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

2.11 While-Loops . . . 74
2.11.1 Data Refinement Rules . . . 75
2.11.2 Autoref Setup . . . 79
2.11.3 Invariants . . . 81
2.11.4 Convenience . . . 89
2.12 Deterministic Monad . . . 90
2.12.1 Deterministic Result Lattice . . . 90
2.13 Partial Function Package Setup . . . 95
2.13.1 Nondeterministic Result Monad . . . 96
2.13.2 Deterministic Result Monad . . . 96
2.14 Transfer Setup . . . 97
2.14.1 Transfer to Deterministic Result Lattice . . . 97
2.14.2 Transfer to Plain Function . . . 98
2.14.3 Total correctness in deterministic monad . . . 98
2.14.4 Relator-Based Transfer . . . 99
2.14.5 Post-Simplification Setup . . . 100
2.15 Foreach Loops . . . 100
2.15.1 Auxilliary Lemmas . . . 100
2.15.2 Definition . . . 101
2.15.3 Proof Rules . . . 102
2.15.4 FOREACH with empty sets . . . 113
2.15.5 Monotonicity . . . 114
2.15.6 Nres-Fold with Interruption (nfoldli) . . . 115
2.15.7 LIST FOREACH combinator . . . 119
2.15.8 FOREACH with duplicates . . . 122
2.15.9 Miscellanneous Utility Lemmas . . . 123
2.16 More Automation . . . 125
2.17 Autoref for the Refinement Monad . . . 126
2.18 Refinement Framework . . . 127
2.18.1 Convenience Constructs . . . 127
2.18.2 Syntax Sugar . . . 127

3 Examples **129**
3.1 Breadth First Search . . . 129
3.1.1 Distances in a Graph . . . 129
3.1.2 Invariants . . . 131
3.1.3 Algorithm . . . 132
3.1.4 Verification Tasks . . . 134
3.2 Machine Words . . . 136
3.2.1 Setup . . . 136
3.2.2 Example . . . 137

4 Conclusion and Future Work **139**

Chapter 1

Introduction

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

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

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

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

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

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

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

1.1 Related Work

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

Chapter 2

Refinement Framework

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

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

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

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

⟨*ML*⟩

end

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

⟨*ML*⟩

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

end

2.1 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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *let-mono*[*refine-mono*]:
f x ≤ *f' x'* ⟹ *Let x f* ≤ *Let x' f'* ⟨*proof*⟩

2.1.1 Uncategorized Lemmas

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

2.1.2 Well-Foundedness

lemma *wf-no-infinite-down-chainI*:
assumes ⋀*f*. ⟦⋀*i*. (*f* (*Suc i*), *f i*)∈*r*⟧ ⟹ *False*
shows *wf r*
⟨*proof*⟩

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*⟩

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)
⟨*proof*⟩

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

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

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

2.1.3 Monotonicity and Orderings

lemma *mono-const*[*simp*, *intro*!]: *mono* $(\lambda\text{-}.\ c)$ ⟨*proof*⟩
lemma *mono-if*: $[\![$*mono S1*; *mono S2*$]\!] \Longrightarrow$
mono $(\lambda F\ s.$ *if* $b\ s$ *then S1* $F\ s$ *else S2* $F\ s)$
⟨*proof*⟩

lemma *mono-infI*: *mono* $f \Longrightarrow$ *mono* $g \Longrightarrow$ *mono* (*inf* $f\ g$)
⟨*proof*⟩

lemma *mono-infI'*:
mono $f \Longrightarrow$ *mono* $g \Longrightarrow$ *mono* $(\lambda x.$ *inf* $(f\ x)\ (g\ x) :: 'b::lattice)$
⟨*proof*⟩

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

lemma *mono-Sup*:

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

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

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

lemma *mono-funpow*: $mono\ (f{::}'a{::}order \Rightarrow 'a) \Longrightarrow mono\ (f\widehat{\ }i)$
⟨*proof*⟩

lemma *mono-id*[*simp*, *intro*!]:
mono id
$mono\ (\lambda x.\ x)$
⟨*proof*⟩

declare *SUP-insert*[*simp*]

lemma (**in** *semilattice-inf*) *le-infD1*:
$a \leq inf\ b\ c \Longrightarrow a \leq b$ ⟨*proof*⟩
lemma (**in** *semilattice-inf*) *le-infD2*:
$a \leq inf\ b\ c \Longrightarrow a \leq c$ ⟨*proof*⟩
lemma (**in** *semilattice-inf*) *inf-leI*:
$[\![\bigwedge x.\ [\![x \leq a;\ x \leq b]\!] \Longrightarrow x \leq c]\!] \Longrightarrow inf\ a\ b \leq c$
⟨*proof*⟩

lemma *top-Sup*: $(top{::}'a{::}complete\text{-}lattice) \in A \Longrightarrow Sup\ A = top$
⟨*proof*⟩
lemma *bot-Inf*: $(bot{::}'a{::}complete\text{-}lattice) \in A \Longrightarrow Inf\ A = bot$
⟨*proof*⟩

lemma *mono-compD*: $mono\ f \Longrightarrow x \leq y \Longrightarrow f\ o\ x \leq f\ o\ y$
⟨*proof*⟩

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*⟩
lemma $\gamma\alpha$-*infl*: $x \leq \gamma(\alpha(x))$
⟨*proof*⟩

lemma α-*mono*: *mono* α
⟨*proof*⟩

lemma γ-*mono*: *mono* γ
⟨*proof*⟩

lemma *dist-*γ[*simp*]:
$\gamma\ (inf\ a\ b) = inf\ (\gamma\ a)\ (\gamma\ b)$
⟨*proof*⟩

lemma *dist-*α[*simp*]:
$\alpha\ (sup\ a\ b) = sup\ (\alpha\ a)\ (\alpha\ b)$
⟨*proof*⟩

end

Fixed Points

lemma *mono-lfp-eqI*:
assumes *MONO*: *mono f*
assumes *FIXP*: $f\ a \leq a$
assumes *LEAST*: $\bigwedge x.\ f\ x = x \Longrightarrow a \leq x$
shows *lfp f* $= a$
⟨*proof*⟩

lemma *mono-gfp-eqI*:
assumes *MONO*: *mono f*
assumes *FIXP*: $a \leq f\ a$
assumes *GREATEST*: $\bigwedge x.\ f\ x = x \Longrightarrow x \leq a$
shows *gfp f* $= a$
⟨*proof*⟩

lemma *lfp-le-gfp'*: *mono f* $\Longrightarrow$ *lfp f x* $\leq$ *gfp f x*
⟨*proof*⟩

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* $\leq P$
⟨*proof*⟩

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*
⟨*proof*⟩

Connecting Complete Lattices and Chain-Complete Partial Orders

lemma (**in** *complete-lattice*) *is-ccpo*: *class.ccpo Sup* (≤) (<)
⟨*proof*⟩

lemma (**in** *complete-lattice*) *is-dual-ccpo*: *class.ccpo Inf* (≥) (>)
⟨*proof*⟩

lemma *ccpo-mono-simp*: *monotone* (≤) (≤) *f* ⟷ *mono f*
⟨*proof*⟩
lemma *ccpo-monoI*: *mono f* ⟹ *monotone* (≤) (≤) *f*
⟨*proof*⟩
lemma *ccpo-monoD*: *monotone* (≤) (≤) *f* ⟹ *mono f*
⟨*proof*⟩

lemma *dual-ccpo-mono-simp*: *monotone* (≥) (≥) *f* ⟷ *mono f*
⟨*proof*⟩
lemma *dual-ccpo-monoI*: *mono f* ⟹ *monotone* (≥) (≥) *f*
⟨*proof*⟩
lemma *dual-ccpo-monoD*: *monotone* (≥) (≥) *f* ⟹ *mono f*
⟨*proof*⟩

lemma *ccpo-lfp-simp*: ⋀*f*. *mono f* ⟹ *ccpo.fixp Sup* (≤) *f* = *lfp f*
⟨*proof*⟩

lemma *ccpo-gfp-simp*: ⋀*f*. *mono f* ⟹ *ccpo.fixp Inf* (≥) *f* = *gfp f*
⟨*proof*⟩

abbreviation *chain-admissible P* ≡ *ccpo.admissible Sup* (≤) *P*
abbreviation *is-chain* ≡ *Complete-Partial-Order.chain* (≤)
lemmas *chain-admissibleI*[*intro?*] = *ccpo.admissibleI*[**where** *lub*=*Sup* **and** *ord*=(≤)]

abbreviation *dual-chain-admissible P* ≡ *ccpo.admissible Inf* (λ*x y*. *y*≤*x*) *P*
abbreviation *is-dual-chain* ≡ *Complete-Partial-Order.chain* (λ*x y*. *y*≤*x*)
lemmas *dual-chain-admissibleI*[*intro?*] =
ccpo.admissibleI[**where** *lub*=*Inf* **and** *ord*=(λ*x y*. *y*≤*x*)]

lemma *dual-chain-iff*[*simp*]: *is-dual-chain C = is-chain C*
⟨*proof*⟩

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

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

lemma *point-chainI*: *is-chain M* ⟹ *is-chain* ((λ*f*. *f x*)‘*M*)
⟨*proof*⟩

We transfer the admissible induction lemmas to complete lattices.

lemma *lfp-cadm-induct*:
⟦*chain-admissible P*; *P* (*Sup* {}); *mono f*; ⋀*x*. *P x* ⟹ *P* (*f x*)⟧ ⟹ *P* (*lfp f*)
⟨*proof*⟩

lemma *gfp-cadm-induct*:
⟦*dual-chain-admissible P*; *P* (*Inf* {}); *mono f*; ⋀*x*. *P x* ⟹ *P* (*f x*)⟧ ⟹ *P* (*gfp f*)
⟨*proof*⟩

Continuity and Kleene Fixed Point Theorem

definition *cont f* ≡ ∀ *C*. *C*≠{} ⟶ *f* (*Sup C*) = *Sup* (*f*‘*C*)
definition *strict f* ≡ *f bot* = *bot*
definition *inf-distrib f* ≡ *strict f* ∧ *cont f*

lemma *contI*[*intro?*]: ⟦⋀*C*. *C*≠{} ⟹ *f* (*Sup C*) = *Sup* (*f*‘*C*)⟧ ⟹ *cont f*
⟨*proof*⟩
lemma *contD*: *cont f* ⟹ *C*≠{} ⟹ *f* (*Sup C*) = *Sup* (*f* ‘ *C*)
⟨*proof*⟩
lemma *contD'*: *cont f* ⟹ *C*≠{} ⟹ *f* (*Sup C*) = *Sup* (*f* ‘ *C*)
⟨*proof*⟩

lemma *strictD*[*dest*]: *strict f* ⟹ *f bot* = *bot*
⟨*proof*⟩
lemma *strictD-simp*[*simp*]: *strict f* ⟹ *f* (*bot*::*'a*::*bot*) = (*bot*::*'a*)
⟨*proof*⟩

lemma *strictI*[*intro?*]: *f bot* = *bot* ⟹ *strict f*
⟨*proof*⟩

lemma *inf-distribD*[*simp*]:
inf-distrib f ⟹ *strict f*
inf-distrib f ⟹ *cont f*

⟨*proof*⟩

lemma *inf-distribI*[*intro?*]: ⟦*strict f*; *cont f*⟧ ⟹ *inf-distrib f*
⟨*proof*⟩

lemma *inf-distribD'*[*simp*]:
fixes *f* :: *'a*::*complete-lattice* ⇒ *'b*::*complete-lattice*
shows *inf-distrib f* ⟹ *f* (*Sup C*) = *Sup* (*f*‘*C*)
⟨*proof*⟩

lemma *inf-distribI'*:
fixes *f* :: *'a*::*complete-lattice* ⇒ *'b*::*complete-lattice*
assumes *B*: ⋀*C*. *f* (*Sup C*) = *Sup* (*f*‘*C*)
shows *inf-distrib f*
⟨*proof*⟩

lemma *cont-is-mono*[*simp*]:
fixes *f* :: *'a*::*complete-lattice* ⇒ *'b*::*complete-lattice*
shows *cont f* ⟹ *mono f*
⟨*proof*⟩

lemma *inf-distrib-is-mono*[*simp*]:
fixes *f* :: *'a*::*complete-lattice* ⇒ *'b*::*complete-lattice*
shows *inf-distrib f* ⟹ *mono f*
⟨*proof*⟩

