Refinement for Monadic Programs

Peter Lammich

December 14, 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 Miscellanneous 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 Miscellanneous 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 ‹*refine-mono-prover.ML*›

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

method-setup *refine-mono* =
‹*Scan.succeed* (*fn ctxt => SIMPLE-METHOD'* (
Refine-Mono-Prover.untriggered-mono-tac ctxt
))›
Refinement framework: *Monotonicity prover*

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

lemma *mono-if*: ⟦*le t t'*; *le e e'*⟧ ⟹ *le* (*If b t e*) (*If b t' e'*) **by** *auto*
lemma *mono-let*: (⋀*x*. *le* (*f x*) (*f' x*)) ⟹ *le* (*Let x f*) (*Let x f'*) **by** *auto*

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

declaration ‹*Refine-Mono-Prover.declare-mono-triggers* @{*thms monoI*}›

end

interpretation *order-mono-setup*: *mono-setup-loc* (≤) :: *'a*::*preorder* ⇒ -
by *standard auto*

declaration ‹*Refine-Mono-Prover.declare-mono-triggers* @{*thms Orderings.monoI*}›

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

end

2.1 Miscellanneous Lemmas and Tools

theory *Refine-Misc*
imports
Automatic-Refinement.Automatic-Refinement
Refine-Mono-Prover
begin

Basic configuration for monotonicity prover:

lemmas [*refine-mono*] = *monoI monotoneI*[*of* (≤) (≤)]
lemmas [*refine-mono*] = *TrueI le-funI order-refl*

lemma *case-prod-mono*[*refine-mono*]:
⟦⋀*a b*. *p*=(*a*,*b*) ⟹ *f a b* ≤ *f' a b*⟧ ⟹ *case-prod f p* ≤ *case-prod f' p*
by (*auto split*: *prod.split*)

lemma *case-option-mono*[*refine-mono*]:
assumes *fn* ≤ *fn'*
assumes ⋀*v*. *x*=*Some v* ⟹ *fs v* ≤ *fs' v*
shows *case-option fn fs x* ≤ *case-option fn' fs' x*
using *assms* **by** (*auto split*: *option.split*)

lemma *case-list-mono*[*refine-mono*]:
assumes *fn* ≤ *fn'*
assumes ⋀*x xs*. *l*=*x*#*xs* ⟹ *fc x xs* ≤ *fc' x xs*
shows *case-list fn fc l* ≤ *case-list fn' fc' l*
using *assms* **by** (*auto split*: *list.split*)

lemma *if-mono*[*refine-mono*]:
assumes *b* ⟹ *m1* ≤ *m1'*
assumes ¬*b* ⟹ *m2* ≤ *m2'*
shows (*if b then m1 else m2*) ≤ (*if b then m1' else m2'*)
using *assms* **by** *auto*

lemma *let-mono*[*refine-mono*]:
f x ≤ *f' x'* ⟹ *Let x f* ≤ *Let x' f'* **by** *auto*

2.1.1 Uncategorized Lemmas

lemma *all-nat-split-at*: ∀ *i*::'*a*::*linorder*<*k*. *P i* ⟹ *P k* ⟹ ∀ *i*>*k*. *P i*

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

2.1.2 Well-Foundedness

lemma *wf-no-infinite-down-chainI*:
assumes $\bigwedge f.\ [\![\bigwedge i.\ (f\ (Suc\ i), f\ i)\in r]\!] \Longrightarrow False$
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*: $\bigwedge s\ s'\ t.\ [\![\ (s,s')\in R;\ (s,t)\in S;\ (x0',s')\in S'^{*}\ ]\!]$
$\Longrightarrow \exists\, t'.\ (s',t')\in S' \wedge (t,t')\in R$
assumes *CLOSED*: *Domain* $S \subseteq S^{*}\,``\{x0\}$
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** ‹$(y,z)\in S$›
by (*rule-tac step.prems*[**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 ∃f'. f' 0 = x0 ∧ (∀ i. (f' i, f' (Suc i))∈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 ∀ i. (f' i,f' (Suc i))∈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' (λi x. SOME x'.
      (x,x')∈S' ∧ (f' (Suc i),x')∈R ∧ (x0', x')∈S'* )
{
  fix i
  note [simp] = g'-def
  have (g' i, g' (Suc i))∈S' ∧ (f' (Suc i),g' (Suc i))∈R
    ∧ (x0',g' (Suc i))∈S'*
  proof (induct i)
    case 0
    from SIM[OF STARTR] CHAIN'[of 0] obtain t' where
      (x0',t')∈S' and (f' (Suc 0),t')∈R by auto
    moreover hence (x0',t')∈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')∈S' and (f' (Suc (Suc i)),t')∈R
      by auto
    moreover from LS Suc have (x0', t')∈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* $S \Longrightarrow$ *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*]:
$(x,y) \in$ *less-than-bool* $\longleftrightarrow$ $x=$*False* $\wedge$ $y=$*True*
by (*auto simp*: *less-than-bool-def*)

definition *greater-bounded* $N \equiv$ *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*]: $(Suc\ x,x) \in$ *greater-bounded* $N \longleftrightarrow Suc\ x \leq N$
by (*auto simp*: *greater-bounded-def*)

2.1.3 Monotonicity and Orderings

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

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

lemma *mono-infI'*:
mono $f \Longrightarrow$ *mono* $g \Longrightarrow$ *mono* $(\lambda x.$ *inf* $(f\ x)\ (g\ x) :: 'b::lattice)$
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 $f :: 'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice$
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 $f :: 'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice$
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 $f :: 'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice$
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*: $mono\ (f::'a::order \Rightarrow 'a) \Longrightarrow mono\ (f\widehat{\ }i)$
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*:
  $a \leq inf\ b\ c \Longrightarrow a \leq b$ **by** (*rule le-infE*)
lemma (**in** *semilattice-inf*) *le-infD2*:
  $a \leq inf\ b\ c \Longrightarrow a \leq c$ **by** (*rule le-infE*)
lemma (**in** *semilattice-inf*) *inf-leI*:
  $[\![\bigwedge x.\ [\![x \leq a;\ x \leq b]\!] \Longrightarrow x \leq c]\!] \Longrightarrow inf\ a\ b \leq c$
  by (*metis inf-le1 inf-le2*)

lemma *top-Sup*: $(top::'a::complete\text{-}lattice) \in A \Longrightarrow Sup\ A = top$
  by (*metis Sup-upper top-le*)
lemma *bot-Inf*: $(bot::'a::complete\text{-}lattice) \in A \Longrightarrow Inf\ A = bot$
  by (*metis Inf-lower le-bot*)

lemma *mono-compD*: $mono\ f \Longrightarrow x \leq y \Longrightarrow f\ o\ x \leq f\ o\ y$
  apply (*rule le-funI*)
  apply (*auto dest*: *monoD le-funD*)
  done

Galois Connections

locale *galois-connection* =
  fixes $\alpha::'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice$ **and** γ
  assumes *galois*: $c \leq \gamma(a) \longleftrightarrow \alpha(c) \leq a$
begin
  lemma $\alpha\gamma$-*defl*: $\alpha(\gamma(x)) \leq x$
  proof −
    have $\gamma\ x \leq \gamma\ x$ **by** *simp*
    with *galois* **show** $\alpha(\gamma(x)) \leq x$ **by** *blast*
  qed
  lemma $\gamma\alpha$-*infl*: $x \leq \gamma(\alpha(x))$
  proof −
    have $\alpha\ x \leq \alpha\ x$ **by** *simp*
    with *galois* **show** $x \leq \gamma(\alpha(x))$ **by** *blast*
  qed

  lemma α-*mono*: *mono* α
  proof
    fix $x::'a$ **and** $y::'a$
    assume $x \leq y$
    also note $\gamma\alpha$-*infl*[*of y*]
    finally show $\alpha\ x \leq \alpha\ y$ **by** (*simp add*: *galois*)
  qed

  lemma γ-*mono*: *mono* γ
    by *rule* (*metis* $\alpha\gamma$-*defl galois inf-absorb1 le-infE*)

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

apply (*rule mono-inf*[*OF* γ*-mono*])
apply (*simp add*: *galois*)
apply (*simp add*: *galois*[*symmetric*])
done

lemma *dist-*α[*simp*]:
α (*sup a b*) = *sup* (α *a*) (α *b*)
by (*metis* (*no-types*) α*-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*: $\bigwedge x.\ f\ x = x \Longrightarrow a \leq x$
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*: $\bigwedge x.\ f\ x = x \Longrightarrow x \leq a$
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* $\Longrightarrow$ *lfp f x* $\leq$ *gfp f x*
by (*rule le-funD*[*OF lfp-le-gfp*])

lemma *lfp-induct'*:
assumes *M*: *mono f*
assumes *IS*: $\bigwedge m.\ [\![\ m \leq lfp\ f;\ m \leq P\ ]\!] \Longrightarrow f\ m \leq P$
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* ≤ *P*
assumes *IS*: ⋀*m*. ⟦
m ≤ *lfp* (λ*s*. *sup m0* (*f s*)); — Assume already established invariants
m ≤ *P*; — Assume invariant
f m ≤ *lfp* (λ*s*. *sup m0* (*f s*)) — Assume that step preserved est. invars
⟧ ⟹ *f m* ≤ *P* — Show that step preserves invariant
shows *lfp* (λ*s*. *sup m0* (*f s*)) ≤ *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* (≤) (<)
apply *unfold-locales*
apply (*erule Sup-upper*)
apply (*erule Sup-least*)
done

lemma (**in** *complete-lattice*) *is-dual-ccpo*: *class.ccpo Inf* (≥) (>)
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* (≤) (≤) *f* ⟷ *mono f*
unfolding *monotone-def mono-def* **by** *simp*
lemma *ccpo-monoI*: *mono f* ⟹ *monotone* (≤) (≤) *f*
by (*simp add*: *ccpo-mono-simp*)
lemma *ccpo-monoD*: *monotone* (≤) (≤) *f* ⟹ *mono f*
by (*simp add*: *ccpo-mono-simp*)

lemma *dual-ccpo-mono-simp*: *monotone* (≥) (≥) *f* ⟷ *mono f*
unfolding *monotone-def mono-def* **by** *auto*

lemma *dual-ccpo-monoI*: *mono f* $\Longrightarrow$ *monotone* $(\geq)$ $(\geq)$ *f*
by (*simp add*: *dual-ccpo-mono-simp*)
lemma *dual-ccpo-monoD*: *monotone* $(\geq)$ $(\geq)$ *f* $\Longrightarrow$ *mono f*
by (*simp add*: *dual-ccpo-mono-simp*)

lemma *ccpo-lfp-simp*: $\bigwedge f$. *mono f* $\Longrightarrow$ *ccpo.fixp Sup* $(\leq)$ *f* = *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*: $\bigwedge f$. *mono f* $\Longrightarrow$ *ccpo.fixp Inf* $(\geq)$ *f* = *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* $\equiv$ *ccpo.admissible Sup* $(\leq)$ *P*
abbreviation *is-chain* $\equiv$ *Complete-Partial-Order.chain* $(\leq)$
lemmas *chain-admissibleI*[*intro?*] = *ccpo.admissibleI*[**where** *lub*=*Sup* **and** *ord*=$(\leq)$]

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

lemma *dual-chain-iff*[*simp*]: *is-dual-chain C* = *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*!]: *is-chain* {}
by (*rule chainI*) *auto*
lemma *is-dual-chain-empty*[*simp*, *intro*!]: *is-dual-chain* {}
by (*rule dual-chainI*) *auto*

lemma *point-chainI*: *is-chain M* $\Longrightarrow$ *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*:
$[\![$*chain-admissible P*; *P* (*Sup* {}); *mono f*; $\bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)]\!] \Longrightarrow P\ (lfp\ f)$
by (*simp only*: *ccpo-mono-simp*[*symmetric*] *ccpo-lfp-simp*[*symmetric*])
(*rule ccpo.fixp-induct*[*OF is-ccpo*])

lemma *gfp-cadm-induct*:
$[\![$*dual-chain-admissible P*; *P* (*Inf* {}); *mono f*; $\bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)]\!] \Longrightarrow P\ (gfp\ f)$
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* $\equiv \forall\ C.\ C{\neq}\{\} \longrightarrow f\ (Sup\ C) = Sup\ (f`C)$
definition *strict f* $\equiv f\ bot = bot$
definition *inf-distrib f* $\equiv$ *strict f* $\wedge$ *cont f*

lemma *contI*[*intro?*]: $[\![\bigwedge C.\ C{\neq}\{\} \Longrightarrow f\ (Sup\ C) = Sup\ (f`C)]\!] \Longrightarrow$ *cont f*
unfolding *cont-def* **by** *auto*
lemma *contD*: *cont f* $\Longrightarrow C{\neq}\{\} \Longrightarrow f\ (Sup\ C) = Sup\ (f\ `\ C)$
unfolding *cont-def* **by** *auto*
lemma *contD'*: *cont f* $\Longrightarrow C{\neq}\{\} \Longrightarrow f\ (Sup\ C) = Sup\ (f\ `\ C)$
by (*fact contD*)

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

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

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

lemma *inf-distribI*[*intro?*]: $[\![$*strict f*; *cont f*$]\!] \Longrightarrow$ *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* $\Longrightarrow f\ (Sup\ C) = Sup\ (f`C)$

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

lemma *inf-distribI'*:
fixes f :: $'a$::*complete-lattice* ⇒ $'b$::*complete-lattice*
assumes B: $\bigwedge C.\ f\ (Sup\ C) = Sup\ (f`C)$
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* ⇒ $'b$::*complete-lattice*
shows *cont f* ⟹ *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* ⇒ $'b$::*complete-lattice*
shows *inf-distrib f* ⟹ *mono f*
by *simp*

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

theorem *gen-kleene-lfp*:
fixes f:: $'a$::*complete-lattice* ⇒ $'a$
assumes *CONT*: *cont f*
shows *lfp* (λ*x*. *sup m* (*f x*)) = (*SUP i*. (*f*^^*i*) *m*)
proof (*rule antisym*)
let *?f* = (λ*x*. *sup m* (*f x*))
let *?K*={ (*f*^^*i*) *m* | *i* . *True*}
note *MONO*=*cont-is-mono*[*OF CONT*]
note *MONO'*[*refine-mono*] = *monoD*[*OF MONO*]
{
fix *i*
have (*f*^^*i*) *m* ≤ *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* α*-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 [$a \mapsto b$] = {(*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-ALIGNI*: *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 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 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
  fun dis-tac goal-ctxt = ALLGOALS (Tagged-Solver.solve-tac goal-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-ALIGNI}
          |> 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** α :: *'c* ⇒ *'a*::*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 α (*if b then s1 else s2*) ≤ (*if b then S1 else S2*)
using *assms* **by** *auto*

lemma *transfer-prod*[*refine-transfer*]:
assumes $\bigwedge a\ b.\ \alpha\ (f\ a\ b) \leq F\ a\ b$
shows α (*case-prod f x*) ≤ (*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 α (*Let x f*) ≤ *Let x F*
using *assms* **by** *auto*

lemma *transfer-option*[*refine-transfer*]:
assumes α *fa* ≤ *Fa*
assumes $\bigwedge x.\ \alpha\ (fb\ x) \leq Fb\ x$
shows α (*case-option fa fb x*) ≤ *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 α (*case-sum fl fr x*) ≤ (*case-sum Fl Fr x*)
using *assms* **by** (*auto split*: *sum.split*)

lemma *transfer-list*[*refine-transfer*]:
assumes α *fn* ≤ *Fn*
assumes $\bigwedge x\ xs.\ \alpha\ (fc\ x\ xs) \leq Fc\ x\ xs$
shows α (*case-list fn fc l*) ≤ *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*: $\bigwedge x\ l\ rec\ rec'\ s.\ [\![\ \bigwedge s.\ \alpha\ (rec\ s) \leq (rec'\ s)\ ]\!]$
$\Longrightarrow \alpha\ (fc\ x\ l\ rec\ s) \leq fc'\ x\ l\ rec'\ s$
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*: $\bigwedge n\ rec\ rec'\ s.\ [\![\ \bigwedge s.\ \alpha\ (rec\ s) \leq rec'\ s\ ]\!]$
$\Longrightarrow \alpha\ (fs\ n\ rec\ s) \leq fs'\ n\ rec'\ s$
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 α-*dist*: $\bigwedge A.\ is\text{-}chain\ A \Longrightarrow \alpha\ (Sup\ A) = Sup\ (\alpha\text{`}A)$
begin
lemma α-*mono*[*simp*, *intro*!]: *mono* α
apply *rule*
apply (*subgoal-tac is-chain* $\{x,y\}$)
apply (*drule* α-*dist*)
apply (*auto simp*: *le-iff-sup*) []
apply (*rule chainI*)
apply *auto*
done

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

Transfer into ccpo

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

Transfer into ccpo with distributive transfer function.

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

  lemma α-*mono*[*simp*, *intro*!]: *mono* α
  proof
    fix $x\ y$ :: $'c$
    assume *LE*: $x \leq y$
    hence *C*[*simp*, *intro*!]: *is-chain* $\{x,y\}$ **by** (*auto intro*: *chainI*)
    from *LE* **have** $\alpha\ x \leq sup\ (\alpha\ x)\ (\alpha\ y)$ **by** *simp*
    also have $\ldots = Sup\ (\alpha`\{x,y\})$ **by** *simp*
    also have $\ldots = \alpha\ (Sup\ \{x,y\})$
      by (*rule* α-*dist*[*symmetric*]) *simp*
    also have $Sup\ \{x,y\} = y$
      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 α-*strict*[*simp*]: $\alpha\ (Sup\ \{\}) = bot$
    using α-*dist*[*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 {} = (λ*x*. *lub* {})
fun-lub lub {*f*} = (λ*x*. *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 ⋀*Z x*. *flat-ord b* (*f Z x*) (*g Z x*)
shows *flat-ord b* (*flatf-fp b f x*) (*flatf-fp b g x*)
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*:
(⋀*x*. *P x b*) ⟹
ccpo.admissible (*flatf-lub b*) (*flatf-ord b*) (λ*g*. ∀ *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*: $\bigwedge a\ x.\ pre\ a\ x \Longrightarrow post\ a\ x\ b$
assumes *MONO*: *flatf-mono b B*
assumes *PRE0*: *pre a x*
assumes *STEP*: $\bigwedge f\ a\ x.$
$[\![\bigwedge a'\ x'.\ pre\ a'\ x' \Longrightarrow post\ a'\ x'\ (f\ x');\ pre\ a\ x;$
flatf-ord b f (*flatf-fp b B*) $]\!]$
$\Longrightarrow post\ a\ x\ (B\ f\ x)$
shows *post a x* (*flatf-fp b B x*)
proof −
define *ub* **where** *ub* = *flatf-fp b B*

have $(\forall x\ a.\ pre\ a\ x \longrightarrow post\ a\ x\ (flatf\text{-}fp\ b\ B\ x))$
$\wedge$ *flatf-ord b* (*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 pointwise 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*: $\bigwedge f\ f'\ x\ x'.$
$[\![\bigwedge x\ x'.\ P\ x\ x' \Longrightarrow tr\ (f\ x)\ (f'\ x');\ P\ x\ x';\ fp' = f']\!]$
$\Longrightarrow tr\ (B\ f\ x)\ (B'\ f'\ x')$
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* ≡ *flat-ord bot*
abbreviation *flat-ge* ≡ *flat-ord top*
abbreviation *flatf-le* ≡ *flatf-ord bot*
abbreviation *flatf-ge* ≡ *flatf-ord top*

The flat ordering implies the lattice ordering

lemma *flat-ord-compat*:
fixes $x\ y :: 'a :: complete\text{-}lattice$
shows
flat-le x y $\Longrightarrow x \leq y$
flat-ge x y $\Longrightarrow x \geq y$
unfolding *flat-ord-def* **by** *auto*

lemma *flatf-ord-compat*:
fixes $x\ y :: 'a \Rightarrow ('b :: complete\text{-}lattice)$
shows
flatf-le x y $\Longrightarrow x \leq y$
flatf-ge x y $\Longrightarrow x \geq y$
by (*auto simp*: *flat-ord-compat le-fun-def fun-ord-def*)

abbreviation *flat-mono-le* ≡ *flat-mono bot*

abbreviation *flat-mono-ge* ≡ *flat-mono top*

abbreviation *flatf-mono-le* ≡ *flatf-mono bot*
abbreviation *flatf-mono-ge* ≡ *flatf-mono top*

abbreviation *flatf-gfp* ≡ *flatf-ord.fixp top*
abbreviation *flatf-lfp* ≡ *flatf-ord.fixp bot*

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

lemma *lfp-eq-flatf-lfp*:
assumes *FM*: *flatf-mono-le B* **and** *M*: *mono B*
shows *lfp B* = *flatf-lfp B*
proof −
from *lfp-unfold*[*OF M*, *symmetric*] **have** *B* (*lfp B*) = *lfp B* **.**
hence *flatf-le* (*B* (*lfp B*)) (*lfp B*) **by** *simp*
with *flatf-ord.fixp-lowerbound*[*OF FM*] **have** *flatf-le* (*flatf-lfp B*) (*lfp B*) **.**
with *flatf-ord-compat* **have** *flatf-lfp B* ≤ *lfp B* **by** *blast*
also
from *flatf-ord.fixp-unfold*[*OF FM*, *symmetric*] **have** *B* (*flatf-lfp B*) = *flatf-lfp B* **.**
hence *B* (*flatf-lfp B*) ≤ *flatf-lfp B* **by** *simp*
with *lfp-lowerbound*[**where** *A*=*flatf-lfp B*] **have** *lfp B* ≤ *flatf-lfp B* **.**
finally show *lfp B* = *flatf-lfp B* **..**
qed

lemma *gfp-eq-flatf-gfp*:
assumes *FM*: *flatf-mono-ge B* **and** *M*: *mono B*
shows *gfp B* = *flatf-gfp B*
proof −
from *gfp-unfold*[*OF M*, *symmetric*] **have** *B* (*gfp B*) = *gfp B* **.**
hence *flatf-ge* (*B* (*gfp B*)) (*gfp B*) **by** *simp*
with *flatf-ord.fixp-lowerbound*[*OF FM*] **have** *flatf-ge* (*flatf-gfp B*) (*gfp B*) **.**
with *flatf-ord-compat* **have** *gfp B* ≤ *flatf-gfp B* **by** *blast*
also
from *flatf-ord.fixp-unfold*[*OF FM*, *symmetric*] **have** *B* (*flatf-gfp B*) = *flatf-gfp B* **.**
hence *flatf-gfp B* ≤ *B* (*flatf-gfp B*) **by** *simp*
with *gfp-upperbound*[**where** *X*=*flatf-gfp B*] **have** *flatf-gfp B* ≤ *gfp B* **.**
finally show *gfp B* = *flatf-gfp B* **.**
qed

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

lemma *wf-fixp-induct*:
— Well-Founded induction for arbitrary fixed points
fixes *a* :: *'a*
assumes *fixp-unfold*: *fp B* = *B* (*fp B*)
assumes *WF*: *wf V*

assumes *P0*: *pre a x*
assumes *STEP*: $\bigwedge f\ a\ x.\ [\![$
$\bigwedge a'\ x'.\ [\![\ pre\ a'\ x';\ (x',x)\in V\ ]\!] \Longrightarrow post\ a'\ x'\ (f\ x');\ fp\ B = f;\ pre\ a\ x$
$]\!] \Longrightarrow post\ a\ x\ (B\ f\ x)$
shows *post a x* (*fp B x*)
proof −
have $\forall a.\ pre\ a\ x \longrightarrow post\ a\ x\ (fp\ B\ x)$
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*::(- ⇒ *'a*::*order-bot*) ⇒ -
assumes *TR-BOT*[*simp*]: $\bigwedge x.\ tr\ bot\ x$
assumes *MONO*: *flatf-mono-le B*
assumes *MONO'*: *flatf-mono-le B'*
assumes *R0*: *P x x'*
assumes *RS*: $\bigwedge f\ f'\ x\ x'.$
$[\![\bigwedge x\ x'.\ P\ x\ x' \Longrightarrow tr\ (f\ x)\ (f'\ x');\ P\ x\ x';\ flatf\text{-}lfp\ B' = f']\!]$
$\Longrightarrow tr\ (B\ f\ x)\ (B'\ f'\ x')$
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*::(- ⇒ *'a*::*order-top*) ⇒ -
assumes *TR-TOP*[*simp*]: $\bigwedge x.\ tr\ x\ top$
assumes *MONO*: *flatf-mono-ge B*
assumes *MONO'*: *flatf-mono-ge B'*
assumes *R0*: *P x x'*
assumes *RS*: $\bigwedge f\ f'\ x\ x'.$
$[\![\bigwedge x\ x'.\ P\ x\ x' \Longrightarrow tr\ (f\ x)\ (f'\ x');\ P\ x\ x';\ flatf\text{-}gfp\ B = f]\!]$
$\Longrightarrow tr\ (B\ f\ x)\ (B'\ f'\ x')$
shows *tr* (*flatf-gfp B x*) (*flatf-gfp B' x'*)
apply (*rule flatf-fixp-transfer*[**where**
tr=$\lambda x\ y.\ tr\ y\ x$ **and** *fp'*=*flatf-gfp B* **and** *P*=$\lambda x\ y.\ P\ y\ x$])
apply (*fact*|*assumption*|*rule flatf-ord.fixp-unfold*[*OF MONO*] *RS*)+

done

lemma *meta-le-everything-if-top*: $(m{=}top) \Longrightarrow (\bigwedge x.\ x \leq (m::'a::order\text{-}top))$
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*]: $\bigwedge a\ a'::'a::complete\text{-}lattice.$
flat-ge a a' $\Longrightarrow$ *flat-ge b b'* $\Longrightarrow$ *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-mono-le B* ∧ *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*

definition *REC* **where** *REC B x* ≡
if (*trimono B*) *then* (*lfp B x*) *else* (*top*::*'a*::*complete-lattice*)
definition *RECT* (*REC*$_T$) **where** *RECT B x* ≡
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* ⟹ *REC B* = *B* (*REC B*)
unfolding *REC-def* [*abs-def*]
by (*simp add*: *lfp-unfold*[*OF trimonoD-mono*, *symmetric*])

lemma *RECT-unfold*: ⟦*trimono B*⟧ ⟹ *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*: ⋀*F x*. (*B F x*) ≤ (*B'* *F x*)
shows (*REC B x*) ≤ (*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*: ⋀*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* ≤ *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*: $\bigwedge a\ x.$ *chain-admissible* $(\lambda b.\ \forall\ a\ x.$ *pre* $a\ x \longrightarrow$ *post* $a\ x\ (b\ x))$
assumes *ADM2*: $\bigwedge a\ x.$ *pre* $a\ x \longrightarrow$ *post* $a\ x$ *bot*
assumes *MONO*: *mono B*
assumes *P0*: *pre a x*
assumes *IS*:
$\bigwedge f\ a\ x.$
$[\![\bigwedge a'\ x'.$ *pre* $a'\ x' \Longrightarrow$ *post* $a'\ x'\ (f\ x')$; *pre a x*;
$f \leq$ (*lfp B*)$]\!]$
$\Longrightarrow$ *post a x* (*B f x*)
shows *post a x* (*lfp B x*)
proof −
define *u* **where** *u* = *lfp B*

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

have $(\forall\ a\ x.$ *pre a x* $\longrightarrow$ *post a x* (*lfp B x*)) $\wedge$ *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*: $\bigwedge f\ arb\ x.\ [\![$
$\bigwedge arb'\ x.$ *pre* $arb'\ x \Longrightarrow f\ x \leq M\ arb'\ x$; *pre arb x*; $f \leq$ *REC body*
$]\!] \Longrightarrow$ *body f x* $\leq M$ *arb x*
shows *REC body x* $\leq M$ *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*])

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*: ⋀*f arb x*. ⟦
⋀*arb' x'*. ⟦*pre arb' x'*; (*x'*,*x*)∈*V*⟧ ⟹ *f x'* ≤ *M arb' x'*;
pre arb x;
RECT body = *f*
⟧ ⟹ *body f x* ≤ *M arb x*
shows *RECT body x* ≤ *M 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*: ⋀*f x*. ⟦ ⋀*x*. *pre x* ⟹ *f x* ≤ *M x*; *pre x*; *f* ≤ *REC body* ⟧
⟹ *body f x* ≤ *M x*
shows *REC body x* ≤ *M x*
by (*rule REC-rule-arb*[**where** *pre*=λ-. *pre* **and** *M*=λ-. *M*, *OF assms*])

lemma *RECT-rule*:
assumes *M*: *trimono body*
assumes *WF*: *wf* (*V*::('*x*×'*x*) *set*)
assumes *I0*: *pre* (*x*::'*x*)
assumes *IS*: ⋀*f x*. ⟦ ⋀*x'*. ⟦*pre x'*; (*x'*,*x*)∈*V*⟧ ⟹ *f x'* ≤ *M x'*; *pre x*;
RECT body = *f*
⟧ ⟹ *body f x* ≤ *M x*
shows *RECT body x* ≤ *M x*
by (*rule RECT-rule-arb*[**where** *pre*=λ-. *pre* **and** *M*=λ-. *M*, *OF assms*])

lemma *REC-rule-arb2*:
assumes *M*: *trimono body*
assumes *I0*: *pre* (*arb*::′*arb*) (*arc*::′*arc*) (*x*::′*x*)
assumes *IS*: $\bigwedge f\ arb\ arc\ x.\ [\![$
$\bigwedge arb'\ arc'\ x'.\ [\![pre\ arb'\ arc'\ x'\ ]\!] \Longrightarrow f\ x' \leq M\ arb'\ arc'\ x';$
pre arb arc x
$]\!] \Longrightarrow body\ f\ x \leq M\ arb\ arc\ x$
shows $REC\ body\ x \leq M\ arb\ arc\ x$
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*: $\bigwedge f\ arb\ arc\ ard\ x.\ [\![$
$\bigwedge arb'\ arc'\ ard'\ x'.\ [\![pre\ arb'\ arc'\ ard'\ x]\!] \Longrightarrow f\ x' \leq M\ arb'\ arc'\ ard'\ x';$
pre arb arc ard x
$]\!] \Longrightarrow body\ f\ x \leq M\ arb\ arc\ ard\ x$
shows $REC\ body\ x \leq M\ arb\ arc\ ard\ x$
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*: $\bigwedge f\ arb\ arc\ x.\ [\![$
$\bigwedge arb'\ arc'\ x'.\ [\![pre\ arb'\ arc'\ x';\ (x',x) \in V]\!] \Longrightarrow f\ x' \leq M\ arb'\ arc'\ x';$
pre arb arc x;
$f \leq RECT\ body$
$]\!] \Longrightarrow body\ f\ x \leq M\ arb\ arc\ x$
shows $RECT\ body\ x \leq M\ arb\ arc\ x$
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*)

assumes *I0*: *pre* (*arb*::′*arb*) (*arc*::′*arc*) (*ard*::′*ard*) (*x*::′*x*)
assumes *IS*: ⋀*f arb arc ard x*. ⟦
⋀*arb′ arc′ ard′ x′*. ⟦*pre arb′ arc′ ard′ x′*; (*x′*,*x*)∈*V*⟧ ⟹ *f x′* ≤ *M arb′ arc′ ard′ x′*;
pre arb arc ard x;
f ≤ *RECT body*
⟧ ⟹ *body f x* ≤ *M arb arc ard x*
shows *RECT body x* ≤ *M 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* ≠ *top*
shows *RECT body x* = *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* ≠ *top*
ultimately show *flatf-gfp body x* = *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*: ⋀*f arb x*.

