; \ \text{it} \subseteq S; \ \text{it}' \subseteq S';$ 
   $\Phi'' \ \text{it } \sigma \ \text{it}' \ \sigma'; \ c \ \sigma; \ c' \ \sigma'; \ (\sigma, \sigma') \in R$ 
]]  $\Rightarrow f \ x \ \sigma$ 
 $\leq \Downarrow (\{(\sigma, \sigma'). (\sigma, \sigma') \in R \wedge \Phi'' (\text{it} - \{x\}) \ \sigma \ (\text{it}' - \{x'\}) \ \sigma'\}) (f' \ x' \ \sigma')$ 
shows FOREACHc S c f  $\sigma \theta \leq \Downarrow R (FOREACHc S' c' f' \sigma \theta')$ 
unfolding FOREACHc-def
apply (rule FOREACHci-refine[where  $\Phi'' = \Phi''$ , OF INJ REFS REF0 REFPHI0])
apply (erule (4) REFC)
apply (rule TrueI)

```

apply (*erule* (9) *REFSTEP*)
done

lemma *FOREACHc-refine-rcg*[*refine*]:

fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S \text{ set}$
fixes $S' :: 'Sa \text{ set}$
assumes *INJ*: *inj-on* α S
assumes *REFS*: $S' = \alpha' S$
assumes *REF0*: $(\sigma 0, \sigma 0') \in R$
assumes *REFC*: $\bigwedge it \ \sigma \ it' \ \sigma'. \llbracket$
 $it' = \alpha' it; it \subseteq S; it' \subseteq S'; (\sigma, \sigma') \in R$
 $\rrbracket \Rightarrow c \ \sigma \longleftrightarrow c' \ \sigma'$
assumes *REFSTEP*: $\bigwedge x \ it \ \sigma \ x' \ it' \ \sigma'. \llbracket$
 $x' = \alpha \ x; x \in it; x' \in it'; it' = \alpha' it; it \subseteq S; it' \subseteq S'; c \ \sigma; c' \ \sigma';$
 $(\sigma, \sigma') \in R$
 $\rrbracket \Rightarrow f \ x \ \sigma \leq \Downarrow R \ (f' \ x' \ \sigma')$
shows *FOREACHc* $S \ c \ f \ \sigma 0 \leq \Downarrow R \ (FOREACHc \ S' \ c' \ f' \ \sigma 0')$
unfolding *FOREACHc-def*
apply (*rule* *FOREACHci-refine-rcg*)
apply (*rule* *assms*)
using *assms* **by** *auto*

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 $\sigma 0 :: 'S$
fixes $\sigma 0' :: 'Sa$
fixes $R :: (('S \text{ set} \times 'S) \times ('Sa \text{ set} \times 'Sa)) \text{ set}$
assumes *INJ*: *inj-on* α S
assumes *REFS*[*simp*]: $S' = \alpha' S$
assumes *REF0*: $((S, \sigma 0), (\alpha' S, \sigma 0')) \in R$
assumes *REFPHI*: $\bigwedge it \ \sigma \ it' \ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi' \ it' \ \sigma';$
 $it' = \alpha' it; ((it, \sigma), (it', \sigma')) \in R$
 $\rrbracket \Rightarrow \Phi \ it \ \sigma$
assumes *REFSTEP*: $\bigwedge x \ it \ \sigma \ x' \ it' \ \sigma'. \llbracket$
 $it \subseteq S; it' \subseteq S'; \Phi \ it \ \sigma; \Phi' \ it' \ \sigma';$
 $x' = \alpha \ x; it' = \alpha' it; ((it, \sigma), (it', \sigma')) \in R;$
 $x \in it; x' \in it'$
 $\rrbracket \Rightarrow f \ x \ \sigma$
 $\leq \Downarrow (\{(\sigma, \sigma'). ((it - \{x\}, \sigma), (it' - \{x'\}, \sigma')) \in R\}) \ (f' \ x' \ \sigma')$
assumes *REF-R-DONE*: $\bigwedge \sigma \ \sigma'. \llbracket \Phi \ \{\} \ \sigma; \Phi' \ \{\} \ \sigma'; ((\{\}, \sigma), (\{\}, \sigma')) \in R \rrbracket$
 $\Rightarrow (\sigma, \sigma') \in R'$
shows *FOREACHi* $\Phi \ S \ f \ \sigma 0 \leq \Downarrow R' \ (FOREACHi \ \Phi' \ S' \ f' \ \sigma 0')$
unfolding *FOREACHi-def*
apply (*rule* *FOREACHci-refine-genR*)
apply (*fact*[*simp*])
using *assms* **by** *auto*


```

using REFSTEP apply auto []
apply (fact|simp)+
done

```

lemma FOREACHi-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 REFPHI0:  $\Phi'' S \sigma 0 (\alpha 'S) \sigma 0'$ 
assumes REFPHI:  $\bigwedge it \sigma it' \sigma'. \llbracket$ 
   $it' = \alpha 'it; it \subseteq S; it' \subseteq S'; \Phi' it' \sigma'; \Phi'' it \sigma it' \sigma'; (\sigma, \sigma') \in R$ 
 $\rrbracket \Rightarrow \Phi it \sigma$ 
assumes REFSTEP:  $\bigwedge x it \sigma x' it' \sigma'. \llbracket$ 
   $x' = \alpha x; x \in it; x' \in it'; it' = \alpha 'it; it \subseteq S; it' \subseteq S';$ 
   $\Phi it \sigma; \Phi' it' \sigma'; \Phi'' it \sigma it' \sigma';$ 
   $(\sigma, \sigma') \in R$ 
 $\rrbracket \Rightarrow f x \sigma$ 
   $\leq \Downarrow (\{(\sigma, \sigma'). (\sigma, \sigma') \in R \wedge \Phi'' (it - \{x\}) \sigma (it' - \{x'\}) \sigma'\}) (f' x' \sigma')$ 
shows FOREACHi  $\Phi S f \sigma 0 \leq \Downarrow R (FOREACHi \Phi' S' f' \sigma 0')$ 
unfolding FOREACHi-def
apply (rule FOREACHci-refine[where  $\Phi'' = \Phi''$ , OF INJ REFS REF0 REFPHI0])
apply (rule refl)
apply (erule (5) REFPHI)
apply (erule (9) REFSTEP)
done