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*⟩

theorem *kleene-lfp*:
fixes *f*:: *'a*::*complete-lattice* ⇒ *'a*
assumes *CONT*: *cont f*
shows *lfp f* = (*SUP i*. (*f*^^*i*) *bot*)
⟨*proof*⟩

theorem
fixes *f*:: *'a*::*complete-lattice* ⇒ *'a*
assumes *CONT*: *cont f*
shows *lfp f* = (*SUP i*. (*f*^^*i*) *bot*)
⟨*proof*⟩

lemma *SUP-funpow-contracting*:
fixes *f* :: *'a* ⇒ (*'a*::*complete-lattice*)
assumes *C*: *cont f*

shows $f\ (SUP\ i.\ (f\widehat{\ }\widehat{\ }i)\ m) \leq (SUP\ i.\ (f\widehat{\ }\widehat{\ }i)\ m)$
⟨*proof*⟩

lemma *gen-kleene-chain-conv*:
fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'a$
assumes C: *cont f*
shows $(SUP\ i.\ (f\widehat{\ }\widehat{\ }i)\ m) = (SUP\ i.\ ((\lambda x.\ sup\ m\ (f\ x))\widehat{\ }\widehat{\ }i)\ bot)$
⟨*proof*⟩

theorem
assumes C: *cont f*
shows $lfp\ (\lambda x.\ sup\ m\ (f\ x)) = (SUP\ i.\ (f\widehat{\ }\widehat{\ }i)\ m)$
⟨*proof*⟩

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

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

2.1.4 Maps

Key-Value Set

lemma *map-to-set-simps*[*simp*]:
$map\text{-}to\text{-}set\ Map.empty = \{\}$
$map\text{-}to\text{-}set\ [a\mapsto b] = \{(a,b)\}$
$map\text{-}to\text{-}set\ (m|`K) = map\text{-}to\text{-}set\ m \cap K\times UNIV$
$map\text{-}to\text{-}set\ (m(x:=None)) = map\text{-}to\text{-}set\ m - \{x\}\times UNIV$
$map\text{-}to\text{-}set\ (m(x\mapsto v)) = map\text{-}to\text{-}set\ m - \{x\}\times UNIV \cup \{(x,v)\}$
$map\text{-}to\text{-}set\ m \cap dom\ m\times UNIV = map\text{-}to\text{-}set\ m$
$m\ k = Some\ v \Longrightarrow (k,v)\in map\text{-}to\text{-}set\ m$
$single\text{-}valued\ (map\text{-}to\text{-}set\ m)$
⟨*proof*⟩

lemma *map-to-set-inj*:
$(k,v)\in map\text{-}to\text{-}set\ m \Longrightarrow (k,v')\in map\text{-}to\text{-}set\ m \Longrightarrow v = v'$
⟨*proof*⟩

end

2.2 Transfer between Domains

theory *RefineG-Transfer*
imports *../Refine-Misc*
begin

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

definition *REFINEG-TRANSFER-POST-SIMP x y* ≡ *x=y*
definition [*simp*]: *REFINEG-TRANSFER-ALIGN x y == True*
lemma *REFINEG-TRANSFER-ALIGNI*: *REFINEG-TRANSFER-ALIGN x y* ⟨*proof*⟩

lemma *START-REFINEG-TRANSFER*:
assumes *REFINEG-TRANSFER-ALIGN d c*
assumes *c*≤*a*
assumes *REFINEG-TRANSFER-POST-SIMP c d*
shows *d*≤*a*
⟨*proof*⟩

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

⟨*proof*⟩

⟨*ML*⟩

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* ⟹ α *s1* ≤ *S1*
assumes ¬*b* ⟹ α *s2* ≤ *S2*
shows α (*if b then s1 else s2*) ≤ (*if b then S1 else S2*)
⟨*proof*⟩

lemma *transfer-prod*[*refine-transfer*]:
assumes ⋀*a b*. α (*f a b*) ≤ *F a b*
shows α (*case-prod f x*) ≤ (*case-prod F x*)
⟨*proof*⟩

lemma *transfer-Let*[*refine-transfer*]:
assumes ⋀*x*. α (*f x*) ≤ *F x*
shows α (*Let x f*) ≤ *Let x F*
⟨*proof*⟩

lemma *transfer-option*[*refine-transfer*]:
assumes α *fa* ≤ *Fa*
assumes ⋀*x*. α (*fb x*) ≤ *Fb x*
shows α (*case-option fa fb x*) ≤ *case-option Fa Fb x*
⟨*proof*⟩

lemma *transfer-sum*[*refine-transfer*]:
assumes ⋀*l*. α (*fl l*) ≤ *Fl l*

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

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

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 $\alpha\ (rec\text{-}list\ fn\ fc\ l\ s) \leq rec\text{-}list\ fn'\ fc'\ l\ s$
⟨*proof*⟩

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 $\alpha\ (rec\text{-}nat\ fn\ fs\ n\ s) \leq rec\text{-}nat\ fn'\ fs'\ n\ s$
⟨*proof*⟩

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*: $\bigwedge A.\ is\text{-}chain\ A \Longrightarrow \alpha\ (Sup\ A) = Sup\ (\alpha`A)$
begin
lemma α-*mono*[*simp*, *intro*!]: *mono* α
⟨*proof*⟩

lemma α-*strict*[*simp*]: $\alpha\ bot = bot$
⟨*proof*⟩
end

Transfer into ccpo

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

Transfer into ccpo with distributive transfer function.

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

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

lemma $\alpha\text{-}mono[simp, intro!]$: $mono\ \alpha$
⟨*proof*⟩

lemma $\alpha\text{-}strict[simp]$: $\alpha\ (Sup\ \{\}) = bot$
⟨*proof*⟩
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* $\{y\ .\ \exists f{\in}F.\ y = f\ x\}$
⟨*proof*⟩

lemma *ccpo-lift*:
assumes *class.ccpo lub le lt*
shows *class.ccpo* (*fun-lub lub*) (*fun-ord le*) (*mk-less* (*fun-ord le*))
⟨*proof*⟩

lemma *fun-lub-simps*[*simp*]:
fun-lub lub $\{\} = (\lambda x.\ lub\ \{\})$
fun-lub lub $\{f\} = (\lambda x.\ lub\ \{f\ x\})$
⟨*proof*⟩

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** $x{\neq}b$ **and** $C{=}\{x\}$
| x **where** $x{\neq}b$ **and** $C{=}\{b,x\}$
⟨*proof*⟩

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

flat-lub b (insert b X) = flat-lub b X
⟨*proof*⟩

lemma *flat-ord-simps*[*simp*]:
flat-ord b b x
⟨*proof*⟩

interpretation *flat-ord*: *ccpo flat-lub b flat-ord b mk-less (flat-ord b)*
⟨*proof*⟩

interpretation *flat-le-mono-setup*: *mono-setup-loc flat-ord b*
⟨*proof*⟩

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)*
⟨*proof*⟩

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*⟩

lemma *flatf-admissible-pointwise*:
(⋀*x. P x b*) ⟹
ccpo.admissible (flatf-lub b) (flatf-ord b) (λg. ∀ x. P x (g x))
⟨*proof*⟩

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*⟩

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*) ($fp'\ x'$)
⟨*proof*⟩

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-lattice*
shows
flat-le x y $\Longrightarrow x \leq y$
flat-ge x y $\Longrightarrow x \geq y$
⟨*proof*⟩

lemma *flatf-ord-compat*:
fixes $x\ y :: 'a \Rightarrow ('b :: \textit{complete-lattice})$
shows
flatf-le $x\ y \Longrightarrow x \leq y$
flatf-ge $x\ y \Longrightarrow x \geq y$
⟨*proof*⟩

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*⟩

lemma *gfp-eq-flatf-gfp*:
assumes *FM*: *flatf-mono-ge B* **and** *M*: *mono B*
shows *gfp B* = *flatf-gfp B*
⟨*proof*⟩

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*⟩

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* $B'\ x'$)
⟨*proof*⟩

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* $B'\ x'$)
⟨*proof*⟩

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

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

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

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

end

2.4 Generic Recursion Combinator for Complete Lattice Structured Domains

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

We define a recursion combinator that asserts monotonicity.

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

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*
∨ (*flatf-fp b1 B x* = *b1* ∧ *flatf-fp b2 B x* = *b2*)
⟨*proof*⟩

lemma *flat-ord-top*[*simp*]: *flat-ord b b x* ⟨*proof*⟩

lemma *lfp-gfp-compare*:
— Least and greatest fixed point are either equal, or bot and top
assumes *MLE*: *flatf-mono-le B* **and** *MGE*: *flatf-mono-ge B*
shows *flatf-lfp B x* = *flatf-gfp B x*
∨ (*flatf-lfp B x* = *bot* ∧ *flatf-gfp B x* = *top*)
⟨*proof*⟩

definition *trimono* :: ((′*a* ⇒ ′*b*) ⇒ ′*a* ⇒ (′*b*::{*bot*,*order*,*top*})) ⇒ *bool*
where *trimono B* ≡ *flatf-mono-ge B* ∧ *mono B*
lemma *trimonoI*[*refine-mono*]:
⟦*flatf-mono-ge B*; *mono B*⟧ ⟹ *trimono B*
⟨*proof*⟩

lemma *trimono-trigger*: *trimono B* ⟹ *trimono B* ⟨*proof*⟩

⟨*ML*⟩

lemma *trimonoD-flatf-ge*: *trimono B* ⟹ *flatf-mono-ge B*
⟨*proof*⟩

lemma *trimonoD-mono*: *trimono B* ⟹ *mono B*
⟨*proof*⟩

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

definition *triords* ≡ {*flat-ge*,(≤)}
lemma *trimono-alt*:
trimono B ⟷ (∀ *ord*∈*fun-ord*‘*triords*. *monotone ord ord B*)
⟨*proof*⟩

lemma *trimonoI'*:
assumes $\bigwedge ord.\ ord \in triords \Longrightarrow monotone\ (fun\text{-}ord\ ord)\ (fun\text{-}ord\ ord)\ B$
shows *trimono B*
⟨*proof*⟩

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

lemma *RECT-gfp-def*: $RECT\ B\ x =$
(*if* (*trimono B*) *then* (*gfp B x*) *else* (*top*::'*a*::*complete-lattice*))
⟨*proof*⟩

lemma *REC-unfold*: $trimono\ B \Longrightarrow REC\ B = B\ (REC\ B)$
⟨*proof*⟩

lemma *RECT-unfold*: $[\![trimono\ B]\!] \Longrightarrow RECT\ B = B\ (RECT\ B)$
⟨*proof*⟩

lemma *REC-mono*[*refine-mono*]:
assumes [*simp*]: *trimono B*
assumes *LE*: $\bigwedge F\ x.\ (B\ F\ x) \leq (B'\ F\ x)$
shows $(REC\ B\ x) \leq (REC\ B'\ x)$
⟨*proof*⟩

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

lemma *REC-le-RECT*: $REC\ body\ x \leq RECT\ body\ x$
⟨*proof*⟩

print-statement *flatf-fp-induct-pointwise*
theorem *lfp-induct-pointwise*:
fixes *a*::'*a*
assumes *ADM1*: $\bigwedge a\ x.\ chain\text{-}admissible\ (\lambda b.\ \forall\, a\ x.\ pre\ a\ x \longrightarrow post\ a\ x\ (b\ x))$
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*⟩