$[\![\bigwedge arb'\ x'.\ [\![pre\ arb'\ x';\ (x',\ x) \in V]\!] \Longrightarrow f\ x' \leq M\ arb'\ x';$
$pre\ arb\ x;\ REC_T\ body = f]\!]$
$\Longrightarrow body\ f\ x \leq M\ arb\ x$
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*: $\bigwedge F\ f\ x.\ [\![\bigwedge x.\ \alpha\ (f\ x) \leq F\ x\ ]\!] \Longrightarrow \alpha\ (b\ f\ x) \leq B\ F\ x$
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*: $\bigwedge F\ f\ x.\ [\![\bigwedge x.\ \alpha\ (f\ x) \leq F\ x\ ]\!] \Longrightarrow \alpha\ (b\ f\ x) \leq B\ F\ x$
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*: $\bigwedge F\ f\ x.\ [\![\bigwedge x.\ \alpha\ (f\ x) \leq F\ x\ ]\!] \Longrightarrow \alpha\ (b\ f\ x) \leq B\ F\ x$
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* α*-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*: $\bigwedge D\ D'\ x\ x'.\ [\![\bigwedge x\ x'.\ P\ x\ x' \Longrightarrow tr\ (D\ x)\ (D'\ x'); P\ x\ x'; RECT\ F = D]\!] \Longrightarrow tr\ (F\ D\ x)\ (F'\ D'\ x')$
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*: $\bigwedge D\ D'\ x\ x'.\ [\![\bigwedge x\ x'.\ P\ x\ x' \Longrightarrow tr\ (D\ x)\ (D'\ x'); P\ x\ x']\!] \Longrightarrow tr\ (F\ D\ x)\ (F'\ D'\ x')$
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*) ⇒ (*unit* ⇒ (*'ma*::*complete-lattice*)) ⇒ *'ma*
fixes *return* :: *unit* ⇒ *'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* ≡ *iASSERT return*
assumes *ASSUME-eq*: *ASSUME* ≡ *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*: ⟦ $\Phi \Longrightarrow M \leq M'$ ⟧ ⟹ $M \leq$ *bind* (*ASSERT* Φ) (λ-. M')
apply (*cases* Φ) **by** (*auto simp*: *ibind-strict imonad1*)

lemma *le-ASSERTI-pres*: ⟦ $\Phi \Longrightarrow M \leq$ *bind* (*ASSERT* Φ) (λ-. M') ⟧
⟹ $M \leq$ *bind* (*ASSERT* Φ) (λ-. M')
apply (*cases* Φ) **by** (*auto simp*: *ibind-strict imonad1*)

lemma *ASSERT-leI*: ⟦ Φ; $\Phi \Longrightarrow M \leq M'$ ⟧ ⟹ *bind* (*ASSERT* Φ) (λ-. M) ≤ M'
apply (*cases* Φ) **by** (*auto simp*: *ibind-strict imonad1*)

lemma *ASSUME-leI*: ⟦ $\Phi \Longrightarrow M \leq M'$ ⟧ ⟹ *bind* (*ASSUME* Φ) (λ-. M) ≤ M'
apply (*cases* Φ) **by** (*auto simp*: *ibind-strict imonad1*)
lemma *ASSUME-leI-pres*: ⟦ $\Phi \Longrightarrow$ *bind* (*ASSUME* Φ) (λ-. M)≤M' ⟧
⟹ *bind* (*ASSUME* Φ) (λ-. M) ≤ M'
apply (*cases* Φ) **by** (*auto simp*: *ibind-strict imonad1*)
lemma *le-ASSUMEI*: ⟦ Φ; $\Phi \Longrightarrow M \leq M'$ ⟧ ⟹ $M \leq$ *bind* (*ASSUME* Φ) (λ-. M')
apply (*cases* Φ) **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* Φ) (λ-. *S*) ≤ *S′* ⟷ (*S′*≠*top* ⟶ Φ) ∧ *S*≤*S′*
by (*cases* Φ) (*auto simp*: *ibind-strict imonad1 simp*: *top-unique*)

lemma *ASSUME-le-iff*:
bind (*ASSUME* Φ) (λ-. *S*) ≤ *S′* ⟷ (Φ ⟶ *S*≤*S′*)
by (*cases* Φ) (*auto simp*: *ibind-strict imonad1*)

lemma *le-ASSERT-iff*:
S ≤ *bind* (*ASSERT* Φ) (λ-. *S′*) ⟷ (Φ ⟶ *S*≤*S′*)
by (*cases* Φ) (*auto simp*: *ibind-strict imonad1*)

lemma *le-ASSUME-iff*:
S ≤ *bind* (*ASSUME* Φ) (λ-. *S′*) ⟷ (*S*≠*bot* ⟶ Φ) ∧ *S*≤*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*)
⇒ (*unit*⇒*′mac*) ⇒ (*′mac*::*complete-lattice*)
and *creturn* :: *unit* ⇒ *′muc* **and** *cASSERT* **and** *cASSUME*
and *abind* :: (*′mua*::*complete-lattice*)
⇒ (*unit*⇒*′maa*) ⇒ (*′maa*::*complete-lattice*)
and *areturn* :: *unit* ⇒ *′mua* **and** *aASSERT* **and** *aASSUME*
and α :: *′mac* ⇒ *′maa*
begin
lemma *transfer-ASSERT*[*refine-transfer*]:
⟦Φ ⟹ α *M* ≤ *M′*⟧
⟹ α (*cbind* (*cASSERT* Φ) (λ-. *M*)) ≤ (*abind* (*aASSERT* Φ) (λ-. *M′*))
apply (*cases* Φ)
apply (*auto simp*: *c.ibind-strict a.ibind-strict c.imonad1 a.imonad1*)
done

lemma *transfer-ASSUME*[*refine-transfer*]:
⟦Φ; Φ ⟹ α *M* ≤ *M′*⟧
⟹ α (*cbind* (*cASSUME* Φ) (λ-. *M*)) ≤ (*abind* (*aASSUME* Φ) (λ-. *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 α
    for abind :: ('mua::complete-lattice)
      ⇒ (unit⇒'maa) ⇒ ('maa::complete-lattice)
    and areturn :: unit ⇒ 'mua and aASSERT and aASSUME
    and α :: 'mac ⇒ 'maa
  begin
    lemma transfer-ASSERT-remove[refine-transfer]:
      ⟦ Φ ⟹ α M ≤ M' ⟧ ⟹ α M ≤ abind (aASSERT Φ) (λ-. M')
      by (rule a.le-ASSERTI)

    lemma transfer-ASSUME-remove[refine-transfer]:
      ⟦ Φ; Φ ⟹ α M ≤ M' ⟧ ⟹ α M ≤ abind (aASSUME Φ) (λ-. 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 $M \leq SPEC\ \Phi$ means that M is correct w.r.t. specification Φ. This suggests the following view on the program lattice: The top-element is the result that is never correct. We call this result *FAIL*. The bottom element is the program that is always correct. It is called *SUCCEED*. An assertion can be encoded by failing if the asserted predicate is not true. Symmetrically, an assumption is encoded by succeeding if the predicate is not true.

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

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

instantiation *nres* :: (*type*) *complete-lattice*
begin
fun *less-eq-nres* **where**
- ≤ *FAILi* ⟷ *True* |
(*RES a*) ≤ (*RES b*) ⟷ *a*⊆*b* |
FAILi ≤ (*RES* -) ⟷ *False*

fun *less-nres* **where**
FAILi < - ⟷ *False* |
(*RES* -) < *FAILi* ⟷ *True* |
(*RES a*) < (*RES b*) ⟷ *a*⊂*b*

fun *sup-nres* **where**
sup - *FAILi* = *FAILi* |
sup FAILi - = *FAILi* |
sup (*RES a*) (*RES b*) = *RES* (*a*∪*b*)

fun *inf-nres* **where**
inf x FAILi = *x* |
inf FAILi x = *x* |
inf (*RES a*) (*RES b*) = *RES* (*a*∩*b*)

definition *Sup X* ≡ *if FAILi*∈*X then FAILi else RES* (⋃{*x* . *RES x* ∈ *X*})
definition *Inf X* ≡ *if* ∃ *x*. *RES x*∈*X then RES* (⋂{*x* . *RES x* ∈ *X*}) *else FAILi*

definition *bot* ≡ *RES* {}
definition *top* ≡ *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*) []
apply (*case-tac x*, *auto*) []
apply (*case-tac x*, *case-tac* [!] *y*, *case-tac* [!] *z*, *auto*) []
apply (*case-tac x*, (*case-tac* [!] *y*)*?*, *auto*) []
apply (*case-tac x*, (*case-tac* [!] *y*)*?*, *simp-all*) []
apply (*case-tac x*, (*case-tac* [!] *y*)*?*, *auto*) []
apply (*case-tac x*, *case-tac* [!] *y*, *case-tac* [!] *z*, *auto*) []
apply (*case-tac x*, (*case-tac* [!] *y*)*?*, *auto*) []
apply (*case-tac x*, (*case-tac* [!] *y*)*?*, *auto*) []
apply (*case-tac x*, *case-tac* [!] *y*, *case-tac* [!] *z*, *auto*) []
apply (*case-tac x*, *auto*) []
apply (*case-tac z*, *fastforce*+) []
apply (*case-tac x*, *auto*) []
apply (*case-tac z*, *fastforce*+) []
apply *auto* []

apply *auto* []
done

end

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

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

lemma *nres-cases*[*case-names FAIL RES*, *cases type*]:
obtains *M*=*FAIL* | *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 ≠ *RES X*
FAIL ≠ *SUCCEED*
FAIL ≠ *RETURN x*
SUCCEED ≠ *FAIL*
SUCCEED ≠ *RETURN x*
RES X ≠ *FAIL*
RETURN x ≠ *FAIL*
RETURN x ≠ *SUCCEED*
unfolding *top-nres-def bot-nres-def RETURN-def*
by *auto*

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

lemma *nres-order-simps*[*simp*]:
⋀*M*. *SUCCEED* ≤ *M*
⋀*M*. *M* ≤ *SUCCEED* ⟷ *M*=*SUCCEED*
⋀*M*. *M* ≤ *FAIL*
⋀*M*. *FAIL* ≤ *M* ⟷ *M*=*FAIL*
⋀*X Y*. *RES X* ≤ *RES Y* ⟷ *X*≤*Y*
⋀*X*. *Sup X* = *FAIL* ⟷ *FAIL*∈*X*
⋀*X f*. *Sup* (*f* ‘ *X*) = *FAIL* ⟷ *FAIL* ∈ *f* ‘ *X*

```
  ⋀X. FAIL = Sup X ⟷ FAIL∈X
  ⋀X f. FAIL = Sup (f ‘ X) ⟷ FAIL ∈ f ‘ X
  ⋀X. FAIL∈X ⟹ Sup X = FAIL
  ⋀X. FAIL∈f ‘ X ⟹ Sup (f ‘ X) = FAIL
  ⋀A. Sup (RES ‘ A) = RES (Sup A)
  ⋀A. Sup (RES ‘ A) = RES (Sup A)
  ⋀A. A≠{} ⟹ Inf (RES‘A) = RES (Inf A)
  ⋀A. A≠{} ⟹ Inf (RES ‘ A) = RES (Inf A)
  Inf {} = FAIL
  Inf UNIV = SUCCEED
  Sup {} = SUCCEED
  Sup UNIV = FAIL
  ⋀x y. RETURN x ≤ RETURN y ⟷ x=y
  ⋀x Y. RETURN x ≤ RES Y ⟷ x∈Y
  ⋀X y. RES X ≤ RETURN y ⟷ X ⊆ {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 ∈ 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 ≡ S≠FAIL
definition inres S x ≡ RETURN x ≤ S

lemma nofail-simps[simp, refine-pw-simps]:
  nofail FAIL ⟷ False
```

nofail (RES X) ⟷ True
nofail (RETURN x) ⟷ True
nofail SUCCEED ⟷ True
unfolding *nofail-def*
by (*simp-all add*: *RETURN-def*)

lemma *inres-simps*[*simp*, *refine-pw-simps*]:
inres FAIL = (λ-. True)
inres (RES X) = (λx. x∈X)
inres (RETURN x) = (λy. x=y)
inres SUCCEED = (λ-. False)
unfolding *inres-def* [*abs-def*]
by (*auto simp add*: *RETURN-def*)

lemma *not-nofail-iff*:
¬nofail S ⟷ S=FAIL **by** (*cases S*) *auto*

lemma *not-nofail-inres*[*simp*, *refine-pw-simps*]:
¬nofail S ⟹ inres S x
apply (*cases S*) **by** *auto*

lemma *intro-nofail*[*refine-pw-simps*]:
S≠FAIL ⟷ nofail S
FAIL≠S ⟷ nofail S
by (*cases S*, *simp-all*)+

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

lemma *pw-le-iff*:
S ≤ S′ ⟷ (nofail S′⟶ (nofail S ∧ (∀ x. inres S x ⟶ inres S′ x)))
apply (*cases S*, *simp-all*)
apply (*case-tac* [!] *S′*, *auto*)
done

lemma *pw-eq-iff*:
S=S′ ⟷ (nofail S = nofail S′ ∧ (∀ x. inres S x ⟷ 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′ ⟷*
(∃ x. inres S x) ⟶ (nofail S ⟷ nofail S′) ∧ (∀ x. inres S x ⟷ inres S′ x)
by (*auto simp* : *flat-ord-def pw-eq-iff*)

lemma *pw-flat-ge-iff*: *flat-ge S S′ ⟷*
(nofail S) ⟶ nofail S′ ∧ (∀ x. inres S x ⟷ 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*:
(*nofail* $S' \longrightarrow$ (*nofail* $S \wedge (\forall x.$ *inres* $S\ x \longrightarrow$ *inres* $S'\ x)$)) $\Longrightarrow S \leq S'$
by (*simp add*: *pw-le-iff*)

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

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

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

lemma *pwD2*:
assumes $S{\leq}S'$ *inres* $S\ x$
shows *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*: ⟦ *nofail* $M' \Longrightarrow M \leq M'$ ⟧ $\Longrightarrow M \leq M'$
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*]:
nofail (*sup a b*) $\longleftrightarrow$ *nofail a* $\wedge$ *nofail b*
apply (*cases a*, *simp*)

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

lemma *pw-sup-inres*[*refine-pw-simps*]:
inres (*sup a b*) *x* ⟷ *inres a x* ∨ *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* ⟷ (∃ *M*∈*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* ⟷ (∃ *M*∈*X*. *inres* (*f M*) *r*)
using *pw-Sup-inres* [*of f* ‘ *X*] **by** *simp*

lemma *pw-Sup-nofail*[*refine-pw-simps*]: *nofail* (*Sup X*) ⟷ (∀ *x*∈*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*)) ⟷ (∀ *x*∈*X*. *nofail* (*f x*))
using *pw-Sup-nofail* [*of f* ‘ *X*] **by** *simp*

lemma *pw-inf-nofail*[*refine-pw-simps*]:
nofail (*inf a b*) ⟷ *nofail a* ∨ *nofail b*
apply (*cases a*, *simp*)
apply (*cases b*, *simp-all*)
done

lemma *pw-inf-inres*[*refine-pw-simps*]:
inres (*inf a b*) *x* ⟷ *inres a x* ∧ *inres b x*
apply (*cases a*, *simp*)
apply (*cases b*, *simp*)
apply (*simp*)
done

lemma *pw-Inf-nofail*[*refine-pw-simps*]: *nofail* (*Inf C*) ⟷ (∃ *x*∈*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*]: *nofail* (*Inf* (*f* ‘ *C*)) ⟷ (∃ *x*∈*C*. *nofail* (*f x*))
using *pw-Inf-nofail* [*of f* ‘ *C*] **by** *simp*

lemma *pw-Inf-inres*[*refine-pw-simps*]: *inres* (*Inf C*) *r* ⟷ (∀ *M*∈*C*. *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*]: *inres* (*Inf* (*f* ‘ *C*)) *r* ⟷ (∀ *M*∈*C*. *inres* (*f M*) *r*)
using *pw-Inf-inres* [*of f* ‘ *C*] **by** *simp*

lemma *nofail-RES-conv*: *nofail m* ⟷ (∃ *M*. *m*=*RES M*) **by** (*cases m*) *auto*

primrec *the-RES* **where** *the-RES* (*RES X*) = *X*
lemma *the-RES-inv*[*simp*]: *nofail m* ⟹ *RES* (*the-RES m*) = *m*
by (*cases m*) *auto*

definition [*refine-pw-simps*]: *nf-inres m x* ≡ *nofail m* ∧ *inres m x*

lemma *nf-inres-RES*[*simp*]: *nf-inres* (*RES X*) *x* ⟷ *x*∈*X*
by (*simp add*: *refine-pw-simps*)

lemma *nf-inres-SPEC*[*simp*]: *nf-inres* (*SPEC* Φ) *x* ⟷ Φ *x*
by (*simp add*: *refine-pw-simps*)

lemma *nofail-antimono-fun*: *f* ≤ *g* ⟹ (*nofail* (*g x*) ⟶ *nofail* (*f x*))
by (*auto simp*: *pw-le-iff dest*: *le-funD*)

Monad Operators

definition *bind* **where** *bind M f* ≡ *case M of*
FAILi ⇒ *FAIL* |
RES X ⇒ *Sup* (*f*‘*X*)

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

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

lemma *bind-RES*: *bind* (*RES X*) *f* = *Sup* (*f*‘*X*) **unfolding** *bind-def*
by (*auto*)

adhoc-overloading
Monad-Syntax.bind Refine-Basic.bind

lemma *pw-bind-nofail*[*refine-pw-simps*]:
nofail (*bind M f*) ⟷ (*nofail M* ∧ (∀ *x*. *inres M x* ⟶ *nofail* (*f x*)))
apply (*cases M*)
by (*auto simp*: *bind-RES refine-pw-simps*)

lemma *pw-bind-inres*[*refine-pw-simps*]:
inres (*bind M f*) = (λ*x*. *nofail M* ⟶ (∃ *y*. (*inres M y* ∧ *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*:
bind M f ≤ *S* ⟷ (*nofail S* ⟶ *nofail M*) ∧
(∀ *x*. *nofail M* ∧ *inres M x* ⟶ *f x* ≤ *S*)
by (*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *pw-bind-leI*: ⟦
nofail S ⟹ *nofail M*; ⋀*x*. ⟦*nofail M*; *inres M x*⟧ ⟹ *f x* ≤ *S*⟧
⟹ *bind M f* ≤ *S*
by (*simp add*: *pw-bind-le-iff*)

lemma *nres-monad1*[*simp*]: *bind* (*RETURN x*) *f* = *f x*
by (*rule pw-eqI*) (*auto simp*: *refine-pw-simps*)
lemma *nres-monad2*[*simp*]: *bind M RETURN* = *M*
by (*rule pw-eqI*) (*auto simp*: *refine-pw-simps*)
lemma *nres-monad3*[*simp*]: *bind* (*bind M f*) *g* = *bind M* (λ*x*. *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 $\bigwedge x.\ RETURN\ x \le m' \Longrightarrow f\ x = f'\ x$
shows $bind\ m\ f = bind\ m'\ f'$
using *assms*
by (*auto simp*: *refine-pw-simps pw-eq-iff pw-le-iff*)

lemma *bind-mono*[*refine-mono*]:
$[\![\ M \le M';\ \bigwedge x.\ RETURN\ x \le M \Longrightarrow f\ x \le f'\ x\ ]\!] \Longrightarrow bind\ M\ f \le bind\ M'\ f'$

$[\![\ flat\text{-}ge\ M\ M';\ \bigwedge x.\ flat\text{-}ge\ (f\ x)\ (f'\ x)\ ]\!] \Longrightarrow flat\text{-}ge\ (bind\ M\ f)\ (bind\ M'\ f')$
apply (*auto simp*: *refine-pw-simps pw-ords-iff*) []
apply (*auto simp*: *refine-pw-simps pw-ords-iff*) []
done

lemma *bind-mono1*[*simp*, *intro*!]: $mono\ (\lambda M.\ bind\ M\ f)$
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*: $bind\ (sup\ M\ N)\ f = sup\ (bind\ M\ f)\ (bind\ N\ f)$
by (*auto simp add*: *pw-eq-iff refine-pw-simps*)

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

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

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

lemma *RES-Sup-RETURN*: $Sup\ (RETURN`X) = RES\ X$

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

2.6.2 VCG Setup

lemma *SPEC-cons-rule*:
assumes $m \leq SPEC\ \Phi$
assumes $\bigwedge x.\ \Phi\ x \Longrightarrow \Psi\ x$
shows $m \leq SPEC\ \Psi$
using *assms* **by** (*auto simp*: *pw-le-iff*)

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

ML ‹
structure Refine = struct

structure vcg = Named-Thms
(*val name* = @{*binding refine-vcg*}
val description = *Refinement Framework*: ^
Verification condition generation rules (*intro*))

structure vcg-cons = Named-Thms
(*val name* = @{*binding refine-vcg-cons*}
val description = *Refinement Framework*: ^
Consequence rules tried by VCG)

structure refine0 = Named-Thms
(*val name* = @{*binding refine0*}
val description = *Refinement Framework*: ^
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*: ^
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 allI}));
  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**
  conc-fun R m $\equiv$ *case m of FAILi* $\Rightarrow$ *FAIL* | *RES X* $\Rightarrow$ *RES* (R^{-1} ‘‘*X*)

definition *abs-fun* ($\Uparrow$) **where**
  abs-fun R m $\equiv$ *case m of FAILi* $\Rightarrow$ *FAIL*
    | *RES X* $\Rightarrow$ *if X*$\subseteq$*Domain R then RES* (*R*‘‘*X*) *else FAIL*

lemma
  conc-fun-FAIL[*simp*]: $\Downarrow R$ *FAIL* = *FAIL* **and**
  conc-fun-RES: $\Downarrow R$ (*RES X*) = *RES* (R^{-1} ‘‘*X*)
  unfolding *conc-fun-def* **by** (*auto split*: *nres.split*)

lemma *abs-fun-simps*[*simp*]:
  $\Uparrow R$ *FAIL* = *FAIL*
  X$\subseteq$*Domain R* $\Longrightarrow$ $\Uparrow R$ (*RES X*) = *RES* (*R*‘‘*X*)
  $\neg$(*X*$\subseteq$*Domain R*) $\Longrightarrow$ $\Uparrow R$ (*RES X*) = *FAIL*
  unfolding *abs-fun-def* **by** (*auto split*: *nres.split*)

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*]:
  nofail ($\Uparrow R$ *M*) $\longleftrightarrow$ (*nofail M* $\wedge$ ($\forall$ *x*. *inres M x* $\longrightarrow$ *x*$\in$*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 (⇑*R M*) *a* ⟷ (*nofail* (⇑*R M*) ⟶ (∃ *c*. *inres M c* ∧ (*c*,*a*)∈*R*))
apply (*cases M*)
apply *simp*
apply (*auto simp*: *abs-fun-def*)
done

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

lemma *pw-conc-inres*[*refine-pw-simps*]:
inres (⇓*R S'*) = (λ*s*. *nofail S'*
⟶ (∃ *s'*. (*s*,*s'*)∈*R* ∧ *inres S' s'*))
apply (*rule ext*)
apply (*cases S'*)
apply (*auto simp*: *conc-fun-RES*)
done

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

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

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

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

lemma *conc-fun-R-mono*:
assumes *R* ⊆ *R'*
shows ⇓*R M* ≤ ⇓*R' M*
using *assms*
by (*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *conc-fun-chain*: ⇓*R* (⇓*S M*) = ⇓(*R O S*) *M*
unfolding *conc-fun-def*
by (*auto split*: *nres.split*)

lemma *conc-Id*[*simp*]: ⇓*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*]:
$\bigwedge A\ B\ C.\ A{\leq}\Downarrow R\ \ B \Longrightarrow B{\leq}\ \ \ C \Longrightarrow A{\leq}\Downarrow R\ \ C$
$\bigwedge A\ B\ C.\ A{\leq}\Downarrow Id\ B \Longrightarrow B{\leq}\Downarrow R\ \ C \Longrightarrow A{\leq}\Downarrow R\ \ C$
$\bigwedge A\ B\ C.\ A{\leq}\Downarrow R\ \ B \Longrightarrow B{\leq}\Downarrow Id\ C \Longrightarrow A{\leq}\Downarrow R\ \ C$

$\bigwedge A\ B\ C.\ A{\leq}\Downarrow Id\ B \Longrightarrow B{\leq}\Downarrow Id\ C \Longrightarrow A{\leq}\ \ \ C$
$\bigwedge A\ B\ C.\ A{\leq}\Downarrow Id\ B \Longrightarrow B{\leq}\ \ \ C \Longrightarrow A{\leq}\ \ \ C$
$\bigwedge A\ B\ C.\ A{\leq}\ \ \ B \Longrightarrow B{\leq}\Downarrow Id\ C \Longrightarrow A{\leq}\ \ \ C$
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*]:
$\bigwedge A\ B\ C.\ [\![\ A \leq B; \Uparrow R\ B \leq C]\!] \Longrightarrow \Uparrow R\ A \leq C$
$\bigwedge A\ B\ C.\ [\![\Uparrow Id\ A \leq B; \Uparrow R\ B \leq C]\!] \Longrightarrow \Uparrow R\ A \leq C$
$\bigwedge A\ B\ C.\ [\![\Uparrow R\ A \leq B; \Uparrow Id\ B \leq C]\!] \Longrightarrow \Uparrow R\ A \leq C$

$\bigwedge A\ B\ C.\ [\![\Uparrow Id\ A \leq B; \Uparrow Id\ B \leq C]\!] \Longrightarrow A \leq C$
$\bigwedge A\ B\ C.\ [\![\Uparrow Id\ A \leq B; B \leq C]\!] \Longrightarrow A \leq C$
$\bigwedge A\ B\ C.\ [\![A \leq B; \Uparrow Id\ B \leq C]\!] \Longrightarrow A \leq C$

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* ≡ *iASSERT RETURN*
definition *ASSUME* **where** *ASSUME* ≡ *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) ⟷ Φ
inres (*ASSERT* Φ) *x*
by (*cases* Φ, *simp-all*)+

lemma *pw-ASSUME*[*refine-pw-simps*]:
nofail (*ASSUME* Φ)
inres (*ASSUME* Φ) *x* ⟷ Φ
by (*cases* Φ, *simp-all*)+

Recursion

lemma *pw-REC-nofail*:
shows *nofail* (*REC B x*) ⟷ *trimono B* ∧
(∃ *F*. (∀ *x*.
nofail (*F x*) ⟶ *nofail* (*B F x*)
∧ (∀ *x'*. *inres* (*B F x*) *x'* ⟶ *inres* (*F x*) *x'*)
) ∧ *nofail* (*F x*))
proof −
have *nofail* (*REC B x*) ⟷ *trimono B* ∧
(∃ *F*. (∀ *x*. *B F x* ≤ *F x*) ∧ *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*:
inres (*REC B x*) *x′* = (*trimono B* ⟶
(∀ *F*. (∀ *x″*.
nofail (*F x″*) ⟶ *nofail* (*B F x″*)
∧ (∀ *x*. *inres* (*B F x″*) *x* ⟶ *inres* (*F x″*) *x*))
⟶ *inres* (*F x*) *x′*))
proof −
have *inres* (*REC B x*) *x′*
⟷ (*trimono B* ⟶ (∀ *F*. (∀ *x″*. *B F x″* ≤ *F x″*) ⟶ *inres* (*F x*) *x′*))
unfolding *REC-def lfp-def*
by (*auto simp*: *refine-pw-simps intro*: *le-funI dest*: *le-funD*)
thus *?thesis* **unfolding** *pw-le-iff* **.**
qed

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

lemma *pw-RECT-nofail*:
shows *nofail* (*RECT B x*) ⟷ *trimono B* ∧
(∀ *F*. (∀ *y*. *nofail* (*B F y*) ⟶
nofail (*F y*) ∧ (∀ *x*. *inres* (*F y*) *x* ⟶ *inres* (*B F y*) *x*)) ⟶
nofail (*F x*))
proof −
have *nofail* (*RECT B x*) ⟷ (*trimono B* ∧ (∀ *F*. (∀ *y*. *F y* ≤ *B F y*) ⟶ *nofail* (*F x*)))
unfolding *RECT-gfp-def gfp-def*
by (*auto simp*: *refine-pw-simps intro*: *le-funI dest*: *le-funD*)
thus *?thesis*
unfolding *pw-le-iff* **.**
qed

lemma *pw-RECT-inres*:
shows *inres* (*RECT B x*) *x′* = (*trimono B* ⟶
(∃ *M*. (∀ *y*. *nofail* (*B M y*) ⟶
nofail (*M y*) ∧ (∀ *x*. *inres* (*M y*) *x* ⟶ *inres* (*B M y*) *x*)) ∧
inres (*M x*) *x′*))
proof −
have *inres* (*RECT B x*) *x′* ⟷ *trimono B* ⟶ (∃ *M*. (∀ *y*. *M y* ≤ *B M y*) ∧ *inres* (*M x*) *x′*)
unfolding *RECT-gfp-def gfp-def*
by (*auto simp*: *refine-pw-simps intro*: *le-funI dest*: *le-funD*)
thus *?thesis* **unfolding** *pw-le-iff* **.**
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*]:
  $m \leq SPEC\ (\lambda\text{-}.\ True) \Longrightarrow m \leq RES\ UNIV$ **by** *auto*

lemma *RETURN-rule*[*refine-vcg*]: $\Phi\ x \Longrightarrow RETURN\ x \leq SPEC\ \Phi$
  by (*auto simp*: *RETURN-def*)
lemma *RES-rule*[*refine-vcg*]: $[\![\bigwedge x.\ x{\in}S \Longrightarrow \Phi\ x]\!] \Longrightarrow RES\ S \leq SPEC\ \Phi$
  by *auto*
lemma *SUCCEED-rule*[*refine-vcg*]: $SUCCEED \leq SPEC\ \Phi$ **by** *auto*
lemma *FAIL-rule*: $False \Longrightarrow FAIL \leq SPEC\ \Phi$ **by** *auto*
lemma *SPEC-rule*[*refine-vcg*]: $[\![\bigwedge x.\ \Phi\ x \Longrightarrow \Phi'\ x]\!] \Longrightarrow SPEC\ \Phi \leq SPEC\ \Phi'$ **by** *auto*

lemma *RETURN-to-SPEC-rule*[*refine-vcg*]: $m{\leq}SPEC\ ((=)\ v) \Longrightarrow m{\leq}RETURN\ v$
  by (*simp add*: *pw-le-iff refine-pw-simps*)

lemma *Sup-img-rule-complete*:
  $(\forall x.\ x{\in}S \longrightarrow f\ x \leq SPEC\ \Phi) \longleftrightarrow Sup\ (f`S) \leq SPEC\ \Phi$
  apply *rule*
  apply (*rule pw-leI*)
  apply (*auto simp*: *pw-le-iff refine-pw-simps*) []
  apply (*intro allI impI*)
  apply (*rule pw-leI*)
  apply (*auto simp*: *pw-le-iff refine-pw-simps*) []
  done

lemma *SUP-img-rule-complete*:
  $(\forall x.\ x{\in}S \longrightarrow f\ x \leq SPEC\ \Phi) \longleftrightarrow Sup\ (f\ `\ S) \leq SPEC\ \Phi$
  using *Sup-img-rule-complete* [*of S f*] **by** *simp*

lemma *Sup-img-rule*[*refine-vcg*]:
  $[\![\ \bigwedge x.\ x{\in}S \Longrightarrow f\ x \leq SPEC\ \Phi\ ]\!] \Longrightarrow Sup(f`S) \leq SPEC\ \Phi$
  by (*auto simp*: *SUP-img-rule-complete*[*symmetric*])