```

lemma FOREACHi-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:  $(\sigma 0, \sigma 0') \in R$ 
assumes REFPHI:  $\bigwedge it \sigma it' \sigma'. \llbracket$ 
   $it' = \alpha 'it; it \subseteq S; it' \subseteq S'; \Phi' it' \sigma'; (\sigma, \sigma') \in R$ 
 $\rrbracket \Rightarrow \Phi it \sigma$ 
assumes REFSTEP:  $\bigwedge x it \sigma x' it' \sigma'. \llbracket$ 
   $x' = \alpha x; x \in it; x' \in it'; it' = \alpha 'it; it \subseteq S; it' \subseteq S';$ 
   $\Phi it \sigma; \Phi' it' \sigma';$ 
   $(\sigma, \sigma') \in R$ 
 $\rrbracket \Rightarrow f x \sigma \leq \Downarrow R (f' x' \sigma')$ 
shows FOREACHi  $\Phi S f \sigma 0 \leq \Downarrow R (FOREACHi \Phi' S' f' \sigma 0')$ 
unfolding FOREACHi-def
apply (rule FOREACHci-refine-rcg)
apply (rule assms)+

```

using *assms apply auto*
done

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 $\sigma 0 :: ' \sigma$

fixes $\sigma 0' :: ' \sigma a$

fixes $R :: (('S \text{ set} \times ' \sigma) \times ('Sa \text{ set} \times ' \sigma a)) \text{ set}$

assumes *INJ*: *inj-on* α S

assumes *REFS[simp]*: $S' = \alpha 'S$

assumes *REF0*: $((S, \sigma 0), (\alpha 'S, \sigma 0')) \in R$

assumes *REFSTEP*: $\bigwedge x \text{ it } \sigma \ x' \ \text{it}' \ \sigma'. \llbracket$

$\text{it} \subseteq S; \text{it}' \subseteq S';$

$x' = \alpha \ x; \text{it}' = \alpha \ \text{it}; ((\text{it}, \sigma), (\text{it}', \sigma')) \in R;$

$x \in \text{it}; x' \in \text{it}'$

$\rrbracket \Rightarrow f \ x \ \sigma$

$\leq \Downarrow (\{(\sigma, \sigma'). ((\text{it} - \{x\}, \sigma), (\text{it}' - \{x'\}, \sigma')) \in R\}) (f' \ x' \ \sigma')$

assumes *REF-R-DONE*: $\bigwedge \sigma \ \sigma'. \llbracket ((\{\}, \sigma), (\{\}, \sigma')) \in R \rrbracket$

$\Rightarrow (\sigma, \sigma') \in R'$

shows *FOREACH* $S \ f \ \sigma 0 \leq \Downarrow R' (FOREACH \ S' \ f' \ \sigma 0')$

unfolding *FOREACH-def*

apply (rule *FOREACHc-refine-genR*)

apply (fact *simp*) +

using *REFSTEP apply auto* \llbracket

apply (fact *simp*) +

done

lemma *FOREACH-refine*:

fixes $\alpha :: 'S \Rightarrow 'Sa$

fixes $S :: 'S \text{ set}$

fixes $S' :: 'Sa \text{ set}$

assumes *INJ*: *inj-on* α 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 *REFSTEP*: $\bigwedge x \ \text{it} \ \sigma \ x' \ \text{it}' \ \sigma'. \llbracket$

$x' = \alpha \ x; x \in \text{it}; x' \in \text{it}'; \text{it}' = \alpha \ \text{it}; \text{it} \subseteq S; \text{it}' \subseteq S';$

$\Phi'' \ \text{it} \ \sigma \ \text{it}' \ \sigma'; (\sigma, \sigma') \in R$

$\rrbracket \Rightarrow f \ x \ \sigma$

$\leq \Downarrow (\{(\sigma, \sigma'). (\sigma, \sigma') \in R \wedge \Phi'' (\text{it} - \{x\}) \ \sigma \ (\text{it}' - \{x'\}) \ \sigma'\}) (f' \ x' \ \sigma')$

shows *FOREACH* $S \ f \ \sigma 0 \leq \Downarrow R (FOREACH \ S' \ f' \ \sigma 0')$

unfolding *FOREACH-def*

apply (rule *FOREACHc-refine*[**where** $\Phi'' = \Phi''$, *OF INJ REFS REF0 REFPHI0*])

apply (rule *refl*)

apply (erule (γ) *REFSTEP*)

done

lemma *FOREACH-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*: $(\sigma 0, \sigma 0') \in R$
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* $S f \sigma 0 \leq \Downarrow R (\text{FOREACH } S' f' \sigma 0')$
unfolding *FOREACH-def*
apply (*rule* *FOREACHc-refine-rcg*)
apply (*rule* *assms*)
using *assms* **by** *auto*

lemma *FOREACHci-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*: $(\sigma 0, \sigma 0') \in R$
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* $S c f \sigma 0 \leq \Downarrow R (\text{FOREACHci } \Phi' S' c' f' \sigma 0')$
unfolding *FOREACHc-def*
apply (*rule* *FOREACHci-refine-rcg*)
apply (*rule* *assms*)
apply (*rule* *assms*)
apply (*rule* *assms*)
apply (*rule* *assms*)
apply (*erule* (\wedge) *REFC*)
apply (*rule* *TrueI*)
apply (*rule* *REFSTEP*, *assumption*+)
done

lemma *FOREACHi-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*: $(\sigma 0, \sigma 0') \in R$
assumes *REFSTEP*: $\bigwedge x \text{ it } \sigma \ x' \ \text{it}' \ \sigma'. \llbracket$
 $x' = \alpha \ x; x \in \text{it}; x' \in \text{it}'; \text{it}' = \alpha' \ \text{it}; \text{it} \subseteq S; \text{it}' \subseteq S';$
 $\Phi' \ \text{it}' \ \sigma';$
 $(\sigma, \sigma') \in R$
 $\rrbracket \implies f \ x \ \sigma \leq \Downarrow R \ (f' \ x' \ \sigma')$
shows *FOREACH S f* $\sigma 0 \leq \Downarrow R \ (\text{FOREACHi } \Phi' \ S' \ f' \ \sigma 0')$
unfolding *FOREACH-def FOREACHi-def*
apply (rule *FOREACHci-refine-rec'*)
apply (rule *assms*)
apply *simp*
apply (rule *REFSTEP, assumption+*)
done

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*: $I \ S \ \sigma 0$
assumes *IP*:
 $\bigwedge x \ \text{it} \ \sigma. \llbracket c \ \sigma; x \in \text{it}; \text{it} \subseteq S; I \ \text{it} \ \sigma; \forall y \in \text{it} - \{x\}. R \ x \ y;$
 $\forall y \in S - \text{it}. R \ y \ x \rrbracket \implies f \ x \ \sigma \leq \text{SPEC} \ (I \ (\text{it} - \{x\}))$
assumes *III*: $\bigwedge \sigma. \llbracket I \ \{\} \ \sigma; c \ \sigma \rrbracket \implies P \ \sigma$
assumes *II2*: $\bigwedge \text{it} \ \sigma. \llbracket \text{it} \subseteq S; I \ \text{it} \ \sigma; \neg c \ \sigma;$
 $\forall x \in \text{it}. \forall y \in S - \text{it}. R \ y \ x \rrbracket \implies P \ \sigma$
shows *FOREACHoci R I S c f* $\sigma 0 \leq \text{SPEC} \ P$
apply (rule *FOREACHoci-rule*[*OF FIN, where I=I, OF I0*])
apply (rule *IP, assumption+*)
apply (*case-tac c* σ)
apply (*blast intro: III*)
apply (*blast intro: II2*)
apply (*blast intro: II2*)
done

lemma *FOREACHci-rule'*[*refine-vcg*]:

assumes *FIN*: *finite S*
assumes *I0*: $I \ S \ \sigma 0$
assumes *IP*:
 $\bigwedge x \ \text{it} \ \sigma. \llbracket x \in \text{it}; \text{it} \subseteq S; I \ \text{it} \ \sigma; c \ \sigma \rrbracket \implies f \ x \ \sigma \leq \text{SPEC} \ (I \ (\text{it} - \{x\}))$
assumes *III*: $\bigwedge \sigma. \llbracket I \ \{\} \ \sigma; c \ \sigma \rrbracket \implies P \ \sigma$
assumes *II2*: $\bigwedge \text{it} \ \sigma. \llbracket \text{it} \subseteq S; I \ \text{it} \ \sigma; \neg c \ \sigma \rrbracket \implies P \ \sigma$
shows *FOREACHci I S c f* $\sigma 0 \leq \text{SPEC} \ P$
unfolding *FOREACHci-def*
by (rule *FOREACHoci-rule'*) (*simp-all add: assms*)

lemma *FOREACHc-rule'*:

assumes *FIN*: *finite S*
assumes *I0*: $I S \sigma 0$
assumes *IP*:
 $\bigwedge x \text{ it } \sigma. \llbracket x \in \text{it}; \text{it} \subseteq S; I \text{ it } \sigma; c \sigma \rrbracket \implies f x \sigma \leq \text{SPEC } (I (\text{it} - \{x\}))$
assumes *II1*: $\bigwedge \sigma. \llbracket I \{ \} \sigma; c \sigma \rrbracket \implies P \sigma$
assumes *II2*: $\bigwedge \text{it } \sigma. \llbracket \text{it} \subseteq S; I \text{ it } \sigma; \neg c \sigma \rrbracket \implies P \sigma$
shows *FOREACHc S c f* $\sigma 0 \leq \text{SPEC } P$
unfolding *FOREACHc-def*
apply (*rule order-trans*[*OF FOREACHci-weaken*], *rule TrueI*)
apply (*rule FOREACHci-rule'*[**where** $I=I$])
using *assms* **by** *auto*

2.15.4 FOREACH with empty sets

lemma *FOREACHoci-emp* [*simp*] :
 $\text{FOREACHoci } R \Phi \{ \} c f \sigma = \text{do } \{ \text{ASSERT } (\Phi \{ \} \sigma); \text{RETURN } \sigma \}$
by (*simp add: FOREACHoci-def FOREACH-cond-def bind-RES*)
(simp add: WHILEIT-unfold)

lemma *FOREACHoi-emp* [*simp*] :
 $\text{FOREACHoi } R \Phi \{ \} f \sigma = \text{do } \{ \text{ASSERT } (\Phi \{ \} \sigma); \text{RETURN } \sigma \}$
by (*simp add: FOREACHoi-def*)

lemma *FOREACHci-emp* [*simp*] :
 $\text{FOREACHci } \Phi \{ \} c f \sigma = \text{do } \{ \text{ASSERT } (\Phi \{ \} \sigma); \text{RETURN } \sigma \}$
by (*simp add: FOREACHci-def*)

lemma *FOREACHc-emp* [*simp*] :
 $\text{FOREACHc } \{ \} c f \sigma = \text{RETURN } \sigma$
by (*simp add: FOREACHc-def*)

lemma *FOREACH-emp* [*simp*] :
 $\text{FOREACH } \{ \} f \sigma = \text{RETURN } \sigma$
by (*simp add: FOREACH-def*)

lemma *FOREACHi-emp* [*simp*] :
 $\text{FOREACHi } \Phi \{ \} f \sigma = \text{do } \{ \text{ASSERT } (\Phi \{ \} \sigma); \text{RETURN } \sigma \}$
by (*simp add: FOREACHi-def*)

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* $L f == ((L \text{ id } (\leq) f f) \wedge (L \text{ id } \text{flat-ge } f f))$

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* .

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*
apply (*unfold trimono-spec-defs*)
apply –
unfolding *FOREACHoci-def FOREACH-to-oci-unfold FOREACH-body-def*
apply (*refine-mono*)
done

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*
 $\square \Rightarrow \text{RETURN } s$
 $| x\#ls \Rightarrow \text{if } c\ s \text{ then do } \{ s \leftarrow f\ x\ s; \text{nfoldli } ls\ c\ f\ s \}$ *else RETURN s*
 $)$

lemma *nfoldli-simps*[*simp*]:
nfoldli $\square\ c\ f\ s = \text{RETURN } s$
nfoldli (*x#ls*) *c f s* =
(if c s then do { s ← f x s; nfoldli ls c f s } else RETURN s)

```

apply (subst nfoldli.simps, simp)+
done

lemma param-nfoldli[param]:
  shows (nfoldli, nfoldli) ∈
    ⟨Ra⟩list-rel → (Rb→Id) → (Ra→Rb→⟨Rb⟩nres-rel) → Rb → ⟨Rb⟩nres-rel
  apply (intro fun-relI)
proof goal-cases
  case (1 l l' c c' f f' s s')
  thus ?case
    apply (induct arbitrary: s s')
    apply (simp only: nfoldli.simps True-implies-equals)
    apply parametricity
    apply (simp only: nfoldli.simps True-implies-equals)
    apply (parametricity)
  done
qed

lemma nfoldli-no-ctd[simp]: ¬ctd s ⇒ nfoldli l ctd f s = RETURN s
  by (cases l) auto

lemma nfoldli-append[simp]: nfoldli (l1@l2) ctd f s = nfoldli l1 ctd f s ≫ nfoldli
  l2 ctd f
  by (induction l1 arbitrary: s) (auto simp: pw-eq-iff refine-pw-simps)

lemma nfoldli-map: nfoldli (map f l) ctd g s = nfoldli l ctd (g o f) s
  apply (induction l arbitrary: s)
  by (auto simp: pw-eq-iff refine-pw-simps)

lemma nfoldli-nfoldli-prod-conv:
  nfoldli l2 ctd (λi. nfoldli l1 ctd (f i)) s = nfoldli (List.product l2 l1) ctd (λ(i,j).
  f i j) s
proof –
  have [simp]: nfoldli (map (Pair a) l) ctd (λ(x, y). f x y) s = nfoldli l ctd (f a) s
    for a l s
  apply (induction l arbitrary: s)
  by (auto simp: pw-eq-iff refine-pw-simps)

  show ?thesis
    by (induction l2 arbitrary: l1 s) auto
qed

```

The fold-function over the nres-monad is transferred to a plain foldli function

```

lemma nfoldli-transfer-plain[refine-transfer]:
  assumes ∧x s. RETURN (f x s) ≤ f' x s
  shows RETURN (foldli l c f s) ≤ (nfoldli l c f' s)
  using assms
  apply (induct l arbitrary: s)
  apply (auto)

```

by (*metis* (*lifting*) *plain-bind*)

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*

$(\text{foldli } l \text{ (case-dres False False } c) (\lambda x s. s \gg f x) (\text{dRETURN } s))$
 $\leq (\text{nfoldli } l c f' s)$

proof (*induct l arbitrary: s*)

case *Nil* **thus** ?*case* **by** *auto*

next

case (*Cons a l*)

thus ?*case*

apply (*auto*)

apply (*cases f a s*)

apply (*cases l, simp-all*) []

apply *simp*

apply (*rule order-trans*[*rotated*])

apply (*rule bind-mono*)

apply (*rule FR*)

apply *assumption*

apply *simp*

apply *simp*

using *FR*[*of a s*]

apply *simp*

done

qed

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

$\llbracket \bigwedge x s. f x s \leq f' x s \rrbracket \Longrightarrow \text{nfoldli } l c f \sigma \leq \text{nfoldli } l c f' \sigma$

$\llbracket \bigwedge x s. \text{flat-ge } (f x s) (f' x s) \rrbracket \Longrightarrow \text{flat-ge } (\text{nfoldli } l c f \sigma) (\text{nfoldli } l c f' \sigma)$

apply (*induct l arbitrary: σ*)

apply *auto*

apply *refine-mono*

apply (*induct l arbitrary: σ*)

apply *auto*

apply *refine-mono*

done

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

$(\text{WHILE}_T^I$

$(\text{FOREACH-cond } c) (\text{FOREACH-body } f) (l, \sigma) \gg$

$(\lambda(-, \sigma). \text{RETURN } \sigma))$

proof (*induct l arbitrary: σ*)

case *Nil* **thus** ?*case* **by** (*subst WHILEIT-unfold*) (*auto simp: FOREACH-cond-def*)


```

next
  case (Cons x ls)
  show ?case
  proof (cases c  $\sigma$ )
    case False thus ?thesis
      apply (subst WHILEIT-unfold)
      unfolding FOREACH-cond-def
      by simp
  next
  case [simp]: True
  from Cons show ?thesis
    apply (subst WHILEIT-unfold)
    unfolding FOREACH-cond-def FOREACH-body-def
    apply clarsimp
    apply (rule Refine-Basic.bind-mono)
    apply simp-all
  done
qed
qed

lemma while-nfoldli:
  do {
    ( $\sigma$ )  $\leftarrow$  WHILET (FOREACH-cond c) (FOREACH-body f) (l, $\sigma$ );
    RETURN  $\sigma$ 
  }  $\leq$  nfoldli l c f  $\sigma$ 
  apply (induct l arbitrary:  $\sigma$ )
  apply (subst WHILET-unfold)
  apply (simp add: FOREACH-cond-def)

  apply (subst WHILET-unfold)
  apply (auto
    simp: FOREACH-cond-def FOREACH-body-def
    intro: bind-mono)
  done

lemma while-eq-nfoldli: do {
  ( $\sigma$ )  $\leftarrow$  WHILET (FOREACH-cond c) (FOREACH-body f) (l, $\sigma$ );
  RETURN  $\sigma$ 
} = nfoldli l c f  $\sigma$ 
  apply (rule antisym)
  apply (rule while-nfoldli)
  apply (rule order-trans[OF nfoldli-while[where I= $\lambda$ -. True]])
  apply (simp add: WHILET-def)
  done

lemma nfoldli-rule:
  assumes I0: I  $\square$  l0  $\sigma$ 0
  assumes IS:  $\bigwedge$  x l1 l2  $\sigma$ .  $\llbracket$  l0=l1@x#l2; I l1 (x#l2)  $\sigma$ ; c  $\sigma$   $\rrbracket \implies$  f x  $\sigma \leq$  SPEC
  (I (l1@[x]) l2)

```