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* ≤ *M arb*′ *x*; *pre arb x*; *f* ≤ *REC body*
⟧ $\Longrightarrow$ *body f x* ≤ *M arb x*
shows *REC body x* ≤ *M arb x*
⟨*proof*⟩

lemma *RECT-rule-arb*:
assumes *M*: *trimono body*
assumes *WF*: *wf* (*V*::(′*x*×′*x*) *set*)
assumes *I0*: *pre* (*arb*::′*arb*) (*x*::′*x*)
assumes *IS*: $\bigwedge$*f arb x*. ⟦
$\bigwedge$*arb*′ *x*′. ⟦*pre arb*′ *x*′; (*x*′,*x*)∈*V*⟧ $\Longrightarrow$ *f x*′ ≤ *M arb*′ *x*′;
pre arb x;
RECT body = *f*
⟧ $\Longrightarrow$ *body f x* ≤ *M arb x*
shows *RECT body x* ≤ *M arb x*
⟨*proof*⟩

lemma *REC-rule*:
fixes *x*::′*x*
assumes *M*: *trimono body*
assumes *I0*: *pre x*
assumes *IS*: $\bigwedge$*f x*. ⟦ $\bigwedge$*x*. *pre x* $\Longrightarrow$ *f x* ≤ *M x*; *pre x*; *f* ≤ *REC body* ⟧
$\Longrightarrow$ *body f x* ≤ *M x*
shows *REC body x* ≤ *M x*
⟨*proof*⟩

lemma *RECT-rule*:
assumes *M*: *trimono body*
assumes *WF*: *wf* (*V*::(′*x*×′*x*) *set*)
assumes *I0*: *pre* (*x*::′*x*)
assumes *IS*: $\bigwedge$*f x*. ⟦ $\bigwedge$*x*′. ⟦*pre x*′; (*x*′,*x*)∈*V*⟧ $\Longrightarrow$ *f x*′ ≤ *M x*′; *pre x*;
RECT body = *f*
⟧ $\Longrightarrow$ *body f x* ≤ *M x*
shows *RECT body x* ≤ *M x*
⟨*proof*⟩

lemma *REC-rule-arb2*:
assumes *M*: *trimono body*
assumes *I0*: *pre* (*arb*::′*arb*) (*arc*::′*arc*) (*x*::′*x*)
assumes *IS*: ⋀*f arb arc x*. ⟦
⋀*arb*′ *arc*′ *x*′. ⟦*pre arb*′ *arc*′ *x*′ ⟧ ⟹ *f x*′ ≤ *M arb*′ *arc*′ *x*′;
pre arb arc x
⟧ ⟹ *body f x* ≤ *M arb arc x*
shows *REC body x* ≤ *M arb arc x*
⟨*proof*⟩

lemma *REC-rule-arb3*:
assumes *M*: *trimono body*
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*′⟧ ⟹ *f x*′ ≤ *M arb*′ *arc*′ *ard*′ *x*′;
pre arb arc ard x
⟧ ⟹ *body f x* ≤ *M arb arc ard x*
shows *REC body x* ≤ *M arb arc ard x*
⟨*proof*⟩

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*: ⋀*f arb arc x*. ⟦
⋀*arb*′ *arc*′ *x*′. ⟦*pre arb*′ *arc*′ *x*′; (*x*′,*x*)∈*V*⟧ ⟹ *f x*′ ≤ *M arb*′ *arc*′ *x*′;
pre arb arc x;
f ≤ *RECT body*
⟧ ⟹ *body f x* ≤ *M arb arc x*
shows *RECT body x* ≤ *M arb arc x*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *RECT-eq-REC*:
— Partial and total correct recursion are equal if total recursion does not fail.
assumes *NT*: $RECT\ body\ x \neq top$
shows $RECT\ body\ x = REC\ body\ x$
⟨*proof*⟩

lemma *RECT-eq-REC-tproof*:
— Partial and total correct recursion are equal if we can provide a termination proof.
fixes $a :: 'a$
assumes *M*: *trimono body*
assumes *WF*: *wf V*
assumes *I0*: *pre a x*
assumes *IS*: $\bigwedge f\ arb\ x.$
$[\![\bigwedge arb'\ x'.\ [\![pre\ arb'\ x';\ (x',\ x) \in V]\!] \Longrightarrow f\ x' \leq M\ arb'\ x';$
$pre\ arb\ x;\ REC_T\ body = f]\!]$
$\Longrightarrow body\ f\ x \leq M\ arb\ x$
assumes *NT*: $M\ a\ x \neq top$
shows $RECT\ body\ x = REC\ body\ x \wedge RECT\ body\ x \leq M\ a\ x$
⟨*proof*⟩

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 $\alpha\ (fr\ x) \leq RECT\ B\ x$
⟨*proof*⟩

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 $\alpha\ (RECT\ b\ x) \leq RECT\ B\ x$
⟨*proof*⟩

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 $\alpha\ (REC\ b\ x) \leq REC\ B\ x$
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *ASSUME-simps*[*simp*,*code*]:
ASSUME True = *return* ()
ASSUME False = *bot*
⟨*proof*⟩

lemma *le-ASSERTI*: $[\![\ \Phi \Longrightarrow M \leq M'\ ]\!] \Longrightarrow M \leq bind\ (ASSERT\ \Phi)\ (\lambda\text{-}.\ M')$

⟨*proof*⟩

lemma *le-ASSERTI-pres*: ⟦ Φ ⟹ *M*≤*bind* (*ASSERT* Φ) (λ-. *M′*) ⟧
⟹ *M* ≤ *bind* (*ASSERT* Φ) (λ-. *M′*)
⟨*proof*⟩

lemma *ASSERT-leI*: ⟦ Φ; Φ ⟹ *M*≤*M′* ⟧ ⟹ *bind* (*ASSERT* Φ) (λ-. *M*) ≤ *M′*
⟨*proof*⟩

lemma *ASSUME-leI*: ⟦ Φ ⟹ *M*≤*M′* ⟧ ⟹ *bind* (*ASSUME* Φ) (λ-. *M*) ≤ *M′*
⟨*proof*⟩
lemma *ASSUME-leI-pres*: ⟦ Φ ⟹ *bind* (*ASSUME* Φ) (λ-. *M*)≤*M′* ⟧
⟹ *bind* (*ASSUME* Φ) (λ-. *M*) ≤ *M′*
⟨*proof*⟩
lemma *le-ASSUMEI*: ⟦ Φ; Φ ⟹ *M*≤*M′* ⟧ ⟹ *M* ≤ *bind* (*ASSUME* Φ) (λ-. *M′*)
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *ASSUME-le-iff*:
bind (*ASSUME* Φ) (λ-. *S*) ≤ *S′* ⟷ (Φ ⟶ *S*≤*S′*)
⟨*proof*⟩

lemma *le-ASSERT-iff*:
S ≤ *bind* (*ASSERT* Φ) (λ-. *S′*) ⟷ (Φ ⟶ *S*≤*S′*)
⟨*proof*⟩

lemma *le-ASSUME-iff*:
S ≤ *bind* (*ASSUME* Φ) (λ-. *S′*) ⟷ (*S*≠*bot* ⟶ Φ) ∧ *S*≤*S′*
⟨*proof*⟩
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'*))
⟨*proof*⟩

lemma *transfer-ASSUME*[*refine-transfer*]:
⟦Φ; Φ ⟹ *α M* ≤ *M'*⟧
⟹ *α* (*cbind* (*cASSUME* Φ) (λ-. *M*)) ≤ (*abind* (*aASSUME* Φ) (λ-. *M'*))
⟨*proof*⟩

end

locale *transfer-generic-Assert-remove* =
a: *generic-Assert abind areturn aASSERT aASSUME* +
transfer α
for *abind* :: (*'mua*::*complete-lattice*)
⇒ (*unit*⇒*'maa*) ⇒ (*'maa*::*complete-lattice*)
and *areturn* :: *unit* ⇒ *'mua* **and** *aASSERT* **and** *aASSUME*
and *α* :: *'mac* ⇒ *'maa*
begin
lemma *transfer-ASSERT-remove*[*refine-transfer*]:
⟦ Φ ⟹ *α M* ≤ *M'* ⟧ ⟹ *α M* ≤ *abind* (*aASSERT* Φ) (λ-. *M'*)
⟨*proof*⟩

lemma *transfer-ASSUME-remove*[*refine-transfer*]:
⟦ Φ; Φ ⟹ *α M* ≤ *M'* ⟧ ⟹ *α M* ≤ *abind* (*aASSUME* Φ) (λ-. *M'*)
⟨*proof*⟩
end

end

2.6 Basic Concepts

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

2.6.1 Nondeterministic Result Lattice and Monad

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

In addition to the return operator, we also introduce the operator *RES*, that embeds a set of results into our lattice. Its synonym for a predicate is *SPEC*.

Program correctness is expressed by refinement, i.e., the expression $M \leq SPEC\ \Phi$ means that M is correct w.r.t. specification Φ. This suggests the following view on the program lattice: The top-element is the result that is never correct. We call this result *FAIL*. The bottom element is the program that is always correct. It is called *SUCCEED*. An assertion can be encoded by failing if the asserted predicate is not true. Symmetrically, an assumption is encoded by succeeding if the predicate is not true.

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

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

instantiation *nres* :: (*type*) *complete-lattice*
begin
fun *less-eq-nres* **where**
- ≤ *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
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *nres-simp-internals*:
RES {} = *SUCCEED*
FAILi = *FAIL*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*

$\bigwedge X\ f.\ Sup\ (f\ `\ X) = FAIL \longleftrightarrow FAIL \in f\ `\ X$
$\bigwedge X.\ FAIL = Sup\ X \longleftrightarrow FAIL \in X$
$\bigwedge X\ f.\ FAIL = Sup\ (f\ `\ X) \longleftrightarrow FAIL \in f\ `\ X$
$\bigwedge X.\ FAIL \in X \Longrightarrow Sup\ X = FAIL$
$\bigwedge X.\ FAIL \in f\ `\ X \Longrightarrow Sup\ (f\ `\ X) = FAIL$
$\bigwedge A.\ Sup\ (RES\ `\ A) = RES\ (Sup\ A)$
$\bigwedge A.\ Sup\ (RES\ `\ A) = RES\ (Sup\ A)$
$\bigwedge A.\ A \neq \{\} \Longrightarrow Inf\ (RES `A) = RES\ (Inf\ A)$
$\bigwedge A.\ A \neq \{\} \Longrightarrow Inf\ (RES\ `\ A) = RES\ (Inf\ A)$
$Inf\ \{\} = FAIL$
$Inf\ UNIV = SUCCEED$
$Sup\ \{\} = SUCCEED$
$Sup\ UNIV = FAIL$
$\bigwedge x\ y.\ RETURN\ x \leq RETURN\ y \longleftrightarrow x = y$
$\bigwedge x\ Y.\ RETURN\ x \leq RES\ Y \longleftrightarrow x \in Y$
$\bigwedge X\ y.\ RES\ X \leq RETURN\ y \longleftrightarrow X \subseteq \{y\}$
⟨*proof*⟩

lemma *Sup-eq-RESE*:
assumes $Sup\ A = RES\ B$
obtains C **where** $A = RES `C$ **and** $B = Sup\ C$
⟨*proof*⟩

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

Pointwise Reasoning

⟨*ML*⟩

definition $nofail\ S \equiv S \neq FAIL$
definition $inres\ S\ x \equiv RETURN\ x \leq S$

lemma *nofail-simps*[*simp*, *refine-pw-simps*]:
$nofail\ FAIL \longleftrightarrow False$
$nofail\ (RES\ X) \longleftrightarrow True$
$nofail\ (RETURN\ x) \longleftrightarrow True$
$nofail\ SUCCEED \longleftrightarrow True$
⟨*proof*⟩

lemma *inres-simps*[*simp*, *refine-pw-simps*]:
$inres\ FAIL = (\lambda\text{-}.\ True)$
$inres\ (RES\ X) = (\lambda x.\ x \in X)$
$inres\ (RETURN\ x) = (\lambda y.\ x = y)$
$inres\ SUCCEED = (\lambda\text{-}.\ False)$
⟨*proof*⟩

lemma *not-nofail-iff*:
$\neg nofail\ S \longleftrightarrow S = FAIL$ ⟨*proof*⟩

lemma *not-nofail-inres*[*simp*, *refine-pw-simps*]:
$\neg nofail\ S \Longrightarrow inres\ S\ x$
⟨*proof*⟩

lemma *intro-nofail*[*refine-pw-simps*]:
$S \neq FAIL \longleftrightarrow nofail\ S$
$FAIL \neq S \longleftrightarrow nofail\ S$
⟨*proof*⟩