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

lemma *bind-rule-complete*: $bind\ M\ f \leq SPEC\ \Phi \longleftrightarrow M \leq SPEC\ (\lambda x.\ f\ x \leq SPEC\ \Phi)$
  by (*auto simp*: *pw-le-iff refine-pw-simps*)
lemma *bind-rule*[*refine-vcg*]:
  $[\![\ M \leq SPEC\ (\lambda x.\ f\ x \leq SPEC\ \Phi)\ ]\!] \Longrightarrow bind\ M\ (\lambda x.\ f\ x) \leq SPEC\ \Phi$
  — Note: η-expanded version helps Isabelle's unification to keep meaningful variable names from the program
  by (*auto simp*: *bind-rule-complete*)

lemma *ASSUME-rule*[*refine-vcg*]: $[\![\Phi \Longrightarrow \Psi\ ()]\!] \Longrightarrow ASSUME\ \Phi \leq SPEC\ \Psi$

by (*cases* Φ) *auto*

lemma *ASSERT-rule*[*refine-vcg*]: ⟦Φ; Φ ⟹ Ψ ()⟧ ⟹ *ASSERT* Φ ≤ *SPEC* Ψ **by** *auto*

lemma *prod-rule*[*refine-vcg*]:
⟦⋀*a b*. *p*=(*a*,*b*) ⟹ *S a b* ≤ *SPEC* Φ⟧ ⟹ *case-prod S p* ≤ *SPEC* Φ
by (*auto split*: *prod.split*)

lemma *prod2-rule*[*refine-vcg*]:
assumes ⋀*a b c d*. ⟦*ab*=(*a*,*b*); *cd*=(*c*,*d*)⟧ ⟹ *f a b c d* ≤ *SPEC* Φ
shows (λ(*a*,*b*) (*c*,*d*). *f a b c d*) *ab cd* ≤ *SPEC* Φ
using *assms*
by (*auto split*: *prod.split*)

lemma *if-rule*[*refine-vcg*]:
⟦ *b* ⟹ *S1* ≤ *SPEC* Φ; ¬*b* ⟹ *S2* ≤ *SPEC* Φ⟧
⟹ (*if b then S1 else S2*) ≤ *SPEC* Φ
by (*auto*)

lemma *option-rule*[*refine-vcg*]:
⟦ *v*=*None* ⟹ *S1* ≤ *SPEC* Φ; ⋀*x*. *v*=*Some x* ⟹ *f2 x* ≤ *SPEC* Φ⟧
⟹ *case-option S1 f2 v* ≤ *SPEC* Φ
by (*auto split*: *option.split*)

lemma *Let-rule*[*refine-vcg*]:
f x ≤ *SPEC* Φ ⟹ *Let x f* ≤ *SPEC* Φ **by** *auto*

lemma *Let-rule′*:
assumes ⋀*x*. *x*=*v* ⟹ *f x* ≤ *SPEC* Φ
shows *Let v* (λ*x*. *f x*) ≤ *SPEC* Φ
using *assms* **by** *simp*

lemma *REC-le-rule*:
assumes *M*: *trimono body*
assumes *I0*: (*x*,*x′*)∈*R*
assumes *IS*: ⋀*f x x′*. ⟦ ⋀*x x′*. (*x*,*x′*)∈*R* ⟹ *f x* ≤ *M x′*; (*x*,*x′*)∈*R* ⟧
⟹ *body f x* ≤ *M x′*
shows *REC body x* ≤ *M x′*
by (*rule REC-rule-arb*[*OF M*, **where** *pre*=λ*x′ x*. (*x*,*x′*)∈*R*, *OF I0 IS*])

Proving Monotonicity

lemma *nr-mono-bind*:

assumes *MA*: *mono A* **and** *MB*: $\bigwedge s.$ *mono* (*B s*)
shows *mono* ($\lambda F\ s.$ *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'*: *mono* ($\lambda F\ s.$ *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*:
$[\![\bigwedge s.\ s{\in}S \Longrightarrow \exists\, s'{\in}S'.\ (s,s'){\in}R]\!] \Longrightarrow RES\ S \leq \Downarrow R\ (RES\ S')$
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*]:
$S \leq SPEC\ \Phi \Longrightarrow S \leq \Downarrow Id\ (SPEC\ \Phi)$ **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 $RETURN\ x \leq \Downarrow R\ (SPEC\ \Phi)$
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 $ai \leq \Downarrow R\ a$
assumes $bi \leq \Downarrow R\ b$
shows $sup\ ai\ bi \leq \Downarrow R\ (sup\ a\ b)$
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*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ inres\ M\ x;\ inres\ M'\ x';$
nofail M; *nofail M'*
$]\!] \Longrightarrow f\ x \leq \Downarrow\ R\ (f'\ x')$
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*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R'\ ]\!]$
$\Longrightarrow f\ x \leq \Downarrow\ R\ (f'\ x')$
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*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ nf\text{-}inres\ M'\ x'$
$]\!] \Longrightarrow f\ x \leq \Downarrow\ R\ (f'\ x')$
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*:
$[\![RES\ X \leq \Downarrow\ R'\ M';$
$\bigwedge x\ x'.\ [\![(x,\ x') \in R';\ x \in X\ ]\!] \Longrightarrow f\ x \leq \Downarrow\ R\ (f'\ x')]\!]$
$\Longrightarrow RES\ X \gg\!= (\lambda x.\ f\ x) \leq \Downarrow\ R\ (M' \gg\!= (\lambda x'.\ f'\ x'))$

$[\![M \leq \Downarrow\ R'\ (RES\ X');$
$\bigwedge x\ x'.\ [\![(x,\ x') \in R';\ x' \in X'\ ]\!] \Longrightarrow f\ x \leq \Downarrow\ R\ (f'\ x')]\!]$
$\Longrightarrow M \gg\!= (\lambda x.\ f\ x) \leq \Downarrow\ R\ (RES\ X' \gg\!= (\lambda x'.\ f'\ x'))$

$[\![RES\ X \leq \Downarrow\ R'\ (RES\ X');$
$\bigwedge x\ x'.\ [\![(x,\ x') \in R';\ x \in X;\ x' \in X']\!] \Longrightarrow f\ x \leq \Downarrow\ R\ (f'\ x')]\!]$
$\Longrightarrow RES\ X \gg\!= (\lambda x.\ f\ x) \leq \Downarrow\ R\ (RES\ X' \gg\!= (\lambda x'.\ f'\ x'))$
by (*auto intro*!: *bind-refine'*)

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

lemma *ASSERT-refine*[*refine*]:
$[\![\ \Phi' \Longrightarrow \Phi\ ]\!] \Longrightarrow ASSERT\ \Phi \leq \Downarrow Id\ (ASSERT\ \Phi')$
by (*cases* Φ') *auto*

lemma *ASSUME-refine*[*refine*]:
$[\![\ \Phi \Longrightarrow \Phi'\ ]\!] \Longrightarrow ASSUME\ \Phi \leq \Downarrow Id\ (ASSUME\ \Phi')$
by (*cases* Φ) *auto*

Assertions and assumptions are treated specially in bindings

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

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

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

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

lemma *ASSUME-refine-left-pres*:
assumes $\Phi \Longrightarrow$ *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* :: $('a \times 'b)\ set \Rightarrow ('b \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$
— Lift assertion over refinement relation
where *lift-assn* $R\ \Phi\ s \equiv \exists s'.\ (s,s') \in R \wedge \Phi\ s'$
lemma *lift-assnI*: $[\![(s,s') \in R;\ \Phi\ s']\!] \Longrightarrow$ *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*: $\bigwedge f\ f'\ x\ x'.\ [\![\ \bigwedge x\ x'.\ (x,x') \in R \Longrightarrow f\ x \leq \Downarrow S\ (f'\ x');\ (x,x') \in R;$
REC body$' = f'\ ]\!]$
$\Longrightarrow$ *body* $f\ x \leq \Downarrow S$ (*body*$'\ f'\ x'$)
shows *REC* $(\lambda f\ x.$ *body* $f\ x)\ x \leq \Downarrow S$ (*REC* $(\lambda f'\ x'.$ *body*$'\ f'\ x')\ x'$)
unfolding *REC-def*
apply (*clarsimp simp add*: *M*)
apply (*rule lfp-induct-pointwise*[**where** *pre*=$\lambda x'\ x.\ (x,x') \in R$ **and** *B*=*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*: $(x,x')\in R$
assumes *RS*: $\bigwedge f\ f'\ x\ x'.\ [\![\ \bigwedge x\ x'.\ (x,x')\in R \Longrightarrow f\ x \leq \Downarrow S\ (f'\ x');\ (x,x')\in R\ ]\!]$
$\Longrightarrow$ *body f x* $\leq\Downarrow S$ (*body' f' x'*)
shows *RECT* ($\lambda f\ x.$ *body f x*) *x* $\leq\Downarrow S$ (*RECT* ($\lambda f'\ x'.$ *body' f' x'*) *x'*)
unfolding *RECT-def*
apply (*clarsimp simp add*: *M*)

apply (*rule flatf-fixp-transfer*[**where**
fp'=*flatf-gfp body*
and *B'*=*body*
and *P*=$\lambda x\ x'.$ (x',x)$\in R$,
OF - - flatf-ord.fixp-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 $[\![b;b']\!] \Longrightarrow S1 \leq \Downarrow R\ S1'$
assumes $[\![\neg b;\neg b']\!] \Longrightarrow S2 \leq \Downarrow R\ S2'$
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 $(m,m')\in R'$
assumes $\bigwedge x\ x'.\ (x,x')\in R' \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
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 $(m,m')\in R \Longrightarrow f\ m \leq \Downarrow S\ (f'\ m')$
shows *Let* $m\ f \leq \Downarrow S$ (*Let* $m'\ f'$)
using *assms* **by** *simp*

lemma *case-option-refine*[*refine*]:
assumes $(v,v')\in\langle Ra\rangle$*option-rel*
assumes $[\![v=None;\ v'=None]\!] \Longrightarrow n \leq \Downarrow\ Rb\ n'$
assumes $\bigwedge x\ x'.\ [\![\ v=Some\ x;\ v'=Some\ x';\ (x,\ x')\in Ra\ ]\!]$
$\Longrightarrow f\ x \leq \Downarrow\ Rb\ (f'\ x')$
shows *case-option* $n\ f\ v \leq \Downarrow Rb$ (*case-option* $n'\ f'\ v'$)
using *assms*
by (*auto split*: *option.split simp*: *option-rel-def*)

lemma *list-case-refine*[*refine*]:
assumes $(li,l)\in\langle S\rangle$*list-rel*
assumes $fni \leq \Downarrow R\ fn$
assumes $\bigwedge xi\ x\ xsi\ xs.\ [\![\ (xi,x)\in S;\ (xsi,xs)\in\langle S\rangle$*list-rel*; $li=xi\#xsi$; $l=x\#xs\ ]\!] \Longrightarrow$ $fci\ xi\ xsi \leq \Downarrow R\ (fc\ x\ xs)$
shows (*case li of* $[] \Rightarrow fni \mid xi\#xsi \Rightarrow fci\ xi\ xsi) \leq \Downarrow R$ (*case l of* $[] \Rightarrow fn \mid x\#xs \Rightarrow fc\ x\ xs$)
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 $M \leq \Downarrow R$ (*Let* $x\ f$) **using** *assms* **by** *auto*

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

lemma *bind2let-refine*[*refine2*]:
assumes $RETURN\ x \leq \Downarrow R'\ M'$
assumes $\bigwedge x'.\ (x,x')\in R' \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
shows *Let* $x\ f \leq \Downarrow R$ (*bind* M' ($\lambda x'.\ f'\ x'$))
using *assms*
apply (*simp add*: *pw-le-iff refine-pw-simps*)
apply *fast*
done

lemma *bind-Let-refine2*[*refine2*]: ⟦
  $m' \leq \Downarrow R'\ (RETURN\ x)$;
  $\bigwedge x'.\ ⟦inres\ m'\ x';\ (x',x)\in R'⟧ \Longrightarrow f'\ x' \leq \Downarrow R\ (f\ x)$
 ⟧ $\Longrightarrow m' \gg\!= (\lambda x'.\ f'\ x') \leq \Downarrow R\ (Let\ x\ (\lambda x.\ f\ x))$
 apply (*simp add*: *pw-le-iff refine-pw-simps*)
 apply *blast*
 done

lemma *bind2letRETURN-refine*[*refine2*]:
 assumes $RETURN\ x \leq \Downarrow R'\ M'$
 assumes $\bigwedge x'.\ (x,x')\in R' \Longrightarrow RETURN\ (f\ x) \leq \Downarrow R\ (f'\ x')$
 shows $RETURN\ (Let\ x\ f) \leq \Downarrow R\ (bind\ M'\ (\lambda x'.\ f'\ x'))$
 using *assms*
 apply (*simp add*: *pw-le-iff refine-pw-simps*)
 apply *fast*
 done

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

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

lemma *if-RETURN-refine* [*refine2*]:
 assumes $b \longleftrightarrow b'$
 assumes $⟦b;b'⟧ \Longrightarrow RETURN\ S1 \leq \Downarrow R\ S1'$
 assumes $⟦\neg b;\neg b'⟧ \Longrightarrow RETURN\ S2 \leq \Downarrow R\ S2'$
 shows $RETURN\ (if\ b\ then\ S1\ else\ S2) \leq \Downarrow R\ (if\ b'\ then\ S1'\ else\ S2')$

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

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

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

lemma *intro-spec-refine*[*refine2*]:
assumes $\bigwedge x.\ x{\in}X \Longrightarrow f\ x \leq \Downarrow R\ M$
shows *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 $\bigwedge x\ x'.\ (x,x'){\in}R' \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
shows *bind* m ($\lambda x.\ f\ x$) $\leq \Downarrow R$ (*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 $\bigwedge x.\ (x,m'){\in}R' \Longrightarrow f\ x \leq \Downarrow R\ m''$
shows *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 *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*:
$[\![\ b \Longrightarrow t \leq m;\ \neg b \Longrightarrow e \leq m\ ]\!] \Longrightarrow If\ b\ t\ e \leq m$ **by** *simp*

lemma *lhs-step-RES*:
$[\![\ \bigwedge x.\ x{\in}X \Longrightarrow RETURN\ x \leq m\ ]\!] \Longrightarrow RES\ X \leq m$
by (*simp add*: *pw-le-iff*)

lemma *lhs-step-SPEC*:
⟦ ⋀x. Φ x ⟹ RETURN x ≤ m ⟧ ⟹ SPEC (λx. Φ x) ≤ m
by (*simp add*: *pw-le-iff*)

lemma *lhs-step-bind*:
fixes m :: 'a nres **and** f :: 'a ⇒ 'b nres
assumes nofail m' ⟹ nofail m
assumes ⋀x. nf-inres m x ⟹ f x ≤ m'
shows do {x←m; f x} ≤ m'
using *assms*
by (*simp add*: *pw-le-iff refine-pw-simps*) *blast*

lemma *rhs-step-bind*:
assumes m ≤ ⇓R m' inres m x ⋀x'. (x,x')∈R ⟹ lhs ≤⇓S (f' x')
shows lhs ≤ ⇓S (m' ≫= f')
using *assms*
by (*simp add*: *pw-le-iff refine-pw-simps*) *blast*

lemma *rhs-step-bind-RES*:
assumes x'∈X'
assumes m ≤ ⇓R (f' x')
shows m ≤ ⇓R (RES X' ≫= f')
using *assms* **by** (*simp add*: *pw-le-iff refine-pw-simps*) *blast*

lemma *rhs-step-bind-SPEC*:
assumes Φ x'
assumes m ≤ ⇓R (f' x')
shows m ≤ ⇓R (SPEC Φ ≫= f')
using *assms* **by** (*simp add*: *pw-le-iff refine-pw-simps*) *blast*

lemma *RES-bind-choose*:
assumes x∈X
assumes m ≤ f x
shows m ≤ RES X ≫= f
using *assms* **by** (*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *pw-RES-bind-choose*:
nofail (RES X ≫= f) ⟷ (∀x∈X. nofail (f x))
inres (RES X ≫= f) y ⟷ (∃x∈X. inres (f x) y)
by (*auto simp*: *refine-pw-simps*)

lemma *prod-case-refine*:
assumes (p',p)∈R1×ᵣR2
assumes ⋀x1' x2' x1 x2. ⟦ p'=(x1',x2'); p=(x1,x2); (x1',x1)∈R1; (x2',x2)∈R2⟧
⟹ f' x1' x2' ≤ ⇓R (f x1 x2)
shows (*case* p' *of* (x1',x2') ⇒ f' x1' x2') ≤⇓R (*case* p *of* (x1,x2) ⇒ 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-rel R* ≡ {(*c*,*a*). *c* ≤ ⇓*R a*}

lemma *nres-rel-def*: ⟨*R*⟩*nres-rel* ≡ {(*c*,*a*). *c* ≤ ⇓*R a*}
by (*simp add*: *nres-rel-def-internal relAPP-def*)

lemma *nres-relD*: (*c*,*a*)∈⟨*R*⟩*nres-rel* ⟹ *c* ≤⇓*R a* **by** (*simp add*: *nres-rel-def*)
lemma *nres-relI*[*refine*]: *c* ≤⇓*R a* ⟹ (*c*,*a*)∈⟨*R*⟩*nres-rel* **by** (*simp add*: *nres-rel-def*)

lemma *nres-rel-comp*: ⟨*A*⟩*nres-rel O* ⟨*B*⟩*nres-rel* = ⟨*A O B*⟩*nres-rel*
by (*auto simp*: *nres-rel-def conc-fun-chain*[*symmetric*] *conc-trans*)

lemma *pw-nres-rel-iff*: (*a*,*b*)∈⟨*A*⟩*nres-rel* ⟷ *nofail* (⇓ *A b*) ⟶ *nofail a* ∧ (∀ *x*. *inres a x* ⟶ *inres* (⇓ *A b*) *x*)
by (*simp add*: *pw-le-iff nres-rel-def*)

lemma *param-SUCCEED*[*param*]: (*SUCCEED*,*SUCCEED*) ∈ ⟨*R*⟩*nres-rel*
by (*auto simp*: *nres-rel-def*)

lemma *param-FAIL*[*param*]: (*FAIL*,*FAIL*) ∈ ⟨*R*⟩*nres-rel*
by (*auto simp*: *nres-rel-def*)

lemma *param-RES*[*param*]:
(*RES*,*RES*) ∈ ⟨*R*⟩*set-rel* → ⟨*R*⟩*nres-rel*
unfolding *set-rel-def nres-rel-def*
by (*fastforce intro*: *RES-refine*)

lemma *param-RETURN*[*param*]:
(*RETURN*,*RETURN*) ∈ *R* → ⟨*R*⟩*nres-rel*
by (*auto simp*: *nres-rel-def RETURN-refine*)

lemma *param-bind*[*param*]:
(*bind*,*bind*) ∈ ⟨*Ra*⟩*nres-rel* → (*Ra*→⟨*Rb*⟩*nres-rel*) → ⟨*Rb*⟩*nres-rel*
by (*auto simp*: *nres-rel-def intro*: *bind-refine dest*: *fun-relD*)

lemma *param-ASSERT-bind*[*param*]: ⟦
(Φ,Ψ) ∈ *bool-rel*;
⟦ Φ; Ψ ⟧ ⟹ (*f*,*g*)∈⟨*R*⟩*nres-rel*
⟧ ⟹ (*ASSERT* Φ ≫ *f*, *ASSERT* Ψ ≫ *g*) ∈ ⟨*R*⟩*nres-rel*
by (*auto intro*: *nres-relI*)

2.6.7 Autoref Setup

consts *i-nres* :: *interface* ⇒ *interface*
lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of nres-rel i-nres*]

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

lemma *param-op-nres-ASSERT-bnd*[*param*]:
assumes Φ′ ⟹ Φ
assumes ⟦Φ′; Φ⟧ ⟹ (*m*,*m*′)∈⟨*R*⟩*nres-rel*
shows (*op-nres-ASSERT-bnd* Φ *m*, *op-nres-ASSERT-bnd* Φ′ *m*′) ∈ ⟨*R*⟩*nres-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* Φ; *m*} ≡ *OP* (*op-nres-ASSERT-bnd* Φ)$*m*
by *simp*

definition [*simp*]: *op-nres-ASSUME-bnd* Φ *m* ≡ *do* {*ASSUME* Φ; *m*}
lemma *id-ASSUME*[*autoref-op-pat-def*]:
do {*ASSUME* Φ; *m*} ≡ *OP* (*op-nres-ASSUME-bnd* Φ)$*m*
by *simp*

end

lemma *autoref-SUCCEED*[*autoref-rules*]: (*SUCCEED*,*SUCCEED*) ∈ ⟨*R*⟩*nres-rel*
by (*auto simp*: *nres-rel-def*)

lemma *autoref-FAIL*[*autoref-rules*]: (*FAIL*,*FAIL*) ∈ ⟨*R*⟩*nres-rel*
by (*auto simp*: *nres-rel-def*)

lemma *autoref-RETURN*[*autoref-rules*]:
(*RETURN*,*RETURN*) ∈ *R* → ⟨*R*⟩*nres-rel*
by (*auto simp*: *nres-rel-def RETURN-refine*)

lemma *autoref-bind*[*autoref-rules*]:
(*bind*,*bind*) ∈ ⟨*R1*⟩*nres-rel* → (*R1*→⟨*R2*⟩*nres-rel*) → ⟨*R2*⟩*nres-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 $\Phi \Longrightarrow (m',m)\in\langle R\rangle nres\text{-}rel$
shows (
m',
(*OP* (*op-nres-ASSERT-bnd* Φ) ::: $\langle R\rangle nres\text{-}rel \to \langle R\rangle nres\text{-}rel$) $ $m)\in\langle R\rangle nres\text{-}rel$
using *assms* **unfolding** *nres-rel-def*
by (*simp add*: *ASSERT-refine-right*)

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

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

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*: $A \leq \Downarrow R \;\; B \Longrightarrow B \leq C \Longrightarrow A \leq \Downarrow R \;\; C$
by (*rule conc-trans-additional*)

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

lemma *pw-ref-I*:
assumes *nofail* S'
$\longrightarrow$ *nofail* $S \wedge (\forall x.\ inres\ S\ x \longrightarrow (\exists s'.\ (x,\ s') \in R \wedge inres\ S'\ s'))$
shows $S \leq \Downarrow R\ S'$
using *assms*
by (*simp add*: *pw-ref-iff*)

Introduce an abstraction relation. Usage: *rule introR*[*where R*=*absRel*]

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

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

lemma *refine-IdI*: $m \leq m' \Longrightarrow m \leq \Downarrow Id\ m'$ **by** *simp*

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

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

lemma *RETURN-ref-RETURND*:
assumes $RETURN\ c \leq \Downarrow R\ (RETURN\ a)$
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** $(x,x')\in R \quad RETURN\ x' \leq m$
using *assms*
by (*auto simp*: *refine-pw-simps pw-le-iff*)

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

lemma *ret-le-down-conv*:
$nofail\ m \Longrightarrow RETURN\ c \leq \Downarrow R\ m \longleftrightarrow (\exists\, a.\ (c,a)\in R \wedge RETURN\ a \leq m)$
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 $nofail\ b \Longrightarrow a \leq SPEC\ (\lambda r.\ (\exists\, x.\ inres\ b\ x \wedge (r,x)\in R)\ )$
shows $a \leq \Downarrow R\ b$
using *assms* **by** (*simp add*: *refine2spec-aux*)

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

lemma *build-rel-SPEC*:
$M \leq SPEC\ (\ \lambda x.\ \Phi\ (\alpha\ x) \wedge I\ x) \Longrightarrow M \leq \Downarrow(build\text{-}rel\ \alpha\ I)\ (SPEC\ \Phi)$
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*: $c \leq \Downarrow Id\ a \Longrightarrow c \leq a$ **by** *simp*

lemma *bind-sim-select-rule*:
assumes $m \gg\!= f' \leq SPEC\ \Psi$
assumes $\bigwedge x.\ [\![nofail\ m;\ inres\ m\ x;\ f'\ x \leq SPEC\ \Psi]\!] \Longrightarrow f\ x \leq SPEC\ \Phi$
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 causes 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 $nofail\ (f\ x) \Longrightarrow f\ x \leq SPEC\ I$
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 $\Phi\ x$
assumes $m \le f\ x$
shows $m \le SPEC\ \Phi \gg\!= f$
using *assms* **by** (*auto simp add*: *pw-le-iff refine-pw-simps*)

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

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

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

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

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

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

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

lemma *nofail-inf-serialize*:
$[\![nofail\ a;\ nofail\ b]\!] \Longrightarrow inf\ a\ b = do\ \{x \leftarrow a;\ ASSUME\ (inres\ b\ x);\ RETURN\ x\}$

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

lemma *conc-fun-SPEC*:
⇓*R* (*SPEC* (λ*x*. Φ *x*)) = *SPEC* (λ*y*. ∃ *x*. (*y*,*x*)∈*R* ∧ Φ *x*)
by (*auto simp*: *pw-eq-iff refine-pw-simps*)

lemma *conc-fun-RETURN*:
⇓*R* (*RETURN x*) = *SPEC* (λ*y*. (*y*,*x*)∈*R*)
by (*auto simp*: *pw-eq-iff refine-pw-simps*)

lemma *use-spec-rule*:
assumes *m* ≤ *SPEC* Ψ
assumes *m* ≤ *SPEC* (λ*s*. Ψ *s* ⟶ Φ *s*)
shows *m* ≤ *SPEC* Φ
using *assms*
by (*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *strengthen-SPEC*: *m* ≤ *SPEC* Φ ⟹ *m* ≤ *SPEC*(λ*s*. *inres m s* ∧ *nofail m* ∧ Φ *s*)
— Strengthen SPEC by adding trivial upper bound for result
by (*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *weaken-SPEC*:
m ≤ *SPEC* Φ ⟹ (⋀*x*. Φ *x* ⟹ Ψ *x*) ⟹ *m* ≤ *SPEC* Ψ
by (*force elim*!: *order-trans*)

lemma *bind-le-nofailI*:
assumes *nofail m*
assumes ⋀*x*. *RETURN x* ≤ *m* ⟹ *f x* ≤ *m*′
shows *m*≫=*f* ≤ *m*′
using *assms*
by (*simp add*: *refine-pw-simps pw-le-iff*) *blast*

lemma *bind-le-shift*:
bind m f ≤ *m*′
⟷ *m* ≤ (*if nofail m*′ *then SPEC* (λ*x*. *f x* ≤ *m*′) *else FAIL*)
by (*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *If-bind-distrib*[*simp*]:
fixes *t e* :: *'a nres*
shows (*If b t e* ≫= (λ*x*. *f x*)) = (*If b* (*t*≫=(λ*x*. *f x*)) (*e*≫=(λ*x*. *f x*)))
by *simp*

lemma *unused-bind-conv*:
assumes *NO-MATCH* (*ASSERT* Φ) *m*
assumes *NO-MATCH* (*ASSUME* Φ) *m*

shows $(m \gg\!= (\lambda x.\ c)) = (ASSERT\ (nofail\ m) \gg\!= (\lambda\text{-}.\ ASSUME\ (\exists x.\ inres\ m\ x) \gg\!= (\lambda x.\ c)))$
by (*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*:
$Let\ x\ f = RETURN\ x \gg\!= f$
by *simp*

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

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

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

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

lemma *if-bind-cond-refine*:
assumes $ci \le RETURN\ b$
assumes $b \Longrightarrow ti \le \Downarrow R\ t$
assumes $\neg b \Longrightarrow ei \le \Downarrow R\ e$
shows $do\ \{b \leftarrow ci;\ if\ b\ then\ ti\ else\ ei\} \le \Downarrow R\ (if\ b\ then\ t\ else\ e)$
using *assms*
by (*auto simp add*: *refine-pw-simps pw-le-iff*)

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

using *assms* **by** *auto*

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

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

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

lemma *le-ASSERT-defI1*:
assumes $c \equiv do\ \{ASSERT\ \Phi;\ m\}$
assumes $\Phi \Longrightarrow m' \leq c$
shows $m' \leq c$
using *assms*
by (*simp add*: *le-ASSERTI*)

lemma *refine-ASSERT-defI1*:
assumes $c \equiv do\ \{ASSERT\ \Phi;\ m\}$
assumes $\Phi \Longrightarrow m' \leq \Downarrow R\ c$
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 $[\![\Phi;\ \Psi]\!] \Longrightarrow m' \leq c$
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 $[\![\Phi;\ \Psi]\!] \Longrightarrow m' \leq \Downarrow R\ c$
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 $\Phi \Longrightarrow m' \leq m$
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* $(RECT\ B\ x) \Longrightarrow RECT\ B\ x = REC\ B\ x$
by (*subst RECT-eq-REC*; *simp-all add*: *nofail-def*)

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

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

Boolean Operations on Specifications

lemma *SPEC-iff*:
assumes $P \leq SPEC\ (\lambda s.\ Q\ s \longrightarrow R\ s)$
and $P \leq SPEC\ (\lambda s.\ \neg\ Q\ s \longrightarrow \neg\ R\ s)$
shows $P \leq SPEC\ (\lambda s.\ Q\ s \longleftrightarrow R\ s)$
using *assms*[*THEN pw-le-iff*[*THEN iffD1*]]
by (*auto intro*!: *pw-leI*)

lemma *SPEC-rule-conjI*:
assumes $A \leq SPEC\ P$ **and** $A \leq SPEC\ Q$
shows $A \leq SPEC\ (\lambda v.\ P\ v \wedge Q\ v)$
proof −
have $A \leq inf\ (SPEC\ P)\ (SPEC\ Q)$ **using** *assms* **by** (*rule-tac inf-greatest*) *assumption*
thus *?thesis* **by** (*auto simp add*:*Collect-conj-eq*)
qed

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

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

Pointwise Reasoning

lemma *inres-if*:
$[\![\ inres\ (if\ P\ then\ Q\ else\ R)\ x;\ [\![P;\ inres\ Q\ x]\!] \Longrightarrow S;\ [\![\neg\ P;\ inres\ R\ x]\!] \Longrightarrow S\ ]\!] \Longrightarrow S$
by (*metis* (*full-types*))

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

lemma *SPEC-nofail*:
$X \leq SPEC\ \Phi \Longrightarrow nofail\ X$
by (*auto dest*: *pwD1*)

lemma *nofail-SPEC*: $nofail\ m \Longrightarrow m \leq SPEC\ (\lambda\text{-}.\ True)$
by (*simp add*: *pw-le-iff*)

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

lemma *nofail-SPEC-triv-refine*: $[\![\ nofail\ m;\ \bigwedge x.\ \Phi\ x\ ]\!] \Longrightarrow m \leq SPEC\ \Phi$
by (*simp add*: *pw-le-iff*)

end

2.7 Less-Equal or Fail

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

A predicate that states refinement or that the LHS fails.