```

assumes FNC:  $\bigwedge l1\ l2\ \sigma. \llbracket l0=l1@l2; I\ l1\ l2\ \sigma; \neg c\ \sigma \rrbracket \implies P\ \sigma$ 
assumes FC:  $\bigwedge \sigma. \llbracket I\ l0 \rrbracket \sigma; c\ \sigma \rrbracket \implies P\ \sigma$ 
shows nfoldli l0 c f  $\sigma0 \leq SPEC\ P$ 
apply (rule order-trans[OF nfoldli-while[
  where  $I=\lambda(l2,\sigma). \exists l1. l0=l1@l2 \wedge I\ l1\ l2\ \sigma$ ]])
unfolding FOREACH-cond-def FOREACH-body-def
apply (refine-vcg WHILEIT-rule[where  $R=measure\ (length\ o\ fst)$ ] refine-vcg)
apply simp
using I0 apply simp

```

```

apply (case-tac a, simp)
apply simp
apply (elim exE conjE)
apply (rule order-trans[OF IS], assumption+)
apply auto []

```

```

apply simp
apply (elim exE disjE2)
using FC apply auto []
using FNC apply auto []
done

```

lemma *nfoldli-leof-rule*:

```

assumes I0:  $I\ []\ l0\ \sigma0$ 
assumes IS:  $\bigwedge x\ l1\ l2\ \sigma. \llbracket l0=l1@x\#l2; I\ l1\ (x\#l2)\ \sigma; c\ \sigma \rrbracket \implies f\ x\ \sigma \leq_n\ SPEC$ 
(I (l1@[x]) l2)
assumes FNC:  $\bigwedge l1\ l2\ \sigma. \llbracket l0=l1@l2; I\ l1\ l2\ \sigma; \neg c\ \sigma \rrbracket \implies P\ \sigma$ 
assumes FC:  $\bigwedge \sigma. \llbracket I\ l0 \rrbracket \sigma; c\ \sigma \rrbracket \implies P\ \sigma$ 
shows nfoldli l0 c f  $\sigma0 \leq_n\ SPEC\ P$ 

```

proof –

```

{
  fix l1 l2  $\sigma$ 
  assume  $l0=l1@l2\ I\ l1\ l2\ \sigma$ 
  hence nfoldli l2 c f  $\sigma \leq_n\ SPEC\ P$ 
  proof (induction l2 arbitrary: l1  $\sigma$ )
    case Nil thus ?case
      apply simp
      apply (cases c  $\sigma$ )
      apply (rule FC; auto; fail)
      apply (rule FNC[of l1 []]; auto; fail)
      done
    next
      case (Cons x l2)
      note [refine-vcg] = Cons.IH[of l1@[x], THEN leof-trans] IS[of l1 x l2  $\sigma$ , THEN
leof-trans]

```

```

show ?case
apply (simp split del: if-split)
apply refine-vcg

```

```

    using Cons.premis FNC by auto
  qed
} from this[of [] l0 σ0] I0 show ?thesis by auto
qed

```

```

lemma nfoldli-refine[refine]:
  assumes (li, l) ∈ ⟨S⟩list-rel
    and (si, s) ∈ R
    and CR: (ci, c) ∈ R → bool-rel
    and [refine]:  $\bigwedge xi\ x\ si\ s. \llbracket (xi,x) \in S; (si,s) \in R; c\ s \rrbracket \implies fi\ xi\ si \leq \Downarrow R (f\ x\ s)$ 
  shows nfoldli li ci fi si  $\leq \Downarrow R$  (nfoldli l c f s)
  using assms(1,2)
proof (induction arbitrary: si s rule: list-rel-induct)
  case Nil thus ?case by simp
next
  case (Cons xi x li l)
  note [refine] = Cons

  show ?case
    apply (simp split del: if-split)
    apply refine-rcg
    using CR Cons.premis by (auto dest: fun-relD)
qed

```

```

lemma nfoldli-invar-refine:
  assumes (li,l) ∈ ⟨S⟩list-rel
  assumes (si,s) ∈ R
  assumes I [] li si
  assumes COND:  $\bigwedge l1i\ l2i\ l1\ l2\ si\ s. \llbracket li=l1i@l2i; l=l1@l2; (l1i,l1) \in \langle S \rangle list-rel; (l2i,l2) \in \langle S \rangle list-rel; I\ l1i\ l2i\ si; (si,s) \in R \rrbracket \implies (ci\ si, c\ s) \in bool-rel$ 
  assumes INV:  $\bigwedge l1i\ xi\ l2i\ si. \llbracket li=l1i@xi\#l2i; I\ l1i\ (xi\#l2i)\ si \rrbracket \implies fi\ xi\ si \leq_n$ 
  SPEC (I (l1i@[xi]) l2i)
  assumes STEP:  $\bigwedge l1i\ xi\ l2i\ l1\ x\ l2\ si\ s. \llbracket li=l1i@xi\#l2i; l=l1@x\#l2; (l1i,l1) \in \langle S \rangle list-rel; (xi,x) \in S; (l2i,l2) \in \langle S \rangle list-rel; I\ l1i\ (xi\#l2i)\ si; (si,s) \in R \rrbracket \implies fi\ xi\ si \leq \Downarrow R (f\ x\ s)$ 
  shows nfoldli li ci fi si  $\leq \Downarrow R$  (nfoldli l c f s)
proof -
  {
    have [refine-dref-RELATES]: RELATES R RELATES S by (auto simp: RELATES-def)

```

```

    note [refine del] = nfoldli-refine

```

```

  fix l1i l2i l1 l2 si s
  assume (l2i,l2) ∈ ⟨S⟩list-rel (l1i,l1) ∈ ⟨S⟩list-rel
  and li=l1i@l2i l=l1@l2

```

```

and (si,s)∈R  I l1i l2i si
hence nfoldli l2i ci fi si ≤ ↓R (nfoldli l2 c f s)
proof (induction arbitrary: si s l1i l1 rule: list-rel-induct)
  case Nil thus ?case by auto
next
case (Cons xi x l2i l2)

show ?case
  apply (simp split del: if-split)
  apply (refine-rcg bind-refine')
  apply (refine-dref-type)
  subgoal using COND[of l1i xi#l2i l1  x#l2 si s] Cons.premss Cons.hyps
by auto
  subgoal apply (rule STEP) using Cons.premss Cons.hyps by auto
  subgoal for si' s'
    apply (rule Cons.IH[of l1i@[xi]  l1@[x]])
    using Cons.premss Cons.hyps
    apply (auto simp: list-rel-append1) apply force
    using INV[of l1i xi l2i si]
    by (auto simp: pw-leof-iff)
  subgoal using Cons.premss by simp
done
qed
}
from this[of li l [] [] si s] assms(1,2,3) show ?thesis by auto
qed

```

lemma foldli-mono-dres-aux1:

```

fixes σ :: 'a :: {order-bot, order-top}
assumes COND: ∧σ σ'. σ ≤ σ' ⇒ c σ ≠ c σ' ⇒ σ = bot ∨ σ' = top
assumes STRICT: ∧x. f x bot = bot  ∧x. f' x top = top
assumes B: σ ≤ σ'
assumes A: ∧a x x'. x ≤ x' ⇒ f a x ≤ f' a x'
shows foldli l c f σ ≤ foldli l c f' σ'
proof -
  { fix l
    have foldli l c f bot = bot by (induct l) (auto simp: STRICT)
  } note [simp] = this
  { fix l
    have foldli l c f' top = top by (induct l) (auto simp: STRICT)
  } note [simp] = this