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

lemma *pw-le-iff*:
$S \leq S' \longleftrightarrow (nofail\ S' \longrightarrow (nofail\ S \wedge (\forall x.\ inres\ S\ x \longrightarrow inres\ S'\ x)))$
⟨*proof*⟩

lemma *pw-eq-iff*:
$S = S' \longleftrightarrow (nofail\ S = nofail\ S' \wedge (\forall x.\ inres\ S\ x \longleftrightarrow inres\ S'\ x))$
⟨*proof*⟩

lemma *pw-flat-le-iff*: *flat-le S S'* $\longleftrightarrow$
$(\exists x.\ inres\ S\ x) \longrightarrow (nofail\ S \longleftrightarrow nofail\ S') \wedge (\forall x.\ inres\ S\ x \longleftrightarrow inres\ S'\ x)$
⟨*proof*⟩

lemma *pw-flat-ge-iff*: *flat-ge S S'* $\longleftrightarrow$
$(nofail\ S) \longrightarrow nofail\ S' \wedge (\forall x.\ inres\ S\ x \longleftrightarrow inres\ S'\ x)$
⟨*proof*⟩

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

lemma *pw-leI*:
$(nofail\ S' \longrightarrow (nofail\ S \wedge (\forall x.\ inres\ S\ x \longrightarrow inres\ S'\ x))) \Longrightarrow S \leq S'$
⟨*proof*⟩

lemma *pw-leI'*:
assumes $nofail\ S' \Longrightarrow nofail\ S$
assumes $\bigwedge x.\ [\![nofail\ S';\ inres\ S\ x]\!] \Longrightarrow inres\ S'\ x$
shows $S \leq S'$
⟨*proof*⟩

lemma *pw-eqI*:
assumes $nofail\ S = nofail\ S'$
assumes $\bigwedge x.\ inres\ S\ x \longleftrightarrow inres\ S'\ x$
shows $S = S'$
⟨*proof*⟩

lemma *pwD1*:
assumes $S \leq S' \quad nofail\ S'$
shows $nofail\ S$
⟨*proof*⟩

lemma *pwD2*:
assumes $S \leq S'$ *inres* $S\ x$
shows *inres* $S'\ x$
⟨*proof*⟩

lemmas *pwD* = *pwD1 pwD2*

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

lemma *le-nofailI*: $[\![\ \mathit{nofail}\ M' \Longrightarrow M \leq M'\ ]\!] \Longrightarrow M \leq M'$
⟨*proof*⟩

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

lemma *pw-sup-nofail*[*refine-pw-simps*]:
$\mathit{nofail}\ (\mathit{sup}\ a\ b) \longleftrightarrow \mathit{nofail}\ a \wedge \mathit{nofail}\ b$
⟨*proof*⟩

lemma *pw-sup-inres*[*refine-pw-simps*]:
$\mathit{inres}\ (\mathit{sup}\ a\ b)\ x \longleftrightarrow \mathit{inres}\ a\ x \vee \mathit{inres}\ b\ x$
⟨*proof*⟩

lemma *pw-Sup-inres*[*refine-pw-simps*]: $\mathit{inres}\ (\mathit{Sup}\ X)\ r \longleftrightarrow (\exists\, M{\in}X.\ \mathit{inres}\ M\ r)$
⟨*proof*⟩

lemma *pw-SUP-inres* [*refine-pw-simps*]: $\mathit{inres}\ (\mathit{Sup}\ (f\ `\ X))\ r \longleftrightarrow (\exists\, M{\in}X.\ \mathit{inres}\ (f\ M)\ r)$
⟨*proof*⟩

lemma *pw-Sup-nofail*[*refine-pw-simps*]: $\mathit{nofail}\ (\mathit{Sup}\ X) \longleftrightarrow (\forall\, x{\in}X.\ \mathit{nofail}\ x)$
⟨*proof*⟩

lemma *pw-SUP-nofail* [*refine-pw-simps*]: $\mathit{nofail}\ (\mathit{Sup}\ (f\ `\ X)) \longleftrightarrow (\forall\, x{\in}X.\ \mathit{nofail}\ (f\ x))$
⟨*proof*⟩

lemma *pw-inf-nofail*[*refine-pw-simps*]:
$\mathit{nofail}\ (\mathit{inf}\ a\ b) \longleftrightarrow \mathit{nofail}\ a \vee \mathit{nofail}\ b$
⟨*proof*⟩

lemma *pw-inf-inres*[*refine-pw-simps*]:
$\mathit{inres}\ (\mathit{inf}\ a\ b)\ x \longleftrightarrow \mathit{inres}\ a\ x \wedge \mathit{inres}\ b\ x$
⟨*proof*⟩

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

lemma *pw-INF-nofail* [*refine-pw-simps*]: *nofail* (*Inf* (*f* ‘ *C*)) ⟷ (∃ *x*∈*C*. *nofail* (*f x*))
⟨*proof*⟩

lemma *pw-Inf-inres*[*refine-pw-simps*]: *inres* (*Inf C*) *r* ⟷ (∀ *M*∈*C*. *inres M r*)
⟨*proof*⟩

lemma *pw-INF-inres* [*refine-pw-simps*]: *inres* (*Inf* (*f* ‘ *C*)) *r* ⟷ (∀ *M*∈*C*. *inres* (*f M*) *r*)
⟨*proof*⟩

lemma *nofail-RES-conv*: *nofail m* ⟷ (∃ *M*. *m*=*RES M*) ⟨*proof*⟩

primrec *the-RES* **where** *the-RES* (*RES X*) = *X*
lemma *the-RES-inv*[*simp*]: *nofail m* ⟹ *RES* (*the-RES m*) = *m*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *nf-inres-SPEC*[*simp*]: *nf-inres* (*SPEC* Φ) *x* ⟷ Φ *x*
⟨*proof*⟩

lemma *nofail-antimono-fun*: *f* ≤ *g* ⟹ (*nofail* (*g x*) ⟶ *nofail* (*f x*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bind-SUCCEED*[*simp*]: *bind SUCCEED f* = *SUCCEED*
⟨*proof*⟩

lemma *bind-RES*: *bind* (*RES X*) *f* = *Sup* (*f*‘*X*) ⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *pw-bind-inres*[*refine-pw-simps*]:
inres (*bind M f*) = (λ*x*. *nofail M* ⟶ (∃ *y*. (*inres M y* ∧ *inres* (*f y*) *x*)))
⟨*proof*⟩

lemma *pw-bind-le-iff*:
bind M f ≤ *S* ⟷ (*nofail S* ⟶ *nofail M*) ∧
(∀ *x*. *nofail M* ∧ *inres M x* ⟶ *f x* ≤ *S*)
⟨*proof*⟩

lemma *pw-bind-leI*: ⟦
nofail S ⟹ *nofail M*; ⋀*x*. ⟦*nofail M*; *inres M x*⟧ ⟹ *f x* ≤ *S*⟧
⟹ *bind M f* ≤ *S*
⟨*proof*⟩

lemma *nres-monad1*[*simp*]: *bind* (*RETURN x*) *f* = *f x*
⟨*proof*⟩
lemma *nres-monad2*[*simp*]: *bind M RETURN* = *M*
⟨*proof*⟩
lemma *nres-monad3*[*simp*]: *bind* (*bind M f*) *g* = *bind M* (λ*x*. *bind* (*f x*) *g*)
⟨*proof*⟩
lemmas *nres-monad-laws* = *nres-monad1 nres-monad2 nres-monad3*

lemma *bind-cong*:
assumes *m*=*m*′
assumes ⋀*x*. *RETURN x* ≤ *m*′ ⟹ *f x* = *f*′ *x*
shows *bind m f* = *bind m*′ *f*′
⟨*proof*⟩

lemma *bind-mono*[*refine-mono*]:
⟦ *M* ≤ *M*′; ⋀*x*. *RETURN x* ≤ *M* ⟹ *f x* ≤ *f*′ *x* ⟧ ⟹ *bind M f* ≤ *bind M*′ *f*′

⟦ *flat-ge M M*′; ⋀*x*. *flat-ge* (*f x*) (*f*′ *x*) ⟧ ⟹ *flat-ge* (*bind M f*) (*bind M*′ *f*′)
⟨*proof*⟩

lemma *bind-mono1*[*simp*, *intro*!]: *mono* (λ*M*. *bind M f*)
⟨*proof*⟩

lemma *bind-mono1*′[*simp*, *intro*!]: *mono bind*
⟨*proof*⟩

lemma *bind-mono2*′[*simp*, *intro*!]: *mono* (*bind M*)
⟨*proof*⟩

lemma *bind-distrib-sup1*: *bind* (*sup M N*) *f* = *sup* (*bind M f*) (*bind N f*)
⟨*proof*⟩

lemma *bind-distrib-sup2*: *bind m* (λ*x*. *sup* (*f x*) (*g x*)) = *sup* (*bind m f*) (*bind m g*)
⟨*proof*⟩

lemma *bind-distrib-Sup1*: *bind* (*Sup M*) *f* = (*SUP m*∈*M*. *bind m f*)
⟨*proof*⟩

lemma *bind-distrib-Sup2*: *F*≠{} ⟹ *bind m* (*Sup F*) = (*SUP f*∈*F*. *bind m f*)
⟨*proof*⟩

lemma *RES-Sup-RETURN*: *Sup* (*RETURN*‘*X*) = *RES X*
⟨*proof*⟩

2.6.2 VCG Setup

lemma *SPEC-cons-rule*:
assumes *m* ≤ *SPEC* Φ
assumes ⋀*x*. Φ *x* ⟹ Ψ *x*
shows *m* ≤ *SPEC* Ψ
⟨*proof*⟩

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

⟨*ML*⟩

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

2.6.3 Data Refinement

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

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

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

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

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

definition *conc-fun* ($\Downarrow$) **where**
conc-fun R m $\equiv$ *case m of FAILi* $\Rightarrow$ *FAIL* | *RES X* $\Rightarrow$ *RES* $(R^{-1}\text{``}X)$

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* $(R\text{``}X)$ *else FAIL*

lemma
conc-fun-FAIL[*simp*]: $\Downarrow R$ *FAIL* = *FAIL* **and**
conc-fun-RES: $\Downarrow R$ *(RES X)* = *RES* $(R^{-1}\text{``}X)$
⟨*proof*⟩

lemma *abs-fun-simps*[*simp*]:
$\Uparrow R$ *FAIL* = *FAIL*
$X \subseteq$ *Domain R* $\Longrightarrow$ $\Uparrow R$ *(RES X)* = *RES* $(R\text{``}X)$
$\neg(X \subseteq$ *Domain R*$) \Longrightarrow$ $\Uparrow R$ *(RES X)* = *FAIL*
⟨*proof*⟩

context fixes *R* **assumes** *SV*: *single-valued R* **begin**
lemma *conc-abs-swap*: $m' \leq \Downarrow R\ m \longleftrightarrow \Uparrow R\ m' \leq m$
⟨*proof*⟩