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

lemma *leofI*[*intro?*]:
assumes $nofail\ m \Longrightarrow m \leq m'$ **shows** $m \leq_n m'$
using *assms* **unfolding** *le-or-fail-def* **by** *auto*

lemma *leofD*:
assumes *nofail m*
assumes $m \leq_n m'$
shows $m \leq m'$
using *assms* **unfolding** *le-or-fail-def* **by** *blast*

lemma *pw-leof-iff*:
$m \leq_n m' \longleftrightarrow (nofail\ m \longrightarrow (\forall x.\ inres\ m\ x \longrightarrow inres\ m'\ x))$
unfolding *le-or-fail-def* **by** (*auto simp add*: *pw-le-iff refine-pw-simps*)

lemma *le-by-leofI*: $[\![\ nofail\ m' \Longrightarrow nofail\ m;\ m \leq_n m'\ ]\!] \Longrightarrow m \leq m'$
by (*auto simp*: *pw-le-iff pw-leof-iff*)

lemma *inres-leof-mono*: $m \leq_n m' \Longrightarrow nofail\ m \Longrightarrow inres\ m\ x \Longrightarrow inres\ m'\ x$
by (*auto simp*: *pw-leof-iff*)

lemma *leof-trans*[*trans*]: ⟦*a* ≤$_n$ *RES X*; *RES X* ≤$_n$ *c*⟧ ⟹ *a* ≤$_n$ *c*
by (*auto simp*: *pw-leof-iff*)

lemma *leof-trans-nofail*: ⟦ *a*≤$_n$*b*; *nofail b*; *b*≤$_n$*c* ⟧ ⟹ *a* ≤$_n$ *c*
by (*auto simp*: *pw-leof-iff*)

lemma *leof-refl*[*simp*]: *a* ≤$_n$ *a*
by (*auto simp*: *pw-leof-iff*)

lemma *leof-RES-UNIV*[*simp*, *intro*!]: *m* ≤$_n$ *RES UNIV*
by (*auto simp*: *pw-leof-iff*)

lemma *leof-FAIL*[*simp*, *intro*!]: *m* ≤$_n$ *FAIL* **by** (*auto simp*: *pw-leof-iff*)
lemma *FAIL-leof*[*simp*, *intro*!]: *FAIL* ≤$_n$ *m*
by (*auto simp*: *le-or-fail-def*)

lemma *leof-lift*:
m ≤ *F* ⟹ *m* ≤$_n$ *F*
by (*auto simp add*: *pw-leof-iff pw-le-iff*)

lemma *leof-RETURN-rule*[*refine-vcg*]:
Φ *m* ⟹ *RETURN m* ≤$_n$ *SPEC* Φ **by** (*simp add*: *pw-leof-iff*)

lemma *leof-bind-rule*[*refine-vcg*]:
⟦ *m* ≤$_n$ *SPEC* (λ*x*. *f x* ≤$_n$ *SPEC* Φ) ⟧ ⟹ *m*≫=*f* ≤$_n$ *SPEC* Φ
by (*auto simp add*: *pw-leof-iff refine-pw-simps*)

lemma *RETURN-leof-RES-iff*[*simp*]: *RETURN x* ≤$_n$ *RES Y* ⟷ *x*∈*Y*
by (*auto simp add*: *pw-leof-iff refine-pw-simps*)

lemma *RES-leof-RES-iff*[*simp*]: *RES X* ≤$_n$ *RES Y* ⟷ *X* ⊆ *Y*
by (*auto simp add*: *pw-leof-iff refine-pw-simps*)

lemma *leof-Let-rule*[*refine-vcg*]: *f x* ≤$_n$ *SPEC* Φ ⟹ *Let x f* ≤$_n$ *SPEC* Φ
by *simp*

lemma *leof-If-rule*[*refine-vcg*]:
⟦*c* ⟹ *t* ≤$_n$ *SPEC* Φ; ¬*c* ⟹ *e* ≤$_n$ *SPEC* Φ⟧ ⟹ *If c t e* ≤$_n$ *SPEC* Φ
by *simp*

lemma *leof-RES-rule*[*refine-vcg*]:
⟦⋀*x*. Ψ *x* ⟹ Φ *x*⟧ ⟹ *SPEC* Ψ ≤$_n$ *SPEC* Φ
⟦⋀*x*. *x*∈*X* ⟹ Φ *x*⟧ ⟹ *RES X* ≤$_n$ *SPEC* Φ
by *auto*

lemma *leof-True-rule*: ⟦⋀*x*. Φ *x*⟧ ⟹ *m* ≤$_n$ *SPEC* Φ
by (*auto simp add*: *pw-leof-iff refine-pw-simps*)

lemma *sup-leof-iff*: (*sup a b* ≤$_n$ *m*) ⟷ (*nofail a* ∧ *nofail b* ⟶ *a*≤$_n$*m* ∧ *b*≤$_n$*m*)

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

lemma *sup-leof-rule*[*refine-vcg*]:
assumes ⟦*nofail a*; *nofail b*⟧ ⟹ $a \leq_n m$
assumes ⟦*nofail a*; *nofail b*⟧ ⟹ $b \leq_n m$
shows *sup a b* $\leq_n$ *m*
using *assms* **by** (*auto simp*: *pw-leof-iff refine-pw-simps*)

lemma *leof-option-rule*[*refine-vcg*]:
⟦*v* = *None* ⟹ *S1* $\leq_n$ *SPEC* Φ; ⋀*x*. *v* = *Some x* ⟹ *f2 x* $\leq_n$ *SPEC* Φ⟧
⟹ (*case v of None* ⇒ *S1* | *Some x* ⇒ *f2 x*) $\leq_n$ *SPEC* Φ
by (*cases v*) *auto*

lemma *ASSERT-leof-rule*[*refine-vcg*]:
assumes Φ ⟹ *m* $\leq_n$ *m'*
shows *do* {*ASSERT* Φ; *m*} $\leq_n$ *m'*
using *assms*
by (*cases* Φ, *auto simp*: *pw-leof-iff*)

lemma *leof-ASSERT-rule*[*refine-vcg*]: ⟦Φ ⟹ *m* $\leq_n$ *m'*⟧ ⟹ *m* $\leq_n$ *ASSERT* Φ ≫ *m'*
by (*auto simp*: *pw-leof-iff refine-pw-simps*)

lemma *leof-ASSERT-refine-rule*[*refine*]: ⟦Φ ⟹ *m* $\leq_n$ ⇓*R m'*⟧ ⟹ *m* $\leq_n$ ⇓*R* (*ASSERT* Φ ≫ *m'*)
by (*auto simp*: *pw-leof-iff refine-pw-simps*)

lemma *ASSUME-leof-iff*: (*ASSUME* Φ $\leq_n$ *SPEC* Ψ) ⟷ (Φ ⟶ Ψ ())
by (*auto simp*: *pw-leof-iff*)

lemma *ASSUME-leof-rule*[*refine-vcg*]:
assumes Φ ⟹ Ψ ()
shows *ASSUME* Φ $\leq_n$ *SPEC* Ψ
using *assms*
by (*auto simp*: *ASSUME-leof-iff*)

lemma *SPEC-rule-conj-leofI1*:
assumes *m* ≤ *SPEC* Φ
assumes *m* $\leq_n$ *SPEC* Ψ
shows *m* ≤ *SPEC* (λ*s*. Φ *s* ∧ Ψ *s*)
using *assms* **by** (*auto simp*: *pw-le-iff pw-leof-iff*)

lemma *SPEC-rule-conj-leofI2*:
assumes *m* $\leq_n$ *SPEC* Φ
assumes *m* ≤ *SPEC* Ψ
shows *m* ≤ *SPEC* (λ*s*. Φ *s* ∧ Ψ *s*)
using *assms* **by** (*auto simp*: *pw-le-iff pw-leof-iff*)

lemma *SPEC-rule-leof-conjI*:
assumes $m \leq_n SPEC\ \Phi \quad m \leq_n SPEC\ \Psi$
shows $m \leq_n SPEC\ (\lambda x.\ \Phi\ x \wedge \Psi\ x)$
using *assms* **by** (*auto simp*: *pw-leof-iff*)

lemma *leof-use-spec-rule*:
assumes $m \leq_n SPEC\ \Psi$
assumes $m \leq_n SPEC\ (\lambda s.\ \Psi\ s \longrightarrow \Phi\ s)$
shows $m \leq_n SPEC\ \Phi$
using *assms* **by** (*auto simp*: *pw-leof-iff refine-pw-simps*)

lemma *use-spec-leof-rule*:
assumes $m \leq_n SPEC\ \Psi$
assumes $m \leq SPEC\ (\lambda s.\ \Psi\ s \longrightarrow \Phi\ s)$
shows $m \leq SPEC\ \Phi$
using *assms* **by** (*auto simp*: *pw-leof-iff pw-le-iff refine-pw-simps*)

lemma *leof-strengthen-SPEC*:
$m \leq_n SPEC\ \Phi \Longrightarrow m \leq_n SPEC\ (\lambda x.\ inres\ m\ x \wedge \Phi\ x)$
by (*auto simp*: *pw-leof-iff*)

lemma *build-rel-SPEC-leof*:
assumes $m \leq_n SPEC\ (\lambda x.\ I\ x \wedge \Phi\ (\alpha\ x))$
shows $m \leq_n \Downarrow(br\ \alpha\ I)\ (SPEC\ \Phi)$
using *assms* **by** (*auto simp*: *build-rel-SPEC-conv*)

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

lemma *ASSERT-leof-defI*:
assumes $c \equiv do\ \{\ ASSERT\ \Phi;\ m'\}$
assumes $\Phi \Longrightarrow m' \leq_n m$
shows $c \leq_n m$
using *assms* **by** (*auto simp*: *pw-leof-iff refine-pw-simps*)

lemma *leof-fun-conv-le*:
$(f\ x \leq_n M\ x) \longleftrightarrow (f\ x \leq (if\ nofail\ (f\ x)\ then\ M\ x\ else\ FAIL))$
by (*auto simp*: *pw-le-iff pw-leof-iff*)

lemma *leof-add-nofailI*: $[\![\ nofail\ m \Longrightarrow m \leq_n m'\ ]\!] \Longrightarrow m \leq_n m'$
by (*auto simp*: *pw-le-iff pw-leof-iff*)

lemma *leof-cons-rule*[*refine-vcg-cons*]:
assumes $m \leq_n SPEC\ Q$
assumes $\bigwedge x.\ Q\ x \Longrightarrow P\ x$

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::($'x \times 'x$) *set*)
assumes *I0*: *pre* (x::$'x$)
assumes *IS*: $\bigwedge f\ x.\ [\![\bigwedge x'.\ [\![pre\ x';\ (x',x) \in V]\!] \Longrightarrow f\ x' \leq_n M\ x';\ pre\ x;$
RECT body = *f*
$]\!] \Longrightarrow$ *body f x* $\leq_n$ *M x*
shows *RECT body x* $\leq_n$ *M x*
apply (*cases* ¬*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*: $\bigwedge x'.\ [\![pre\ x';\ (x',\ xa) \in V]\!] \Longrightarrow REC_T\ body\ x' \leq$ (*if nofail* (*REC*$_T$ *body x'*) *then M x' else FAIL*)
assume *a2*: $\bigwedge x\ f.\ [\![\bigwedge x'.\ [\![pre\ x';\ (x',\ x) \in V]\!] \Longrightarrow f\ x' \leq$ (*if nofail* (*f x'*) *then M x' else FAIL*); *pre x*; *REC*$_T$ *body* = *f*$]\!] \Longrightarrow$ *body f x* $\leq$ (*if nofail* (*body f x*) *then M x else FAIL*)
assume *a3*: *pre xa*
assume *a4*: *nofail* (*REC*$_T$ *body xa*)
assume *a5*: *trimono body*
have *f6*: $\forall x.\ \neg$ *pre x* $\vee$ (*x*, *xa*) $\notin$ *V* $\vee$ (*if nofail* (*REC*$_T$ *body x*) *then REC*$_T$ *body x* $\leq$ *M x else REC*$_T$ *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* $\leq$ (*if nofail* (*f xa*) *then M xa else FAIL*)) $\vee \neg$ *pre x* $\vee$ *REC*$_T$ *body* $\neq$ *f*) $\vee$ *body f x* $\leq$ (*if nofail* (*body f x*) *then M x else FAIL*)
using *a2* **by** *blast*
obtain *xx* :: ($'x \Rightarrow 'a$ *nres*) $\Rightarrow$ $'x \Rightarrow 'x$ **where**
f8: $\forall$ *x0 x1*. ($\exists$ *v2*. (*pre v2* $\wedge$ (*v2*, *x1*) $\in$ *V*) $\wedge \neg$ *x0 v2* $\leq$ (*if nofail* (*x0 v2*) *then M v2 else FAIL*)) = ((*pre* (*xx x0 x1*) $\wedge$ (*xx x0 x1*, *x1*) $\in$ *V*) $\wedge \neg$ *x0* (*xx x0 x1*) $\leq$ (*if nofail* (*x0* (*xx x0 x1*)) *then M* (*xx x0 x1*) *else FAIL*))
by *moura*
have *f9*: $\forall$ *x0 x1*. (*x0* (*xx x0 x1*) $\leq$ (*if nofail* (*x0* (*xx x0 x1*)) *then M* (*xx x0 x1*) *else FAIL*)) = (*if nofail* (*x0* (*xx x0 x1*)) *then x0* (*xx x0 x1*) $\leq$ *M* (*xx x0 x1*) *else x0* (*xx x0 x1*) $\leq$ *FAIL*)
by *presburger*
have *nofail* (*body* (*REC*$_T$ *body*) *xa*)
using *a5 a4* **by** (*metis* (*no-types*) *RECT-unfold*)
then show *body* (*REC*$_T$ *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: ^
      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->bool) (ctxt:Proof.context) i thm =
  if Thm.nprems-of thm >= 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 \n ^ 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*]:
$RELATES\ R \Longrightarrow (a,b)\in R \Longrightarrow (a,b)\in R$ **.**
lemma *RELATESI-refspec*[*refine-dref-pattern*]:
$RELATES\ R \Longrightarrow S \leq\Downarrow R\ S' \Longrightarrow S \leq\Downarrow R\ S'$ **.**

Allows refine-rules to add *RELATES* goals if they introduce hidden relations

lemma *RELATES-pattern*[*refine-dref-pattern*]: $RELATES\ R \Longrightarrow RELATES\ R$ **.**
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*]:
$RELATES\ Ra \Longrightarrow RELATES\ Rb \Longrightarrow RELATES\ (\langle Ra,Rb\rangle prod\text{-}rel)$
by (*simp add*: *RELATES-def prod-rel-def*)

declare *prod-rel-sv*[*refine-hsimp*]
lemma *prod-rel-iff*[*refine-hsimp*]:
(*(a,b),(a′,b′)*)∈⟨*A,B*⟩*prod-rel* ⟷ (*a,a′*)∈*A* ∧ (*b,b′*)∈*B*
by (*auto simp*: *prod-rel-def*)

lemmas [*refine-hsimp*] = *prod-rel-id-simp*

lemma *option-rel-RELATES*[*refine-dref-RELATES*]:
RELATES Ra ⟹ *RELATES* (⟨*Ra*⟩*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 ⟹ *RELATES* (⟨*R*⟩*set-rel*) **by** (*simp add*: *RELATES-def*)

lemma *set-rel-empty-eq*: (*S,S′*)∈⟨*X*⟩*set-rel* ⟹ *S*={} ⟷ *S′*={}
by (*auto simp*: *set-rel-def*)

lemma *set-rel-sngD*: ({*a*},{*b*})∈⟨*R*⟩*set-rel* ⟹ (*a,b*)∈*R*
by (*auto simp*: *set-rel-def*)

lemma *Image-insert*[*refine-hsimp*]:
(*a,b*)∈*R* ⟹ *single-valued R* ⟹ *R‘‘insert a A* = *insert b* (*R‘‘A*)
by (*auto dest*: *single-valuedD*)

lemmas [*refine-hsimp*] = *Image-Un*

lemma *Image-Diff*[*refine-hsimp*]:
single-valued (*converse R*) ⟹ *R‘‘*(*A*−*B*) = *R‘‘A* − *R‘‘B*
by (*auto dest*: *single-valuedD*)

lemma *Image-Inter*[*refine-hsimp*]:
single-valued (*converse R*) ⟹ *R‘‘*(*A*∩*B*) = *R‘‘A* ∩ *R‘‘B*
by (*auto dest*: *single-valuedD*)

lemma *list-rel-RELATES*[*refine-dref-RELATES*]:
RELATES R ⟹ *RELATES* (⟨*R*⟩*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* $\Longrightarrow$ *RELATES* ($\langle R\rangle$*nres-rel*)
by (*simp add*: *RELATES-def*)

end

2.9 More Combinators

theory *Refine-More-Comb*
imports *Refine-Basic Refine-Heuristics Refine-Leof*
begin

OBTAIN Combinator

Obtain value with given property, asserting that there exists one.

definition *OBTAIN* :: ('*a* $\Rightarrow$ *bool*) $\Rightarrow$ '*a nres*
where
OBTAIN P $\equiv$ *ASSERT* ($\exists x.$ *P x*) $\gg$ *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*]: $[\![\exists x.$ *P x*; $\bigwedge x.$ *P x* $\Longrightarrow$ *Q x* $]\!] \Longrightarrow$ *OBTAIN P* $\leq$ *SPEC Q*
unfolding *OBTAIN-def*
by *auto*
lemma *OBTAIN-refine-iff*: *OBTAIN P* $\leq\Downarrow R$ (*OBTAIN Q*) $\longleftrightarrow$ (*Ex Q* $\longrightarrow$ *Ex P* $\wedge$ *Collect P* $\subseteq$ R^{-1} ``*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* $\Longrightarrow$ *Ex P*
assumes $\bigwedge x\ y.$ $[\![Q\ x;\ P\ y]\!] \Longrightarrow \exists x'.$ $(y,x')\in R \wedge Q\ x'$
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* $\Rightarrow$ *bool*) $\Rightarrow$ '*a option nres*

where *SELECT P* ≡ *if* ∃*x*. *P x then RES* {*Some x*| *x*. *P x*} *else RETURN None*

lemma *SELECT-rule*[*refine-vcg*]:
assumes ⋀*x*. *P x* ⟹ *Q* (*Some x*)
assumes ∀*x*. ¬*P x* ⟹ *Q None*
shows *SELECT P* ≤ *SPEC Q*
unfolding *SELECT-def*
using *assms*
by *auto*

lemma *SELECT-refine-iff*: (*SELECT P* ≤⇓(⟨*R*⟩*option-rel*) (*SELECT P′*))
⟷ (
(*Ex P′* ⟶ *Ex P*) ∧
(∀*x*. *P x* ⟶ (∃*x′*. (*x*,*x′*)∈*R* ∧ *P′ x′*))
)
by (*force simp*: *SELECT-def pw-le-iff refine-pw-simps*)

lemma *SELECT-refine*[*refine*]:
assumes *RELATES R*
assumes ⋀*x′*. *P′ x′* ⟹ ∃*x*. *P x*
assumes ⋀*x*. *P x* ⟹ ∃*x′*. (*x*,*x′*)∈*R* ∧ *P′ x′*
shows *SELECT P* ≤⇓(⟨*R*⟩*option-rel*) (*SELECT P′*)
unfolding *SELECT-refine-iff* **using** *assms* **by** *blast*

lemma *SELECT-as-SPEC*: *SELECT P* = *SPEC* (λ*None* ⇒ ∀*x*. ¬*P x* | *Some x* ⇒ *P x*)
unfolding *SELECT-def* **by** (*auto simp*: *pw-eq-iff split*: *option.split*)

lemma *SELECT-pw*[*refine-pw-simps*]:
nofail (*SELECT P*)
inres (*SELECT P*) *r* ⟷ (*r*=*None* ⟶ (∀*x*. ¬*P x*)) ∧ (∀*x*. *r*=*Some x* ⟶ *P x*)
unfolding *SELECT-def*
by *auto*

lemma *SELECT-pw-simps*[*simp*]:
nofail (*SELECT P*)
inres (*SELECT P*) *None* ⟷ (∀*x*. ¬*P x*)
inres (*SELECT P*) (*Some x*) ⟷ *P x*
by (*auto simp*: *refine-pw-simps*)

end

2.10 Generic While-Combinator

theory *RefineG-While*
imports

RefineG-Recursion
HOL−Library.While-Combinator
begin

definition
WHILEI-body bind return I b f ≡
(λ*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 ≡ *REC* (*WHILEI-body bind return I b f*) *s0*
definition
iWHILEIT bind return I b f s0 ≡ *RECT* (*WHILEI-body bind return I b f*) *s0*
definition *iWHILE bind return* ≡ *iWHILEI bind return* (λ-. *True*)
definition *iWHILET bind return* ≡ *iWHILEIT bind return* (λ-. *True*)

lemma *mono-prover-monoI*[*refine-mono*]:
monotone (*fun-ord* (≤)) (*fun-ord* (≤)) *B* ⟹ *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* ⇒ (*'a*⇒*'m*) ⇒ (*'m*::*complete-lattice*)
fixes *return* :: *'a* ⇒ *'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* (λ*x*. *bind* (*f x*) *g*)
assumes *ibind-mono-ge*: ⟦*flat-ge m m'*; ⋀*x*. *flat-ge* (*f x*) (*f' x*)⟧
⟹ *flat-ge* (*bind m f*) (*bind m' f'*)
assumes *ibind-mono*: ⟦(≤) *m m'*; ⋀*x*. (≤) (*f x*) (*f' x*)⟧
⟹ (≤) (*bind m f*) (*bind m' f'*)
assumes *WHILEIT-eq*: *WHILEIT* ≡ *iWHILEIT bind return*
assumes *WHILEI-eq*: *WHILEI* ≡ *iWHILEI bind return*
assumes *WHILET-eq*: *WHILET* ≡ *iWHILET bind return*
assumes *WHILE-eq*: *WHILE* ≡ *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*]:
⟦*trimono B*; ⋀*D x*. *B D x* ≤ *B*′ *D x*⟧ ⟹ *REC B x* ≤ *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*]:
⟦*trimono B*; ⋀*D x*. *B D x* ≤ *B*′ *D x*⟧ ⟹ *RECT B x* ≤ *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*: ⋀*x*. *I x* ⟹ *I*′ *x*
shows *WHILEI I*′ *b f x* ≤ *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*: $\bigwedge x.\ I\ x \Longrightarrow I'\ x$
shows *WHILEIT I' b f x* ≤ *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* ≤ *WHILEIT I b f s*
unfolding *WHILEI-def WHILEIT-def*
by (*rule REC-le-RECT*)

While without Annotated Invariant

lemma *WHILE-unfold*:
WHILE b f s = (*if b s then bind* (*f s*) (*WHILE b f*) *else return s*)
unfolding *WHILE-def*
apply (*subst WHILEI-unfold*)
apply *simp*
done

lemma *WHILET-unfold*:
WHILET b f s = (*if b s then bind* (*f s*) (*WHILET b f*) *else return s*)
unfolding *WHILET-def*
apply (*subst WHILEIT-unfold*)
apply *simp*
done

lemma *transfer-WHILEIT-esc*[*refine-transfer*]:
assumes *REF*: $\bigwedge x.$ *return* (*f x*) ≤ *F x*
shows *return* (*while b f x*) ≤ *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*: $\bigwedge x.\ return\ (f\ x) \leq F\ x$
shows $return\ (while\ b\ f\ x) \leq WHILET\ b\ F\ x$
unfolding *WHILET-def*
using *assms* **by** (*rule transfer-WHILEIT-esc*)

lemma *WHILE-mono-prover-rule*[*refine-mono*]:
$[\![\bigwedge x.\ f\ x \leq f'\ x]\!] \Longrightarrow WHILE\ b\ f\ s0 \leq WHILE\ b\ f'\ s0$
$[\![\bigwedge x.\ f\ x \leq f'\ x]\!] \Longrightarrow WHILEI\ I\ b\ f\ s0 \leq WHILEI\ I\ b\ f'\ s0$
$[\![\bigwedge x.\ f\ x \leq f'\ x]\!] \Longrightarrow WHILET\ b\ f\ s0 \leq WHILET\ b\ f'\ s0$
$[\![\bigwedge x.\ f\ x \leq f'\ x]\!] \Longrightarrow WHILEIT\ I\ b\ f\ s0 \leq WHILEIT\ I\ b\ f'\ s0$

$[\![\bigwedge x.\ flat\text{-}ge\ (f\ x)\ (f'\ x)]\!] \Longrightarrow flat\text{-}ge\ (WHILET\ b\ f\ s0)\ (WHILET\ b\ f'\ s0)$
$[\![\bigwedge x.\ flat\text{-}ge\ (f\ x)\ (f'\ x)]\!] \Longrightarrow flat\text{-}ge\ (WHILEIT\ I\ b\ f\ s0)\ (WHILEIT\ I\ b\ f'\ s0)$
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*: $[\![\alpha\ m \leq M;\ \bigwedge x.\ \alpha\ (f\ x) \leq F\ x\ ]\!]$
$\Longrightarrow \alpha\ (cbind\ m\ f) \leq abind\ M\ F$
assumes *transfer-return*: $\alpha\ (creturn\ x) \leq areturn\ x$
begin

lemma *transfer-WHILEIT*[*refine-transfer*]:
assumes *REF*: $\bigwedge x.\ \alpha\ (f\ x) \leq F\ x$
shows $\alpha\ (cWHILEIT\ I\ b\ f\ x) \leq aWHILEIT\ I\ b\ F\ x$
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 $\alpha\ (cWHILEI\ I\ b\ f\ x) \leq aWHILEI\ I\ b\ F\ x$
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 $\alpha\ (cWHILE\ b\ f\ x) \leq aWHILE\ b\ F\ x$
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 $\alpha\ (cWHILET\ b\ f\ x) \leq aWHILET\ b\ F\ x$
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*: $SPEC\ \Phi = Sup\ \{return\ x\ |\ x.\ \Phi\ x\}$
assumes *ibind-rule*: $[\![\ M \leq SPEC\ (\lambda x.\ f\ x \leq SPEC\ \Phi)\ ]\!] \Longrightarrow bind\ M\ f \leq SPEC\ \Phi$
begin

lemma *ireturn-eq*: $return\ x = SPEC\ ((=)\ x)$
unfolding *iSPEC-eq* **by** *auto*

lemma *iSPEC-rule*: $(\bigwedge x.\ \Phi\ x \Longrightarrow \Psi\ x) \Longrightarrow SPEC\ \Phi \leq SPEC\ \Psi$
unfolding *iSPEC-eq*
by (*auto intro*: *Sup-mono*)

lemma *ireturn-rule*: $\Phi\ x \Longrightarrow return\ x \leq SPEC\ \Phi$
unfolding *ireturn-eq*
by (*auto intro*: *iSPEC-rule*)

lemma *WHILEI-rule*:
assumes *I0*: *I s*
assumes *ISTEP*: $\bigwedge s.\ [\![I\ s;\ b\ s]\!] \Longrightarrow f\ s \leq SPEC\ I$

assumes *CONS*: $\bigwedge s.\ [\![\ I\ s;\ \neg\ b\ s\ ]\!] \Longrightarrow \Phi\ s$
shows *WHILEI I b f s* $\leq$ *SPEC* Φ
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*: $\bigwedge s.\ [\![\ I\ s;\ b\ s\ ]\!] \Longrightarrow f\ s \leq SPEC\ (\lambda s'.\ I\ s' \wedge (s',s) \in R)$
assumes *PHI*: $\bigwedge s.\ [\![\ I\ s;\ \neg\ b\ s\ ]\!] \Longrightarrow \Phi\ s$
shows *WHILEIT I b f s* $\leq$ *SPEC* Φ

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*: $\bigwedge s.\ [\![I\ s;\ b\ s]\!] \Longrightarrow f\ s \leq SPEC\ I$
assumes *CONS*: $\bigwedge s.\ [\![\ I\ s;\ \neg\ b\ s\ ]\!] \Longrightarrow \Phi\ s$