show ?thesis
using B
apply (induct l arbitrary: σ σ')
apply (auto simp: A STRICT dest!: COND)
done

```

qed

lemma *foldli-mono-dres-aux2*:

assumes *STRICT*: $\bigwedge x. f\ x\ bot = bot \quad \bigwedge x. f'\ x\ top = top$
assumes *A*: $\bigwedge a\ x\ x'. x \leq x' \implies f\ a\ x \leq f'\ a\ x'$
shows *foldli l (case-dres False False c) f σ*
 \leq *foldli l (case-dres False False c) f' σ*
apply (*rule foldli-mono-dres-aux1*)
apply (*simp-all split: dres.split-asm add: STRICT A*)
done

lemma *foldli-mono-dres[refine-mono]*:

assumes *A*: $\bigwedge a\ x. f\ a\ x \leq f'\ a\ x$
shows *foldli l (case-dres False False c) (λx s. dbind s (f x)) σ*
 \leq *foldli l (case-dres False False c) (λx s. dbind s (f' x)) σ*
apply (*rule foldli-mono-dres-aux2*)
apply (*simp-all*)
apply (*rule dbind-mono*)
apply (*simp-all add: A*)
done

partial-function (*drec*) *dfoldli where*

dfoldli l c f s = (case l of
 $\square \Rightarrow dRETURN\ s$
 $| x\#\!ls \Rightarrow \text{if } c\ s \text{ then do } \{ s \leftarrow f\ x\ s; dfoldli\ ls\ c\ f\ s \}$ *else* $dRETURN\ s$
 $)$

lemma *dfoldli-simps[simp]*:

dfoldli $\square\ c\ f\ s = dRETURN\ s$
dfoldli ($x\#\!ls$) $c\ f\ s =$
 $(\text{if } c\ s \text{ then do } \{ s \leftarrow f\ x\ s; dfoldli\ ls\ c\ f\ s \}$ *else* $dRETURN\ s)$
apply (*subst dfoldli.simps, simp*)
done

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

$\llbracket \bigwedge x\ s. f\ x\ s \leq f'\ x\ s \rrbracket \implies dfoldli\ l\ c\ f\ \sigma \leq dfoldli\ l\ c\ f'\ \sigma$
 $\llbracket \bigwedge x\ s. flat\text{-}ge\ (f\ x\ s)\ (f'\ x\ s) \rrbracket \implies flat\text{-}ge\ (dfoldli\ l\ c\ f\ \sigma)\ (dfoldli\ l\ c\ f'\ \sigma)$
apply (*induct l arbitrary: σ*)
apply *auto*
apply *refine-mono*

apply (*induct l arbitrary: σ*)
apply *auto*
apply *refine-mono*
done

lemma *foldli-dres-pres-FAIL[simp]*:

foldli l (case-dres False False c) (λx s. dbind s (f x)) dFAIL = dFAIL

by (*cases l*) *auto*

lemma *foldli-dres-pres-SUCCEED*[*simp*]:

foldli l (case-dres False False c) ($\lambda x s. dbind s (f x)$) dSUCCEED = dSUCCEED
by (*cases l*) *auto*

lemma *dfoldli-by-foldli*: *dfoldli l c f σ*

= foldli l (case-dres False False c) ($\lambda x s. dbind s (f x)$) (dRETURN σ)
apply (*induction l arbitrary: σ*)
apply *simp*
apply (*clarsimp intro!: ext*)
apply (*rename-tac a l x*)
apply (*case-tac f a x*)
apply *auto*
done

lemma *foldli-mono-dres-flat*[*refine-mono*]:

assumes *A: $\bigwedge a x. flat-ge (f a x) (f' a x)$*
shows *flat-ge (foldli l (case-dres False False c) ($\lambda x s. dbind s (f x)$) σ)*
(foldli l (case-dres False False c) ($\lambda x s. dbind s (f' x)$) σ)
apply (*cases σ*)
apply (*simp-all add: dfoldli-by-foldli[symmetric]*)
using *A* **apply** *refine-mono*
done

lemma *dres-foldli-ne-bot*[*refine-transfer*]:

assumes *1: $\sigma \neq dSUCCEED$*
assumes *2: $\bigwedge x \sigma. f x \sigma \neq dSUCCEED$*
shows *foldli l c ($\lambda x s. s \gg f x$) $\sigma \neq dSUCCEED$*
using *1* **apply** (*induct l arbitrary: σ*)
apply *simp*
apply (*simp split: dres.split, intro allI impI*)
apply *rprems*
using *2*
apply (*simp add: dres-ne-bot-basic*)
done

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 $\equiv \langle R \rangle list-rel O$ br set distinct*

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

shows (*nfoldli*, *nfoldli*)
 $\in \langle Ra \rangle list\text{-rel} \rightarrow (Rb \rightarrow bool\text{-rel}) \rightarrow (Ra \rightarrow Rb \rightarrow \langle Rb \rangle nres\text{-rel}) \rightarrow Rb \rightarrow \langle Rb \rangle nres\text{-rel}$
by (*rule param-nfoldli*)

This constant is a placeholder to be converted to custom operations by pattern rules

definition *it-to-sorted-list* $R\ s$

$\equiv SPEC (\lambda l. distinct\ l \wedge s = set\ l \wedge sorted\text{-wrt}\ R\ l)$

definition *LIST-FOREACH* $\Phi\ tsl\ c\ f\ \sigma\ \theta \equiv do\ \{$

$xs \leftarrow tsl;$
 $(-, \sigma) \leftarrow WHILE_T^{\lambda(it, \sigma). \exists xs'. xs = xs' @ it \wedge \Phi (set\ it)\ \sigma}$
 (*FOREACH-cond* c) (*FOREACH-body* f) ($xs, \sigma\ \theta$);
 RETURN σ

lemma *FOREACHoci-by-LIST-FOREACH*:

FOREACHoci $R\ \Phi\ S\ c\ f\ \sigma\ \theta = do\ \{$
 ASSERT (*finite* S);
LIST-FOREACH Φ (*it-to-sorted-list* $R\ S$) $c\ f\ \sigma\ \theta$
 $\}$

unfolding *OP-def FOREACHoci-def LIST-FOREACH-def it-to-sorted-list-def*
by *simp*

Patterns that convert FOREACH-constructs to *LIST-FOREACH*

context begin interpretation *autoref-syn* .

lemma *FOREACH-patterns*[*autoref-op-pat-def*]:

FOREACH ^{I} $s\ f \equiv FOREACH_{OC}^{\lambda-. True, I\ s} (\lambda-. True)\ f$
FOREACHci $I\ s\ c\ f \equiv FOREACHoci (\lambda-. True)\ I\ s\ c\ f$
FOREACH ^{R, Φ} $s\ c\ f \equiv \lambda\sigma. do\ \{$
 ASSERT (*finite* s);
Autoref-Tagging.OP (*LIST-FOREACH* Φ) (*it-to-sorted-list* $R\ s$) $c\ f\ \sigma$
 $\}$
FOREACH $s\ f \equiv FOREACHoci (\lambda-. True)\ (\lambda-. True)\ s\ (\lambda-. True)\ f$
FOREACHoi $R\ I\ s\ f \equiv FOREACHoci R\ I\ s\ (\lambda-. True)\ f$
FOREACHc $s\ c\ f \equiv FOREACHoci (\lambda-. True)\ (\lambda-. True)\ s\ c\ f$

unfolding

FOREACHoci-by-LIST-FOREACH[*abs-def*]
FOREACHc-def[*abs-def*]
FOREACH-def[*abs-def*]
FOREACHci-def[*abs-def*]
FOREACHi-def[*abs-def*]
FOREACHoi-def[*abs-def*]

by *simp-all*

end

definition $LIST-FOREACH' \text{ } tsl \ c \ f \ \sigma \equiv do \ \{xs \leftarrow \text{ } tsl; \text{ } nfoldli \ xs \ c \ f \ \sigma\}$

lemma $LIST-FOREACH'-param[param]$:
shows $(LIST-FOREACH', LIST-FOREACH')$
 $\in (\langle\langle Rv \rangle list-rel \rangle nres-rel \rightarrow (R\sigma \rightarrow bool-rel))$
 $\rightarrow (Rv \rightarrow R\sigma \rightarrow \langle R\sigma \rangle nres-rel) \rightarrow R\sigma \rightarrow \langle R\sigma \rangle nres-rel)$
unfolding $LIST-FOREACH'-def[abs-def]$
by *parametricity*

lemma $LIST-FOREACH-autoref[autoref-rules]$:
shows $(LIST-FOREACH', LIST-FOREACH \ \Phi) \in$
 $(\langle\langle Rv \rangle list-rel \rangle nres-rel \rightarrow (R\sigma \rightarrow bool-rel))$
 $\rightarrow (Rv \rightarrow R\sigma \rightarrow \langle R\sigma \rangle nres-rel) \rightarrow R\sigma \rightarrow \langle R\sigma \rangle nres-rel)$

proof (*intro fun-relI nres-relI*)

fix $tsl \ tsl' \ c \ c' \ f \ f' \ \sigma \ \sigma'$

assume $[param]$:

$(tsl, tsl') \in \langle\langle Rv \rangle list-rel \rangle nres-rel$

$(c, c') \in R\sigma \rightarrow bool-rel$

$(f, f') \in Rv \rightarrow R\sigma \rightarrow \langle R\sigma \rangle nres-rel$

$(\sigma, \sigma') \in R\sigma$

have $LIST-FOREACH' \text{ } tsl \ c \ f \ \sigma \leq \Downarrow R\sigma \ (LIST-FOREACH' \text{ } tsl' \ c' \ f' \ \sigma')$

apply (*rule nres-relD*)

by *parametricity*

also have $LIST-FOREACH' \text{ } tsl' \ c' \ f' \ \sigma'$

$\leq LIST-FOREACH \ \Phi \text{ } tsl' \ c' \ f' \ \sigma'$

apply (*rule refine-IdD*)

unfolding $LIST-FOREACH-def \ LIST-FOREACH'-def$

apply *refine-rcg*

apply *simp*

apply (*rule nfoldli-while*)

done

finally show

$LIST-FOREACH' \text{ } tsl \ c \ f \ \sigma \leq \Downarrow R\sigma \ (LIST-FOREACH \ \Phi \text{ } tsl' \ c' \ f' \ \sigma')$

qed

context begin interpretation *trimono-spec* .

lemma $LIST-FOREACH'-mono[unfolded \ trimono-spec-defs, \ refine-mono]$:

trimono-spec $(R \ o \ R \ o \ M2 \ o \ R) \ LIST-FOREACH'$

apply (*unfold trimono-spec-defs*)

apply $-$

unfolding $LIST-FOREACH'-def$

by *refine-mono+*

end

lemma $LIST-FOREACH'-transfer-plain[refine-transfer]$:


```

assumes RETURN  $tsl \leq tsl'$ 
assumes  $\bigwedge x \sigma. RETURN (f x \sigma) \leq f' x \sigma$ 
shows RETURN (foldli  $tsl c f \sigma$ )  $\leq LIST-FOREACH' tsl' c f' \sigma$ 
apply (rule order-trans[rotated])
unfolding LIST-FOREACH'-def
using assms
apply refine-transfer
by simp

```

thm *refine-transfer*

```

lemma LIST-FOREACH'-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  $xs (case-dres False False c) (\lambda x s. s \gg f x) (dRETURN \sigma)$ 
  })  $\leq LIST-FOREACH' tsl' c f' \sigma$ 
unfolding LIST-FOREACH'-def
using assms
by refine-transfer

```

Simplification rules to summarize iterators

```

lemma [refine-transfer-post-simp]:
  do {
     $xs \leftarrow dRETURN tsl$ ;
    foldli  $xs c f \sigma$ 
  } = foldli  $tsl c f \sigma$ 
by simp

```

```

lemma [refine-transfer-post-simp]:
  (let  $xs = tsl$  in foldli  $xs c f \sigma$ ) = foldli  $tsl c f \sigma$ 
by simp

```

```

lemma LFO-pre-refine:
assumes  $(li, l) \in \langle A \rangle list-set-rel$ 
assumes  $(ci, c) \in R \rightarrow bool-rel$ 
assumes  $(fi, f) \in A \rightarrow R \rightarrow \langle R \rangle nres-rel$ 
assumes  $(s0i, s0) \in R$ 
shows LIST-FOREACH' (RETURN  $li$ )  $ci fi s0i \leq \Downarrow R (FOREACHci I l c f s0)$ 
proof –
from assms(1) have [simp]: finite l by (auto simp: list-set-rel-def br-def)
show ?thesis
unfolding FOREACHc-def FOREACHci-def FOREACHoci-by-LIST-FOREACH
apply simp
apply (rule LIST-FOREACH-autoref[param-fo, THEN nres-relD])
using assms

```

```

apply auto
apply (auto simp: it-to-sorted-list-def nres-rel-def pw-le-iff refine-pw-simps
        list-set-rel-def br-def)
done
qed

```

```

lemma LFOci-refine:
  assumes  $(li, l) \in \langle A \rangle \text{list-set-rel}$ 
  assumes  $\bigwedge s \text{ si. } (si, s) \in R \implies ci \text{ si} \longleftrightarrow c \text{ s}$ 
  assumes  $\bigwedge x \text{ xi } s \text{ si. } [(xi, x) \in A; (si, s) \in R] \implies fi \text{ xi } si \leq \Downarrow R (f \text{ x } s)$ 
  assumes  $(s0i, s0) \in R$ 
  shows  $nfoldli \text{ li } ci \text{ fi } s0i \leq \Downarrow R (\text{FOREACH}ci \text{ I l c f } s0)$ 
proof –
  from assms LFO-pre-refine[of li l A ci c R fi f s0i s0] show ?thesis
  unfolding fun-rel-def nres-rel-def LIST-FOREACH'-def
  apply (simp add: pw-le-iff refine-pw-simps)
  apply blast+
  done
qed

```

```

lemma LFOc-refine:
  assumes  $(li, l) \in \langle A \rangle \text{list-set-rel}$ 
  assumes  $\bigwedge s \text{ si. } (si, s) \in R \implies ci \text{ si} \longleftrightarrow c \text{ s}$ 
  assumes  $\bigwedge x \text{ xi } s \text{ si. } [(xi, x) \in A; (si, s) \in R] \implies fi \text{ xi } si \leq \Downarrow R (f \text{ x } s)$ 
  assumes  $(s0i, s0) \in R$ 
  shows  $nfoldli \text{ li } ci \text{ fi } s0i \leq \Downarrow R (\text{FOREACH}c \text{ l c f } s0)$ 
  unfolding FOREACHc-def
  by (rule LFOci-refine[OF assms])

```

```

lemma LFO-refine:
  assumes  $(li, l) \in \langle A \rangle \text{list-set-rel}$ 
  assumes  $\bigwedge x \text{ xi } s \text{ si. } [(xi, x) \in A; (si, s) \in R] \implies fi \text{ xi } si \leq \Downarrow R (f \text{ x } s)$ 
  assumes  $(s0i, s0) \in R$ 
  shows  $nfoldli \text{ li } (\lambda-. \text{True}) \text{ fi } s0i \leq \Downarrow R (\text{FOREACH} \text{ l f } s0)$ 
  unfolding FOREACH-def
  apply (rule LFOc-refine)
  apply (rule assms | simp)+
  done

```