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

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*))
⟨*proof*⟩

lemma *pw-abs-inres*[*refine-pw-simps*]:
inres $(\Uparrow R\ M)\ a \longleftrightarrow$ (*nofail* $(\Uparrow R\ M) \longrightarrow (\exists c.$ *inres M c* $\wedge$ $(c,a) \in R)$)
⟨*proof*⟩

lemma *pw-conc-nofail*[*refine-pw-simps*]:
nofail $(\Downarrow R\ S)$ = *nofail S*
⟨*proof*⟩

lemma *pw-conc-inres*[*refine-pw-simps*]:
inres $(\Downarrow R\ S') = (\lambda s.$ *nofail S'*
$\longrightarrow (\exists s'.\ (s,s') \in R \wedge$ *inres S' s'*$))$
⟨*proof*⟩

lemma *abs-fun-strict*[*simp*]:
$\Uparrow$ *R SUCCEED* = *SUCCEED*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

Transitivity Reasoner Setup

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

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

$\bigwedge A\ B\ C.\ A{\leq}\Downarrow Id\ B \Longrightarrow B{\leq}\Downarrow Id\ C \Longrightarrow A{\leq}\ \ \ C$

$\bigwedge A\ B\ C.\ A{\leq}{\Downarrow}Id\ B \Longrightarrow B{\leq}\quad C \Longrightarrow A{\leq}\quad C$
$\bigwedge A\ B\ C.\ A{\leq}\quad B \Longrightarrow B{\leq}{\Downarrow}Id\ C \Longrightarrow A{\leq}\quad C$
⟨*proof*⟩

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

lemma *abs-trans-additional*[*trans*]:
$\bigwedge A\ B\ C.\ [\![\ A \leq B;\ \Uparrow R\ B \leq C]\!] \Longrightarrow \Uparrow R\ A \leq C$
$\bigwedge A\ B\ C.\ [\![\Uparrow Id\ A \leq B;\ \Uparrow R\ B \leq C]\!] \Longrightarrow \Uparrow R\ A \leq C$
$\bigwedge A\ B\ C.\ [\![\Uparrow R\ A \leq B;\ \Uparrow Id\ B \leq C]\!] \Longrightarrow \Uparrow R\ A \leq C$

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

⟨*proof*⟩

2.6.4 Derived Program Constructs

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

ASSUME and ASSERT

definition *ASSERT* **where** *ASSERT* ≡ *iASSERT RETURN*
definition *ASSUME* **where** *ASSUME* ≡ *iASSUME RETURN*
interpretation *assert?*: *generic-Assert bind RETURN ASSERT ASSUME*
⟨*proof*⟩

Order matters!

lemmas [*refine-vcg*] = *ASSERT-leI*
lemmas [*refine-vcg*] = *le-ASSUMEI*
lemmas [*refine-vcg*] = *le-ASSERTI*
lemmas [*refine-vcg*] = *ASSUME-leI*

lemma *pw-ASSERT*[*refine-pw-simps*]:
nofail (*ASSERT* Φ) ⟷ Φ
inres (*ASSERT* Φ) *x*
⟨*proof*⟩

lemma *pw-ASSUME*[*refine-pw-simps*]:
nofail (*ASSUME* Φ)
inres (*ASSUME* Φ) *x* ⟷ Φ
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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 ≤ *SPEC* (λ-. *True*) ⟹ *m* ≤ *RES UNIV* ⟨*proof*⟩

lemma *RETURN-rule*[*refine-vcg*]: Φ *x* ⟹ *RETURN x* ≤ *SPEC* Φ
⟨*proof*⟩
lemma *RES-rule*[*refine-vcg*]: ⟦⋀*x*. *x*∈*S* ⟹ Φ *x*⟧ ⟹ *RES S* ≤ *SPEC* Φ

⟨*proof*⟩
lemma *SUCCEED-rule*[*refine-vcg*]: *SUCCEED* ≤ *SPEC* Φ ⟨*proof*⟩
lemma *FAIL-rule*: *False* ⟹ *FAIL* ≤ *SPEC* Φ ⟨*proof*⟩
lemma *SPEC-rule*[*refine-vcg*]: ⟦⋀*x*. Φ *x* ⟹ Φ′ *x*⟧ ⟹ *SPEC* Φ ≤ *SPEC* Φ′
⟨*proof*⟩

lemma *RETURN-to-SPEC-rule*[*refine-vcg*]: *m*≤*SPEC* ((=) *v*) ⟹ *m*≤*RETURN*
v
⟨*proof*⟩

lemma *Sup-img-rule-complete*:
(∀ *x*. *x*∈*S* ⟶ *f x* ≤ *SPEC* Φ) ⟷ *Sup* (*f*‘*S*) ≤ *SPEC* Φ
⟨*proof*⟩

lemma *SUP-img-rule-complete*:
(∀ *x*. *x*∈*S* ⟶ *f x* ≤ *SPEC* Φ) ⟷ *Sup* (*f* ‘ *S*) ≤ *SPEC* Φ
⟨*proof*⟩

lemma *Sup-img-rule*[*refine-vcg*]:
⟦ ⋀*x*. *x*∈*S* ⟹ *f x* ≤ *SPEC* Φ ⟧ ⟹ *Sup*(*f*‘*S*) ≤ *SPEC* Φ
⟨*proof*⟩

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

lemma *bind-rule-complete*: *bind M f* ≤ *SPEC* Φ ⟷ *M* ≤ *SPEC* (λ*x*. *f x* ≤
SPEC Φ)
⟨*proof*⟩
lemma *bind-rule*[*refine-vcg*]:
⟦ *M* ≤ *SPEC* (λ*x*. *f x* ≤ *SPEC* Φ) ⟧ ⟹ *bind M* (λ*x*. *f x*) ≤ *SPEC* Φ
— Note: η-expanded version helps Isabelle’s unification to keep meaningful variable names from the program
⟨*proof*⟩

lemma *ASSUME-rule*[*refine-vcg*]: ⟦Φ ⟹ Ψ ()⟧ ⟹ *ASSUME* Φ ≤ *SPEC* Ψ
⟨*proof*⟩

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

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

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
⟨*proof*⟩

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

⟦ $b \Longrightarrow S1 \leq SPEC\ \Phi$; $\neg b \Longrightarrow S2 \leq SPEC\ \Phi$⟧
$\Longrightarrow$ (*if b then S1 else S2*) $\leq SPEC\ \Phi$
⟨*proof*⟩

lemma *option-rule*[*refine-vcg*]:
⟦ $v{=}None \Longrightarrow S1 \leq SPEC\ \Phi$; $\bigwedge x.\ v{=}Some\ x \Longrightarrow f2\ x \leq SPEC\ \Phi$⟧
$\Longrightarrow$ *case-option S1 f2 v* $\leq SPEC\ \Phi$
⟨*proof*⟩

lemma *Let-rule*[*refine-vcg*]:
$f\ x \leq SPEC\ \Phi \Longrightarrow Let\ x\ f \leq SPEC\ \Phi$ ⟨*proof*⟩

lemma *Let-rule′*:
assumes $\bigwedge x.\ x{=}v \Longrightarrow f\ x \leq SPEC\ \Phi$
shows *Let v* $(\lambda x.\ f\ x) \leq SPEC\ \Phi$
⟨*proof*⟩

lemma *REC-le-rule*:
assumes *M*: *trimono body*
assumes *I0*: $(x,x')\in R$
assumes *IS*: $\bigwedge f\ x\ x'.$ ⟦ $\bigwedge x\ x'.\ (x,x')\in R \Longrightarrow f\ x \leq M\ x'$; $(x,x')\in R$ ⟧
$\Longrightarrow$ *body f x* $\leq M\ x'$
shows *REC body x* $\leq M\ x'$
⟨*proof*⟩

Proving Monotonicity

lemma *nr-mono-bind*:
assumes *MA*: *mono A* **and** *MB*: $\bigwedge s.$ *mono* $(B\ s)$
shows *mono* $(\lambda F\ s.$ *bind* $(A\ F\ s)$ $(\lambda s'.\ B\ s\ F\ s'))$
⟨*proof*⟩

lemma *nr-mono-bind′*: *mono* $(\lambda F\ s.$ *bind* $(f\ s)\ F)$
⟨*proof*⟩

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*]: $S \leq \Downarrow Id\ S$ ⟨*proof*⟩

lemma *RES-refine*:
$[\![\bigwedge s.\ s{\in}S \Longrightarrow \exists\, s'{\in}S'.\ (s,s'){\in}R]\!] \Longrightarrow RES\ S \leq \Downarrow R\ (RES\ S')$
⟨*proof*⟩

lemma *SPEC-refine*:
assumes $S \leq SPEC\ (\lambda x.\ \exists\, x'.\ (x,x'){\in}R \wedge \Phi\ x')$
shows $S \leq \Downarrow R\ (SPEC\ \Phi)$
⟨*proof*⟩

lemma *Id-SPEC-refine*[*refine*]:
$S \leq SPEC\ \Phi \Longrightarrow S \leq \Downarrow Id\ (SPEC\ \Phi)$ ⟨*proof*⟩

lemma *RETURN-refine*[*refine*]:
assumes $(x,x'){\in}R$
shows $RETURN\ x \leq \Downarrow R\ (RETURN\ x')$
⟨*proof*⟩

lemma *RETURN-SPEC-refine*:
assumes $\exists\, x'.\ (x,x'){\in}R \wedge \Phi\ x'$
shows $RETURN\ x \leq \Downarrow R\ (SPEC\ \Phi)$
⟨*proof*⟩

lemma *FAIL-refine*[*refine*]: $X \leq \Downarrow R\ FAIL$ ⟨*proof*⟩
lemma *SUCCEED-refine*[*refine*]: $SUCCEED \leq \Downarrow R\ X'$ ⟨*proof*⟩

lemma *sup-refine*[*refine*]:
assumes $ai \leq \Downarrow R\ a$
assumes $bi \leq \Downarrow R\ b$
shows $sup\ ai\ bi \leq \Downarrow R\ (sup\ a\ b)$
⟨*proof*⟩

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

lemma *bind-refine′*:
fixes $R' :: ('a{\times}'b)\ set$ **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 $bind\ M\ (\lambda x.\ f\ x) \leq \Downarrow\ R\ (bind\ M'\ (\lambda x'.\ f'\ x'))$
⟨*proof*⟩

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* $M\ (\lambda x.\ f\ x) \leq \Downarrow R\ (bind\ M'\ (\lambda x'.\ f'\ x'))$
⟨*proof*⟩

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* $M\ (\lambda x.\ f\ x) \leq \Downarrow R\ (bind\ M'\ (\lambda x'.\ f'\ x'))$
⟨*proof*⟩

Special cases for refinement of binding to *RES* statements

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

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

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

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

lemma *ASSERT-refine*[*refine*]:
$[\![\ \Phi' \Longrightarrow \Phi\ ]\!] \Longrightarrow ASSERT\ \Phi \leq \Downarrow Id\ (ASSERT\ \Phi')$
⟨*proof*⟩

lemma *ASSUME-refine*[*refine*]:
$[\![\ \Phi \Longrightarrow \Phi'\ ]\!] \Longrightarrow ASSUME\ \Phi \leq \Downarrow Id\ (ASSUME\ \Phi')$
⟨*proof*⟩

Assertions and assumptions are treated specially in bindings

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

⟨*proof*⟩

lemma *ASSERT-refine-left*:
assumes Φ
assumes Φ ⟹ $S \leq \Downarrow R\ S'$
shows $do\{ASSERT\ \Phi;\ S\} \leq \Downarrow R\ S'$
⟨*proof*⟩

lemma *ASSUME-refine-right*:
assumes Φ
assumes Φ ⟹ $S \leq \Downarrow R\ S'$
shows $S \leq \Downarrow R\ (do\ \{ASSUME\ \Phi;\ S'\})$
⟨*proof*⟩

lemma *ASSUME-refine-left*:
assumes Φ ⟹ $S \leq \Downarrow R\ S'$
shows $do\ \{ASSUME\ \Phi;\ S\} \leq \Downarrow R\ S'$
⟨*proof*⟩

lemma *ASSUME-refine-left-pres*:
assumes Φ ⟹ $do\ \{ASSUME\ \Phi;\ S\} \leq \Downarrow R\ S'$
shows $do\ \{ASSUME\ \Phi;\ S\} \leq \Downarrow R\ S'$
⟨*proof*⟩