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*: ⋀*s*. ⟦ *I s*; *b s* ⟧ ⟹ *f s* ≤ *SPEC* (λ*s'*. *I s'* ∧ (*s'*,*s*)∈*R*)
assumes *PHI*: ⋀*s*. ⟦ *I s*; ¬ *b s* ⟧ ⟹ Φ *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-Leof Generic/RefineG-While
begin

definition *WHILEIT* (*WHILE*$_T$⁻) **where**
WHILEIT ≡ *iWHILEIT bind RETURN*
definition *WHILEI* (*WHILE*⁻) **where** *WHILEI* ≡ *iWHILEI bind RETURN*
definition *WHILET* (*WHILE*$_T$) **where** *WHILET* ≡ *iWHILET bind RETURN*
definition *WHILE* **where** *WHILE* ≡ *iWHILE 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 $I\ s \Longrightarrow M \leq \Downarrow R\ (WHILEI\ I\ b\ f\ s)$
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 $I\ s \Longrightarrow M \leq \Downarrow R\ (WHILEIT\ I\ b\ f\ s)$
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*: $\bigwedge W\ s\ s'.\ [\![\bigwedge s\ s'.\ (s,s')\in R \Longrightarrow W\ s \leq M\ s';\ (s,s')\in R]\!] \Longrightarrow$
$do\ \{ASSERT\ (I\ s);\ if\ b\ s\ then\ bind\ (f\ s)\ W\ else\ RETURN\ s\} \leq M\ s'$
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*: $I'\ s' \Longrightarrow (s,s')\in R'$
assumes *RI*: $\bigwedge s\ s'.\ [\![\ (s,s')\in R';\ I'\ s'\ ]\!] \Longrightarrow I\ s$
assumes *RB*: $\bigwedge s\ s'.\ [\![\ (s,s')\in R';\ I'\ s';\ I\ s;\ b'\ s'\ ]\!] \Longrightarrow b\ s$

assumes *RS*: $\bigwedge s\ s'.\ [\![\ (s,s')\in R';\ I'\ s';\ I\ s;\ b\ s\ ]\!]$
$\Longrightarrow f\ s \le sup\ (\Downarrow R'\ (do\ \{ASSUME\ (b'\ s');\ f'\ s'\}))\ (\Downarrow R'\ (RETURN\ s'))$
assumes *R-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ \neg b\ x;\ \neg b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow (x,x')\in R$
shows *WHILEI I b f s* $\le \Downarrow R$ (*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*=*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*: $I'\ x' \Longrightarrow (x,x')\in R'$
assumes *IREF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ I'\ x'\ ]\!] \Longrightarrow I\ x$
assumes *COND-REF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ b\ x;\ b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow f\ x \le \Downarrow R'\ (f'\ x')$
assumes *R-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ \neg b\ x;\ \neg b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow (x,x')\in R$
shows *WHILEI I b f x* $\le \Downarrow R$ (*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*: $\bigwedge s\ s'.\ [\![\ (s,s')\in R';\ b'\ s'\ ]\!] \Longrightarrow b\ s$

assumes *RS*: $\bigwedge s\ s'.\ [\![\ (s,s')\in R';\ b\ s\ ]\!]$
$\Longrightarrow f\ s \leq sup\ (\Downarrow R'\ (do\ \{ASSUME\ (b'\ s');\ f'\ s'\}))\ (\Downarrow R'\ (RETURN\ s'))$
assumes *R-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ \neg b\ x;\ \neg b'\ x'\ ]\!] \Longrightarrow (x,x')\in R$
shows *WHILE b f s* $\leq \Downarrow R$ (*WHILE b' f' s'*)
unfolding *WHILE-def*
apply (*rule WHILEI-invisible-refine-genR*)
apply (*rule assms*, (*assumption*+)*?*)+
done

lemma *WHILE-refine-genR*:
assumes *R0*: $(x,x')\in R'$
assumes *COND-REF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R'\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ b\ x;\ b'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R'\ (f'\ x')$
assumes *R-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ \neg b\ x;\ \neg b'\ x'\ ]\!] \Longrightarrow (x,x')\in R$
shows *WHILE b f x* $\leq \Downarrow R$ (*WHILE b' f' x'*)
unfolding *WHILE-def*
apply (*rule WHILEI-refine-genR*)
apply (*rule assms*, (*assumption*+)*?*)+
done

lemma *WHILE-refine-genR'*:
assumes *R0*: $(x,x')\in R'$
assumes *COND-REF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ I'\ x'\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ b\ x;\ b'\ x';\ I'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R'\ (f'\ x')$
assumes *R-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ \neg b\ x;\ \neg b'\ x'\ ]\!] \Longrightarrow (x,x')\in R$
shows *WHILE b f x* $\leq \Downarrow R$ (*WHILEI I' b' f' x'*)
unfolding *WHILE-def*
apply (*rule WHILEI-refine-genR*)
apply (*rule 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 *WHILEI-invisible-refine*:
assumes *R0*: $I'\ s' \Longrightarrow (s,s')\in R$
assumes *RI*: $\bigwedge s\ s'.\ [\![\ (s,s')\in R;\ I'\ s'\ ]\!] \Longrightarrow I\ s$
assumes *RB*: $\bigwedge s\ s'.\ [\![\ (s,s')\in R;\ I'\ s';\ I\ s;\ b'\ s'\ ]\!] \Longrightarrow b\ s$
assumes *RS*: $\bigwedge s\ s'.\ [\![\ (s,s')\in R;\ I'\ s';\ I\ s;\ b\ s\ ]\!]$
$\Longrightarrow f\ s \leq sup\ (\Downarrow R\ (do\ \{ASSUME\ (b'\ s');\ f'\ s'\}))\ (\Downarrow R\ (RETURN\ s'))$
shows *WHILEI I b f s* $\leq \Downarrow R$ (*WHILEI I' b' f' s'*)
apply (*rule WHILEI-invisible-refine-genR*[**where** *R'*=*R*])
apply (*rule assms*, (*assumption*+)*?*)+
done

lemma *WHILEI-refine*[*refine*]:
assumes *R0*: $I'\ x' \Longrightarrow (x,x')\in R$
assumes *IREF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I'\ x'\ ]\!] \Longrightarrow I\ x$
assumes *COND-REF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow f\ x \le \Downarrow R\ (f'\ x')$
shows *WHILEI I b f x* $\le \Downarrow R$ (*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*: $\bigwedge s\ s'.\ [\![\ (s,s')\in R;\ b'\ s'\ ]\!] \Longrightarrow b\ s$
assumes *RS*: $\bigwedge s\ s'.\ [\![\ (s,s')\in R;\ b\ s\ ]\!]$
$\Longrightarrow f\ s \le sup\ (\Downarrow R\ (do\ \{ASSUME\ (b'\ s');\ f'\ s'\}))\ (\Downarrow R\ (RETURN\ s'))$
shows *WHILE b f s* $\le \Downarrow R$ (*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*: $\bigwedge W\ s\ s'.\ [\![\bigwedge s\ s'.\ (s,s')\in R \Longrightarrow W\ s \le M\ s';\ (s,s')\in R]\!] \Longrightarrow$
do {*if b s then bind* (*f s*) *W else RETURN s*} $\le M\ s'$
shows *WHILE b f s* $\le M\ s'$
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*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x'\ ]\!] \Longrightarrow f\ x \le \Downarrow R\ (f'\ x')$
shows *WHILE b f x* $\le \Downarrow R$ (*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*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I'\ x'\ ]\!] \Longrightarrow b\ x = b'\ x'$

assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
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*: $[\![\Phi;\ \Phi \Longrightarrow S \leq S']\!] \Longrightarrow$ (*if* Φ *then S else FAIL*) $\leq S'$
by *auto*
lemma *ref-AIFI*: $[\![\Phi \Longrightarrow S \leq \Downarrow R\ S']\!] \Longrightarrow S \leq \Downarrow R$ (*if* Φ *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*: $I'\ x' \Longrightarrow (x,x')\in R'$
assumes *IREF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ I'\ x'\ ]\!] \Longrightarrow I\ x$
assumes *COND-REF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ b\ x;\ b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R'\ (f'\ x')$
assumes *R-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ \neg b\ x;\ \neg b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow (x,x')\in R$
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*+) []
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*: $\bigwedge x\ x'.\ (x,x')\in R' \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ b\ x;\ b'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R'\ (f'\ x')$
assumes *R-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ \neg b\ x;\ \neg b'\ x'\ ]\!] \Longrightarrow (x,x')\in R$

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*: $I'\ x' \Longrightarrow (x,x')\in R'$
assumes *COND-REF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ I'\ x'\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ b\ x;\ b'\ x';\ I'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R'\ (f'\ x')$
assumes *R-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ \neg b\ x;\ \neg b'\ x';\ I'\ x'\ ]\!] \Longrightarrow (x,x')\in R$
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*: $I'\ x' \Longrightarrow (x,x')\in R$
assumes *IREF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I'\ x'\ ]\!] \Longrightarrow I\ x$
assumes *COND-REF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
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*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
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*: $I'\ x' \Longrightarrow (x,x')\in R$
assumes *COND-REF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I'\ x'\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
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*: $I'\ x' \Longrightarrow (x,x')\in R$
assumes *INV0*: $[\![\ I'\ x';\ (x,x')\in R\ ]\!] \Longrightarrow I\ x$
assumes *COND-REF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow f\ x \le \Downarrow R\ (f'\ x')$
assumes *STEP-INV*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I\ x;\ I'\ x';\ nofail\ (f\ x)\ ]\!] \Longrightarrow f\ x \le SPEC\ I$
shows *WHILEI I b f x* $\le\Downarrow R$ (*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*)∈*R* ∧ *I c* }
])
using *assms*
by (*auto intro*: *add-invar-refineI*)

lemma *WHILEIT-refine-new-invar*:
assumes *R0*: $I'\ x' \Longrightarrow (x,x')\in R$
assumes *INV0*: $[\![\ I'\ x';\ (x,x')\in R\ ]\!] \Longrightarrow I\ x$
assumes *COND-REF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow f\ x \le \Downarrow R\ (f'\ x')$
assumes *STEP-INV*:
$\bigwedge x\ x'.\ [\![\ nofail\ (f\ x);\ (x,x')\in R;\ b\ x;\ b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow f\ x \le SPEC\ I$
shows *WHILEIT I b f x* $\le\Downarrow R$ (*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*)∈*R* ∧ *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 ≡ *OP* (*WHILEIT I*)
WHILEI I ≡ *OP* (*WHILEI I*)
by *auto*

lemma *autoref-WHILET*[*autoref-rules*]:
assumes $\bigwedge x\ x'.\ [\![\ (x,x')\in R]\!] \Longrightarrow (c\ x, c'\$x')\in Id$
assumes $\bigwedge x\ x'.\ [\![\ REMOVE\text{-}INTERNAL\ c'\ x';\ (x,x')\in R]\!]$

$\Longrightarrow (f\ x, f'\$x') \in \langle R\rangle nres\text{-}rel$
assumes $(s,s')\in R$
shows ($WHILET\ c\ f\ s$,
$(OP\ WHILET ::: (R\to Id)\to(R\to\langle R\rangle nres\text{-}rel)\to R\to\langle R\rangle nres\text{-}rel)\$c'\$f'\$s')$
$\in \langle R\rangle nres\text{-}rel$
using *assms*
by (*auto simp add*: *nres-rel-def fun-rel-def intro*!: *WHILET-refine*)

lemma *autoref-WHILEIT*[*autoref-rules*]:
assumes $\bigwedge x\ x'.\ [\![I\ x';\ (x,x')\in R]\!] \Longrightarrow (c\ x, c'\$x') \in Id$
assumes $\bigwedge x\ x'.\ [\![REMOVE\text{-}INTERNAL\ c'\ x';\ I\ x';\ (x,x')\in R]\!] \Longrightarrow (f\ x, f'\$x') \in \langle R\rangle nres\text{-}rel$
assumes $I\ s' \Longrightarrow (s,s')\in R$
shows ($WHILET\ c\ f\ s$,
$(OP\ (WHILEIT\ I) ::: (R\to Id)\to(R\to\langle R\rangle nres\text{-}rel)\to R\to\langle R\rangle nres\text{-}rel)\$c'\$f'\$s'$
$)\in\langle R\rangle nres\text{-}rel$
using *assms*
by (*auto simp add*: *nres-rel-def fun-rel-def intro*!: *WHILET-refine'*)

lemma *autoref-WHILEIT'*[*autoref-rules*]:
assumes $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I\ x']\!] \Longrightarrow (c\ x, c'\$x') \in Id$
assumes $\bigwedge x\ x'.\ [\![\ REMOVE\text{-}INTERNAL\ c'\ x';\ (x,x')\in R;\ I\ x']\!]$
$\Longrightarrow (f\ x, f'\$x') \in \langle R\rangle nres\text{-}rel$
shows ($WHILET\ c\ f$,
$(OP\ (WHILEIT\ I) ::: (R\to Id)\to(R\to\langle R\rangle nres\text{-}rel)\to R\to\langle R\rangle nres\text{-}rel)\$c'\$f'$
$)\in R\to\langle R\rangle nres\text{-}rel$
unfolding *autoref-tag-defs*
by (*parametricity*
add: *autoref-WHILEIT*[*unfolded autoref-tag-defs*]
assms[*unfolded autoref-tag-defs*])

lemma *autoref-WHILE*[*autoref-rules*]:
assumes $\bigwedge x\ x'.\ [\![\ (x,x')\in R]\!] \Longrightarrow (c\ x, c'\$x') \in Id$
assumes $\bigwedge x\ x'.\ [\![\ REMOVE\text{-}INTERNAL\ c'\ x';\ (x,x')\in R]\!]$
$\Longrightarrow (f\ x, f'\$x') \in \langle R\rangle nres\text{-}rel$
assumes $(s,s')\in R$
shows ($WHILE\ c\ f\ s$,
$(OP\ WHILE ::: (R\to Id)\to(R\to\langle R\rangle nres\text{-}rel)\to R\to\langle R\rangle nres\text{-}rel)\$c'\$f'\$s'$
$)\in\langle R\rangle nres\text{-}rel$
using *assms*
by (*auto simp add*: *nres-rel-def fun-rel-def intro*!: *WHILE-refine*)

lemma *autoref-WHILE'*[*autoref-rules*]:
assumes $\bigwedge x\ x'.\ [\![\ (x,x')\in R]\!] \Longrightarrow (c\ x, c'\$x') \in Id$
assumes $\bigwedge x\ x'.\ [\![\ REMOVE\text{-}INTERNAL\ c'\ x';\ (x,x')\in R]\!]$
$\Longrightarrow (f\ x, f'\$x') \in \langle R\rangle nres\text{-}rel$
shows ($WHILE\ c\ f$,
$(OP\ WHILE ::: (R\to Id)\to(R\to\langle R\rangle nres\text{-}rel)\to R\to\langle R\rangle nres\text{-}rel)\$c'\$f'$
$)\in R\to\langle R\rangle nres\text{-}rel$

using *assms*
by (*auto simp add*: *nres-rel-def fun-rel-def intro*!: *WHILE-refine*)

lemma *autoref-WHILEI*[*autoref-rules*]:
assumes $\bigwedge x\ x'.\ [\![I\ x';\ (x,x')\in R]\!] \Longrightarrow (c\ x, c'\$x') \in Id$
assumes $\bigwedge x\ x'.\ [\![REMOVE\text{-}INTERNAL\ c'\ x';\ I\ x';\ (x,x')\in R]\!] \Longrightarrow (f\ x, f'\$x') \in \langle R\rangle nres\text{-}rel$
assumes $I\ s' \Longrightarrow (s,s')\in R$
shows ($WHILE\ c\ f\ s$,
($OP\ (WHILEI\ I) ::: (R\to Id) \to (R\to\langle R\rangle nres\text{-}rel) \to R \to \langle R\rangle nres\text{-}rel)\$c'\$f'\$s'$
)$\in\langle R\rangle nres\text{-}rel$
using *assms*
by (*auto simp add*: *nres-rel-def fun-rel-def intro*!: *WHILE-refine'*)

lemma *autoref-WHILEI'*[*autoref-rules*]:
assumes $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I\ x']\!] \Longrightarrow (c\ x, c'\$x') \in Id$
assumes $\bigwedge x\ x'.\ [\![\ REMOVE\text{-}INTERNAL\ c'\ x';\ (x,x')\in R;\ I\ x']\!]$
$\Longrightarrow (f\ x, f'\$x') \in \langle R\rangle nres\text{-}rel$
shows ($WHILE\ c\ f$,
($OP\ (WHILEI\ I) ::: (R\to Id) \to (R\to\langle R\rangle nres\text{-}rel) \to R \to \langle R\rangle nres\text{-}rel)\$c'\$f'$
$)\in R \to \langle R\rangle nres\text{-}rel$
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 $RETURN\ s \le m$
shows $REC\ (\lambda D\ s.\ sup\ (a\ s \gg\!= D)\ (b\ s))\ s \le lfp\ (\lambda x.\ sup\ m\ (x \gg\!= a)) \gg\!= b$
(**is** $REC\ ?F\ s \le lfp\ ?f \gg\!= b$)
proof (*rule REC-rule*[**where** $pre = \lambda s.\ RETURN\ s \le lfp\ ?f$], *refine-mono*)
show $RETURN\ s \le lfp\ (\lambda x.\ sup\ m\ (x \gg\!= a))$
apply (*subst lfp-unfold*, *tagged-solver*)
using *assms* **apply** (*simp add*: *le-supI1*)
done
next
fix *f x*
assume *IH*: $\bigwedge x.\ RETURN\ x \le lfp\ ?f \Longrightarrow$
$f\ x \le lfp\ ?f \gg\!= b$

```
    and PRE: RETURN x ≤ lfp ?f

    show  sup (a x ⤜ f) (b x) ≤ lfp ?f ⤜ b
    proof (rule sup-least)
      show b x ≤ lfp ?f ⤜ b
      using PRE
        by (simp add: pw-le-iff refine-pw-simps) blast
    next
      from PRE have a x ≤ lfp ?f ⤜ a
        by (simp add: pw-le-iff refine-pw-simps) blast
      also have ... ≤ lfp ?f
        apply (subst (2) lfp-unfold, tagged-solver)
        apply (simp add: le-supI2)
        done
      finally show a x ⤜ f ≤ lfp ?f ⤜ b
        using IH
        by (simp add: pw-le-iff refine-pw-simps) blast
    qed
  qed

  private corollary tailrec-transform1:
    fixes m :: 'a nres
    shows m⤜REC (λD s. sup (a s ⤜ D) (b s)) ≤ lfp (λx. sup m (x⤜a)) ⤜
b
    apply (cases nofail m)
    apply (erule bind-le-nofailI)
    apply (erule tailrec-transform-aux1)
    apply (simp add: not-nofail-iff lfp-const)
    done


  private lemma tailrec-transform-aux2:
    fixes m :: 'a nres
    shows lfp (λx. sup m (x⤜a))
      ≤ m ⤜ REC (λD s. sup (a s ⤜ D) (RETURN s))
    (is lfp ?f ≤ m ⤜ 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 ((λx. x ⤜ a) ^^ i) m ≤ m ⤜ REC (λD s. sup (a s ⤜ D) (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 (λD s. sup (a s ⤜ D) (RETURN
s)) s ⤜ b
  ≤ REC (λD s. sup (a s ⤜ 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:
  lfp (λx. sup m (bind x a)) ⤜ b ≤ m ⤜ REC (λD s. sup (a s ⤜ D) (b s))
proof −
  from bind-mono(1)[OF tailrec-transform-aux2 order-refl]
  have lfp (λx. sup m (bind x a)) ⤜ b
    ≤ m ⤜ (λx. REC (λD s. sup (a s ⤜ D) (RETURN s)) x ⤜ b)
    by simp
  also from bind-mono(1)[OF order-refl tailrec-transform-aux3]
  have ... ≤ m ⤜ REC (λD s. sup (a s ⤜ D) (b s)) .
  finally show ?thesis .
qed

definition tailrec-body a b ≡ (λD s. sup (bind (a s) D) (b s))

theorem tailrec-transform:
  m ⤜ REC (λD s. sup (a s ⤜ D) (b s)) = lfp (λx. sup m (bind x a)) ⤜ b
  apply (rule antisym)
  apply (rule tailrec-transform1)
  apply (rule tailrec-transform2)
  done

theorem tailrec-transform':
```

```
  m ≫= REC (tailrec-body a b) = lfp (λx. sup m (bind x a)) ≫= b
  unfolding tailrec-body-def
  by (rule tailrec-transform)

lemma WHILE c f =
  REC (tailrec-body
    (λs. do {ASSUME (c s); f s})
    (λs. do {ASSUME (¬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
    (λs. do {ASSERT (I s); ASSUME (c s); f s})
    (λs. do {ASSERT (I s); ASSUME (¬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
    (λs. do {ASSERT (I s); ASSUME (c s); f s})
    (λs. do {ASSERT (I s); ASSUME (¬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 ≡
  (λx. sup m (do {
     s ← x;
     - ← ASSERT (I s);
     - ← ASSUME (c s);
     f s
   }))

lemma WHILEI-lfp-conv: m ≫= WHILEI I c f =
  do {
    s ← lfp (WHILEI-lfp-body I m c f);
    ASSERT (I s);
    ASSUME (¬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* ≡ *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* ≡ *do* {*x*←*m*; *ASSUME* (*c x*); *RETURN x*}
definition *filter-ASSERT c m* ≡ *do* {*x*←*m*; *ASSERT* (*c x*); *RETURN x*}

lemma [*refine-pw-simps*]: *nofail* (*filter-ASSUME c m*) ⟷ *nofail m*
unfolding *filter-ASSUME-def*
by (*simp add*: *refine-pw-simps*)

lemma [*refine-pw-simps*]: *inres* (*filter-ASSUME c m*) *x*
⟷ (*nofail m* ⟶ *inres m x* ∧ *c x*)
unfolding *filter-ASSUME-def*
by (*simp add*: *refine-pw-simps*)

lemma *msii-is-invar*:
m ≤ *msii I m c f*
m ≤ *msii I m c f* ⟹ *bind* (*filter-ASSUME c* (*filter-ASSERT I m*)) *f* ≤ *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* ⤜ *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*: $\bigwedge m.\ [\![m \leq msii\ I\ m0\ c\ f;\ m \leq P;$
$filter\text{-}ASSUME\ c\ (filter\text{-}ASSERT\ I\ m) \gg\!= f \leq msii\ I\ m0\ c\ f$
$]\!] \Longrightarrow filter\text{-}ASSUME\ c\ (filter\text{-}ASSERT\ I\ m) \gg\!= f \leq P$
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: $[\![\ m0{=}RES\ X;\ x{\in}X\ ]\!] \Longrightarrow rwof\ m0\ cond\ step\ x$
| *step*: $[\![\ rwof\ m0\ cond\ step\ x;\ cond\ x;\ step\ x = RES\ Y;\ y{\in}Y\ ]\!]$
$\Longrightarrow rwof\ m0\ cond\ step\ y$

lemma *establish-rwof-invar*:
assumes *I*: $m0 \leq_n SPEC\ I$
assumes *S*: $\bigwedge s.\ [\![\ rwof\ m0\ cond\ step\ s;\ I\ s;\ cond\ s\ ]\!]$
$\Longrightarrow step\ s \leq_n SPEC\ I$
assumes *rwof m0 cond step s*
shows *I s*
using *assms*(*3*)
apply *induct*
using *I* **apply** *auto* []
using *S* **apply** *fastforce* []
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*: $\bigwedge s.\ [\![\ rwof\ m0\ cond\ step\ s;\ I\ s;\ cond\ s\ ]\!]$
$\Longrightarrow step\ s \leq_n SPEC\ I$
shows *is-rwof-invar m0 cond step I*
using *assms* **unfolding** *is-rwof-invar-def* **by** *blast*

lemma *rwof-cons*: ⟦*is-rwof-invar m0 cond step I*; *rwof m0 cond step s*⟧ ⟹ *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* ≤ *SPEC I*
assumes *S*: ⋀*s*. ⟦ *rwof m0 cond step s*; *I s*; *cond s* ⟧ ⟹ *step s* ≤ *SPEC I*
shows *m0* ≫= *WHILE cond step* ≤ *SPEC* (λ*s*. *rwof m0 cond step s* ∧ ¬*cond s* ∧ *I s*)
proof −
from *I0* **obtain** *M0* **where** [*simp*]: *m0* = *RES M0* **and** *M0* ⊆ *Collect I*
by (*cases m0*) *auto*

show *?thesis*
apply *simp*
apply *refine-vcg*
apply (*rule WHILE-rule*[**where** *I*=λ*s*. *I s* ∧ *rwof m0 cond step s*])
proof −
fix *s* **assume** *s*∈*M0* **thus** *I s* ∧ *rwof m0 cond step s*
using *I0* **by** (*auto intro*: *rwof.init*) []

next
fix *s*
assume *A*: *I s* ∧ *rwof m0 cond step s* **and** *C*: *cond s*
hence *step s* ≤ *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* ≤ *SPEC* (*rwof m0 cond step*) **by** *auto*
finally (*SPEC-rule-conjI*)
show *step s* ≤ *SPEC* (λ*s*. *I s* ∧ *rwof m0 cond step s*) **.**
qed *auto*
qed

Filtering out states that satisfy the loop condition

definition *filter-nb* :: (*'a* ⇒ *bool*) ⇒ *'a nres* ⇒ *'a nres* **where**
filter-nb b I ≡ *do* {*s*←*I*; *ASSUME* (¬*b s*); *RETURN s*}

lemma *pw-filter-nb*[*refine-pw-simps*]:
nofail (*filter-nb b I*) ⟷ *nofail I*
inres (*filter-nb b I*) *x* ⟷ (*nofail I* ⟶ *inres I x* ∧ ¬*b x*)
unfolding *filter-nb-def*
by (*simp-all add*: *refine-pw-simps*)

lemma *filter-nb-mono*: *m*≤*m'* ⟹ *filter-nb cond m* ≤ *filter-nb cond m'*

unfolding *filter-nb-def*
by *refine-mono*

lemma *filter-nb-cont*:
filter-nb cond (*Sup M*) = *Sup* {*filter-nb cond m* | *m*. *m* ∈ *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-nb cond FAIL* = *FAIL*
by (*simp add*: *filter-nb-def*)

lemma *filter-nb-RES*[*simp*]: *filter-nb cond* (*RES X*) = *RES* {*x*∈*X*. ¬*cond x*}
by (*simp add*: *pw-eq-iff refine-pw-simps*)

Bounded while-loop

lemma *WHILE-rule-gen-le*:
assumes *I0*: *m0* ≤ *I*
assumes *ISTEP*: ⋀*s*. ⟦*RETURN s* ≤ *I*; *b s*⟧ ⟹ *f s* ≤ *I*
shows *m0* ≫= *WHILE b f* ≤ *filter-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*=λ*s*. *RETURN s* ≤ *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* ⇒ (*'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 ← *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 ≫= WHILE cond step = do {*
x ← m;
if cond x then step x ≫= WHILE cond step
else RETURN x
}
by (*subst WHILE-unfold*[*abs-def*]) (*rule refl*)

lemma *WHILE-bounded-aux1*:
filter-nb cond (*bounded-WHILE n cond step m*) ≤ *m ≫= 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 ≫= WHILE cond step
≤ *filter-nb cond* (*Sup* {*bounded-WHILE n cond step m* | *n. 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 *RETURN s* ≤ *Sup* {*bounded-WHILE n cond step m* |*n. True*}
then obtain *n* **where** *RETURN s* ≤ *bounded-WHILE n cond step m*
by (*fold inres-def*) (*auto simp*: *refine-pw-simps*)
moreover assume *cond s*
ultimately have *step s* ≤ *bounded-WHILE* (*Suc n*) *cond step m*
by (*simp add*: *pw-le-iff refine-pw-simps*, *blast*)
thus *step s* ≤ *Sup* {*bounded-WHILE n cond step m* |*n. True*}
by (*metis* (*mono-tags*, *lifting*) *Sup-upper2 mem-Collect-eq*)
qed

lemma *WHILE-bounded*:
m ≫= 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 ∃ n. RETURN s ≤ (bounded-WHILE n cond step m0)
  using assms
  apply induction
  apply (rule-tac x=0 in exI)
  apply simp

  apply clarsimp
  apply (rule-tac x=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 ≠ FAIL
  shows ∃ n'<n. ∃ x X.
      bounded-WHILE n' cond step m0 = RES X
    ∧ x∈X ∧ cond x ∧ step x = FAIL
  using assms
proof (induction n)
  case 0 thus ?case by simp
next
  case (Suc n)
  assume bounded-WHILE (Suc n) cond step m0 = FAIL
  hence bounded-WHILE n cond step m0 = FAIL
    ∨ (∃ X x. bounded-WHILE n cond step m0 = RES X
        ∧ x∈X ∧ cond x ∧ step x = FAIL)
    (is ?C1 ∨ ?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 = RES X ∧ x∈X ∧ cond x ∧ step x =
```

```
FAIL
      by blast
    hence 2: n' < 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 = RES X
  assumes x∈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.prems(1) obtain Xh where
    BWN: bounded-WHILE n cond step m0 = RES Xh
    and ∀ x∈Xh. cond x ⟶ nofail (step x)
    and X = {x'. ∃ x∈Xh. cond x ∧ inres (step x) x'} ∪ {x. x∈Xh ∧ ¬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.prems(2) obtain xh X' where
      xh∈Xh and
    C: cond xh ⟶ step xh = RES X' ∧ x∈X' and NC: ¬cond xh ⟶ 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 ‹xh∈Xh›])
        apply assumption
        apply (simp-all add: C) [2]

      apply (rule Suc.IH[OF BWN])
      apply (simp add: NC ‹xh∈Xh›)
    done
qed


lemma rwof-FAIL-imp-WHILE-FAIL:
```

assumes *RW*: *rwof m0 cond step s*
and *C*: *cond s*
and *S*: *step s = FAIL*
shows *m0 ≫= WHILE cond step = FAIL*
proof −
from *rwof-in-bounded-WHILE*[*OF RW*] **obtain** *n* **where**
RETURN s ≤ bounded-WHILE n cond step m0 **..**
with *C* **have** *step s ≤ bounded-WHILE* (*Suc n*) *cond step m0*
by (*auto simp add*: *pw-le-iff refine-pw-simps*)
with *S* **have** *bounded-WHILE* (*Suc n*) *cond step m0 = FAIL* **by** *simp*
with *WHILE-bounded-aux1*[*of cond Suc n step m0*] **show** *?thesis*
by (*simp del*: *bounded-WHILE.simps*)
qed

lemma *pw-bounded-WHILE-RES-rwof*: ⟦ *nofail* (*bounded-WHILE n cond step m0*);
inres (*bounded-WHILE n cond step m0*) *x* ⟧ ⟹ *rwof m0 cond step x*
using *bounded-WHILE-RES-rwof*[*of n cond step m0 - x*]
by (*auto simp add*: *pw-eq-iff*)

corollary *WHILE-nofail-imp-rwof-nofail*:
assumes *nofail* (*m0 ≫= WHILE cond step*)
assumes *RW*: *rwof m0 cond step s*
assumes *C*: *cond s*
shows *nofail* (*step s*)
apply (*rule ccontr*) **apply** (*simp add*: *nofail-def*)
using *assms rwof-FAIL-imp-WHILE-FAIL*[*OF RW C*]
by *auto*

lemma *WHILE-le-WHILEI*: *WHILE b f s ≤ WHILEI I b f s*
unfolding *WHILE-def*
by (*rule WHILEI-weaken*) *simp*

corollary *WHILEI-nofail-imp-rwof-nofail*:
assumes *NF*: *nofail* (*m0 ≫= WHILEI I cond step*)
assumes *RW*: *rwof m0 cond step s*
assumes *C*: *cond s*
shows *nofail* (*step s*)
proof −
from *NF* **have** *nofail* (*m0 ≫= WHILE cond step*)
using *WHILE-le-WHILEI*
by (*fastforce simp*: *pw-le-iff refine-pw-simps*)
from *WHILE-nofail-imp-rwof-nofail*[*OF this RW C*] **show** *?thesis* **.**
qed

corollary *WHILET-nofail-imp-rwof-nofail*:
assumes *NF*: *nofail* (*m0 ≫= WHILET cond step*)
assumes *RW*: *rwof m0 cond step s*

assumes *C*: *cond s*
shows *nofail* (*step s*)
proof −
from *NF* **have** *nofail* (*m0* ⤜ *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* ⤜ *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* ⤜ *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 ⟹ ∃ *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* ⤜ *WHILE cond step*)
shows *m0* ⤜ *WHILE cond step* = *SPEC* (λ*s*. *rwof m0 cond step s* ∧ ¬*cond s*)
proof −
from *NF WHILE-bounded*[*of m0 cond step*] **have** *NF′*:
nofail (*Sup* {*filter-nb cond m* |*m*. ∃ *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*. ∃ *n*. *m* = *bounded-WHILE n cond step m0*}
≤ *SPEC* (λ*s*. *rwof m0 cond step s* ∧ ¬ *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* (λ*s*. *rwof m0 cond step s* ∧ ¬ *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* ⤜ *WHILE cond step*)
assumes *R*: *rwof m0 cond step s*
assumes *C*: *cond s*
shows *inres* (*step s*) *s'* ⟹ *rwof m0 cond step s'*
and *nofail* (*step s*)
using *WHILE-nofail-imp-rwof-nofail*[*OF NF R C*] *pw-rwof-step'*[*OF - assms*(2−)]
by *simp-all*

lemma *rwof-step*:
⟦ *nofail* (*m0* ⤜ *WHILE cond step*); *rwof m0 cond step s*; *cond s* ⟧
⟹ *step s* ≤ *SPEC* (*rwof m0 cond step*)
using *pw-rwof-step*[*of m0 cond step s*]
by (*simp add*: *pw-le-iff refine-pw-simps*)

lemma (**in** −) *rwof-leof-init*: *m* ≤$_n$ *SPEC* (*rwof m c f*)
apply *rule*
using *rwof.init*
apply (*fastforce simp*: *nofail-RES-conv*)
done

lemma (**in** −) *rwof-leof-step*: ⟦*rwof m c f s*; *c s*⟧ ⟹ *f s* ≤$_n$ *SPEC* (*rwof m c f*)
apply *rule*
using *rwof.step*
apply (*fastforce simp*: *nofail-RES-conv*)
done

lemma (**in** −) *rwof-step-refine*:
assumes *NF*: *nofail* (*m0* ⤜ *WHILE cond step*)
assumes *A*: *rwof m0 cond step' s*
assumes *FR*: ⋀*s*. ⟦ *rwof m0 cond step s*; *cond s* ⟧ ⟹ *step' s* ≤$_n$ *step s*
shows *rwof m0 cond 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* ⤜ *WHILE c f* = *m* ⤜ *WHILEI* (*rwof m c f*) *c f*
proof (*rule antisym*)
have *m* ⤜ *WHILEI* (*rwof m c f*) *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* ≤$_n$ ⇓*R m′*
assumes *C*: ⋀*s s′*. ⟦(*s*,*s′*)∈*R*; *rwof m c f s*; *rwof m′ c′ f′ s′*⟧ ⟹ *c s* = *c′ s′*
assumes *S*: ⋀*s s′*. ⟦(*s*,*s′*)∈*R*; *rwof m c f s*; *rwof m′ c′ f′ s′*; *c s*; *c′ s′*⟧ ⟹ *f s* ≤$_n$ ⇓*R* (*f′ s′*)
shows ∃ *s′*. (*s*,*s′*)∈*R* ∧ *rwof m′ 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*
≡ {(*s*,*s′*). *rwof init cond step s* ∧ *cond s* ∧ *inres* (*step s*) *s′*}

lemma *rwof-rel-spec*: ⟦*rwof init cond step s*; *cond s*⟧
⟹ *step s* ≤$_n$ *SPEC* (λ*s′*. (*s*,*s′*)∈*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 ∃ *s0*. *inres init s0* ∧ (*s0*,*s*)∈(*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: ⋀ i. (f i, f (Suc i)) ∈ rwof-rel init cond 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) ∈ (rwof-rel init cond step)*     inres init s0
sn = f 0
    using rwof-reachable by metis
  then obtain f where P: ⋀ i. (f i, f (Suc i)) ∈ rwof-rel init cond 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    ⋀ i. (?f i, ?f (Suc i)) ∈ rwof-rel init cond step
      using step by (auto split: nat.splits)
    then show ?case using step by blast
  qed auto

  from P have [simp]: ⋀i. cond (f i)
    unfolding rwof-rel-def by auto

  from P have SIR: ⋀i. inres (step (f i)) (f (Suc i))
    unfolding rwof-rel-def by auto

  define F where F = (WHILEI-body (≫=) RETURN I cond step)