```

lemma LFOi-refine:
  assumes  $(li, l) \in \langle A \rangle \text{list-set-rel}$ 
  assumes  $\bigwedge x \text{ xi } s \text{ si. } [(xi, x) \in A; (si, s) \in R] \implies fi \text{ xi } si \leq \Downarrow R (f \text{ x } s)$ 
  assumes  $(s0i, s0) \in R$ 
  shows  $nfoldli \text{ li } (\lambda-. \text{True}) \text{ fi } s0i \leq \Downarrow R (\text{FOREACH}i \text{ I l f } s0)$ 
  unfolding FOREACHi-def
  apply (rule LFOci-refine)
  apply (rule assms | simp)+

```

done

lemma *LIST-FOREACH'-refine*: $LIST-FOREACH' \text{ } \text{tsl}' \text{ } c' \text{ } f' \text{ } \sigma' \leq LIST-FOREACH \Phi \text{ } \text{tsl}' \text{ } c' \text{ } f' \text{ } \sigma'$
apply (*rule refine-IdD*)
unfolding *LIST-FOREACH-def LIST-FOREACH'-def*
apply *refine-rcg*
apply *simp*
apply (*rule nfoldli-while*)
done

lemma *LIST-FOREACH'-eq*: $LIST-FOREACH (\lambda-. True) \text{ } \text{tsl}' \text{ } c' \text{ } f' \text{ } \sigma' = (LIST-FOREACH' \text{ } \text{tsl}' \text{ } c' \text{ } f' \text{ } \sigma')$
apply (*rule antisym*)
subgoal
apply (*rule refine-IdD*)
unfolding *LIST-FOREACH-def LIST-FOREACH'-def while-eq-nfoldli[symmetric]*

apply (*refine-rcg WHILEIT-refine-new-invar*)
unfolding *WHILET-def*
apply (*rule WHILEIT-refine-new-invar*)

apply *refine-dref-type*
apply *clarsimp-all*
unfolding *FOREACH-body-def FOREACH-cond-def*
apply *refine-vcg*
apply (*auto simp: refine-pw-simps pw-le-iff neq-Nil-conv*)
done
subgoal by (*rule LIST-FOREACH'-refine*)
done

2.15.8 FOREACH with duplicates

definition *FOREACHcd* $S \text{ } c \text{ } f \text{ } \sigma \equiv do \{$
ASSERT (*finite S*);
 $l \leftarrow SPEC (\lambda l. set \text{ } l = S);$
nfoldli $l \text{ } c \text{ } f \text{ } \sigma$
 $\}$

lemma *FOREACHcd-rule*:
assumes *finite S₀*
assumes *I0: I { } S₀ σ₀*
assumes *STEP: $\bigwedge S1 \text{ } S2 \text{ } x \text{ } \sigma. \llbracket S_0 = insert \text{ } x \text{ } (S1 \cup S2); I \text{ } S1 \text{ } (insert \text{ } x \text{ } S2) \text{ } \sigma; c \text{ } \sigma \rrbracket \implies f \text{ } x \text{ } \sigma \leq SPEC (I (insert \text{ } x \text{ } S1) \text{ } S2)$*
assumes *INTR: $\bigwedge S1 \text{ } S2 \text{ } \sigma. \llbracket S_0 = S1 \cup S2; I \text{ } S1 \text{ } S2 \text{ } \sigma; \neg c \text{ } \sigma \rrbracket \implies \Phi \text{ } \sigma$*
assumes *COMPL: $\bigwedge \sigma. \llbracket I \text{ } S_0 \text{ } \{ \} \text{ } \sigma; c \text{ } \sigma \rrbracket \implies \Phi \text{ } \sigma$*
shows $FOREACHcd \text{ } S_0 \text{ } c \text{ } f \text{ } \sigma_0 \leq SPEC \Phi$
unfolding *FOREACHcd-def*
apply *refine-vcg*

apply *fact*
apply (*rule* *ifoldli-rule*[**where** $I = \lambda l1\ l2\ \sigma.\ I\ (set\ l1)\ (set\ l2)\ \sigma$])
subgoal using *I0* **by** *auto*
subgoal using *STEP* **by** *auto*
subgoal using *INTR* **by** *auto*
subgoal using *COMPL* **by** *auto*
done

definition *FOREACHcdi*

$:: ('a\ set \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow bool)$
 $\Rightarrow 'a\ set \Rightarrow ('b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow 'b\ nres) \Rightarrow 'b \Rightarrow 'b\ nres$
where
 $FOREACHcdi\ I \equiv FOREACHcd$

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

assumes *finite* S_0
assumes *I0*: $I\ \{\}\ S_0\ \sigma_0$
assumes *STEP*: $\bigwedge S1\ S2\ x\ \sigma.\ \llbracket S_0 = insert\ x\ (S1 \cup S2); I\ S1\ (insert\ x\ S2)\ \sigma; c\ \sigma \rrbracket \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (insert\ x\ S1)\ S2)$
assumes *INTR*: $\bigwedge S1\ S2\ \sigma.\ \llbracket S_0 = S1 \cup S2; I\ S1\ S2\ \sigma; \neg c\ \sigma \rrbracket \Longrightarrow \Phi\ \sigma$
assumes *COMPL*: $\bigwedge \sigma.\ \llbracket I\ S_0\ \{\}\ \sigma; c\ \sigma \rrbracket \Longrightarrow \Phi\ \sigma$
shows $FOREACHcdi\ I\ S_0\ c\ f\ \sigma_0 \leq SPEC\ \Phi$
unfolding *FOREACHcdi-def*
using *assms*
by (*rule* *FOREACHcd-rule*)

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*: $\bigwedge \sigma'\ \sigma.\ (\sigma', \sigma) \in R \Longrightarrow (c'\ \sigma', c\ \sigma) \in bool\ rel$
assumes *F*: $\bigwedge x'\ x\ \sigma'\ \sigma.\ \llbracket (x', x) \in S; (\sigma', \sigma) \in R \rrbracket \Longrightarrow f'\ x'\ \sigma' \leq \Downarrow R\ (f\ x\ \sigma)$
shows $FOREACHcd\ s'\ c'\ f'\ \sigma' \leq \Downarrow R\ (FOREACHcdi\ I\ s\ c\ f\ \sigma)$

proof –

have [*refine-dref-RELATES*]: *RELATES* S **by** (*simp* *add*: *RELATES-def*)

from *SV* **obtain** $\alpha\ I$ **where** [*simp*]: $S = br\ \alpha\ I$ **by** (*rule* *single-valued-as-brE*)

with S **have** [*simp*]: $s = \alpha\ s'$ **and** [*simp*]: $\forall x \in s'.\ I\ x$

by (*auto* *simp*: *br-set-rel-alt*)

show *?thesis*

unfolding *FOREACHcd-def* *FOREACHcdi-def*

apply *refine-rcg*

apply *refine-dref-type*

subgoal **by** *simp*

subgoal

apply (*auto* *simp*: *pw-le-iff* *refine-pw-simps*)

```

  using S
  apply (rule-tac x=map  $\alpha$  x in exI)
  apply (auto simp: map-in-list-rel-conv)
  done
  subgoal using R0 by auto
  subgoal using C by auto
  subgoal using F by auto
  done
qed

```

lemma *FOREACHc-refines-FOREACHcd-aux*:
shows $FOREACHc\ s\ c\ f\ \sigma \leq FOREACHcd\ s\ c\ f\ \sigma$
unfolding *FOREACHc-def FOREACHci-def FOREACHoci-by-LIST-FOREACH*
LIST-FOREACH'-eq
LIST-FOREACH'-def FOREACHcd-def it-to-sorted-list-def
apply (rule refine-IdD)
apply (refine-rcg)
apply refine-dref-type
apply auto
done

lemmas *FOREACHc-refines-FOREACHcd[refine]*
 $=\ order-trans[OF\ FOREACHc-refines-FOREACHcd-aux\ FOREACHcd-refine]$

2.15.9 Miscellaneous Utility Lemmas

lemma *map-foreach*:
assumes *finite S*
shows $FOREACH\ S\ (\lambda x\ \sigma.\ RETURN\ (insert\ (f\ x)\ \sigma))\ R0 \leq SPEC\ ((=)\ (R0 \cup f'S))$
apply (rule *FOREACH-rule*[**where** $I=\lambda it\ \sigma.\ \sigma=R0 \cup f'(S-it)$])
apply (auto intro: *assms*)
done

lemma *map-sigma-foreach*:
fixes $f :: 'a \times 'b \Rightarrow 'c$
assumes *finite A*
assumes $\bigwedge x. x \in A \implies finite\ (B\ x)$
shows $FOREACH\ A\ (\lambda a\ \sigma.\ FOREACH\ (B\ a)\ (\lambda b\ \sigma.\ RETURN\ (insert\ (f\ (a,b))\ \sigma))\ \sigma)$
 $R0 \leq SPEC\ ((=)\ (R0 \cup f'Sigma\ A\ B))$
apply (rule *FOREACH-rule*[**where** $I=\lambda it\ \sigma.\ \sigma=R0 \cup f'(Sigma\ (A-it)\ B)$])
apply (auto intro: *assms*) [2]
apply (rule-tac $I=\lambda it'\ \sigma.\ \sigma=R0 \cup f'(Sigma\ (A - it)\ B)$
 $\cup f'(\{x\} \times (B\ x - it'))$
in *FOREACH-rule*)

```

apply (auto intro: assms) [2]
apply (rule refine-vcg)
apply auto []
apply auto []
apply auto []
done

```

lemma *map-sigma-sigma-foreach*:

```

fixes f :: 'a × ('b × 'c) ⇒ '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)))  $\sigma$ )  $\sigma$ 
    ) R0 ≤ SPEC ((=) (R0 ∪ f'(Sigma A ( $\lambda a. \text{Sigma } (B\ a)\ (C\ a))))$ 
  ) apply (rule FOREACH-rule[where
    I= $\lambda it\ \sigma. \sigma = R0 \cup f'(\text{Sigma } (A - it)\ (\lambda a. \text{Sigma } (B\ a)\ (C\ a)))$ )]
  ) apply (auto intro: assms) [2]
  ) apply (rule-tac
    I= $\lambda it'\ \sigma. \sigma = R0 \cup f'(\text{Sigma } (A - it)\ (\lambda a. \text{Sigma } (B\ a)\ (C\ a)))$ 
    ∪ f'({x} × (Sigma (B x - it') (C x)))
  ) in FOREACH-rule)
  ) apply (auto intro: assms) [2]
  ) apply (rule-tac
    I= $\lambda it''\ \sigma. \sigma = R0 \cup f'(\text{Sigma } (A - it)\ (\lambda a. \text{Sigma } (B\ a)\ (C\ a)))$ 
    ∪ f'({x} × (Sigma (B x - ita) (C x)))
    ∪ f'({x} × ({xa} × (C x xa - it'')))
  ) in FOREACH-rule)
  ) apply (auto intro: assms) [2]

apply auto
done

```

lemma *bij-set-rel-for-inj*:

```

fixes R
defines  $\alpha \equiv \text{fun-of-rel } R$ 
assumes bijjective R (s,s') ∈ ⟨R⟩ set-rel
shows inj-on  $\alpha\ s\ s' = \alpha\ s$ 
— To be used when generating refinement conditions for foreach-loops
using assms
unfolding bijjective-def set-rel-def  $\alpha$ -def fun-of-rel-def[abs-def]
apply (auto intro!: inj-onI ImageI simp: image-def)
apply (metis (mono-tags) Domain.simps contra-subsetD tfl-some)
apply (metis (mono-tags) someI)
apply (metis DomainE contra-subsetD tfl-some)

```

done

lemma *nfoldli-by-idx-gen*:

shows $\text{nfoldli } (\text{drop } k \ l) \ c \ f \ s = \text{nfoldli } [k..<\text{length } l] \ c \ (\lambda i \ s. \text{do } \{$
 $\text{ASSERT } (i < \text{length } l);$
 $\text{let } x = !i;$
 $f \ x \ s$
 $\}) \ s$

proof (*cases* $k \leq \text{length } l$)

case *False* **thus** *?thesis* **by** *auto*

next

case *True* **thus** *?thesis*

proof (*induction arbitrary: s rule: inc-induct*)

case *base* **thus** *?case*

by *auto*

next

case (*step* k)

from *step.hyps* **have** $1: \text{drop } k \ l = !k \ \# \ \text{drop } (\text{Suc } k) \ l$

by (*auto simp: Cons-nth-drop-Suc*)

from *step.hyps* **have** $2: [k..<\text{length } l] = k\#[\text{Suc } k..<\text{length } l]$

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

show *?case*

unfolding $1 \ 2$

by (*auto simp: step.IH[abs-def] step.hyps*)

qed

qed

lemma *nfoldli-by-idx*:

$\text{nfoldli } l \ c \ f \ s = \text{nfoldli } [0..<\text{length } l] \ c \ (\lambda i \ s. \text{do } \{$
 $\text{ASSERT } (i < \text{length } l);$
 $\text{let } x = !i;$
 $f \ x \ s$
 $\}) \ s$

using *nfoldli-by-idx-gen[of 0]* **by** *auto*

lemma *nfoldli-map-inv*:

assumes *inj g*

shows $\text{nfoldli } l \ c \ f = \text{nfoldli } (\text{map } g \ l) \ c \ (\lambda x \ s. f \ (\text{the-inv } g \ x) \ s)$

using *assms*

apply (*induction l*)

subgoal **by** *auto*

subgoal **by** (*auto simp: the-inv-f-f*)

done

lemma *nfoldli-shift*:

fixes *ofs* $:: \text{nat}$

shows $nfoldli\ l\ c\ f = nfoldli\ (map\ (\lambda i. i+ofs)\ l)\ c\ (\lambda x\ s. do\ \{ASSERT\ (x \geq ofs);\ f\ (x - ofs)\ s\})$
by $(induction\ l)\ auto$

lemma *nfoldli-foreach-shift*:

shows $nfoldli\ [a..<b]\ c\ f = nfoldli\ [a+ofs..<b+ofs]\ c\ (\lambda x\ s. do\ \{ASSERT\ (x \geq ofs);\ f\ (x - ofs)\ s\})$
using *nfoldli-shift[of [a..<b] c f ofs]*
by $(auto\ simp: map-add-upt')$

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 –

have $nfoldli\ l\ (\lambda f. \neg f)\ (\lambda y. RETURN\ (y=x))\ s \leq SPEC\ (\lambda r. r \longleftrightarrow s \vee x \in set\ l)$
for s

apply $(induction\ l\ arbitrary: s)$

subgoal **by** $auto$

subgoal **for** $a\ l\ s$

apply $clarsimp$

apply $(rule\ order-trans)$

apply $rprems$

by $auto$

done

from $this[of\ False]$ **show** $?thesis$ **by** $auto$

qed

definition *sum-impl* :: $('a \Rightarrow 'b::comm-monoid-add\ nres) \Rightarrow 'a\ set \Rightarrow 'b\ nres$
where

$sum-impl\ g\ S \equiv FOREACH\ S\ (\lambda x\ a. do\ \{ b \leftarrow g\ x; RETURN\ (a+b)\})\ 0$

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 $sum-impl\ gi\ S \leq SPEC\ (\lambda r. r=sum\ g\ S)$

unfolding *sum-impl-def*

apply $(refine-vcg\ FOREACH-rule[where\ I=\lambda it\ a. a = sum\ g\ (S - it)])$

apply $(auto\ simp: it-step-insert-iff\ algebra-simps)$

done

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 ‹

```

signature REFINE-AUTOMATION = sig
  type extraction = {
    pattern: term,    (* Pattern to be defined as own constant *)
    gen-thm: thm,    (* Code eq generator: [| c==rhs; ... |] ==> c == ... *)
    gen-tac: local-theory -> tactic' (* Solves remaining premises of gen-thm *)
  }

  val extract-as-def: (string * typ) list -> string -> term
    -> local-theory -> ((term * thm) * local-theory)

  val extract-recursion-egs: extraction list -> string -> thm
    -> local-theory -> local-theory

  val add-extraction: string -> extraction -> theory -> theory

  val prepare-code-thms-cmd: string list -> thm -> local-theory -> local-theory

  val define-concrete-fun: extraction list option -> binding ->
    Token.src list -> indexname list -> thm ->
    cterm list -> local-theory -> (thm * thm) * local-theory

  val mk-qualified: string -> bstring -> binding

  val prepare-cd-pattern: Proof.context -> cterm -> cterm
  val add-cd-pattern: cterm -> Context.generic -> Context.generic
  val del-cd-pattern: cterm -> Context.generic -> Context.generic
  val get-cd-patterns: Proof.context -> cterm list

  val add-vc-rec-thm: thm -> Context.generic -> Context.generic
  val del-vc-rec-thm: thm -> Context.generic -> Context.generic
  val get-vc-rec-thms: Proof.context -> thm list

  val add-vc-solve-thm: thm -> Context.generic -> Context.generic
  val del-vc-solve-thm: thm -> Context.generic -> Context.generic
  val get-vc-solve-thms: Proof.context -> thm list

  val vc-solve-tac: Proof.context -> bool -> tactic'
  val vc-solve-modifiers: Method.modifier parser list

  val setup: theory -> theory

```

end

structure Refine-Automation :REFINE-AUTOMATION = struct

```

type extraction = {
  pattern: term, (* Pattern to be defined as own constant *)
  gen-thm: thm, (* Code eq generator: [] c==rhs; ... [] ==> c == ... *)
  gen-tac: local-theory -> tactic' (* Solves remaining premises of gen-thm *)
}

structure extractions = Generic-Data (
  type T = extraction list Syntab.table
  val empty = Syntab.empty
  val extend = I
  val merge = Syntab.merge-list ((op =) o apply2 #pattern)
)

fun add-extraction name ex =
  Context.theory-map (extractions.map (
    Syntab.update-list ((op =) o apply2 #pattern) (name,ex)))

(*
  Define new constant name for subterm t in context bnd.
  Returns replacement for t using the new constant and definition
  theorem.
*)
fun extract-as-def bnd name t lthy = let
  val loose = rev (loose-bnos t);

  val rnames = #1 (Variable.names-of lthy |> fold-map (Name.variant o #1)
bnd);
  val rfrees = map (fn (name,(-,typ)) => Free (name,typ)) (rnames ~~ bnd);
  val t' = subst-bounds (rfrees,t);
  val params = map Bound (rev loose);

  val param-vars
    = (Library.foldl (fn (l,i) => nth rfrees i :: l) ([],loose));
  val param-types = map fastype-of param-vars;

  val def-t = Logic.mk-equals
    (list-comb (Free (name,param-types ---> fastype-of t'),param-vars),t');

  val ((lhs-t,(-,def-thm)),lthy)
    = Specification.definition NONE [] [] (Binding.empty-atts,def-t) lthy;

  (*val - = tracing xxx;*)
  val app-t = list-comb (lhs-t, params);
in
  ((app-t,def-thm),lthy)

```

end;

fun mk-qualified basename q = Binding.qualify true basename (Binding.name q);

fun extract-recursion-egs exs basename orig-def-thm lthy = let

val thy = Proof-Context.theory-of lthy

*val pat-net : extraction Item-Net.T =
Item-Net.init ((op =) o apply2 #pattern) (fn {pattern, ...} => [pattern])
|> fold Item-Net.update exs*

local

*fun tr env t ctx = let
(* Recurse for subterms *)
val (t,ctx) = case t of
t1\$t2 => let
val (t1,ctx) = tr env t1 ctx
val (t2,ctx) = tr env t2 ctx
in
(t1\$t2,ctx)
end
| Abs (x,T,t) => let
val (t',ctx) = tr ((x,T)::env) t ctx
in (Abs (x,T,t'),ctx) end
| - => (t,ctx)*

(Check if we match a pattern *)
val ex =
Item-Net.retrieve-matching pat-net t
|> get-first (fn ex =>
case
try (Pattern.first-order-match thy (#pattern ex,t))
(Vartab.empty,Vartab.empty)
of NONE => NONE | SOME - => SOME ex
)*

in

*case ex of
NONE => (t,ctx)
| SOME ex => let
(* Extract as new constant *)
val (idx,defs,lthy) = ctx
val name = (basename ^ - ^ string-of-int idx)
val ((t,def-thm),lthy) = extract-as-def env name t lthy
val ctx = (idx+1,(def-thm,ex)::defs,lthy)
in
(t,ctx)
end*

```

    end
  in
    fun transform t lthy = let
      val (t,(-,defs,lthy)) = tr [] t (0,[],lthy)
    in
      ((t,defs),lthy)
    end
  end
end

(* Import theorem and extract RHS *)
val ((-,orig-def-thm'),lthy) = yield-singleton2
  (Variable.import true) orig-def-thm lthy;
val (lhs,rhs) = orig-def-thm' |> Thm.prop-of |> Logic.dest-equals;

(* Transform RHS, generating new constants *)
val ((rhs',defs),lthy) = transform rhs lthy;
val def-thms = map #1 defs

(* Register definitions of generated constants *)
val (-,lthy)
  = Local-Theory.note ((mk-qualified basename defs,[]),def-thms) lthy;

(* Obtain new def-thm *)
val def-unfold-ss =
  put-simpset HOL-basic-ss lthy addsimps (orig-def-thm::def-thms)
val new-def-thm = Goal.prove-internal lthy
  [] (Logic.mk-equals (lhs,rhs') |> Thm.cterm-of lthy) (K (simp-tac def-unfold-ss
1))

(* Obtain new theorem by folding with defs of generated constants *)
(* TODO: Maybe cleaner to generate eq-thm and prove by unfold, refl *)
(*val new-def-thm
  = Library.foldr (fn (dt,t) => Local-Defs.fold lthy [dt] t)
  (def-thms,orig-def-thm');*)

(* Prepare code equations *)
fun mk-code-thm lthy (def-thm,{gen-thm, gen-tac, ...}) = let
  val ((-,def-thm),lthy') = yield-singleton2
    (Variable.import true) def-thm lthy;
  val thm = def-thm RS gen-thm;
  val tac = SOLVED' (gen-tac lthy')
    ORELSE' (simp-tac def-unfold-ss THEN' gen-tac lthy')

  val thm = the (SINGLE (ALLGOALS tac) thm);
  val thm = singleton (Variable.export lthy' lthy) thm;
in
  thm
end;

```

```

val code-thms = map (mk-code-thm lthy) defs;

val - = if forall Thm.no-prems code-thms then () else
  warning Unresolved premises in code theorems

val (-,lthy) = Local-Theory.note
  ((mk-qualified basename code,@{attributes [code]}),new-def-thm::code-thms)
  lthy;

in
  lthy
end;

fun prepare-code-thms-cmd names thm lthy = let
  fun name-of (Const (n,-)) = n
    | name-of (Free (n,-)) = n
    | name-of - = raise (THM (No definitional theorem,0,[thm]));

  val (lhs,-) = thm |> Thm.prop-of |> Logic.dest-equals;
  val basename = lhs |> strip-comb |> #1
    |> name-of
    |> Long-Name.base-name;

  val exs-tab = extractions.get (Context.Proof lthy)
  fun get-exs name =
    case Symtab.lookup exs-tab name of
      NONE => error (No such extraction mode: ^ name)
    | SOME exs => exs

  val exs = case names of
    [] => Symtab.dest-list exs-tab |> map #2
  | - => map get-exs names |> flat

  val - = case exs of [] => error No extraction patterns selected | - => ()

  val lthy = extract-recursion-egs exs basename thm lthy
in
  lthy
end;

fun extract-concrete-fun - [] concl =
  raise TERM (Conclusion does not match any extraction pattern,[concl])
| extract-concrete-fun thy (pat::pats) concl = (
  case Refine-Util.fo-matchp thy pat concl of
    NONE => extract-concrete-fun thy pats concl
  | SOME [t] => t
  | SOME (t::-) => (
    warning (concrete-definition: Pattern has multiple holes, taking

```

```

      ^first one: ^@{make-string} pat
    ); t)
  | - => (warning (concrete-definition: Ignoring invalid pattern
    ^@{make-string} pat);
    extract-concrete-fun thy pats concl)
)