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

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

For backward compatibility, as *intro refine* still seems to be used instead of *refine-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* $R\ \Phi\ s$
⟨*proof*⟩

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* $(\lambda f'\ x'.\ body'\ f'\ x')\ x'$)
⟨*proof*⟩

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* $(\lambda f'\ x'.\ body'\ f'\ x')\ x'$)
⟨*proof*⟩

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*$'$)
⟨*proof*⟩

lemma *Let-unfold-refine*[*refine*]:
assumes $f\ x \leq \Downarrow R\ (f'\ x')$
shows *Let x f* $\leq \Downarrow R$ (*Let x*$'$ *f*$'$)
⟨*proof*⟩

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

lemma *Let-refine*:
assumes $(m,m')\in R'$
assumes $\bigwedge x\ x'.\ (x,x')\in R' \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
shows *Let m* $(\lambda x.\ f\ x) \leq \Downarrow R$ (*Let m*$'$ $(\lambda x'.\ f'\ x')$)
⟨*proof*⟩

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*$'$)
⟨*proof*⟩

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*$'$)

⟨*proof*⟩

lemma *list-case-refine*[*refine*]:
assumes (*li*,*l*)∈⟨*S*⟩*list-rel*
assumes *fni* ≤⇓*R fn*
assumes ⋀*xi x xsi xs*. ⟦ (*xi*,*x*)∈*S*; (*xsi*,*xs*)∈⟨*S*⟩*list-rel*; *li*=*xi*#*xsi*; *l*=*x*#*xs* ⟧ ⟹ *fci xi xsi* ≤⇓*R* (*fc x xs*)
shows (*case li of* [] ⇒ *fni* | *xi*#*xsi* ⇒ *fci xi xsi*) ≤ ⇓*R* (*case l of* [] ⇒ *fn* | *x*#*xs* ⇒ *fc x xs*)
⟨*proof*⟩

It is safe to split conjunctions in refinement goals.

declare *conjI*[*refine*]

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

lemma *remove-Let-refine*[*refine2*]:
assumes *M* ≤ ⇓*R* (*f x*)
shows *M* ≤ ⇓*R* (*Let x f*) ⟨*proof*⟩

lemma *intro-Let-refine*[*refine2*]:
assumes *f x* ≤ ⇓*R M'*
shows *Let x f* ≤ ⇓*R M'* ⟨*proof*⟩

lemma *bind2let-refine*[*refine2*]:
assumes *RETURN x* ≤ ⇓*R' M'*
assumes ⋀*x'*. (*x*,*x'*)∈*R'* ⟹ *f x* ≤ ⇓*R* (*f' x'*)
shows *Let x f* ≤ ⇓*R* (*bind M'* (λ*x'*. *f' x'*))
⟨*proof*⟩

lemma *bind-Let-refine2*[*refine2*]: ⟦
m' ≤⇓*R'* (*RETURN x*);
⋀*x'*. ⟦*inres m' x'*; (*x'*,*x*)∈*R*⟧ ⟹ *f' x'* ≤ ⇓*R* (*f x*)
⟧ ⟹ *m'*≫=(λ*x'*. *f' x'*) ≤ ⇓*R* (*Let x* (λ*x*. *f x*))
⟨*proof*⟩

lemma *bind2letRETURN-refine*[*refine2*]:
assumes *RETURN x* ≤ ⇓*R' M'*
assumes ⋀*x'*. (*x*,*x'*)∈*R'* ⟹ *RETURN* (*f x*) ≤ ⇓*R* (*f' x'*)
shows *RETURN* (*Let x f*) ≤ ⇓*R* (*bind M'* (λ*x'*. *f' x'*))
⟨*proof*⟩

lemma *RETURN-as-SPEC-refine*[*refine2*]:
assumes *M* ≤ *SPEC* (λ*c*. (*c*,*a*)∈*R*)
shows *M* ≤ ⇓*R* (*RETURN a*)
⟨*proof*⟩

lemma *RETURN-as-SPEC-refine-old*:
⋀*M R*. *M* ≤ ⇓*R* (*SPEC* (λ*x*. *x*=*v*)) ⟹ *M* ≤⇓*R* (*RETURN v*)

⟨*proof*⟩

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

⟨*proof*⟩

lemma *RES-sng-as-SPEC-refine*[*refine2*]:
assumes $M \leq SPEC\ (\lambda c.\ (c,a)\in R)$
shows $M \leq \Downarrow R\ (RES\ \{a\})$
⟨*proof*⟩

lemma *intro-spec-refine-iff*:
$(bind\ (RES\ X)\ f \leq \Downarrow R\ M) \longleftrightarrow (\forall x\in X.\ f\ x \leq \Downarrow R\ M)$
⟨*proof*⟩

lemma *intro-spec-refine*[*refine2*]:
assumes $\bigwedge x.\ x\in X \Longrightarrow f\ x \leq \Downarrow R\ M$
shows $bind\ (RES\ X)\ (\lambda x.\ f\ x) \leq \Downarrow R\ M$
⟨*proof*⟩

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

Replacing a let by a deterministic computation

lemma *let2bind-refine*:
assumes $m \leq \Downarrow R'\ (RETURN\ m')$
assumes $\bigwedge x\ x'.\ (x,x')\in R' \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
shows $bind\ m\ (\lambda x.\ f\ x) \leq \Downarrow R\ (Let\ m'\ (\lambda x'.\ f'\ x'))$
⟨*proof*⟩

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 $bind\ m\ (\lambda x.\ f\ x) \leq \Downarrow R\ m''$
⟨*proof*⟩

lemma *intro-bind-refine-id*:
assumes $m \leq (SPEC\ ((=)\ m'))$
assumes $f\ m' \leq \Downarrow R\ m''$
shows $bind\ m\ f \leq \Downarrow R\ m''$
⟨*proof*⟩

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

lemma *lhs-step-If*:
$[\![\ b \Longrightarrow t \le m;\ \neg b \Longrightarrow e \le m\ ]\!] \Longrightarrow \mathit{If}\ b\ t\ e \le m$ ⟨*proof*⟩

lemma *lhs-step-RES*:
$[\![\ \bigwedge x.\ x{\in}X \Longrightarrow \mathit{RETURN}\ x \le m\ ]\!] \Longrightarrow \mathit{RES}\ X \le m$
⟨*proof*⟩

lemma *lhs-step-SPEC*:
$[\![\ \bigwedge x.\ \Phi\ x \Longrightarrow \mathit{RETURN}\ x \le m\ ]\!] \Longrightarrow \mathit{SPEC}\ (\lambda x.\ \Phi\ x) \le m$
⟨*proof*⟩

lemma *lhs-step-bind*:
fixes $m :: 'a\ nres$ **and** $f :: 'a \Rightarrow 'b\ nres$
assumes $\mathit{nofail}\ m' \Longrightarrow \mathit{nofail}\ m$
assumes $\bigwedge x.\ \mathit{nf\text{-}inres}\ m\ x \Longrightarrow f\ x \le m'$
shows $\mathit{do}\ \{x{\leftarrow}m;\ f\ x\} \le m'$
⟨*proof*⟩

lemma *rhs-step-bind*:
assumes $m \le \Downarrow R\ m'$ $\quad \mathit{inres}\ m\ x$ $\quad \bigwedge x'.\ (x,x'){\in}R \Longrightarrow \mathit{lhs} \le \Downarrow S\ (f'\ x')$
shows $\mathit{lhs} \le \Downarrow S\ (m' \gg\!= f')$
⟨*proof*⟩

lemma *rhs-step-bind-RES*:
assumes $x'{\in}X'$
assumes $m \le \Downarrow R\ (f'\ x')$
shows $m \le \Downarrow R\ (\mathit{RES}\ X' \gg\!= f')$
⟨*proof*⟩

lemma *rhs-step-bind-SPEC*:
assumes $\Phi\ x'$
assumes $m \le \Downarrow R\ (f'\ x')$
shows $m \le \Downarrow R\ (\mathit{SPEC}\ \Phi \gg\!= f')$
⟨*proof*⟩

lemma *RES-bind-choose*:
assumes $x{\in}X$
assumes $m \le f\ x$
shows $m \le \mathit{RES}\ X \gg\!= f$
⟨*proof*⟩

lemma *pw-RES-bind-choose*:
$\mathit{nofail}\ (\mathit{RES}\ X \gg\!= f) \longleftrightarrow (\forall x{\in}X.\ \mathit{nofail}\ (f\ x))$
$\mathit{inres}\ (\mathit{RES}\ X \gg\!= f)\ y \longleftrightarrow (\exists x{\in}X.\ \mathit{inres}\ (f\ x)\ y)$
⟨*proof*⟩

lemma *prod-case-refine*:
assumes $(p',p)\in R1\times_r R2$
assumes $\bigwedge x1'\ x2'\ x1\ x2.\ [\![\ p'=(x1',x2');\ p=(x1,x2);\ (x1',x1)\in R1;\ (x2',x2)\in R2\ ]\!] \Longrightarrow f'\ x1'\ x2' \leq \Downarrow R\ (f\ x1\ x2)$
shows $(case\ p'\ of\ (x1',x2') \Rightarrow f'\ x1'\ x2') \leq \Downarrow R\ (case\ p\ of\ (x1,x2) \Rightarrow f\ x1\ x2)$
⟨*proof*⟩

2.6.6 Relators

declare *fun-relI*[*refine*]

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

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

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

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

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

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

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

lemma *param-RES*[*param*]:
$(RES,RES) \in \langle R\rangle set\text{-}rel \to \langle R\rangle nres\text{-}rel$
⟨*proof*⟩

lemma *param-RETURN*[*param*]:
$(RETURN,RETURN) \in R \to \langle R\rangle nres\text{-}rel$
⟨*proof*⟩

lemma *param-bind*[*param*]:
$(bind,bind) \in \langle Ra\rangle nres\text{-}rel \to (Ra\to\langle Rb\rangle nres\text{-}rel) \to \langle Rb\rangle nres\text{-}rel$
⟨*proof*⟩

lemma *param-ASSERT-bind*[*param*]: [⟦
$(\Phi,\Psi) \in bool\text{-}rel$;
⟦ Φ; Ψ ⟧ $\Longrightarrow (f,g)\in\langle R\rangle nres\text{-}rel$

$[\![]\!] \Longrightarrow (ASSERT\ \Phi \gg f,\ ASSERT\ \Psi \gg g) \in \langle R \rangle nres\text{-}rel$
⟨*proof*⟩

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 $\Phi' \Longrightarrow \Phi$
assumes $[\![\Phi'; \Phi]\!] \Longrightarrow (m,m') \in \langle R \rangle nres\text{-}rel$
shows (*op-nres-ASSERT-bnd* Φ *m*, *op-nres-ASSERT-bnd* Φ′ *m*′) ∈ ⟨*R*⟩*nres-rel*
⟨*proof*⟩

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *id-ASSERT*[*autoref-op-pat-def*]:
do {*ASSERT* Φ; *m*} ≡ *OP* (*op-nres-ASSERT-bnd* Φ)$*m*
⟨*proof*⟩

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*
⟨*proof*⟩

end

lemma *autoref-SUCCEED*[*autoref-rules*]: (*SUCCEED*,*SUCCEED*) ∈ ⟨*R*⟩*nres-rel*
⟨*proof*⟩

lemma *autoref-FAIL*[*autoref-rules*]: (*FAIL*,*FAIL*) ∈ ⟨*R*⟩*nres-rel*
⟨*proof*⟩

lemma *autoref-RETURN*[*autoref-rules*]:
(*RETURN*,*RETURN*) ∈ *R* → ⟨*R*⟩*nres-rel*
⟨*proof*⟩

lemma *autoref-bind*[*autoref-rules*]:
(*bind*,*bind*) ∈ ⟨*R1*⟩*nres-rel* → (*R1*→⟨*R2*⟩*nres-rel*) → ⟨*R2*⟩*nres-rel*
⟨*proof*⟩