  {
    assume M: trimono F
    define f' where f' x = (if x∈range f then FAIL else gfp F x) for x

    have f' ≤ F f'
      unfolding f'-def
      apply (rule le-funI)
      apply (case-tac x∈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 ⋀x. f' x ≤ 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: ⋀i s. c (i,s) ⟷ i<u
  assumes F0: F {} s0 = s0
  assumes Fs: ⋀s i X. ⟦l≤i; i<u⟧
    ⟹ f (i, (F X s)) ≤ SPEC (λ(i',r). i'=i+1 ∧ r=F (insert (list!i) X) s)
  shows WHILET c f (l,s0)
    ≤ SPEC (λ(-,r). r=F {list!i | i. l≤i ∧ i<u} s0)
  apply (cases ¬(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 = measure (λ(i,-). u−i)
    and I = λ(i,s). l≤i ∧ i≤u ∧ s = F {list!j | j. l≤j ∧ 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* ≡ *dFAILi*
definition *bot-dres* ≡ *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 |
  inf - dSUCCEEDi = dSUCCEEDi

lemma inf-dres-addsimps[simp]:
  inf x dSUCCEEDi = dSUCCEEDi
  inf dSUCCEEDi x = dSUCCEEDi
  inf x dFAILi = x
  inf (dRETURN v) x ≠ dFAILi
  apply (case-tac [!] x)
  apply simp-all
  done

definition Sup-dres S ≡
  if S⊆{dSUCCEEDi} then dSUCCEEDi
  else if dFAILi∈S then dFAILi
  else if ∃ a b. a≠b ∧ dRETURN a∈S ∧ dRETURN b∈S then dFAILi
  else dRETURN (THE x. dRETURN x ∈ S)

definition Inf-dres S ≡
  if S⊆{dFAILi} then dFAILi
  else if dSUCCEEDi∈S then dSUCCEEDi
  else if ∃ a b. a≠b ∧ dRETURN a∈S ∧ dRETURN b∈S then dSUCCEEDi
  else dRETURN (THE x. dRETURN x ∈ S)

fun less-eq-dres where
  less-eq-dres dSUCCEEDi - ⟷ True |
  less-eq-dres - dFAILi ⟷ True |
  less-eq-dres (dRETURN (a::'a)) (dRETURN b) ⟷ a=b |
  less-eq-dres - - ⟷ False

definition less-dres where less-dres (a::'a dres) b ⟷ a≤b ∧ ¬ b≤a

lemma less-eq-dres-split-conv:
  a≤b ⟷ (case (a,b) of
      (dSUCCEEDi,-) ⇒ True
    | (-,dFAILi) ⇒ True
    | (dRETURN (a::'a), dRETURN b) ⇒ a=b
    | - ⇒ False
  )
  by (auto split: dres.split)

lemma inf-dres-split-conv:
  inf a b = (case (a,b) of
    (dFAILi,x) ⇒ x
  | (x,dFAILi) ⇒ x
  | (dRETURN a, dRETURN b) ⇒ (if a=b then dRETURN b else dSUCCEEDi)
  | - ⇒ 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 ≡ (bot::'a dres)
abbreviation dFAIL ≡ (top::'a dres)

lemma dres-cases[cases type, case-names dSUCCEED dRETURN dFAIL]:
 obtains x=dSUCCEED | r where x=dRETURN r |   x=dFAIL
 unfolding bot-dres-def top-dres-def by (cases x) auto

lemmas [simp] = dres.case(1,2)[folded top-dres-def bot-dres-def]

lemma dres-order-simps[simp]:
 x≤dSUCCEED ⟷ x=dSUCCEED
 dFAIL≤x ⟷ x=dFAIL
 dRETURN r ≠ dFAIL
 dRETURN r ≠ dSUCCEED
 dFAIL ≠ dRETURN r
 dSUCCEED ≠ dRETURN r
 dFAIL≠dSUCCEED
 dSUCCEED≠dFAIL
 x=y ⟹ inf (dRETURN x) (dRETURN y) = dRETURN y
 x≠y ⟹ inf (dRETURN x) (dRETURN y) = dSUCCEED
 x=y ⟹ sup (dRETURN x) (dRETURN y) = dRETURN y
 x≠y ⟹ 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-Sup-cases:
 obtains S⊆{dSUCCEED} and Sup S = dSUCCEED
 | dFAIL∈S and Sup S = dFAIL
 | a b where a≠b   dRETURN a∈S   dRETURN b∈S   dFAIL∉S   Sup S =
dFAIL
 | a where S ⊆ {dSUCCEED, dRETURN a}   dRETURN a∈S   Sup S =
dRETURN a
proof −
 show ?thesis
  apply (cases S⊆{dSUCCEED})
  apply (rule that(1), assumption)
  apply (simp add: Sup-dres-def bot-dres-def)

  apply (cases dFAIL∈S)
```

```
    apply (rule that(2), assumption)
    apply (simp add: Sup-dres-def bot-dres-def top-dres-def)

    apply (cases ∃ a b. a≠b ∧ dRETURN a∈S ∧ dRETURN b∈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 ∃ a. dRETURN a ∈ 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⊆{dFAIL} and Inf S = dFAIL
  | dSUCCEED∈S and Inf S = dSUCCEED
  | a b where a≠b    dRETURN a∈S    dRETURN b∈S    dSUCCEED∉S    Inf
S = dSUCCEED
  | a where S ⊆ {dFAIL, dRETURN a}    dRETURN a∈S    Inf S = dRETURN
a
proof −
  show ?thesis
    apply (cases S⊆{dFAIL})
    apply (rule that(1), assumption)
    apply (simp add: Inf-dres-def top-dres-def)

    apply (cases dSUCCEED∈S)
    apply (rule that(2), assumption)
    apply (simp add: Inf-dres-def bot-dres-def top-dres-def)

    apply (cases ∃ a b. a≠b ∧ dRETURN a∈S ∧ dRETURN b∈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 ∃ a. dRETURN a ∈ 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 ⟹
dRETURN r ∈ *M* ⟹ *dRETURN s* ∈ *M* ⟹ *r*=*s*
by (*metis chainD less-eq-dres.simps*(*4*))

lemma *dres-Sup-chain-cases*:
assumes *CHAIN*: *is-chain M*
obtains *M* ⊆ {*dSUCCEED*} *Sup M* = *dSUCCEED*
| *r* **where** *M* ⊆ {*dSUCCEED*,*dRETURN r*} *dRETURN r*∈*M* *Sup M* = *dRETURN r*
| *dFAIL*∈*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* ⊆ {*dFAIL*} *Inf M* = *dFAIL*
| *r* **where** *M* ⊆ {*dFAIL*,*dRETURN r*} *dRETURN r*∈*M* *Inf M* = *dRETURN r*
| *dSUCCEED*∈*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*

adhoc-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* (λ*x*. *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*]:
⟦ *M* ≤ *M'*; ⋀*x*. *dRETURN x* ≤ *M* ⟹ *f x* ≤ *f' x* ⟧ ⟹ *dbind M f* ≤ *dbind M' f'*
⟦ *flat-ge M M'*; ⋀*x*. *flat-ge* (*f x*) (*f' x*) ⟧ ⟹ *flat-ge* (*dbind M f*) (*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*: ⋀*s*. *mono* (*B s*)
shows *mono* (λ*F s*. *dbind* (*A F s*) (λ*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*]) []
apply (*auto dest*: *monoD*[*OF MB*, *THEN le-funD*]) []
done

lemma *dr-mono-bind'*: *mono* (λ*F s*. *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*]:
dbind dSUCCEED f = *dSUCCEED*
dbind dFAIL f = *dFAIL*
by (*simp-all*)

definition *dASSERT* ≡ *iASSERT dRETURN*
definition *dASSUME* ≡ *iASSUME dRETURN*
interpretation *dres-assert*: *generic-Assert dbind dRETURN dASSERT dASSUME*
apply *unfold-locales*
by (*auto simp*: *dASSERT-def dASSUME-def*)

definition *dWHILEIT* ≡ *iWHILEIT dbind dRETURN*
definition *dWHILEI* ≡ *iWHILEI dbind dRETURN*
definition *dWHILET* ≡ *iWHILET dbind dRETURN*
definition *dWHILE* ≡ *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* ≠ *dSUCCEED*

lemma *dres-ne-bot-basic*[*refine-transfer*]:
dFAIL ≠ *dSUCCEED*
⋀*x*. *dRETURN x* ≠ *dSUCCEED*
⋀*m f*. ⟦ *m*≠*dSUCCEED*; ⋀*x*. *f x* ≠ *dSUCCEED* ⟧ ⟹ *dbind m f* ≠ *dSUCCEED*
⋀Φ. *dASSERT* Φ ≠ *dSUCCEED*
⋀*b m1 m2*. ⟦ *m1*≠*dSUCCEED*; *m2*≠*dSUCCEED* ⟧ ⟹ *If b m1 m2* ≠ *dSUCCEED*
⋀*x f*. ⟦ ⋀*x*. *f x*≠*dSUCCEED* ⟧ ⟹ *Let x f* ≠ *dSUCCEED*

⋀*g p*. ⟦ ⋀*x1 x2*. *g x1 x2* ≠ *dSUCCEED* ⟧ ⟹ *case-prod g p* ≠ *dSUCCEED*
⋀*fn fs x*.
⟦ *fn*≠*dSUCCEED*; ⋀*v*. *fs v* ≠ *dSUCCEED* ⟧ ⟹ *case-option fn fs x* ≠ *dSUCCEED*
⋀*fn fc x*.
⟦ *fn*≠*dSUCCEED*; ⋀*x xs*. *fc x xs* ≠ *dSUCCEED* ⟧ ⟹ *case-list fn fc x* ≠ *dSUCCEED*
apply (*auto split*: *prod.split option.split list.split*)
apply (*case-tac m*, *auto*) []
apply (*case-tac* Φ, *auto*) []
done

lemma *dres-ne-bot-RECT*[*refine-transfer*]:
assumes *A*: ⋀*f x*. ⟦ ⋀*x*. *f x* ≠ *dSUCCEED* ⟧ ⟹ *B f x* ≠ *dSUCCEED*
shows *RECT B x* ≠ *dSUCCEED*
unfolding *RECT-def*
apply (*split if-split*)
apply (*intro impI conjI*)
apply *simp-all*

apply (*rule flatf-fp-induct-pointwise*[**where** *pre*=λ- -. *True* **and** *B*=*B* **and** *b*=*top* **and** *post*=λ- - *m*. *m*≠*dSUCCEED*])
apply (*simp-all add*: *trimonoD A*)
done

lemma *dres-ne-bot-dWHILEIT*[*refine-transfer*]:
assumes ⋀*x*. *f x* ≠ *dSUCCEED*
shows *dWHILEIT I b f s* ≠ *dSUCCEED* **using** *assms*
unfolding *dWHILEIT-def iWHILEIT-def WHILEI-body-def*
apply *refine-transfer*
done

lemma *dres-ne-bot-dWHILET*[*refine-transfer*]:
assumes ⋀*x*. *f x* ≠ *dSUCCEED*
shows *dWHILET b f s* ≠ *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* $(\lambda f::'a \Rightarrow 'b\ nres.$
$(\forall x0.\ f\ x0 \leq SPEC\ (P\ x0)))$
apply (*rule ccpo.admissibleI*)
apply (*unfold fun-lub-def*)
apply (*intro allI impI*)
apply (*rule Sup-least*)
apply *auto*
done

declaration ‹*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*)›

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;
$\bigwedge y.$ *monotone* (*fun-ord* $(\leq)$) $(\leq)$ $(\lambda f.\ C\ y\ f)$ $]\!] \Longrightarrow$
monotone (*fun-ord* $(\leq)$) $(\leq)$ $(\lambda f.\ bind\ (B\ f)\ (\lambda y.\ C\ y\ f))$
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* $(\lambda f::'a \Rightarrow 'b\ dres.$
$(\forall x.\ P\ x \longrightarrow$
$(f\ x \neq dFAIL \wedge$
$(\forall r.\ f\ x = dRETURN\ r \longrightarrow Q\ x\ r))))$
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]: ⋀A x. {y. ∃f∈A. y = f x} = (λf. 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 ‹Partial-Function.init drec @{term drec.fixp-fun}
  @{term drec.mono-body} @{thm drec.fixp-rule-uc} @{thm drec.fixp-induct-uc}
  NONE›

lemma drec-bind-mono-pfun[partial-function-mono]:
  fixes C :: 'a ⇒ ('b ⇒ 'c dres) ⇒ ('d dres)
  shows
  ⟦ monotone (fun-ord (≤)) (≤) B;
    ⋀y. monotone (fun-ord (≤)) (≤) (λf. C y f) ⟧ ⟹
     monotone (fun-ord (≤)) (≤) (λf. dbind (B f) (λy. 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 ≡ case r of*
dSUCCEEDi ⇒ SUCCEED
| *dFAILi ⇒ FAIL*
| *dRETURN x ⇒ 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 ≤ nres-of b ⟷ a≤b
nres-of a < nres-of b ⟷ a<b
is-chain A ⟹ nres-of (Sup A) = Sup (nres-of‘A)
is-chain A ⟹ 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 ⋀*x*. *nres-of* (*f x*) ≤ *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*]:
⟦*RETURN x* ≤ *M*; ⋀*x*. *RETURN* (*f x*) ≤ *F x*⟧ ⟹ *RETURN* (*Let x f*) ≤ *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 *RETURN* (*the-res S*) ≤ *S′*. Here, *the-res* extracts the result from a deterministic monad. As *the-res* is executable, the above shows that (*the-res S*) is always a correct result.

fun *the-res* **where** *the-res* (*dRETURN x*) = *x*

The following lemma converts a proof-obligation with result extraction to a transfer proof obligation, and a proof obligation that the program yields not bottom.

Note that this rule has to be applied manually, as, otherwise, it would interfere with the default setup, that tries to generate a plain function.

lemma *the-resI*:
assumes *nres-of S* ≤ *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* ≡ *x*
lemma *detTAGI*: *x* = *detTAG x* **unfolding** *detTAG-def* **by** *simp*
lemma *autoref-detI*:
assumes (*b*,*a*)∈⟨*R*⟩*nres-rel*
assumes *RETURN c* ≤ *b*
assumes *c* = *detTAG d*
shows (*RETURN d*, *a*)∈⟨*R*⟩*nres-rel*
using *assms*
unfolding *nres-rel-def detTAG-def*
by *simp*

2.14.4 Relator-Based Transfer

definition *dres-nres-rel-internal-def*:
dres-nres-rel R ≡ {(*c*,*a*). *nres-of c* ≤ ⇓ *R a*}

lemma *dres-nres-rel-def*: ⟨*R*⟩*dres-nres-rel* ≡ {(*c*,*a*). *nres-of c* ≤ ⇓ *R a*}
by (*simp add*: *dres-nres-rel-internal-def relAPP-def*)

lemma *dres-nres-relI*[*intro?*]: *nres-of c* ≤ ⇓ *R a* ⟹ (*c*,*a*)∈⟨*R*⟩*dres-nres-rel*
by (*simp add*: *dres-nres-rel-def*)

lemma *dres-nres-relD*: (*c*,*a*)∈⟨*R*⟩*dres-nres-rel* ⟹ *nres-of c* ≤ ⇓ *R a*
by (*simp add*: *dres-nres-rel-def*)

lemma *dres-nres-rel-as-br-conv*:
⟨*R*⟩*dres-nres-rel* = *br nres-of* (λ-. *True*) *O* ⟨*R*⟩*nres-rel*
unfolding *dres-nres-rel-def br-def nres-rel-def* **by** *auto*

definition *plain-nres-rel-internal-def*:
plain-nres-rel R ≡ {(*c*,*a*). *RETURN c* ≤ ⇓ *R a*}

lemma *plain-nres-rel-def*: ⟨*R*⟩*plain-nres-rel* ≡ {(*c*,*a*). *RETURN c* ≤ ⇓ *R a*}
by (*simp add*: *plain-nres-rel-internal-def relAPP-def*)

lemma *plain-nres-relI*[*intro?*]: *RETURN c* ≤ ⇓ *R a* ⟹ (*c*,*a*)∈⟨*R*⟩*plain-nres-rel*
by (*simp add*: *plain-nres-rel-def*)

lemma *plain-nres-relD*: (*c*,*a*)∈⟨*R*⟩*plain-nres-rel* ⟹ *RETURN c* ≤ ⇓ *R a*
by (*simp add*: *plain-nres-rel-def*)

lemma *plain-nres-rel-as-br-conv*:
⟨*R*⟩*plain-nres-rel* = *br RETURN* (λ-. *True*) *O* ⟨*R*⟩*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 ⊆ *S* ⟹ *x*∈*it* ⟹ *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* ≡ λ(*xs*, *σ*). *do* {
let x = *hd xs*; *σ′*←*f x σ*; *RETURN* (*tl xs*,*σ′*)
}

definition *FOREACH-cond* **where** *FOREACH-cond c* ≡ (λ(*xs*,*σ*). *xs*≠[] ∧ *c σ*)

Foreach with continuation condition, order and annotated invariant:

definition *FOREACHoci* (*FOREACH*$_{OC}$$^{\text{-,-}}$) **where** *FOREACHoci R* Φ *S c f σ0* ≡ *do* {
ASSERT (*finite S*);
xs ← *SPEC* (λ*xs*. *distinct xs* ∧ *S* = *set xs* ∧ *sorted-wrt R xs*);
(-,*σ*) ← *WHILEIT*
(λ(*it*,*σ*). ∃ *xs′*. *xs* = *xs′* @ *it* ∧ Φ (*set it*) *σ*) (*FOREACH-cond c*) (*FOREACH-body f*) (*xs*,*σ0*);
RETURN σ }

Foreach with continuation condition and annotated invariant:

definition *FOREACHci* (*FOREACH*$_{C}$$^{\text{-}}$) **where** *FOREACHci* ≡ *FOREACHoci* (λ- -. *True*)

Foreach with continuation condition:

definition *FOREACHc* (*FOREACH*$_{C}$) **where** *FOREACHc* ≡ *FOREACHci* (λ- -. *True*)

Foreach with annotated invariant:

definition *FOREACHi* (*FOREACH*$^{\text{-}}$) **where**
FOREACHi Φ *S* ≡ *FOREACHci* Φ *S* (λ-. *True*)

Foreach with annotated invariant and order:

definition *FOREACHoi* (*FOREACH*$_{O}$$^{\text{-,-}}$) **where**
FOREACHoi R Φ *S* ≡ *FOREACHoci R* Φ *S* (λ-. *True*)

Basic foreach

definition *FOREACH S* ≡ *FOREACHc S* (λ-. *True*)

lemmas *FOREACH-to-oci-unfold*
= *FOREACHci-def FOREACHc-def FOREACHi-def FOREACHoi-def FOREACH-def*