(* Define concrete function from refinement lemma *)
fun define-concrete-fun gen-code fun-name attribs-raw param-names thm pats
  (orig-lthy:local-theory) =
let
  val lthy = orig-lthy;
  val ((-,inst),thm'),lthy) = yield-singleton2 (Variable.import true) thm lthy;

  val concl = thm' |> Thm.concl-of

  (*val ((typ-subst,term-subst),lthy)
    = Variable.import-inst true [concl] lthy;
  val concl = Term-Subst.instantiate (typ-subst,term-subst) concl;
  *)

  val term-subst = build (inst |> Vars.fold (cons o apsnd Thm.term-of))

  val param-terms = map (fn name =>
    case AList.lookup (fn (n,v) => n = #1 v) term-subst name of
      NONE => raise TERM (No such variable:
        ^Term.string-of-vname name,[concl])
      | SOME t => t
    ) param-names;

  val f-term = extract-concrete-fun (Proof-Context.theory-of lthy) pats concl;

  val lhs-type = map Term.fastype-of param-terms ---> Term.fastype-of f-term;
  val lhs-term
    = list-comb ((Free (Binding.name-of fun-name,lhs-type)),param-terms);
  val def-term = Logic.mk-equals (lhs-term,f-term)
    |> fold Logic.all param-terms;

  val attribs = map (Attrib.check-src lthy) attribs-raw;

  val ((-,(-,def-thm)),lthy) = Specification.definition
    (SOME (fun-name,NONE,Mixfix.NoSyn)) [] [] ((Binding.empty,attribs),def-term)
lthy;

  val folded-thm = Local-Defs.fold lthy [def-thm] thm';

  val (-,lthy)
    = Local-Theory.note

```

```

    ((mk-qualified (Binding.name-of fun-name) refine,[]),[folded-thm])
    lthy;

val lthy = case gen-code of
  NONE => lthy
| SOME modes =>
  extract-recursion-eqs modes (Binding.name-of fun-name) def-thm lthy

in
  ((def-thm,folded-thm),lthy)
end;

val cd-pat-eq = apply2 (Thm.term-of #> Refine-Util.anorm-term) #> (op aconv)

structure cd-patterns = Generic-Data (
  type T = cterm list
  val empty = []
  val extend = I
  val merge = merge cd-pat-eq
)

fun prepare-cd-pattern ctxt pat =
  case Thm.term-of pat |> fastype-of of
    @{typ bool} =>
      Thm.term-of pat
      |> HOLogic.mk-Trueprop
      |> Thm.cterm-of ctxt
  | - => pat

fun add-cd-pattern pat context =
  cd-patterns.map (insert cd-pat-eq (prepare-cd-pattern (Context.proof-of context)
pat)) context

fun del-cd-pattern pat context =
  cd-patterns.map (remove cd-pat-eq (prepare-cd-pattern (Context.proof-of context)
pat)) context

val get-cd-patterns = cd-patterns.get o Context.Proof

structure rec-thms = Named-Thms (
  val name = @{binding vcs-rec}
  val description = VC-Solver: Recursive intro rules
)

structure solve-thms = Named-Thms (
  val name = @{binding vcs-solve}

```

```

    val description = VC-Solver: Solve rules
  )

  val add-vc-rec-thm = rec-thms.add-thm
  val del-vc-rec-thm = rec-thms.del-thm
  val get-vc-rec-thms = rec-thms.get

  val add-vc-solve-thm = solve-thms.add-thm
  val del-vc-solve-thm = solve-thms.del-thm
  val get-vc-solve-thms = solve-thms.get

  val rec-modifiers = [
    Args.$$$ rec -- Scan.option Args.add -- Args.colon
    >> K (Method.modifier rec-thms.add here),
    Args.$$$ rec -- Scan.option Args.del -- Args.colon
    >> K (Method.modifier rec-thms.del here)
  ];

  val solve-modifiers = [
    Args.$$$ solve -- Scan.option Args.add -- Args.colon
    >> K (Method.modifier solve-thms.add here),
    Args.$$$ solve -- Scan.option Args.del -- Args.colon
    >> K (Method.modifier solve-thms.del here)
  ];

  val vc-solve-modifiers =
    clasimp-modifiers @ rec-modifiers @ solve-modifiers;

  fun vc-solve-tac ctxt no-pre = let
    val rthms = rec-thms.get ctxt
    val sthms = solve-thms.get ctxt
    val pre-tac = if no-pre then K all-tac else clarsimp-tac ctxt
    val tac = SELECT-GOAL (auto-tac ctxt)
  in
    TRY o pre-tac
    THEN-ALL-NEW-FWD (TRY o REPEAT-ALL-NEW-FWD (resolve-tac ctxt
rthms))
    THEN-ALL-NEW-FWD (TRY o SOLVED' (resolve-tac ctxt sthms THEN-ALL-NEW-FWD
tac))
  end

  val setup = I
    #> rec-thms.setup
    #> solve-thms.setup

end;
>

```


setup *Refine-Automation.setup*

```

setup <
  let
    fun parse-cpat cxt = let
      val (t, (context, tks)) = Scan.lift Args.embedded-inner-syntax cxt
      val ctxt = Context.proof-of context
      val t = Proof-Context.read-term-pattern ctxt t
    in
      (Thm.ctrm-of ctxt t, (context, tks))
    end

    fun do-p f = Scan.repeat1 parse-cpat >> (fn pats =>
      Thm.declaration-attribute (K (fold f pats)))
  in
    Attrib.setup @({binding cd-patterns} (
      Scan.lift Args.add |-- do-p Refine-Automation.add-cd-pattern
    || Scan.lift Args.del |-- do-p Refine-Automation.del-cd-pattern
    || do-p Refine-Automation.add-cd-pattern
    )
      Add/delete concrete-definition pattern
    end
  >

```

ML <*Outer-Syntax.local-theory*

```

@{command-keyword concrete-definition}
Define function from refinement theorem
(Parse.binding
  -- Parse.opt-attribs
  -- Scan.optional (@{keyword for} |-- Scan.repeat1 Args.var) []
  --| @{keyword uses} -- Parse.thm
  -- Scan.optional (@{keyword is} |-- Scan.repeat1 Args.embedded-inner-syntax)
)
>> (fn (((name,attribs),params),raw-thm),pats) => fn lthy => let
  val thm =
    case Attrib.eval-thms lthy [raw-thm] of
      [thm] => thm
    | - => error Expecting exactly one theorem;
  val pats = case pats of
    [] => Refine-Automation.get-cd-patterns lthy
  | l => map (Proof-Context.read-term-pattern lthy #> Thm.ctrm-of lthy #>
    Refine-Automation.prepare-cd-pattern lthy) l
in

```

```

    Refine-Automation.define-concrete-fun
      NONE name attribs params thm pats lthy
    |> snd
  end))
>

```

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 <
  let
    val modes = (Scan.optional
      (@{keyword } |-- Parse.list1 Parse.name --| @{keyword }))) []
  in
    Outer-Syntax.local-theory
    @command-keyword prepare-code-thms}
    Refinement framework: Prepare theorems for code generation
    (modes -- Parse.thms1
      >> (fn (modes,raw-thms) => fn lthy => let
        val thms = Attrib.eval-thms lthy raw-thms
      in
        fold (Refine-Automation.prepare-code-thms-cmd modes) thms lthy
      end)
    )
  end
>

```

Command: *prepare-code-thms (modes) thm* where the (*mode*)-part is optional.

Set up code-equations for recursions in constant defined by *thm*. The op-

tional *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  $f\ x \equiv B\ f\ x$ 
  unfolding D
  apply (subst RECT-unfold)
  by (rule M)
```

```
lemma gen-code-thm-REC:
  fixes x
  assumes D:  $f \equiv REC\ B$ 
  assumes M: trimono B
  shows  $f\ x \equiv B\ f\ x$ 
  unfolding D
  apply (subst REC-unfold)
  by (rule M)
```

```
setup <
  Refine-Automation.add-extraction nres {
    pattern = Logic.verify-global @{term REC x},
    gen-thm = @{thm gen-code-thm-RECT},
    gen-tac = Refine-Mono-Prover.mono-tac
  }
#>
  Refine-Automation.add-extraction nres {
    pattern = Logic.verify-global @{term RECT x},
    gen-thm = @{thm gen-code-thm-RECT},
    gen-tac = Refine-Mono-Prover.mono-tac
  }
>
```

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.

```
method-setup vc-solve =
  <Scan.lift (Args.mode nopre)
  --| Method.sections Refine-Automation.vc-solve-modifiers >>
  (fn (nopre) => fn ctxt => SIMPLE-METHOD (
    CHANGED (ALLGOALS (Refine-Automation.vc-solve-tac ctxt nopre))
  ))> Try to solve verification conditions
```

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  $(RETURN\ c, a) \in \langle R \rangle nres\text{-}rel \quad RETURN\ c \leq \Downarrow R\ a$ 
  using assms
  unfolding nres-rel-def
  by simp-all
```

```
lemma autoref-monadicI2:
  assumes  $(b,a) \in \langle R \rangle nres\text{-}rel$ 
  assumes  $nres\text{-}of\ c \leq b$ 
  shows  $(nres\text{-}of\ c, a) \in \langle R \rangle nres\text{-}rel \quad nres\text{-}of\ c \leq \Downarrow R\ a$ 
  using assms
  unfolding nres-rel-def
  by simp-all
```

```
lemmas autoref-monadicI = autoref-monadicI1 autoref-monadicI2
```

```
ML <
```

```
structure Autoref-Monadic = struct
```

```
  val cfg-plain = Attrib.setup-config-bool @{binding autoref-plain} (K false)
```

```
  fun autoref-monadic-tac ctxt = let
    open Autoref-Tacticals
    val ctxt = Autoref-Phases.init-data ctxt
    val plain = Config.get ctxt cfg-plain
    val trans-thms = if plain then [] else @{thms the-resI}
```

```
  in
    resolve-tac ctxt @{thms autoref-monadicI}
    THEN'
    IF-SOLVED (Autoref-Phases.all-phases-tac ctxt)
      (RefineG-Transfer.post-transfer-tac trans-thms ctxt)
      (K all-tac) (* Autoref failed *)
```

```
  end
```

```
end
```

```
>
```

```

method-setup autoref-monadic = <let
  open Refine-Util Autoref-Monadic
  val autoref-flags =
    parse-bool-config trace Autoref-Phases.cfg-trace
    || parse-bool-config debug Autoref-Phases.cfg-debug
    || parse-bool-config plain Autoref-Monadic.cfg-plain

  in
    parse-paren-lists autoref-flags
  >>
  ( fn - => fn ctxt => SIMPLE-METHOD' (
    let
      val ctxt = Config.put Autoref-Phases.cfg-keep-goal true ctxt
    in autoref-monadic-tac ctxt end
    ))

  end

>
  Automatic Refinement and Determinization for the Monadic Refinement Frame-
  work

end