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *autoref-ASSERT*[*autoref-rules*]:
assumes Φ ⟹ (*m′*,*m*)∈⟨*R*⟩*nres-rel*
shows (
m′,
(*OP* (*op-nres-ASSERT-bnd* Φ) ::: ⟨*R*⟩*nres-rel* → ⟨*R*⟩*nres-rel*) $ *m*)∈⟨*R*⟩*nres-rel*
⟨*proof*⟩

lemma *autoref-ASSUME*[*autoref-rules*]:
assumes *SIDE-PRECOND* Φ
assumes Φ ⟹ (*m′*,*m*)∈⟨*R*⟩*nres-rel*
shows (
m′,
(*OP* (*op-nres-ASSUME-bnd* Φ) ::: ⟨*R*⟩*nres-rel* → ⟨*R*⟩*nres-rel*) $ *m*)∈⟨*R*⟩*nres-rel*
⟨*proof*⟩

lemma *autoref-REC*[*autoref-rules*]:
assumes (*B*,*B′*)∈(*Ra*→⟨*Rr*⟩*nres-rel*) → *Ra* → ⟨*Rr*⟩*nres-rel*
assumes *DEFER trimono B*
shows (*REC B*,
(*OP REC*
::: ((*Ra*→⟨*Rr*⟩*nres-rel*) → *Ra* → ⟨*Rr*⟩*nres-rel*) → *Ra* → ⟨*Rr*⟩*nres-rel*)$*B′*
) ∈ *Ra* → ⟨*Rr*⟩*nres-rel*
⟨*proof*⟩

theorem *param-RECT*[*param*]:
assumes (*B*, *B′*) ∈ (*Ra* → ⟨*Rr*⟩*nres-rel*) → *Ra* → ⟨*Rr*⟩*nres-rel*
and *trimono B*
shows (*REC*$_T$ *B*, *REC*$_T$ *B′*)∈ *Ra* → ⟨*Rr*⟩*nres-rel*
⟨*proof*⟩

lemma *autoref-RECT*[*autoref-rules*]:
assumes (*B*,*B′*) ∈ (*Ra*→⟨*Rr*⟩*nres-rel*) → *Ra*→⟨*Rr*⟩*nres-rel*
assumes *DEFER trimono B*
shows (*RECT B*,
(*OP RECT*
::: ((*Ra*→⟨*Rr*⟩*nres-rel*) → *Ra* → ⟨*Rr*⟩*nres-rel*) → *Ra* → ⟨*Rr*⟩*nres-rel*)$*B′*
) ∈ *Ra* → ⟨*Rr*⟩*nres-rel*
⟨*proof*⟩

end

2.6.8 Convenience Rules

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

lemma *ref-two-step*: *A*≤⇓*R B* ⟹ *B*≤*C* ⟹ *A*≤⇓*R C*
⟨*proof*⟩

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

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 $S \leq \Downarrow R\ S'$
⟨*proof*⟩

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

lemma *introR*: $(a,a') \in R \Longrightarrow (a,a') \in R$ ⟨*proof*⟩

lemma *intro-prgR*: $c \leq \Downarrow R\ a \Longrightarrow c \leq \Downarrow R\ a$ ⟨*proof*⟩

lemma *refine-IdI*: $m \leq m' \Longrightarrow m \leq \Downarrow Id\ m'$ ⟨*proof*⟩

lemma *le-ASSERTI-pres*:
assumes $\Phi \Longrightarrow S \leq do\ \{ASSERT\ \Phi;\ S'\}$
shows $S \leq do\ \{ASSERT\ \Phi;\ S'\}$
⟨*proof*⟩

lemma *RETURN-ref-SPECD*:
assumes $RETURN\ c \leq \Downarrow R\ (SPEC\ \Phi)$
obtains a **where** $(c,a) \in R \quad \Phi\ a$
⟨*proof*⟩

lemma *RETURN-ref-RETURND*:
assumes $RETURN\ c \leq \Downarrow R\ (RETURN\ a)$
shows $(c,a) \in R$
⟨*proof*⟩

lemma *return-refine-prop-return*:
assumes *nofail* m
assumes $RETURN\ x \leq \Downarrow R\ m$
obtains x' **where** $(x,x') \in R \quad RETURN\ x' \leq m$
⟨*proof*⟩

lemma *ignore-snd-refine-conv*:
$(m \leq \Downarrow(R \times_r UNIV)\ m') \longleftrightarrow m \gg\!= (RETURN\ o\ fst) \leq \Downarrow R\ (m' \gg\!= (RETURN\ o\ fst))$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *SPEC-eq-is-RETURN*:
SPEC ((=) *x*) = *RETURN x*
SPEC (λ*x*. *x*=*y*) = *RETURN y*
⟨*proof*⟩

lemma *RETURN-SPEC-conv*: *RETURN r* = *SPEC* (λ*x*. *x*=*r*)
⟨*proof*⟩

lemma *refine2spec-aux*:
a ≤ ⇓*R b* ⟷ ((*nofail b* ⟶ *a* ≤ *SPEC* (λ*r*. (∃ *x*. *inres b x* ∧ (*r*,*x*)∈*R*))))
⟨*proof*⟩

lemma *refine2specI*:
assumes *nofail b* ⟹ *a* ≤ *SPEC* (λ*r*. (∃ *x*. *inres b x* ∧ (*r*,*x*)∈*R*))
shows *a* ≤ ⇓*R b*
⟨*proof*⟩

lemma *specify-left*:
assumes *m* ≤ *SPEC* Φ
assumes ⋀*x*. Φ *x* ⟹ *f x* ≤ *M*
shows *do* { *x* ← *m*; *f x* } ≤ *M*
⟨*proof*⟩

lemma *build-rel-SPEC*:
M ≤ *SPEC* (λ*x*. Φ (α *x*) ∧ *I x*) ⟹ *M* ≤ ⇓(*build-rel* α *I*) (*SPEC* Φ)
⟨*proof*⟩

lemma *build-rel-SPEC-conv*: ⇓(*br* α *I*) (*SPEC* Φ) = *SPEC* (λ*x*. *I x* ∧ Φ (α *x*))
⟨*proof*⟩

lemma *refine-IdD*: *c* ≤ ⇓*Id a* ⟹ *c* ≤ *a* ⟨*proof*⟩

lemma *bind-sim-select-rule*:
assumes *m*⤜*f*′ ≤ *SPEC* Ψ
assumes ⋀*x*. ⟦*nofail m*; *inres m x*; *f*′ *x*≤*SPEC* Ψ⟧ ⟹ *f x*≤*SPEC* Φ
shows *m*⤜*f* ≤ *SPEC* Φ
— Simultaneously select a result from assumption and verification goal. Useful to work with assumptions that restrict the current program to be verified.
⟨*proof*⟩

lemma *assert-bind-spec-conv*: *ASSERT* Φ ≫ *m* ≤ *SPEC* Ψ ⟷ (Φ ∧ *m* ≤ *SPEC* Ψ)
— 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.
⟨*proof*⟩

lemma *summarize-ASSERT-conv*: *do* {*ASSERT* Φ; *ASSERT* Ψ; *m*} = *do* {*ASSERT* (Φ ∧ Ψ); *m*}
⟨*proof*⟩

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

lemma *le-RES-nofailI*:
assumes *a*≤*RES x*
shows *nofail a*
⟨*proof*⟩

lemma *add-invar-refineI*:
assumes *f x* ≤⇓*R* (*f′ x′*)
and *nofail* (*f x*) ⟹ *f x* ≤ *SPEC I*
shows *f x* ≤ ⇓ {(*c*, *a*). (*c*, *a*) ∈ *R* ∧ *I c*} (*f′ x′*)
⟨*proof*⟩

lemma *bind-RES-RETURN-eq*: *bind* (*RES X*) (λ*x*. *RETURN* (*f x*)) =
RES { *f x* | *x*. *x*∈*X* }
⟨*proof*⟩

lemma *bind-RES-RETURN2-eq*: *bind* (*RES X*) (λ(*x*,*y*). *RETURN* (*f x y*)) =
RES { *f x y* | *x y*. (*x*,*y*)∈*X* }
⟨*proof*⟩

lemma *le-SPEC-bindI*:
assumes Φ *x*
assumes *m* ≤ *f x*
shows *m* ≤ *SPEC* Φ ≫= *f*
⟨*proof*⟩

lemma *bind-assert-refine*:
assumes *m1* ≤ *SPEC* Φ
assumes ⋀*x*. Φ *x* ⟹ *m2 x* ≤ *m′*
shows *do* {*x*←*m1*; *ASSERT* (Φ *x*); *m2 x*} ≤ *m′*
⟨*proof*⟩

lemma *RETURN-refine-iff*[*simp*]: *RETURN x* ≤⇓*R* (*RETURN y*) ⟷ (*x*,*y*)∈*R*
⟨*proof*⟩

lemma *RETURN-RES-refine-iff*:
RETURN x ≤⇓*R* (*RES Y*) ⟷ (∃ *y*∈*Y*. (*x*,*y*)∈*R*)
⟨*proof*⟩

lemma *RETURN-RES-refine*:

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

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

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

lemma *inf-RETURN-SPEC*[*simp*]:
$inf\ (RETURN\ x)\ (SPEC\ (\lambda y.\ \Phi\ y)) = SPEC\ (\lambda y.\ y{=}x \wedge \Phi\ x)$
$inf\ (SPEC\ (\lambda y.\ \Phi\ y))\ (RETURN\ x) = SPEC\ (\lambda y.\ y{=}x \wedge \Phi\ x)$
⟨*proof*⟩

lemma *RES-sng-eq-RETURN*: $RES\ \{x\} = RETURN\ x$
⟨*proof*⟩

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

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

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

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

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

lemma *weaken-SPEC*:
$m \leq SPEC\ \Phi \Longrightarrow (\bigwedge x.\ \Phi\ x \Longrightarrow \Psi\ x) \Longrightarrow m \leq SPEC\ \Psi$
⟨*proof*⟩

lemma *bind-le-nofailI*:
assumes *nofail m*
assumes $\bigwedge x.\ RETURN\ x \le m \Longrightarrow f\ x \le m'$
shows $m \gg\!= f \le m'$
⟨*proof*⟩

lemma *bind-le-shift*:
$bind\ m\ f \le m'$
$\longleftrightarrow m \le (if\ nofail\ m'\ then\ SPEC\ (\lambda x.\ f\ x \le m')\ else\ FAIL)$
⟨*proof*⟩

lemma *If-bind-distrib*[*simp*]:
fixes $t\ e :: 'a\ nres$
shows $(If\ b\ t\ e \gg\!= (\lambda x.\ f\ x)) = (If\ b\ (t \gg\!= (\lambda x.\ f\ x))\ (e \gg\!= (\lambda x.\ f\ x)))$
⟨*proof*⟩

lemma *unused-bind-conv*:
assumes *NO-MATCH* $(ASSERT\ \Phi)\ m$
assumes *NO-MATCH* $(ASSUME\ \Phi)\ m$
shows $(m \gg\!= (\lambda x.\ c)) = (ASSERT\ (nofail\ m) \gg\!= (\lambda\text{-}.\ ASSUME\ (\exists x.\ inres\ m\ x) \gg\!= (\lambda x.\ c)))$
⟨*proof*⟩