2.15.3 Proof Rules

lemma *FOREACHoci-rule*[*refine-vcg*]:
assumes *FIN*: *finite S*
assumes *I0*: *I S σ0*

```
  assumes IP:
    ⋀x it σ. ⟦ c σ; x∈it; it⊆S; I it σ; ∀y∈it − {x}. R x y;
              ∀y∈S − it. R y x ⟧ ⟹ f x σ ≤ SPEC (I (it−{x}))
  assumes II1: ⋀σ. ⟦I {} σ⟧ ⟹ P σ
  assumes II2: ⋀it σ. ⟦ it≠{}; it⊆S; I it σ; ¬c σ;
                        ∀x∈it. ∀y∈S − it. R y x ⟧ ⟹ P σ
  shows FOREACHoci R I S c f σ0 ≤ SPEC P
  unfolding FOREACHoci-def
  apply (intro refine-vcg)

  apply (rule FIN)

  apply (subgoal-tac wf (measure (λ(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' σ xs)
defer
  apply (simp, elim conjE exE)+
  apply (rename-tac x s xs' σ xs)
defer
proof −
  fix xs' σ xs

  assume I-xs': I (set xs') σ
     and sorted-xs-xs': sorted-wrt R (xs @ xs')
     and dist: distinct xs    distinct xs'    set xs ∩ set xs' = {}
     and S-eq: S = set xs ∪ set xs'

  from S-eq have set xs' ⊆ S by simp
  from dist S-eq have S-diff: S − set xs' = set xs by blast

  { assume xs' ≠ []    c σ
    from ‹xs' ≠ []› obtain x xs'' where xs'-eq: xs' = x # xs'' by (cases xs', auto)

    have x-in-xs': x ∈ set xs' and x-nin-xs'': x ∉ set xs''
      using ‹distinct xs'› unfolding xs'-eq by simp-all

    from IP[of σ x set xs', OF ‹c σ› x-in-xs' ‹set xs' ⊆ S› ‹I (set xs') σ›] x-nin-xs''
        sorted-xs-xs' S-diff
    show f (hd xs') σ ≤ SPEC
          (λx. (∃xs'a. xs @ xs' = xs'a @ tl xs') ∧
               I (set (tl xs')) x)
      apply (simp add: xs'-eq)
      apply (simp add: sorted-wrt-append)
```

```
    done
  }

  { assume xs' = [] ∨ ¬(c σ)
    show P σ
    proof (cases xs' = [])
      case True thus P σ using ‹I (set xs') σ› by (simp add: II1)
    next
      case False note xs'-neq-nil = this
      with ‹xs' = [] ∨ ¬ c σ› have ¬ c σ by simp

      from II2 [of set xs' σ] S-diff sorted-xs-xs'
      show P σ
        apply (simp add: xs'-neq-nil S-eq ‹¬ c σ› I-xs')
        apply (simp add: sorted-wrt-append)
      done
    qed
  }
qed

lemma FOREACHoi-rule[refine-vcg]:
  assumes FIN: finite S
  assumes I0: I S σ0
  assumes IP:
    ⋀x it σ. ⟦ x∈it; it⊆S; I it σ; ∀y∈it − {x}. R x y;
               ∀y∈S − it. R y x ⟧ ⟹ f x σ ≤ SPEC (I (it−{x}))
  assumes II1: ⋀σ. ⟦I {} σ⟧ ⟹ P σ
  shows FOREACHoi R I S f σ0 ≤ 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 σ0
  assumes IP:
    ⋀x it σ. ⟦ x∈it; it⊆S; I it σ; c σ ⟧ ⟹ f x σ ≤ SPEC (I (it−{x}))
  assumes II1: ⋀σ. ⟦I {} σ⟧ ⟹ P σ
  assumes II2: ⋀it σ. ⟦ it≠{}; it⊆S; I it σ; ¬c σ ⟧ ⟹ P σ
  shows FOREACHci I S c f σ0 ≤ 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 α :: 'S ⇒ 'Sa — Abstraction mapping of elements
  fixes S :: 'S 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 *RR-OK*: $\bigwedge x\ y.\ [\![x \in S;\ y \in S;\ RR\ x\ y]\!] \Longrightarrow RR'\ (\alpha\ x)\ (\alpha\ y)$
assumes *REF0*: $((S,\sigma 0),(\alpha`S,\sigma 0')) \in R$
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it \subseteq S;\ it' \subseteq S';\ \Phi'\ it'\ \sigma';\ \Phi\ it\ \sigma;$
$\forall x \in S-it.\ \forall y \in it.\ RR\ x\ y;\ \forall x \in S'-it'.\ \forall y \in it'.\ RR'\ x\ y;$
$it' = \alpha`it;\ ((it,\sigma),(it',\sigma')) \in R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it \subseteq S;\ it' \subseteq S';\ \Phi'\ it'\ \sigma';$
$\forall x \in S-it.\ \forall y \in it.\ RR\ x\ y;\ \forall x \in S'-it'.\ \forall y \in it'.\ RR'\ x\ y;$
$it' = \alpha`it;\ ((it,\sigma),(it',\sigma')) \in R$
$]\!] \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$it \subseteq S;\ it' \subseteq S';\ \Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';$
$\forall x \in S-it.\ \forall y \in it.\ RR\ x\ y;\ \forall x \in S'-it'.\ \forall y \in it'.\ RR'\ x\ y;$
$x' = \alpha\ x;\ it' = \alpha`it;\ ((it,\sigma),(it',\sigma')) \in R;$
$x \in it;\ \forall y \in it-\{x\}.\ RR\ x\ y;$
$x' \in it';\ \forall y' \in it'-\{x'\}.\ RR'\ x'\ y';$
$c\ \sigma;\ c'\ \sigma'$
$]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\ \sigma').\ ((it-\{x\},\sigma),(it'-\{x'\},\sigma')) \in R\})\ (f'\ x'\ \sigma')$
assumes *REF-R-DONE*: $\bigwedge \sigma\ \sigma'.\ [\![\ \Phi\ \{\}\ \sigma;\ \Phi'\ \{\}\ \sigma';\ ((\{\},\sigma),(\{\},\sigma')) \in R\ ]\!]$
$\Longrightarrow (\sigma,\sigma') \in R'$
assumes *REF-R-BRK*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it \subseteq S;\ it' \subseteq S';\ \Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';$
$\forall x \in S-it.\ \forall y \in it.\ RR\ x\ y;\ \forall x \in S'-it'.\ \forall y \in it'.\ RR'\ x\ y;$
$it' = \alpha`it;\ ((it,\sigma),(it',\sigma')) \in R;$
$it \neq \{\};\ it' \neq \{\};$
$\neg c\ \sigma;\ \neg c'\ \sigma'$
$]\!] \Longrightarrow (\sigma,\sigma') \in R'$
shows $FOREACHoci\ RR\ \Phi\ S\ c\ f\ \sigma 0 \leq \Downarrow R'\ (FOREACHoci\ RR'\ \Phi'\ S'\ c'\ f'\ \sigma 0')$

supply [[*simproc del*: *defined-all*]]
unfolding *FOREACHoci-def*
apply (*refine-rcg WHILEIT-refine-genR*[**where**
$R' = \{((xs,\sigma),(xs',\sigma'))\ .$
$xs' = map\ \alpha\ xs \wedge$
$set\ xs \subseteq S \wedge set\ xs' \subseteq S' \wedge$
$(\forall x \in S - set\ xs.\ \forall y \in set\ xs.\ RR\ x\ y) \wedge$
$(\forall x \in S' - set\ xs'.\ \forall y \in set\ xs'.\ RR'\ x\ y) \wedge$
$((set\ xs,\sigma),(set\ xs',\sigma')) \in R\ \}$
])

using *REFS INJ* **apply** (*auto dest*: *finite-imageD*) []
apply (*rule intro-prgR*[**where** *R*={(*xs*,*xs'*) . *xs'*=*map* α *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*=α,*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-tl*)
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 *α* :: *'S* ⇒ *'Sa*
fixes *S* :: *'S set*
fixes *S'* :: *'Sa set*
assumes *INJ*: *inj-on α S*
assumes *REFS*: *S'* = *α*‘*S*
assumes *REF0*: (*σ*0,*σ*0')∈*R*
assumes *RR-OK*: ⋀*x y*. ⟦*x* ∈ *S*; *y* ∈ *S*; *RR x y*⟧ ⟹ *RR'* (*α x*) (*α y*)
assumes *REFPHI0*: Φ'' *S σ*0 (*α*‘*S*) *σ*0'
assumes *REFC*: ⋀*it σ it' σ'*. ⟦

$it'{=}\alpha`it$; $it{\subseteq}S$; $it'{\subseteq}S'$; $\Phi'\ it'\ \sigma'$; $\Phi''\ it\ \sigma\ it'\ \sigma'$; $\Phi\ it\ \sigma$; $(\sigma,\sigma')\in R$
$\rrbracket \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'{=}\alpha`it$; $it{\subseteq}S$; $it'{\subseteq}S'$; $\Phi'\ it'\ \sigma'$; $\Phi''\ it\ \sigma\ it'\ \sigma'$; $(\sigma,\sigma')\in R$
$\rrbracket \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![\forall y{\in}it{-}\{x\}.\ RR\ x\ y$;
$x'{=}\alpha\ x$; $x{\in}it$; $x'{\in}it'$; $it'{=}\alpha`it$; $it{\subseteq}S$; $it'{\subseteq}S'$;
$\Phi\ it\ \sigma$; $\Phi'\ it'\ \sigma'$; $\Phi''\ it\ \sigma\ it'\ \sigma'$; $c\ \sigma$; $c'\ \sigma'$;
$(\sigma,\sigma')\in R$
$\rrbracket \Longrightarrow f\ x\ \sigma$
$\le \Downarrow(\{(\sigma,\sigma').\ (\sigma,\sigma')\in R \wedge \Phi''\ (it-\{x\})\ \sigma\ (it'-\{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows *FOREACHoci RR* Φ *S c f* $\sigma 0 \le \Downarrow R$ (*FOREACHoci RR'* Φ' *S' c' f'* $\sigma 0'$)

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* []
using *REFPHI* **apply** *auto* []
using *REFSTEP* **apply** *auto* []
apply *auto* []
apply *auto* []
done

lemma *FOREACHoci-refine-rcg*[*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 *RR-OK*: $\bigwedge x\ y.\ [\![x \in S;\ y \in S;\ RR\ x\ y]\!] \Longrightarrow RR'\ (\alpha\ x)\ (\alpha\ y)$
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'{=}\alpha`it$; $it{\subseteq}S$; $it'{\subseteq}S'$; $\Phi'\ it'\ \sigma'$; $\Phi\ it\ \sigma$; $(\sigma,\sigma')\in R$
$\rrbracket \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'{=}\alpha`it$; $it{\subseteq}S$; $it'{\subseteq}S'$; $\Phi'\ it'\ \sigma'$; $(\sigma,\sigma')\in R$
$\rrbracket \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![\ \forall y{\in}it{-}\{x\}.\ RR\ x\ y$;
$x'{=}\alpha\ x$; $x{\in}it$; $x'{\in}it'$; $it'{=}\alpha`it$; $it{\subseteq}S$; $it'{\subseteq}S'$;
$\Phi\ it\ \sigma$; $\Phi'\ it'\ \sigma'$; $c\ \sigma$; $c'\ \sigma'$;
$(\sigma,\sigma')\in R$
$\rrbracket \Longrightarrow f\ x\ \sigma \le \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACHoci RR* Φ *S c f* $\sigma 0 \le \Downarrow R$ (*FOREACHoci RR'* Φ' *S' c' f'* $\sigma 0'$)
apply (*rule FOREACHoci-refine*[**where** $\Phi''{=}\lambda$- - - -. *True*])
apply (*rule assms*)+

using *assms* **by** *simp-all*

lemma *FOREACHoci-weaken*:
assumes *IREF*: $\bigwedge it\ \sigma.\ it{\subseteq}S \Longrightarrow I\ it\ \sigma \Longrightarrow I'\ it\ \sigma$
shows $FOREACHoci\ RR\ I'\ S\ c\ f\ \sigma 0 \leq FOREACHoci\ RR\ I\ S\ c\ f\ \sigma 0$
apply (*rule FOREACHoci-refine-rcg*[**where** α=*id* **and** R=*Id*, *simplified*])
apply (*auto intro*: *IREF*)
done

lemma *FOREACHoci-weaken-order*:
assumes *RRREF*: $\bigwedge x\ y.\ x \in S \Longrightarrow y \in S \Longrightarrow RR\ x\ y \Longrightarrow RR'\ x\ y$
shows $FOREACHoci\ RR\ I\ S\ c\ f\ \sigma 0 \leq FOREACHoci\ RR'\ I\ S\ c\ f\ \sigma 0$
apply (*rule FOREACHoci-refine-rcg*[**where** α=*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 $S :: 'S\ set$ — 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 *RR-OK*: $\bigwedge x\ y.\ [\![x \in S;\ y \in S;\ RR\ x\ y]\!] \Longrightarrow RR'\ (\alpha\ x)\ (\alpha\ y)$
assumes *REF0*: $((S,\sigma 0),(\alpha `S,\sigma 0')) \in R$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';\ \Phi'\ it'\ \sigma';$
$\forall x{\in}S{-}it.\ \forall y{\in}it.\ RR\ x\ y;\ \forall x{\in}S'{-}it'.\ \forall y{\in}it'.\ RR'\ x\ y;$
$it'{=}\alpha `it;\ ((it,\sigma),(it',\sigma'))\in R$
$]\!] \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';\ \Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';$
$\forall x{\in}S{-}it.\ \forall y{\in}it.\ RR\ x\ y;\ \forall x{\in}S'{-}it'.\ \forall y{\in}it'.\ RR'\ x\ y;$
$x'{=}\alpha\ x;\ it'{=}\alpha `it;\ ((it,\sigma),(it',\sigma'))\in R;$
$x{\in}it;\ \forall y{\in}it{-}\{x\}.\ RR\ x\ y;$
$x'{\in}it';\ \forall y'{\in}it'{-}\{x'\}.\ RR'\ x'\ y'$
$]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\ \sigma').\ ((it{-}\{x\},\sigma),(it'{-}\{x'\},\sigma'))\in R\})\ (f'\ x'\ \sigma')$
assumes *REF-R-DONE*: $\bigwedge\sigma\ \sigma'.\ [\![\ \Phi\ \{\}\ \sigma;\ \Phi'\ \{\}\ \sigma';\ ((\{\},\sigma),(\{\},\sigma'))\in R\ ]\!]$
$\Longrightarrow (\sigma,\sigma')\in R'$
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* []

apply (*fact* | *simp*)+
done

lemma *FOREACHoi-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 *RR-OK*: $\bigwedge x\ y.\ [\![x \in S;\ y \in S;\ RR\ x\ y]\!] \Longrightarrow RR'\ (\alpha\ x)\ (\alpha\ y)$
assumes *REFPHI0*: $\Phi''\ S\ \sigma 0\ (\alpha`S)\ \sigma 0'$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';\ \Phi'\ it'\ \sigma';\ \Phi''\ it\ \sigma\ it'\ \sigma';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![\forall y\in it-\{x\}.\ RR\ x\ y;$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ \Phi''\ it\ \sigma\ it'\ \sigma';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma$
$\le \Downarrow(\{(\sigma,\sigma').\ (\sigma,\sigma')\in R \wedge \Phi''\ (it-\{x\})\ \sigma\ (it'-\{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows *FOREACHoi RR* $\Phi\ S\ f\ \sigma 0 \le \Downarrow R\ (FOREACHoi\ RR'\ \Phi'\ S'\ f'\ \sigma 0')$
unfolding *FOREACHoi-def*
apply (*rule FOREACHoci-refine* [*of* α - - - - - - - Φ'])
apply (*simp-all add*: *assms*)
done

lemma *FOREACHoi-refine-rcg*[*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 *RR-OK*: $\bigwedge x\ y.\ [\![x \in S;\ y \in S;\ RR\ x\ y]\!] \Longrightarrow RR'\ (\alpha\ x)\ (\alpha\ y)$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';\ \Phi'\ it'\ \sigma';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![\forall y\in it-\{x\}.\ RR\ x\ y;$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \le \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACHoi RR* $\Phi\ S\ f\ \sigma 0 \le \Downarrow R\ (FOREACHoi\ RR'\ \Phi'\ S'\ f'\ \sigma 0')$
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 $S :: 'S\ set$ — 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*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';\ \Phi'\ it'\ \sigma';\ \Phi\ it\ \sigma;$
$it'{=}\alpha`it;\ ((it,\sigma),(it',\sigma'))\in R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';\ \Phi'\ it'\ \sigma';$
$it'{=}\alpha`it;\ ((it,\sigma),(it',\sigma'))\in R$
$]\!] \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';\ \Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';$
$x'{=}\alpha\ x;\ it'{=}\alpha`it;\ ((it,\sigma),(it',\sigma'))\in R;$
$x{\in}it;\ x'{\in}it';$
$c\ \sigma;\ c'\ \sigma'$
$]\!] \Longrightarrow f\ x\ \sigma$
$\le \Downarrow(\{(\sigma,\ \sigma').\ ((it{-}\{x\},\sigma),(it'{-}\{x'\},\sigma'))\in R\})\ (f'\ x'\ \sigma')$
assumes *REF-R-DONE*: $\bigwedge\sigma\ \sigma'.\ [\![\ \Phi\ \{\}\ \sigma;\ \Phi'\ \{\}\ \sigma';\ ((\{\},\sigma),(\{\},\sigma'))\in R\ ]\!]$
$\Longrightarrow (\sigma,\sigma')\in R'$
assumes *REF-R-BRK*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';\ \Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';$
$it'{=}\alpha`it;\ ((it,\sigma),(it',\sigma'))\in R;$
$it{\neq}\{\};\ it'{\neq}\{\};$
$\neg c\ \sigma;\ \neg c'\ \sigma'$
$]\!] \Longrightarrow (\sigma,\sigma')\in R'$
shows *FOREACHci* $\Phi\ S\ c\ f\ \sigma 0 \le \Downarrow R'\ (FOREACHci\ \Phi'\ S'\ c'\ f'\ \sigma 0')$
unfolding *FOREACHci-def*
apply (*rule FOREACHoci-refine-genR*)
apply (*fact*|*simp*)+
using *REFC* **apply** *auto* []
using *REFPHI* **apply** *auto* []
using *REFSTEP* **apply** *auto* []
apply (*fact*|*simp*)+
using *REF-R-BRK* **apply** *auto* []
done

lemma *FOREACHci-refine*:
fixes α :: *'S* ⇒ *'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*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';\ \Phi'\ it'\ \sigma';\ \ \Phi''\ it\ \sigma\ it'\ \sigma';\ \Phi\ it\ \sigma;\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';\ \Phi'\ it'\ \sigma';\ \Phi''\ it\ \sigma\ it'\ \sigma';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ \ \Phi''\ it\ \sigma\ it'\ \sigma';\ c\ \sigma;\ c'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma$
$\le \Downarrow(\{(\sigma,\ \sigma').\ (\sigma,\ \sigma')\in R\wedge \Phi''\ (it-\{x\})\ \sigma\ (it'-\{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows *FOREACHci* $\Phi\ S\ c\ f\ \sigma 0 \le \Downarrow R\ (FOREACHci\ \Phi'\ S'\ c'\ f'\ \sigma 0')$
unfolding *FOREACHci-def*
apply (*rule FOREACHoci-refine* [*of* α - - - - - - - Φ''])
apply (*simp-all add*: *assms*)
done

lemma *FOREACHci-refine-rcg*[*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 *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';\ \Phi'\ it'\ \sigma';\ \Phi\ it\ \sigma;\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';\ \Phi'\ it'\ \sigma';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ c\ \sigma;\ c'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \le \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACHci* $\Phi\ S\ c\ f\ \sigma 0 \le \Downarrow R\ (FOREACHci\ \Phi'\ S'\ c'\ f'\ \sigma 0')$
apply (*rule FOREACHci-refine*[**where** $\Phi''=\lambda$- - - -. *True*])
apply (*rule assms*)+
using *assms* **by** *auto*

lemma *FOREACHci-weaken*:
assumes *IREF*: $\bigwedge it\ \sigma.\ it\subseteq S \Longrightarrow I\ it\ \sigma \Longrightarrow I'\ it\ \sigma$
shows *FOREACHci* $I'\ S\ c\ f\ \sigma 0 \le FOREACHci\ I\ S\ c\ f\ \sigma 0$
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 σ0*
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ [\![\ x{\in}it;\ it{\subseteq}S;\ I\ it\ \sigma\ ]\!] \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it{-}\{x\}))$
assumes *II*: $\bigwedge\sigma.\ [\![I\ \{\}\ \sigma]\!] \Longrightarrow P\ \sigma$
shows *FOREACHi I S f σ0* $\leq$ *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 σ0*
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ [\![\ x{\in}it;\ it{\subseteq}S;\ I\ it\ \sigma;\ c\ \sigma\ ]\!] \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it{-}\{x\}))$
assumes *II1*: $\bigwedge\sigma.\ [\![I\ \{\}\ \sigma]\!] \Longrightarrow P\ \sigma$
assumes *II2*: $\bigwedge it\ \sigma.\ [\![\ it{\neq}\{\};\ it{\subseteq}S;\ I\ it\ \sigma;\ \neg c\ \sigma\ ]\!] \Longrightarrow P\ \sigma$
shows *FOREACHc S c f σ0* $\leq$ *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 σ0*
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ [\![\ x{\in}it;\ it{\subseteq}S;\ I\ it\ \sigma\ ]\!] \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it{-}\{x\}))$
assumes *II*: $\bigwedge\sigma.\ [\![I\ \{\}\ \sigma]\!] \Longrightarrow P\ \sigma$
shows *FOREACH S f σ0* $\leq$ *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 α :: *'S* $\Rightarrow$ *'Sa* — Abstraction mapping of elements
fixes *S* :: *'S set* — Concrete set
fixes *S'* :: *'Sa set* — Abstract set
fixes *σ0* :: *'σ*
fixes *σ0'* :: *'σa*
fixes *R* :: (*('S set* × *'σ)* × *('Sa set* × *'σa)) set*
assumes *INJ*: *inj-on α S*
assumes *REFS*[*simp*]: *S'* = *α*‘*S*
assumes *REF0*: *((S,σ0),(α*‘*S,σ0'*)) ∈ *R*
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
it⊆*S*; *it'*⊆*S'*;
it'=*α*‘*it*; *((it,σ),(it',σ'))*∈*R*

⟧ ⟹ $c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'$. ⟦
$it \subseteq S$; $it' \subseteq S'$;
$x' = \alpha\ x$; $it' = \alpha `it$; $((it,\sigma),(it',\sigma')) \in R$;
$x \in it$; $x' \in it'$;
$c\ \sigma$; $c'\ \sigma'$
⟧ ⟹ $f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\ \sigma').\ ((it - \{x\},\sigma),(it' - \{x'\},\sigma')) \in R\})\ (f'\ x'\ \sigma')$
assumes *REF-R-DONE*: $\bigwedge \sigma\ \sigma'$. ⟦ $((\{\},\sigma),(\{\},\sigma')) \in R$ ⟧
⟹ $(\sigma,\sigma') \in R'$
assumes *REF-R-BRK*: $\bigwedge it\ \sigma\ it'\ \sigma'$. ⟦
$it \subseteq S$; $it' \subseteq S'$;
$it' = \alpha `it$; $((it,\sigma),(it',\sigma')) \in R$;
$it \neq \{\}$; $it' \neq \{\}$;
$\neg c\ \sigma$; $\neg c'\ \sigma'$
⟧ ⟹ $(\sigma,\sigma') \in R'$
shows *FOREACHc S c f* $\sigma 0 \leq \Downarrow R'$ (*FOREACHc S' c' f'* $\sigma 0'$)
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 $S :: 'S\ set$
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*: $\bigwedge it\ \sigma\ it'\ \sigma'$. ⟦
$it' = \alpha `it$; $it \subseteq S$; $it' \subseteq S'$; $\Phi''\ it\ \sigma\ it'\ \sigma'$; $(\sigma,\sigma') \in R$
⟧ ⟹ $c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'$. ⟦
$x' = \alpha\ x$; $x \in it$; $x' \in it'$; $it' = \alpha `it$; $it \subseteq S$; $it' \subseteq S'$;
$\Phi''\ it\ \sigma\ it'\ \sigma'$; $c\ \sigma$; $c'\ \sigma'$; $(\sigma,\sigma') \in R$
⟧ ⟹ $f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\ \sigma').\ (\sigma,\ \sigma') \in R \wedge \Phi''\ (it - \{x\})\ \sigma\ (it' - \{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows *FOREACHc S c f* $\sigma 0 \leq \Downarrow R$ (*FOREACHc S' c' f'* $\sigma 0'$)
unfolding *FOREACHc-def*
apply (*rule FOREACHci-refine*[**where** $\Phi''=\Phi''$, *OF INJ REFS REF0 REF-PHI0*])
apply (*erule* (4) *REFC*)
apply (*rule TrueI*)

apply (*erule* (*9*) *REFSTEP*)
done

lemma *FOREACHc-refine-rcg*[*refine*]:
fixes α :: *'S* ⇒ *'Sa*
fixes *S* :: *'S set*
fixes *S'* :: *'Sa set*
assumes *INJ*: *inj-on* α *S*
assumes *REFS*: *S'* = α‘*S*
assumes *REF0*: (σ*0*,σ*0'*)∈*R*
assumes *REFC*: ⋀*it* σ *it'* σ'. ⟦
it'=α‘*it*; *it*⊆*S*; *it'*⊆*S'*; (σ,σ')∈*R*
⟧ ⟹ *c* σ ⟷ *c'* σ'
assumes *REFSTEP*: ⋀*x it* σ *x' it'* σ'. ⟦
x'=α *x*; *x*∈*it*; *x'*∈*it'*; *it'*=α‘*it*; *it*⊆*S*; *it'*⊆*S'*; *c* σ; *c'* σ';
(σ,σ')∈*R*
⟧ ⟹ *f x* σ ≤ ⇓*R* (*f' x'* σ')
shows *FOREACHc S c f* σ*0* ≤ ⇓*R* (*FOREACHc S' c' f'* σ*0'*)
unfolding *FOREACHc-def*
apply (*rule FOREACHci-refine-rcg*)
apply (*rule assms*)+
using *assms* **by** *auto*

lemma *FOREACHi-refine-genR*:
fixes α :: *'S* ⇒ *'Sa* — Abstraction mapping of elements
fixes *S* :: *'S set* — Concrete set
fixes *S'* :: *'Sa set* — Abstract set
fixes σ*0* :: *'*σ
fixes σ*0'* :: *'*σ*a*
fixes *R* :: ((*'S set* × *'*σ) × (*'Sa set* × *'*σ*a*)) *set*
assumes *INJ*: *inj-on* α *S*
assumes *REFS*[*simp*]: *S'* = α‘*S*
assumes *REF0*: ((*S*,σ*0*),(α‘*S*,σ*0'*)) ∈ *R*
assumes *REFPHI*: ⋀*it* σ *it'* σ'. ⟦
it⊆*S*; *it'*⊆*S'*; Φ' *it'* σ';
it'=α‘*it*; ((*it*,σ),(*it'*,σ'))∈*R*
⟧ ⟹ Φ *it* σ
assumes *REFSTEP*: ⋀*x it* σ *x' it'* σ'. ⟦
it⊆*S*; *it'*⊆*S'*; Φ *it* σ; Φ' *it'* σ';
x'=α *x*; *it'*=α‘*it*; ((*it*,σ),(*it'*,σ'))∈*R*;
x∈*it*; *x'*∈*it'*
⟧ ⟹ *f x* σ
≤ ⇓({(σ, σ'). ((*it*−{*x*},σ),(*it'*−{*x'*},σ'))∈*R*}) (*f' x'* σ')
assumes *REF-R-DONE*: ⋀σ σ'. ⟦ Φ {} σ; Φ' {} σ'; (({},σ),({},σ'))∈*R* ⟧
⟹ (σ,σ')∈*R'*
shows *FOREACHi* Φ *S f* σ*0* ≤ ⇓*R'* (*FOREACHi* Φ' *S' f'* σ*0'*)
unfolding *FOREACHi-def*
apply (*rule FOREACHci-refine-genR*)
apply (*fact*|*simp*)+

using *REFSTEP* **apply** *auto* []
apply (*fact*|*simp*)+
done

lemma *FOREACHi-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 *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';\ \Phi'\ it'\ \sigma';\ \Phi''\ it\ \sigma\ it'\ \sigma';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ \Phi''\ it\ \sigma\ it'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma$
$\le \Downarrow(\{(\sigma,\sigma').\ (\sigma,\sigma')\in R\wedge\Phi''\ (it-\{x\})\ \sigma\ (it'-\{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows *FOREACHi* $\Phi\ S\ f\ \sigma 0 \le \Downarrow R\ (FOREACHi\ \Phi'\ S'\ f'\ \sigma 0')$
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 $S :: 'S\ set$
fixes $S' :: 'Sa\ set$
assumes *INJ*: *inj-on* $\alpha\ S$
assumes *REFS*: $S' = \alpha`S$
assumes *REF0*: $(\sigma 0,\sigma 0')\in R$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';\ \Phi'\ it'\ \sigma';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \le \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACHi* $\Phi\ S\ f\ \sigma 0 \le \Downarrow R\ (FOREACHi\ \Phi'\ S'\ f'\ \sigma 0')$
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 $S :: 'S\ set$ — 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 *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$it \subseteq S;\ it' \subseteq S';$
$x' = \alpha\ x;\ it' = \alpha `it;\ ((it,\sigma),(it',\sigma')) \in R;$
$x \in it;\ x' \in it'$
$]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\ \sigma').\ ((it - \{x\},\sigma),(it' - \{x'\},\sigma')) \in R\})\ (f'\ x'\ \sigma')$
assumes *REF-R-DONE*: $\bigwedge \sigma\ \sigma'.\ [\![\ ((\{\},\sigma),(\{\},\sigma')) \in R\ ]\!]$
$\Longrightarrow (\sigma,\sigma') \in R'$
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* []
apply (*fact*|*simp*)+
done

lemma *FOREACH-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 *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x' = \alpha\ x;\ x \in it;\ x' \in it';\ it' = \alpha `it;\ it \subseteq S;\ it' \subseteq S';$
$\Phi''\ it\ \sigma\ it'\ \sigma';\ (\sigma,\sigma') \in R$
$]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\ \sigma').\ (\sigma,\ \sigma') \in R \wedge \Phi''\ (it - \{x\})\ \sigma\ (it' - \{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows $FOREACH\ S\ f\ \sigma 0 \leq \Downarrow R\ (FOREACH\ S'\ f'\ \sigma 0')$
unfolding *FOREACH-def*
apply (*rule FOREACHc-refine*[**where** $\Phi''=\Phi''$, *OF INJ REFS REF0 REFPHI0*])
apply (*rule refl*)
apply (*erule* (7) *REFSTEP*)
done

lemma *FOREACH-refine-rcg*[*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 *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \leq \Downarrow R\ (f'\ x'\ \sigma')$
shows $FOREACH\ S\ f\ \sigma 0 \leq \Downarrow R\ (FOREACH\ S'\ f'\ \sigma 0')$
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 $S :: 'S\ set$
fixes $S' :: 'Sa\ set$
assumes *INJ*: *inj-on* $\alpha\ S$
assumes *REFS*: $S' = \alpha`S$
assumes *REF0*: $(\sigma 0,\sigma 0')\in R$
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';\ \Phi'\ it'\ \sigma';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi'\ it'\ \sigma';\ c\ \sigma;\ c'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \leq \Downarrow R\ (f'\ x'\ \sigma')$
shows $FOREACHc\ S\ c\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHci\ \Phi'\ S'\ c'\ f'\ \sigma 0')$
unfolding *FOREACHc-def*
apply (*rule FOREACHci-refine-rcg*)
apply (*rule assms*)
apply (*rule assms*)
apply (*rule assms*)
apply (*erule* (4) *REFC*)
apply (*rule TrueI*)
apply (*rule REFSTEP*, *assumption*+)
done

lemma *FOREACHi-refine-rcg'*[*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 *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi'\ it'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \le \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACH S f* $\sigma 0 \le \Downarrow R$ (*FOREACHi* $\Phi'\ S'\ f'\ \sigma 0'$)
unfolding *FOREACH-def FOREACHi-def*
apply (*rule FOREACHci-refine-rcg'*)
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*:
$\bigwedge x\ it\ \sigma.\ [\![\ c\ \sigma;\ x\in it;\ it\subseteq S;\ I\ it\ \sigma;\ \forall y\in it - \{x\}.\ R\ x\ y;$
$\forall y\in S - it.\ R\ y\ x\ ]\!] \Longrightarrow f\ x\ \sigma \le SPEC\ (I\ (it-\{x\}))$
assumes *II1*: $\bigwedge\sigma.\ [\![I\ \{\}\ \sigma;\ c\ \sigma]\!] \Longrightarrow P\ \sigma$
assumes *II2*: $\bigwedge it\ \sigma.\ [\![\ it\subseteq S;\ I\ it\ \sigma;\ \neg c\ \sigma;$
$\forall x\in it.\ \forall y\in S - it.\ R\ y\ x\ ]\!] \Longrightarrow P\ \sigma$
shows *FOREACHoci R I S c f* $\sigma 0 \le SPEC\ P$
apply (*rule FOREACHoci-rule*[*OF FIN*, **where** $I=I$, *OF I0*])
apply (*rule IP*, *assumption*+)
apply (*case-tac c* σ)
apply (*blast intro*: *II1*)
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*:
$\bigwedge x\ it\ \sigma.\ [\![\ x\in it;\ it\subseteq S;\ I\ it\ \sigma;\ c\ \sigma\ ]\!] \Longrightarrow f\ x\ \sigma \le SPEC\ (I\ (it-\{x\}))$
assumes *II1*: $\bigwedge\sigma.\ [\![I\ \{\}\ \sigma;\ c\ \sigma]\!] \Longrightarrow P\ \sigma$
assumes *II2*: $\bigwedge it\ \sigma.\ [\![\ it\subseteq S;\ I\ it\ \sigma;\ \neg c\ \sigma\ ]\!] \Longrightarrow P\ \sigma$
shows *FOREACHci I S c f* $\sigma 0 \le SPEC\ P$
unfolding *FOREACHci-def*
by (*rule FOREACHoci-rule'*) (*simp-all add*: *assms*)

lemma *FOREACHc-rule'*:

assumes *FIN*: *finite S*
assumes *I0*: *I S σ0*
assumes *IP*:
⋀*x it σ*. ⟦ *x*∈*it*; *it*⊆*S*; *I it σ*; *c σ* ⟧ ⟹ *f x σ* ≤ *SPEC* (*I* (*it*−{*x*}))
assumes *II1*: ⋀*σ*. ⟦*I* {} *σ*; *c σ*⟧ ⟹ *P σ*
assumes *II2*: ⋀*it σ*. ⟦ *it*⊆*S*; *I it σ*; ¬*c σ* ⟧ ⟹ *P σ*
shows *FOREACHc S c f σ0* ≤ *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*] :
FOREACHoci R Φ {} *c f σ* = *do* {*ASSERT* (Φ {} *σ*); *RETURN σ*}
by (*simp add*: *FOREACHoci-def FOREACH-cond-def bind-RES*)
(*simp add*: *WHILEIT-unfold*)

lemma *FOREACHoi-emp* [*simp*] :
FOREACHoi R Φ {} *f σ* = *do* {*ASSERT* (Φ {} *σ*); *RETURN σ*}
by (*simp add*: *FOREACHoi-def*)

lemma *FOREACHci-emp* [*simp*] :
FOREACHci Φ {} *c f σ* = *do* {*ASSERT* (Φ {} *σ*); *RETURN σ*}
by (*simp add*: *FOREACHci-def*)

lemma *FOREACHc-emp* [*simp*] :
FOREACHc {} *c f σ* = *RETURN σ*
by (*simp add*: *FOREACHc-def*)

lemma *FOREACH-emp* [*simp*] :
FOREACH {} *f σ* = *RETURN σ*
by (*simp add*: *FOREACH-def*)

lemma *FOREACHi-emp* [*simp*] :
FOREACHi Φ {} *f σ* = *do* {*ASSERT* (Φ {} *σ*); *RETURN σ*}
by (*simp add*: *FOREACHi-def*)

2.15.5 Monotonicity

definition *lift-refl P c f g* == ∀ *x*. *P c* (*f x*) (*g x*)
definition *lift-mono P c f g* == ∀ *x y*. *c x y* ⟶ *P c* (*f x*) (*g y*)
definition *lift-mono1 P c f g* == ∀ *x y*. (∀ *a*. *c* (*x a*) (*y a*)) ⟶ *P c* (*f x*) (*g y*)
definition *lift-mono2 P c f g* == ∀ *x y*. (∀ *a b*. *c* (*x a b*) (*y a b*)) ⟶ *P c* (*f x*) (*g y*)

definition *trimono-spec L f* == ((*L id* (≤) *f f*) ∧ (*L id flat-ge f f*))

lemmas *trimono-atomize* = *atomize-imp atomize-conj atomize-all*

lemmas *trimono-deatomize = trimono-atomize*[*symmetric*]

lemmas *trimono-spec-defs = trimono-spec-def lift-refl-def*[*abs-def*] *comp-def id-def lift-mono-def*[*abs-def*] *lift-mono1-def*[*abs-def*] *lift-mono2-def*[*abs-def*] *trimono-deatomize*

locale *trimono-spec* **begin**
abbreviation *R* ≡ *lift-refl*
abbreviation *M* ≡ *lift-mono*
abbreviation *M1* ≡ *lift-mono1*
abbreviation *M2* ≡ *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*
[] ⇒ *RETURN s*
| *x*#*ls* ⇒ *if c s then do* { *s*←*f x s*; *nfoldli ls c f s*} *else RETURN s*
)

lemma *nfoldli-simps*[*simp*]:
nfoldli [] *c f s* = *RETURN s*
nfoldli (*x*#*ls*) *c f s* =
(*if c s then do* { *s*←*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 (λ*i*. *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 transfered 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 $l :: 'a\ list$ **and** $c :: 'b \Rightarrow bool$
assumes *FR*: $\bigwedge x\ s.\ nres\text{-}of\ (f\ x\ s) \leq f'\ x\ s$
shows *nres-of*
(*foldli l* (*case-dres False False c*) ($\lambda x\ s.\ s \gg\!= f\ x$) (*dRETURN s*))
$\leq$ (*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*]:
$[\![\bigwedge x\ s.\ f\ x\ s \leq f'\ x\ s\ ]\!] \Longrightarrow nfoldli\ l\ c\ f\ \sigma \leq nfoldli\ l\ c\ f'\ \sigma$
$[\![\bigwedge x\ s.\ flat\text{-}ge\ (f\ x\ s)\ (f'\ x\ s)\ ]\!] \Longrightarrow flat\text{-}ge\ (nfoldli\ l\ c\ f\ \sigma)\ (nfoldli\ l\ c\ f'\ \sigma)$
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*: *nfoldli l c f* σ
$\leq$
(${WHILE_T}^I$
(*FOREACH-cond c*) (*FOREACH-body f*) (*l*, σ) $\gg\!=$
(λ(-, σ). *RETURN* σ))
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 σ)
    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 {
    (-,σ) ← WHILE_T (FOREACH-cond c) (FOREACH-body f) (l,σ);
    RETURN σ
  } ≤ nfoldli l c f σ
  apply (induct l arbitrary: σ)
  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 {
    (-,σ) ← WHILE_T (FOREACH-cond c) (FOREACH-body f) (l,σ);
    RETURN σ
  } = nfoldli l c f σ
  apply (rule antisym)
  apply (rule while-nfoldli)
  apply (rule order-trans[OF nfoldli-while[where I=λ-. True]])
  apply (simp add: WHILET-def)
  done

lemma nfoldli-rule:
  assumes I0: I [] l0 σ0
  assumes IS: ⋀x l1 l2 σ. ⟦ l0=l1@x#l2; I l1 (x#l2) σ; c σ ⟧ ⟹ f x σ ≤ SPEC
(I (l1@[x]) l2)
```

```
  assumes FNC: ⋀l1 l2 σ. ⟦ l0=l1@l2; I l1 l2 σ; ¬c σ ⟧ ⟹ P σ
  assumes FC: ⋀σ. ⟦ I l0 [] σ; c σ ⟧ ⟹ P σ
  shows nfoldli l0 c f σ0 ≤ SPEC P
  apply (rule order-trans[OF nfoldli-while[
    where I=λ(l2,σ). ∃l1. l0=l1@l2 ∧ I l1 l2 σ]])
  unfolding FOREACH-cond-def FOREACH-body-def
  apply (refine-rcg 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 σ0
  assumes IS: ⋀x l1 l2 σ. ⟦ l0=l1@x#l2; I l1 (x#l2) σ; c σ ⟧ ⟹ f x σ ≤n SPEC
(I (l1@[x]) l2)
  assumes FNC: ⋀l1 l2 σ. ⟦ l0=l1@l2; I l1 l2 σ; ¬c σ ⟧ ⟹ P σ
  assumes FC: ⋀σ. ⟦ I l0 [] σ; c σ ⟧ ⟹ P σ
  shows nfoldli l0 c f σ0 ≤n SPEC P
proof −
  {
    fix l1 l2 σ
    assume l0=l1@l2    I l1 l2 σ
    hence nfoldli l2 c f σ ≤n SPEC P
    proof (induction l2 arbitrary: l1 σ)
      case Nil thus ?case
        apply simp
        apply (cases c σ)
        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 σ,THEN
leof-trans]

      show ?case
        apply (simp split del: if-split)
        apply refine-vcg
```

```
      using Cons.prems 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]: ⋀xi x si s. ⟦ (xi,x)∈S; (si,s)∈R; c s ⟧ ⟹ fi xi si ≤ ⇓R (f x s)
  shows nfoldli li ci fi si ≤ ⇓ 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.prems 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: ⋀l1i l2i l1 l2 si s. ⟦
    li=l1i@l2i; l=l1@l2; (l1i,l1)∈⟨S⟩list-rel; (l2i,l2)∈⟨S⟩list-rel;
    I l1i l2i si; (si,s)∈R⟧ ⟹ (ci si, c s)∈bool-rel
  assumes INV: ⋀l1i xi l2i si. ⟦li=l1i@xi#l2i; I l1i (xi#l2i) si⟧ ⟹ fi xi si ≤n
SPEC (I (l1i@[xi]) l2i)
  assumes STEP: ⋀l1i xi l2i l1 x l2 si s. ⟦
    li=l1i@xi#l2i; l=l1@x#l2; (l1i,l1)∈⟨S⟩list-rel; (xi,x)∈S; (l2i,l2)∈⟨S⟩list-rel;
    I l1i (xi#l2i) si; (si,s)∈R⟧ ⟹ fi xi si ≤ ⇓R (f x s)
  shows nfoldli li ci fi si ≤ ⇓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.prems Cons.hyps
by auto
        subgoal apply (rule STEP) using Cons.prems Cons.hyps by auto
        subgoal for si' s'
          apply (rule Cons.IH[of l1i@[xi]    l1@[x]])
          using Cons.prems 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.prems 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*: $\bigwedge a\ x\ x'.\ x \leq x' \Longrightarrow f\ a\ x \leq f'\ a\ x'$
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*) ($\lambda x\ s.$ *dbind s* (*f x*)) *σ*
$\leq$ *foldli l* (*case-dres False False c*) ($\lambda x\ s.$ *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*
[] ⇒ *dRETURN s*
| *x*#*ls* ⇒ *if c s then do* { *s*←*f x s*; *dfoldli ls c f s*} *else dRETURN s*
)

lemma *dfoldli-simps*[*simp*]:
dfoldli [] *c f s* = *dRETURN s*
dfoldli (*x*#*ls*) *c f s* =
(*if c s then do* { *s*←*f x s*; *dfoldli ls c f s*} *else dRETURN s*)
apply (*subst dfoldli.simps*, *simp*)+
done

lemma *dfoldli-mono*[*refine-mono*]:
⟦ $\bigwedge x\ s.\ f\ x\ s \leq f'\ x\ s$ ⟧ ⟹ *dfoldli l c f σ* ≤ *dfoldli l c f' σ*
⟦ $\bigwedge x\ s.$ *flat-ge* (*f x s*) (*f' x s*) ⟧ ⟹ *flat-ge* (*dfoldli l c f σ*) (*dfoldli l c f' σ*)
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*) ($\lambda x\ s.$ *dbind s* (*f x*)) *dFAIL* = *dFAIL*

by (*cases l*) *auto*

lemma *foldli-dres-pres-SUCCEED*[*simp*]:
foldli l (*case-dres False False c*) (λ*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*) (λ*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*: ⋀*a x*. *flat-ge* (*f a x*) (*f*′ *a x*)
shows *flat-ge* (*foldli l* (*case-dres False False c*) (λ*x s*. *dbind s* (*f x*)) *σ*)
(*foldli l* (*case-dres False False c*) (λ*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*: *σ* ≠ *dSUCCEED*
assumes *2*: ⋀*x σ*. *f x σ* ≠ *dSUCCEED*
shows *foldli l c* (λ*x s*. *s* ⤜ *f x*) *σ* ≠ *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* ≡ ⟨*R*⟩*list-rel O br set distinct*

lemma *autoref-nfoldli*[*autoref-rules*]:
shows (*nfoldli*, *nfoldli*)
∈ ⟨*Ra*⟩*list-rel* → (*Rb* → *bool-rel*) → (*Ra* → *Rb* → ⟨*Rb*⟩*nres-rel*) → *Rb* → ⟨*Rb*⟩*nres-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*
≡ *SPEC* (λ*l*. *distinct l* ∧ *s* = *set l* ∧ *sorted-wrt R l*)

definition *LIST-FOREACH* Φ *tsl c f* σ*0* ≡ *do* {
xs ← *tsl*;
(-,σ) ← $WHILE_T$λ(*it*, σ). ∃*xs'*. *xs* = *xs'* @ *it* ∧ Φ (*set it*) σ
(*FOREACH-cond c*) (*FOREACH-body f*) (*xs*, σ*0*);
RETURN σ}

lemma *FOREACHoci-by-LIST-FOREACH*:
FOREACHoci R Φ *S c f* σ*0* = *do* {
ASSERT (*finite S*);
LIST-FOREACH Φ (*it-to-sorted-list R S*) *c f* σ*0*
}
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* ≡ $FOREACH_{OC}{}^{\lambda\text{- -. }True,I}$ *s* (λ-. *True*) *f*
FOREACHci I s c f ≡ *FOREACHoci* (λ- -. *True*) *I s c f*
$FOREACH_{OC}{}^{R,\Phi}$ *s c f* ≡ λσ. *do* {
ASSERT (*finite s*);
Autoref-Tagging.OP (*LIST-FOREACH* Φ) (*it-to-sorted-list R s*) *c f* σ
}
FOREACH s f ≡ *FOREACHoci* (λ- -. *True*) (λ- -. *True*) *s* (λ-. *True*) *f*
FOREACHoi R I s f ≡ *FOREACHoci R I s* (λ-. *True*) *f*
FOREACHc s c f ≡ *FOREACHoci* (λ- -. *True*) (λ- -. *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′ tsl c f σ* ≡ *do* {*xs* ← *tsl*; *nfoldli xs c f σ*}

lemma *LIST-FOREACH′-param*[*param*]:
shows (*LIST-FOREACH′*,*LIST-FOREACH′*)
∈ (⟨⟨*Rv*⟩*list-rel*⟩*nres-rel* → (*Rσ*→*bool-rel*)
→ (*Rv* → *Rσ* → ⟨*Rσ*⟩*nres-rel*) → *Rσ* → ⟨*Rσ*⟩*nres-rel*)
unfolding *LIST-FOREACH′-def*[*abs-def*]
by *parametricity*

lemma *LIST-FOREACH-autoref*[*autoref-rules*]:
shows (*LIST-FOREACH′*, *LIST-FOREACH* Φ) ∈
(⟨⟨*Rv*⟩*list-rel*⟩*nres-rel* → (*Rσ*→*bool-rel*)
→ (*Rv* → *Rσ* → ⟨*Rσ*⟩*nres-rel*) → *Rσ* → ⟨*Rσ*⟩*nres-rel*)
proof (*intro fun-relI nres-relI*)
fix *tsl tsl′ c c′ f f′ σ σ′*
assume [*param*]:
(*tsl*,*tsl′*)∈⟨⟨*Rv*⟩*list-rel*⟩*nres-rel*
(*c*,*c′*)∈*Rσ*→*bool-rel*
(*f*,*f′*)∈*Rv*→*Rσ*→⟨*Rσ*⟩*nres-rel*
(*σ*,*σ′*)∈*Rσ*

have *LIST-FOREACH′ tsl c f σ* ≤ ⇓*Rσ* (*LIST-FOREACH′ tsl′ c′ f′ σ′*)
apply (*rule nres-relD*)
by *parametricity*
also have *LIST-FOREACH′ tsl′ c′ f′ σ′*
≤ *LIST-FOREACH* Φ *tsl′ c′ f′ σ′*
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′ tsl c f σ ≤ ⇓ *Rσ* (*LIST-FOREACH* Φ *tsl′ c′ f′ σ′*)
.
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* ≤ *tsl'*
assumes ⋀*x σ*. *RETURN* (*f x σ*) ≤ *f' x σ*
shows *RETURN* (*foldli tsl c f σ*) ≤ *LIST-FOREACH' tsl' c f' σ*
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* ≤ *tsl'*
assumes ⋀*x σ*. *nres-of* (*f x σ*) ≤ *f' x σ*
shows *nres-of* (
do {
xs←*tsl*;
foldli xs (*case-dres False False c*) (λ*x s*. *s*≫=*f x*) (*dRETURN σ*)
}) ≤ *LIST-FOREACH' tsl' c f' σ*
unfolding *LIST-FOREACH'-def*
using *assms*
by *refine-transfer*

Simplification rules to summarize iterators

lemma [*refine-transfer-post-simp*]:
do {
xs ← *dRETURN tsl*;
foldli xs c f σ
} = *foldli tsl c f σ*
by *simp*

lemma [*refine-transfer-post-simp*]:
(*let xs* = *tsl in foldli xs c f σ*) = *foldli tsl c f σ*
by *simp*

lemma *LFO-pre-refine*:
assumes (*li*,*l*)∈⟨*A*⟩*list-set-rel*
assumes (*ci*,*c*)∈*R* → *bool-rel*
assumes (*fi*,*f*)∈*A*→*R*→⟨*R*⟩*nres-rel*
assumes (*s0i*,*s0*)∈*R*
shows *LIST-FOREACH'* (*RETURN li*) *ci fi s0i* ≤ ⇓*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 $(li,l)\in\langle A\rangle$*list-set-rel*
assumes $\bigwedge s\ si.\ (si,s)\in R \Longrightarrow ci\ si \longleftrightarrow c\ s$
assumes $\bigwedge x\ xi\ s\ si.\ [\![(xi,x)\in A;\ (si,s)\in R]\!] \Longrightarrow fi\ xi\ si \le \Downarrow R\ (f\ x\ s)$
assumes $(s0i,s0)\in R$
shows *nfoldli li ci fi s0i* $\le \Downarrow R$ (*FOREACHci 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 $(li,l)\in\langle A\rangle$*list-set-rel*
assumes $\bigwedge s\ si.\ (si,s)\in R \Longrightarrow ci\ si \longleftrightarrow c\ s$
assumes $\bigwedge x\ xi\ s\ si.\ [\![(xi,x)\in A;\ (si,s)\in R]\!] \Longrightarrow fi\ xi\ si \le \Downarrow R\ (f\ x\ s)$
assumes $(s0i,s0)\in R$
shows *nfoldli li ci fi s0i* $\le \Downarrow R$ (*FOREACHc l c f s0*)
unfolding *FOREACHc-def*
by (*rule LFOci-refine*[*OF assms*])

lemma *LFO-refine*:
assumes $(li,l)\in\langle A\rangle$*list-set-rel*
assumes $\bigwedge x\ xi\ s\ si.\ [\![(xi,x)\in A;\ (si,s)\in R]\!] \Longrightarrow fi\ xi\ si \le \Downarrow R\ (f\ x\ s)$
assumes $(s0i,s0)\in R$
shows *nfoldli li* (λ-. *True*) *fi s0i* $\le \Downarrow R$ (*FOREACH l f s0*)
unfolding *FOREACH-def*
apply (*rule LFOc-refine*)
apply (*rule assms* | *simp*)+
done

lemma *LFOi-refine*:
assumes $(li,l)\in\langle A\rangle$*list-set-rel*
assumes $\bigwedge x\ xi\ s\ si.\ [\![(xi,x)\in A;\ (si,s)\in R]\!] \Longrightarrow fi\ xi\ si \le \Downarrow R\ (f\ x\ s)$
assumes $(s0i,s0)\in R$
shows *nfoldli li* (λ-. *True*) *fi s0i* $\le \Downarrow R$ (*FOREACHi I l f s0*)
unfolding *FOREACHi-def*
apply (*rule LFOci-refine*)
apply (*rule assms* | *simp*)+

done

lemma *LIST-FOREACH'-refine*: *LIST-FOREACH' tsl' c' f' σ' ≤ LIST-FOREACH Φ tsl' c' f' σ'*
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* (λ- -. *True*) *tsl' c' f' σ' = (LIST-FOREACH' tsl' c' f' σ')*
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 c f σ ≡ do* {
ASSERT (*finite S*);
l ← *SPEC* (λ*l*. *set l = S*);
nfoldli l c f σ
}

lemma *FOREACHcd-rule*:
assumes *finite* S_0
assumes *I0*: *I* {} S_0 σ_0
assumes *STEP*: ⋀*S1 S2 x σ*. ⟦ S_0 = *insert x* (*S1*∪*S2*); *I S1* (*insert x S2*) *σ*; *c σ* ⟧ ⟹ *f x σ ≤ SPEC* (*I* (*insert x S1*) *S2*)
assumes *INTR*: ⋀*S1 S2 σ*. ⟦ S_0 = *S1*∪*S2*; *I S1 S2 σ*; ¬*c σ* ⟧ ⟹ Φ *σ*
assumes *COMPL*: ⋀*σ*. ⟦ *I* S_0 {} *σ*; *c σ* ⟧ ⟹ Φ *σ*
shows *FOREACHcd* S_0 *c f* σ_0 ≤ *SPEC* Φ
unfolding *FOREACHcd-def*
apply *refine-vcg*