```

2.18 Refinement Framework

```

theory Refine-Monadic
imports
  Refine-Chapter
  Refine-Basic
  Refine-Leof
  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* ≡

$REC (\lambda D x. do \{ ASSERT (pre\ x); r \leftarrow body\ D\ x; ASSERT (post\ x\ r); RETURN\ r \})\ x$

theorem *REC-annot-rule*[*refine-vcg*]:

assumes *M*: *trimono body*

and *P*: *pre x*

and *S*: $\bigwedge f\ x. [\bigwedge x. pre\ x \implies f\ x \leq SPEC\ (post\ x); pre\ x] \implies body\ f\ x \leq SPEC\ (post\ x)$

and *C*: $\bigwedge r. post\ x\ r \implies \Phi\ r$

shows *REC-annot pre post body x* $\leq SPEC\ \Phi$

proof –

from *trimono body* **have** [*refine-mono*]:

$\bigwedge f\ g\ x\ xa. (\bigwedge x. flat\text{-}ge\ (f\ x)\ (g\ x)) \implies flat\text{-}ge\ (body\ f\ x)\ (body\ g\ x)$

$\bigwedge f\ g\ x\ xa. (\bigwedge x. f\ x \leq g\ x) \implies body\ f\ x \leq body\ g\ x$

apply –

unfolding *trimono-def monotone-def fun-ord-def mono-def le-fun-def*

apply (*auto*)

done

show *?thesis*

unfolding *REC-annot-def*

apply (*rule order-trans*[**where** *y*=*SPEC (post x)*])

apply (*refine-rcg*

refine-vcg

REC-rule[**where** *pre*=*pre* **and** *M*= $\lambda x. SPEC\ (post\ x)$]

order-trans[*OF S*]

)

apply *fact*

apply *simp*

using *C* **apply** (*auto*) []

done

qed

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 *WHILET* (*while_T*)

notation *WHILEI* (*while⁻*)

notation *WHILET* (*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: $\llbracket \text{dist } v \ d \ v h; v' \in \text{succ } v h \rrbracket \implies \text{dist } v \ (\text{Suc } d) \ v'$

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*]: $\text{dist } v \ d \ v' \implies \text{conn } v \ v'$
and *connI-id*[*intro*]: $\text{conn } v \ v$
and *connI-succ*[*intro*]: $\text{conn } v \ v' \implies v'' \in \text{succ } v' \implies \text{conn } v \ v''$
by (*auto simp: conn-def intro: dist-suc dist-z*)

lemma *min-distI2*:

$\llbracket \text{conn } v \ v'; \bigwedge d. \llbracket \text{dist } v \ d \ v'; \bigwedge d'. \text{dist } v \ d' \ v' \implies d \leq d' \rrbracket \implies Q \ d \rrbracket$
 $\implies Q \ (\text{min-dist } v \ v')$
unfolding *min-dist-def*
by (*rule LeastI2-wellorder*[**where** $Q=Q$ **and** $a=\text{SOME } d. \text{dist } v \ d \ v'$])
(*auto simp: conn-def intro: someI*)

lemma *min-distI-eq*:

$\llbracket \text{dist } v \ d \ v'; \bigwedge d'. \text{dist } v \ d' \ v' \implies d \leq d' \rrbracket \implies \text{min-dist } v \ v' = d$
by (*force intro: min-distI2 simp: conn-def*)

Two nodes are connected by a path of length 0, iff they are equal.

lemma *dist-z-iff*[*simp*]: $\text{dist } v \ 0 \ v' \longleftrightarrow v'=v$
by (*auto elim: dist.cases intro: dist.intros*)

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*]: $\text{min-dist } v \ v = 0$
by (*rule min-distI2*) (*auto intro: dist-z*)

lemma *min-dist-z-iff*[*simp*]: $\text{conn } v \ v' \implies \text{min-dist } v \ v' = 0 \longleftrightarrow v'=v$
by (*rule min-distI2*) (*auto intro: dist-z*)

lemma *min-dist-is-dist*: $\text{conn } v \ v' \implies \text{dist } v \ (\text{min-dist } v \ v') \ v'$
by (*auto intro: min-distI2*)

lemma *min-dist-minD*: $\text{dist } v \ d \ v' \implies \text{min-dist } v \ v' \leq d$
by (*auto intro: min-distI2*)

We also provide introduction and destruction rules for the pattern *min-dist* $v \ v' = \text{Suc } d$.

lemma *min-dist-succ*:

[[*conn* v v' ; $v'' \in \text{succ } v'$]] $\implies \text{min-dist } v$ $v'' \leq \text{Suc } (\text{min-dist } v$ $v')$
by (*rule min-distI2*[**where** $v'=v'$])
 (*auto elim: dist.cases intro!: min-dist-minD dist-suc*)

lemma *min-dist-suc*:

assumes c : *conn* v v' *min-dist* v $v' = \text{Suc } d$
shows $\exists v''$. *conn* v $v'' \wedge v' \in \text{succ } v'' \wedge \text{min-dist } v$ $v'' = d$
proof –
from *min-dist-is-dist*[*OF* $c(1)$]
have *min-dist* v $v' = \text{Suc } d \longrightarrow ?thesis$
proof cases
case (*dist-suc* d' v'') **then show** $?thesis$
using *min-dist-succ*[*of* v v'' v'] *min-dist-minD*[*of* v d v'']
by (*auto simp: connI*)
qed simp
with c **show** $?thesis$ **by simp**
qed

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 *conn src* v *min-dist src* $v = d$ **and** $d' < d$
shows $\exists v'$. *conn src* $v' \wedge \text{min-dist src } v' = d'$
using *assms*
proof (*induct d arbitrary: v*)
case (*Suc* d) **with** *min-dist-suc*[*of src v*] **show** $?case$
by (*cases d' = d*) *auto*
qed auto

Lemma *min-dist-less* can be weakened to $d' \leq d$.

corollary *min-dist-le*:

assumes c : *conn src* v **and** d' : $d' \leq \text{min-dist src } v$
shows $\exists v'$. *conn src* $v' \wedge \text{min-dist src } v' = d'$
using *min-dist-less*[*OF* c , *of min-dist src v d*] $d' c$
by (*auto simp: le-less*)

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 $bfs-invar' src dst \sigma \equiv let (f, V, C, N, d) = \sigma in$

$$\begin{aligned}
 & (\neg f \rightarrow (\\
 & \quad N \subseteq V \wedge dst \notin V - (C \cup N) \wedge \\
 & \quad (\forall v \in C. conn src v \wedge min-dist src v = d) \wedge \\
 & \quad (\forall v \in N. conn src v \wedge min-dist src v = Suc d) \wedge \\
 & \quad (\forall v \in V. conn src v \wedge min-dist src v \leq Suc d) \wedge \\
 & \quad (\forall v. conn src v \wedge min-dist src v \leq d \rightarrow v \in V) \wedge \\
 & \quad (\forall v. conn src v \wedge min-dist src v = Suc d \rightarrow v \in N \cup ((\bigcup (succ' C)) - V)) \\
 &)) \wedge (\\
 & \quad f \rightarrow conn src dst \wedge min-dist src dst = d \\
 &)
 \end{aligned}$$

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 σ

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 $FE-invar V N v it \sigma \equiv let (V', N') = \sigma in$

$$\begin{aligned}
 & V' = V \cup (succ v - it) \wedge \\
 & N' = N \cup ((succ v - it) - V)
 \end{aligned}$$

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 *do* c *end* f σ_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=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 ∪ succ v, C - {v}, N ∪ (succ v - V), d)
    using assms
  proof (simp (no-asm-use) add: bfs-invar'-def, intro conjI, goal-cases)
    case 6 then show ?case
      by (metis Un-iff Diff-iff connI-succ le-SucE min-dist-succ)
    next
      case 7 then show ?case
        by (metis Un-iff connI-succ min-dist-succ)
  qed blast+

```

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)
  using assms
  proof (simp (no-asm-use) add: bfs-invar'-def, intro conjI impI allI, goal-cases)
    case 3 then show ?case
      by (metis le-SucI)
    next
      case 4 then show ?case
        by (metis le-SucE subsetD)
    next
      case (5 v) then show ?case
        using min-dist-suc[of src v Suc d] by metis
    next
      case 6 then show ?case
        by (metis Suc-n-not-le-n)
  qed blast+

```

The invariant holds initially.

```

lemma invar-init: bfs-invar src dst (False, {src}, {src}, {}, 0)
  by (auto simp: bfs-invar-def bfs-invar'-def empty-asm-def)
  (metis min-dist-suc min-dist-z-iff)

```

The invariant is preserved if we break the loop.

```

lemma invar-break:
  assumes bfs-invar src dst (False, V, C, N, d)
  assumes dst ∈ C
  shows bfs-invar src dst (True, {}, {}, {}, d)
  using assms unfolding bfs-invar-def bfs-invar'-def empty-asm-def
  by auto

```

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 min-dist src dst = d
using assms unfolding bfs-invar'-def
by auto

```

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: conn src dst — There is a path between src and dst.
  assumes INV: bfs-invar' src dst (False, V, {}, {}, d)
  shows False
proof —

```

We make a case-distinction whether $d' \leq d$:

```

have min-dist src dst ≤ d ∨ d + 1 ≤ min-dist src dst by auto
moreover {

```

Due to the invariant, any node with a path of length at most *d* is contained in *V*

```

from INV have  $\forall v. \text{conn } src \ v \wedge \text{min-dist } src \ v \leq d \longrightarrow v \in V$ 
unfolding bfs-invar'-def by auto
moreover

```

This applies in the first case, hence we obtain that $dst \in V$

```

assume A: min-dist src dst ≤ d
moreover from C have dist src (min-dist src dst) dst
by (blast intro: min-dist-is-dist)
ultimately have dst ∈ V by blast

```

However, as *C* and *N* are empty, the invariant also implies that *dst* is not in *V*:

```

moreover from INV have dst ∉ V unfolding bfs-invar'-def by auto

```

This yields a contradiction:

```

ultimately have False by blast
} moreover {

```

In the case $d+1 \leq d'$, we also obtain a node that has a shortest path of length $d+1$:

```

assume d+1 ≤ min-dist src dst
with min-dist-le[OF C] obtain v' where
  conn src v' min-dist src v' = d + 1
by auto

```

However, the invariant states that such nodes are either in *N* or are successors of *C*. As *N* and *C* are both empty, we again get a contradiction.


```

    with INV have False unfolding bfs-invar'-def by auto
  } ultimately show ?thesis by blast
qed

```

Finally, we prove our algorithm correct: The following theorem solves both verification tasks.

Note that a proposition of the form $S \sqsubseteq SPEC \Phi$ 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  $\Rightarrow$   $\neg$  conn src dst
    | Some d  $\Rightarrow$  conn src dst  $\wedge$  min-dist src dst = d)

```

```

theorem bfs-correct: bfs src dst  $\leq$  bfs-spec src dst

```

```

  unfolding bfs-def bfs-spec-def
  apply (intro refine-vcg)

```

```

  unfolding FE-invar-def
  apply (auto simp:
    intro: invar-init invar-break )

```

```

  unfolding bfs-invar-def
  apply (auto simp:
    intro: invar-succ-step invar-empty-step invar-final-succeed)

```

```

  unfolding empty-asm-def
  apply (auto intro: invar-final-fail)

```

```

  unfolding bfs-invar'-def
  apply auto
done

```

```

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*]: $\text{word-nat-rel} \equiv \text{build-rel } (\text{unat}) (\lambda-. \text{True})$

lemma $\text{word-nat-RELEATES}[\text{refine-dref-RELATES}]$:
 $\text{RELATES word-nat-rel by (simp add: RELATES-def)}$

lemma [*simp, relator-props*]:
 $\text{single-valued word-nat-rel unfolding word-nat-rel-def}$
by *blast*

lemma [*simp*]: $\text{single-valuedp } (\lambda c a. a = \text{unat } c)$
by (*rule single-valuedpI*) *blast*

lemma [*simp, relator-props*]: $\text{single-valued } (\text{converse word-nat-rel})$
by (*auto intro: injI*)

lemmas [*refine-hsimp*] =
 $\text{word-less-nat-alt word-le-nat-alt unat-sub iffD1[OF unat-add-lem]}$

3.2.2 Example

type-synonym $\text{word32} = 32 \text{ word}$

definition $\text{test} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat set nres}$ **where** $\text{test } x0 \ y0 \equiv \text{do } \{$
 $\text{let } S = \{\};$
 $(S, -, -) \leftarrow \text{WHILE } (\lambda(S, x, y). x > 0) (\lambda(S, x, y). \text{do } \{$
 $\text{let } S = S \cup \{y\};$
 $\text{let } x = x - 1;$
 $\text{ASSERT } (y < x0 + y0);$
 $\text{let } y = y + 1;$
 $\text{RETURN } (S, x, y)$
 $\}) (S, x0, y0);$
 $\text{RETURN } S$
 $\}$

lemma $y0 > 0 \implies \text{test } x0 \ y0 \leq \text{SPEC } (\lambda S. S = \{y0 .. y0 + x0 - 1\})$
— Choosen pre-condition to get least trouble when proving

unfolding *test-def*

apply (*intro WHILE-rule[where I = $\lambda(S, x, y).$*

$x + y = x0 + y0 \wedge x \leq x0 \wedge$
 $S = \{y0 .. y0 + (x0 - x) - 1\}$
refine-vcg)

by *auto*

definition *test-impl* :: *word32* \Rightarrow *word32* \Rightarrow *word32* *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 $(x,x') \in \text{word-nat-rel}$

assumes $(y,y') \in \text{word-nat-rel}$

shows $\text{test-impl } x \ y \leq \Downarrow(\langle \text{word-nat-rel} \rangle \text{set-rel}) (\text{test } x' \ y')$

proof –

from *assms* **show** *?thesis*

unfolding *test-impl-def test-def*

apply (*refine-rcg*)

apply (*refine-dref-type*)

apply (*auto simp: refine-hsimp refine-rel-defs*)

done

qed

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.