```
  apply fact
  apply (rule nfoldli-rule[where I = λl1 l2 σ. I (set l1) (set l2) σ])
  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 ⇒ 'a set ⇒ 'b ⇒ bool)
    ⇒ 'a set ⇒ ('b ⇒ bool) ⇒ ('a ⇒ 'b ⇒ 'b nres) ⇒ 'b ⇒ 'b nres
  where
  FOREACHcdi I ≡ FOREACHcd

lemma FOREACHcdi-rule[refine-vcg]:
  assumes finite S₀
  assumes I0: I {} S₀ σ₀
  assumes STEP: ⋀S1 S2 x σ. ⟦ S₀ = insert x (S1∪S2); I S1 (insert x S2) σ; c
σ ⟧ ⟹ f x σ ≤ SPEC (I (insert x S1) S2)
  assumes INTR: ⋀S1 S2 σ. ⟦ S₀ = S1∪S2; I S1 S2 σ; ¬c σ ⟧ ⟹ Φ σ
  assumes COMPL: ⋀σ. ⟦ I S₀ {} σ; c σ ⟧ ⟹ Φ σ
  shows FOREACHcdi I S₀ c f σ₀ ≤ SPEC Φ
  unfolding FOREACHcdi-def
  using assms
  by (rule FOREACHcd-rule)

lemma FOREACHcd-refine[refine]:
  assumes [simp]: finite s'
  assumes S: (s',s)∈⟨S⟩set-rel
  assumes SV: single-valued S
  assumes R0: (σ',σ)∈R
  assumes C: ⋀σ' σ. (σ',σ)∈R ⟹ (c' σ', c σ)∈bool-rel
  assumes F: ⋀x' x σ' σ. ⟦(x', x) ∈ S; (σ', σ) ∈ R⟧
    ⟹ f' x' σ' ≤ ⇓ R (f x σ)
  shows FOREACHcd s' c' f' σ' ≤ ⇓R (FOREACHcdi I s c f σ)
proof −
  have [refine-dref-RELATES]: RELATES S by (simp add: RELATES-def)

  from SV obtain α I where [simp]: S=br α I by (rule single-valued-as-brE)
  with S have [simp]: s=α`s' and [simp]: ∀x∈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 α 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 σ ≤ FOREACHcd s c f σ
  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 Miscellanneous Utility Lemmas

```
lemma map-foreach:
  assumes finite S
  shows FOREACH S (λx σ. RETURN (insert (f x) σ)) R0 ≤ SPEC ((=) (R0
∪ f`S))
  apply (rule FOREACH-rule[where I=λit σ. σ=R0 ∪ f`(S−it)])
  apply (auto intro: assms)
  done

lemma map-sigma-foreach:
  fixes f :: 'a × 'b ⇒ 'c
  assumes finite A
  assumes ⋀x. x∈A ⟹ finite (B x)
  shows FOREACH A (λa σ.
    FOREACH (B a) (λb σ. RETURN (insert (f (a,b)) σ)) σ
  ) R0 ≤ SPEC ((=) (R0 ∪ f`Sigma A B))
  apply (rule FOREACH-rule[where I=λit σ. σ=R0 ∪ f`(Sigma (A−it) B)])
  apply (auto intro: assms) [2]

  apply (rule-tac I=λit' σ. σ=R0 ∪ f`(Sigma (A − it) B)
    ∪ f`({x} × (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 $f :: 'a \times ('b \times 'c) \Rightarrow 'd$
assumes *finite A*
assumes $\bigwedge a.\ a{\in}A \Longrightarrow$ *finite* $(B\ a)$
assumes $\bigwedge a\ b.\ [\![a{\in}A;\ b{\in}B\ a]\!] \Longrightarrow$ *finite* $(C\ a\ b)$
shows *FOREACH A* $(\lambda a\ \sigma.$
FOREACH $(B\ a)$ $(\lambda b\ \sigma.$
FOREACH $(C\ a\ b)$ $(\lambda c\ \sigma.$
RETURN (*insert* $(f\ (a,(b,c)))\ \sigma)\ \sigma)\ \sigma$
$)\ R0 \leq SPEC\ ((=)\ (R0 \cup f`Sigma\ A\ (\lambda a.\ Sigma\ (B\ a)\ (C\ a))))$
apply (*rule FOREACH-rule*[**where**
$I{=}\lambda it\ \sigma.\ \sigma{=}R0 \cup f`(Sigma\ (A{-}it)\ (\lambda a.\ Sigma\ (B\ a)\ (C\ a)))$])
apply (*auto intro*: *assms*) [*2*]
apply (*rule-tac*
$I{=}\lambda it'\ \sigma.\ \sigma{=}R0 \cup f`(Sigma\ (A - it)\ (\lambda a.\ Sigma\ (B\ a)\ (C\ a)))$
$\cup\ f`(\{x\} \times (\ Sigma\ (B\ x - it')\ (C\ x)))$
in *FOREACH-rule*)
apply (*auto intro*: *assms*) [*2*]
apply (*rule-tac*
$I{=}\lambda it''\ \sigma.\ \sigma{=}R0 \cup f`(Sigma\ (A - it)\ (\lambda a.\ Sigma\ (B\ a)\ (C\ a)))$
$\cup\ f`(\{x\} \times (\ Sigma\ (B\ x - ita)\ (C\ x)))$
$\cup\ f`(\{x\} \times (\{xa\} \times (C\ x\ xa - it'')))$

in *FOREACH-rule*)
apply (*auto intro*: *assms*) [*2*]

apply *auto*
done

lemma *bij-set-rel-for-inj*:
fixes *R*
defines $\alpha \equiv$ *fun-of-rel R*
assumes *bijective R* $(s,s')\in\langle R\rangle$*set-rel*
shows *inj-on* α *s* $s' = \alpha`s$
— To be used when generating refinement conditions for foreach-loops
using *assms*
unfolding *bijective-def set-rel-def* α*-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 nfoldli (drop k l) c f s = nfoldli [k..<length l] c (λi s. do {
      ASSERT (i<length l);
      let x = l!i;
      f x s
    }) s
proof (cases k≤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: drop k l = l!k # drop (Suc k) l
      by (auto simp: Cons-nth-drop-Suc)
    from step.hyps have 2: [k..<length l] = k#[Suc k..<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:
  nfoldli l c f s = nfoldli [0..<length l] c (λi s. do {
    ASSERT (i<length l);
    let x = l!i;
    f x s
  }) s
  using nfoldli-by-idx-gen[of 0] by auto

lemma nfoldli-map-inv:
  assumes inj g
  shows nfoldli l c f = nfoldli (map g l) c (λx s. f (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 :: nat
```

shows *nfoldli l c f = nfoldli (map (λi. i+ofs) l) c (λx s. do {ASSERT (x≥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 (λx s. do{ASSERT(x≥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 (λf. ¬f) (λy -. RETURN (y=x)) False ≤ SPEC (λr. r ⟷ x∈set l)*
proof −
have *nfoldli l (λf. ¬f) (λy -. RETURN (y=x)) s ≤ SPEC (λr. r ⟷ s ∨ x∈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 ⇒ 'b::comm-monoid-add nres) ⇒ 'a set ⇒ 'b nres*
where
sum-impl g S ≡ FOREACH S (λx a. do { b ← g x; RETURN (a+b)}) 0

lemma *sum-impl-correct*:
assumes [*simp*]: *finite S*
assumes [*refine-vcg*]: *⋀x. x∈S ⟹ gi x ≤ SPEC (λr. r=g x)*
shows *sum-impl gi S ≤ SPEC (λr. r=sum g S)*
unfolding *sum-impl-def*
apply (*refine-vcg FOREACH-rule*[**where** *I=λit 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-eqs: 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 Symtab.table
    val empty = Symtab.empty
    val merge = Symtab.merge-list ((op =) o apply2 #pattern)
  )

  fun add-extraction name ex =
    Context.theory-map (extractions.map (
      Symtab.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 xxxx;*)
    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-eqs 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

  (* 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-eqs 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 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 Parse.embedded-inner-syntax cxt
      val ctxt = Context.proof-of context
      val t = Proof-Context.read-term-pattern ctxt t
    in
      (Thm.cterm-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 Parse.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.cterm-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 ≡ RECT B
  assumes M: trimono B
  shows f x ≡ B f x
  unfolding D
  apply (subst RECT-unfold)
  by (rule M)

lemma gen-code-thm-REC:
  fixes x
  assumes D: f ≡ REC B
  assumes M: trimono B
  shows f x ≡ B f x
  unfolding D
  apply (subst REC-unfold)
  by (rule M)

setup ‹
  Refine-Automation.add-extraction nres {
    pattern = Logic.varify-global @{term REC x},
    gen-thm = @{thm gen-code-thm-REC},
    gen-tac = Refine-Mono-Prover.mono-tac
  }
  #>
  Refine-Automation.add-extraction nres {
    pattern = Logic.varify-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 $RETURN\ c \le b$
shows $(RETURN\ c,\ a)\in\langle R\rangle nres\text{-}rel \quad RETURN\ c \le \Downarrow R\ a$
using *assms*
unfolding *nres-rel-def*
by *simp-all*

lemma *autoref-monadicI2*:
assumes $(b,a)\in\langle R\rangle nres\text{-}rel$
assumes $nres\text{-}of\ c \le b$
shows $(nres\text{-}of\ c,\ a)\in\langle R\rangle nres\text{-}rel \quad nres\text{-}of\ c \le \Downarrow R\ a$
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 Framework

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 (λ*D x*. *do* {*ASSERT* (*pre x*); *r*←*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*: ⋀*f x*. ⟦⋀*x*. *pre x* ⟹ *f x* ≤ *SPEC* (*post x*); *pre x*⟧
⟹ *body f x* ≤ *SPEC* (*post x*)
and *C*: ⋀*r*. *post x r* ⟹ Φ *r*
shows *REC-annot pre post body x* ≤ *SPEC* Φ
proof −
from ‹*trimono body*› **have** [*refine-mono*]:
⋀*f g x xa*. (⋀*x*. *flat-ge* (*f x*) (*g x*)) ⟹ *flat-ge* (*body f x*) (*body g x*)
⋀*f g x xa*. (⋀*x*. *f x* ≤ *g x*) ⟹ *body f x* ≤ *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*=λ*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* ⇒ *'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* ⇒ *nat* ⇒ *'vertex* ⇒ *bool* **where**

dist-z: *dist v 0 v* |
dist-suc: ⟦*dist v d vh*; *v′ ∈ succ vh* ⟧ ⟹ *dist v* (*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′* = (*LEAST d*. *dist v d v′*)
definition *conn v v′* = (∃ *d*. *dist v d v′*)

Properties

In this subsection, we prove some properties of paths.

lemma
shows *connI*[*intro*]: *dist v d v′* ⟹ *conn v v′*
and *connI-id*[*intro*]: *conn v v*
and *connI-succ*[*intro*]: *conn v v′* ⟹ *v″ ∈ succ v′* ⟹ *conn v v″*
by (*auto simp*: *conn-def intro*: *dist-suc dist-z*)

lemma *min-distI2*:
⟦ *conn v v′* ; ⋀*d*. ⟦ *dist v d v′*; ⋀*d′*. *dist v d′ v′* ⟹ *d* ≤ *d′* ⟧ ⟹ *Q d* ⟧
⟹ *Q* (*min-dist v v′*)
unfolding *min-dist-def*
by (*rule LeastI2-wellorder*[**where** *Q*=*Q* **and** *a*=*SOME d*. *dist v d v′*])
(*auto simp*: *conn-def intro*: *someI*)

lemma *min-distI-eq*:
⟦ *dist v d v′*; ⋀*d′*. *dist v d′ v′* ⟹ *d* ≤ *d′* ⟧ ⟹ *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*]: *dist v 0 v′* ⟷ *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*]: *min-dist v v* = *0*
by (*rule min-distI2*) (*auto intro*: *dist-z*)

lemma *min-dist-z-iff*[*simp*]: *conn v v′* ⟹ *min-dist v v′* = *0* ⟷ *v′*=*v*
by (*rule min-distI2*) (*auto intro*: *dist-z*)

lemma *min-dist-is-dist*: *conn v v′* ⟹ *dist v* (*min-dist v v′*) *v′*
by (*auto intro*: *min-distI2*)
lemma *min-dist-minD*: *dist v d v′* ⟹ *min-dist v v′* ≤ *d*
by (*auto intro*: *min-distI2*)

We also provide introduction and destruction rules for the pattern *min-dist v v′* = *Suc d*.

```
lemma min-dist-succ:
  ⟦ conn v v'; v'' ∈ succ v' ⟧ ⟹ min-dist v v'' ≤ Suc (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' = Suc d
  shows ∃ v''. conn v v'' ∧ v' ∈ succ v'' ∧ min-dist v v'' = d
proof −
  from min-dist-is-dist[OF c(1)]
  have min-dist v v' = Suc d ⟶ ?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 ∃ v'. conn src v' ∧ 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' ≤ min-dist src v
  shows ∃ v'. conn src v' ∧ 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 *break*ed, 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 *break*ed, d is the distance of the shortest path between *src* and *dst*.

definition *bfs-invar′ src dst σ* ≡ *let* $(f,V,C,N,d)=\sigma$ *in*
$(\neg f \longrightarrow ($
$N \subseteq V \wedge dst \notin V - (C \cup N) \wedge$
$(\forall v{\in}C.\ conn\ src\ v \wedge min\text{-}dist\ src\ v = d) \wedge$
$(\forall v{\in}N.\ conn\ src\ v \wedge min\text{-}dist\ src\ v = Suc\ d) \wedge$
$(\forall v{\in}V.\ conn\ src\ v \wedge min\text{-}dist\ src\ v \leq Suc\ d) \wedge$
$(\forall v.\ conn\ src\ v \wedge min\text{-}dist\ src\ v \leq d \longrightarrow v \in V) \wedge$
$(\forall v.\ conn\ src\ v \wedge min\text{-}dist\ src\ v = Suc\ d \longrightarrow v \in N \cup ((\bigcup (succ`C)) - V))$
$)) \wedge ($
$f \longrightarrow conn\ src\ dst \wedge min\text{-}dist\ src\ dst = d$
$)$

The second part of the invariant, *empty-assm*, just states that C can only be empty if N is also empty.

definition *empty-assm σ* ≡ *let* $(f,V,C,N,d)=\sigma$ *in*
$C=\{\} \longrightarrow N=\{\}$

Finally, we define the invariant of the outer loop, *bfs-invar*, as the conjunction of both parts:

definition *bfs-invar src dst σ* ≡ *bfs-invar′ src dst σ* ∧
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 σ* ≡ *let* $(V',N')=\sigma$ *in*
$V'=V \cup (succ\ v - it) \wedge$
$N'=N \cup ((succ\ v - it) - V)$

3.1.3 Algorithm

The following algorithm is a straightforward transcription of the algorithm given in the assignment to the monadic style featured by our framework. We briefly explain the (mainly syntactic) differences:

- The initialization of the variables occur after the loop in our formulation. This is just a syntactic difference, as our loop construct has the form *WHILEI I c 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 $v \leftarrow SPEC\ (\lambda v.\ v \in C)$ nondeterministically 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 ⇒ 'vertex ⇒
  (nat option nres)
  where bfs src dst ≡ do {
  (f,-,-,-,d) ← WHILEI (bfs-invar src dst) (λ(f,V,C,N,d). f=False ∧ C≠{})
    (λ(f,V,C,N,d). do {
      v ← SPEC (λv. v∈C); let C = C−{v};
      if v=dst then RETURN (True,{},{},{},d)
      else do {
        (V,N) ← FOREACHi (FE-invar V N v) (succ v) (λw (V,N).
          if (w∉V) then do {
            ASSERT (w∉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∈C
  assumes v≠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-assm-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-assm-def
  by auto
```

If we have *break*ed 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** ∀ *v*. *conn src v* ∧ *min-dist src v* ≤ *d* ⟶ *v*∈*V*
unfolding *bfs-invar′-def* **by** *auto*
moreover

This applies in the first case, hence we obtain that *dst*∈*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* ≤ *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 ≡ SPEC (
  λr. case r of None ⇒ ¬ conn src dst
           | Some d ⇒ conn src dst ∧ min-dist src dst = d)
theorem bfs-correct: bfs src dst ≤ 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-assm-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*]: *word-nat-rel* ≡ *build-rel* (*unat*) (λ-. *True*)
lemma *word-nat-RELEATES*[*refine-dref-RELATES*]:
RELATES word-nat-rel **by** (*simp add*: *RELATES-def*)

lemma [*simp*, *relator-props*]:
single-valued word-nat-rel **unfolding** *word-nat-rel-def*
by *blast*

lemma [*simp*]: *single-valuedp* (λ*c a*. *a* = *unat c*)
by (*rule single-valuedpI*) *blast*

lemma [*simp*, *relator-props*]: *single-valued* (*converse word-nat-rel*)
by (*auto intro*: *injI*)

lemmas [*refine-hsimp*] =
word-less-nat-alt word-le-nat-alt unat-sub iffD1[*OF unat-add-lem*]

3.2.2 Example

type-synonym *word32* = *32 word*

definition *test* :: *nat* ⇒ *nat* ⇒ *nat set nres* **where** *test x0 y0* ≡ *do* {
let S={};
(*S*,-,-) ← *WHILE* (λ(*S*,*x*,*y*). *x*>*0*) (λ(*S*,*x*,*y*). *do* {
let S=*S*∪{*y*};
let x=*x* − *1*;
ASSERT (*y*<*x0*+*y0*);
let y=*y* + *1*;
RETURN (*S*,*x*,*y*)
}) (*S*,*x0*,*y0*);
RETURN S
}

lemma *y0*>*0* ⟹ *test x0 y0* ≤ *SPEC* (λ*S*. *S*={*y0* .. *y0* + *x0* − *1*})
— Choosen pre-condition to get least trouble when proving
unfolding *test-def*
apply (*intro WHILE-rule*[**where** *I*=λ(*S*,*x*,*y*).
x+*y*=*x0*+*y0* ∧ *x*≤*x0* ∧
S={*y0* .. *y0* + (*x0*−*x*) − *1*}]
refine-vcg)

```
  by auto

definition test-impl :: word32 ⇒ word32 ⇒ word32 set nres where
  test-impl x y ≡ do {
    let S={};
    (S,-,-) ← WHILE (λ(S,x,y). x>0) (λ(S,x,y). do {
      let S=S∪{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^LENGTH(32)
  assumes (x,x')∈word-nat-rel
  assumes (y,y')∈word-nat-rel
  shows test-impl x y ≤ ⇓(⟨word-nat-rel⟩set-rel) (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 linked[1]. 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

[1] This 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.