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

lemma *let-to-bind-conv*:
$Let\ x\ f = RETURN\ x \gg\!= f$
⟨*proof*⟩

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

lemma *pull-out-let-conv*: $RETURN\ (Let\ x\ f) = Let\ x\ (\lambda x.\ RETURN\ (f\ x))$
⟨*proof*⟩

lemma *push-in-let-conv*:
$Let\ x\ (\lambda x.\ RETURN\ (f\ x)) = RETURN\ (Let\ x\ f)$
$Let\ x\ (RETURN\ o\ f) = RETURN\ (Let\ x\ f)$
⟨*proof*⟩

lemma *pull-out-RETURN-case-option*:
$case\text{-}option\ (RETURN\ a)\ (\lambda v.\ RETURN\ (f\ v))\ x = RETURN\ (case\text{-}option\ a\ f\ x)$
⟨*proof*⟩

lemma *if-bind-cond-refine*:
assumes $ci \le RETURN\ b$
assumes $b \Longrightarrow ti \le \Downarrow R\ t$

assumes $\neg b \Longrightarrow ei \leq \Downarrow R\ e$
shows $do\ \{b \leftarrow ci;\ if\ b\ then\ ti\ else\ ei\} \leq \Downarrow R\ (if\ b\ then\ t\ else\ e)$
⟨*proof*⟩

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

⟨*proof*⟩

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

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

lemma *unused-bind-RES-ne*[*simp*]: $X \neq \{\} \Longrightarrow do\ \{\ - \leftarrow RES\ X;\ m\} = m$
⟨*proof*⟩

lemma *le-ASSERT-defI1*:
assumes $c \equiv do\ \{ASSERT\ \Phi;\ m\}$
assumes $\Phi \Longrightarrow m' \leq c$
shows $m' \leq c$
⟨*proof*⟩

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

lemma *le-ASSERT-defI2*:
assumes $c \equiv do\ \{ASSERT\ \Phi;\ ASSERT\ \Psi;\ m\}$
assumes $[\![\Phi;\ \Psi]\!] \Longrightarrow m' \leq c$
shows $m' \leq c$
⟨*proof*⟩

lemma *refine-ASSERT-defI2*:
assumes $c \equiv do\ \{ASSERT\ \Phi;\ ASSERT\ \Psi;\ m\}$
assumes $[\![\Phi;\ \Psi]\!] \Longrightarrow m' \leq \Downarrow R\ c$
shows $m' \leq \Downarrow R\ c$
⟨*proof*⟩

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

shows $c \leq m$
⟨*proof*⟩

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

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

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

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

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 $P \leq SPEC\ (\lambda s.\ Q\ s \longleftrightarrow R\ s)$
⟨*proof*⟩

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*⟩

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

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

Pointwise Reasoning

lemma *inres-if*:
$[\![\ inres\ (if\ P\ then\ Q\ else\ R)\ x;\ [\![P;\ inres\ Q\ x]\!] \Longrightarrow S;\ [\![\neg\ P;\ inres\ R\ x]\!] \Longrightarrow S\ ]\!] \Longrightarrow S$
⟨*proof*⟩

lemma *inres-SPEC*:

inres M x $\Longrightarrow$ *M* $\leq$ *SPEC* Φ $\Longrightarrow$ Φ *x*
⟨*proof*⟩

lemma *SPEC-nofail*:
X $\leq$ *SPEC* Φ $\Longrightarrow$ *nofail X*
⟨*proof*⟩

lemma *nofail-SPEC*: *nofail m* $\Longrightarrow$ *m* $\leq$ *SPEC* (λ-. *True*)
⟨*proof*⟩

lemma *nofail-SPEC-iff*: *nofail m* $\longleftrightarrow$ *m* $\leq$ *SPEC* (λ-. *True*)
⟨*proof*⟩

lemma *nofail-SPEC-triv-refine*: ⟦ *nofail m*; $\bigwedge x.$ Φ *x* ⟧ $\Longrightarrow$ *m* $\leq$ *SPEC* Φ
⟨*proof*⟩

end

2.7 Less-Equal or Fail

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

A predicate that states refinement or that the LHS fails.

definition *le-or-fail* :: *'a nres* $\Rightarrow$ *'a nres* $\Rightarrow$ *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'*
⟨*proof*⟩

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

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

lemma *le-by-leofI*: ⟦ *nofail m'* $\Longrightarrow$ *nofail m*; *m*$\leq_n$*m'* ⟧ $\Longrightarrow$ *m* $\leq$ *m'*
⟨*proof*⟩

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

lemma *leof-trans*[*trans*]: $[\![a \leq_n \mathit{RES}\ X;\ \mathit{RES}\ X \leq_n c]\!] \Longrightarrow a \leq_n c$
⟨*proof*⟩

lemma *leof-trans-nofail*: $[\![\ a \leq_n b;\ \mathit{nofail}\ b;\ b \leq_n c\ ]\!] \Longrightarrow a \leq_n c$
⟨*proof*⟩

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

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

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

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

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

lemma *leof-bind-rule*[*refine-vcg*]:
$[\![\ m \leq_n \mathit{SPEC}\ (\lambda x.\ f\ x \leq_n \mathit{SPEC}\ \Phi)\ ]\!] \Longrightarrow m \gg\!= f \leq_n \mathit{SPEC}\ \Phi$
⟨*proof*⟩

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

lemma *RES-leof-RES-iff*[*simp*]: $\mathit{RES}\ X \leq_n \mathit{RES}\ Y \longleftrightarrow X \subseteq Y$
⟨*proof*⟩

lemma *leof-Let-rule*[*refine-vcg*]: $f\ x \leq_n \mathit{SPEC}\ \Phi \Longrightarrow \mathit{Let}\ x\ f \leq_n \mathit{SPEC}\ \Phi$
⟨*proof*⟩

lemma *leof-If-rule*[*refine-vcg*]:
$[\![c \Longrightarrow t \leq_n \mathit{SPEC}\ \Phi;\ \neg c \Longrightarrow e \leq_n \mathit{SPEC}\ \Phi]\!] \Longrightarrow \mathit{If}\ c\ t\ e \leq_n \mathit{SPEC}\ \Phi$
⟨*proof*⟩

lemma *leof-RES-rule*[*refine-vcg*]:
$[\![\bigwedge x.\ \Psi\ x \Longrightarrow \Phi\ x]\!] \Longrightarrow \mathit{SPEC}\ \Psi \leq_n \mathit{SPEC}\ \Phi$
$[\![\bigwedge x.\ x \in X \Longrightarrow \Phi\ x]\!] \Longrightarrow \mathit{RES}\ X \leq_n \mathit{SPEC}\ \Phi$
⟨*proof*⟩

lemma *leof-True-rule*: $[\![\bigwedge x.\ \Phi\ x]\!] \Longrightarrow m \leq_n \mathit{SPEC}\ \Phi$

⟨*proof*⟩

lemma *sup-leof-iff*: (*sup a b* $\leq_n$ *m*) $\longleftrightarrow$ (*nofail a* $\wedge$ *nofail b* $\longrightarrow$ *a*$\leq_n$*m* $\wedge$ *b*$\leq_n$*m*)

⟨*proof*⟩

lemma *sup-leof-rule*[*refine-vcg*]:
assumes ⟦*nofail a*; *nofail b*⟧ $\Longrightarrow$ *a*$\leq_n$*m*
assumes ⟦*nofail a*; *nofail b*⟧ $\Longrightarrow$ *b*$\leq_n$*m*
shows *sup a b* $\leq_n$ *m*
⟨*proof*⟩

lemma *leof-option-rule*[*refine-vcg*]:
⟦*v* = *None* $\Longrightarrow$ *S1* $\leq_n$ *SPEC* Φ; $\bigwedge$*x*. *v* = *Some x* $\Longrightarrow$ *f2 x* $\leq_n$ *SPEC* Φ⟧
$\Longrightarrow$ (*case v of None* $\Rightarrow$ *S1* | *Some x* $\Rightarrow$ *f2 x*) $\leq_n$ *SPEC* Φ
⟨*proof*⟩

lemma *ASSERT-leof-rule*[*refine-vcg*]:
assumes Φ $\Longrightarrow$ *m* $\leq_n$ *m*′
shows *do* {*ASSERT* Φ; *m*} $\leq_n$ *m*′
⟨*proof*⟩

lemma *leof-ASSERT-rule*[*refine-vcg*]: ⟦Φ $\Longrightarrow$ *m* $\leq_n$ *m*′⟧ $\Longrightarrow$ *m* $\leq_n$ *ASSERT* Φ $\gg$ *m*′
⟨*proof*⟩

lemma *leof-ASSERT-refine-rule*[*refine*]: ⟦Φ $\Longrightarrow$ *m* $\leq_n$ $\Downarrow$*R m*′⟧ $\Longrightarrow$ *m* $\leq_n$ $\Downarrow$*R* (*ASSERT* Φ $\gg$ *m*′)
⟨*proof*⟩

lemma *ASSUME-leof-iff*: (*ASSUME* Φ $\leq_n$ *SPEC* Ψ) $\longleftrightarrow$ (Φ $\longrightarrow$ Ψ ())
⟨*proof*⟩

lemma *ASSUME-leof-rule*[*refine-vcg*]:
assumes Φ $\Longrightarrow$ Ψ ()
shows *ASSUME* Φ $\leq_n$ *SPEC* Ψ
⟨*proof*⟩

lemma *SPEC-rule-conj-leofI1*:
assumes *m* $\leq$ *SPEC* Φ
assumes *m* $\leq_n$ *SPEC* Ψ
shows *m* $\leq$ *SPEC* (λ*s*. Φ *s* $\wedge$ Ψ *s*)
⟨*proof*⟩

lemma *SPEC-rule-conj-leofI2*:
assumes *m* $\leq_n$ *SPEC* Φ
assumes *m* $\leq$ *SPEC* Ψ
shows *m* $\leq$ *SPEC* (λ*s*. Φ *s* $\wedge$ Ψ *s*)

⟨*proof*⟩

lemma *SPEC-rule-leof-conjI*:
assumes $m \leq_n SPEC\ \Phi \quad m \leq_n SPEC\ \Psi$
shows $m \leq_n SPEC\ (\lambda x.\ \Phi\ x \wedge \Psi\ x)$
⟨*proof*⟩

lemma *leof-use-spec-rule*:
assumes $m \leq_n SPEC\ \Psi$
assumes $m \leq_n SPEC\ (\lambda s.\ \Psi\ s \longrightarrow \Phi\ s)$
shows $m \leq_n SPEC\ \Phi$
⟨*proof*⟩

lemma *use-spec-leof-rule*:
assumes $m \leq_n SPEC\ \Psi$
assumes $m \leq SPEC\ (\lambda s.\ \Psi\ s \longrightarrow \Phi\ s)$
shows $m \leq SPEC\ \Phi$
⟨*proof*⟩

lemma *leof-strengthen-SPEC*:
$m \leq_n SPEC\ \Phi \Longrightarrow m \leq_n SPEC\ (\lambda x.\ inres\ m\ x \wedge \Phi\ x)$
⟨*proof*⟩

lemma *build-rel-SPEC-leof*:
assumes $m \leq_n SPEC\ (\lambda x.\ I\ x \wedge \Phi\ (\alpha\ x))$
shows $m \leq_n \Downarrow(br\ \alpha\ I)\ (SPEC\ \Phi)$
⟨*proof*⟩

lemma *RETURN-as-SPEC-refine-leof*[*refine2*]:
assumes $M \leq_n SPEC\ (\lambda c.\ (c,a) \in R)$
shows $M \leq_n \Downarrow R\ (RETURN\ a)$
⟨*proof*⟩

lemma *ASSERT-leof-defI*:
assumes $c \equiv do\ \{\ ASSERT\ \Phi;\ m'\}$
assumes $\Phi \Longrightarrow m' \leq_n m$
shows $c \leq_n m$
⟨*proof*⟩

lemma *leof-fun-conv-le*:
$(f\ x \leq_n M\ x) \longleftrightarrow (f\ x \leq (if\ nofail\ (f\ x)\ then\ M\ x\ else\ FAIL))$
⟨*proof*⟩

lemma *leof-add-nofailI*: $[\![\ nofail\ m \Longrightarrow m \leq_n m'\ ]\!] \Longrightarrow m \leq_n m'$
⟨*proof*⟩

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

shows $m \leq_n SPEC\ P$
⟨*proof*⟩

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 $RECT\ body\ x \leq_n M\ x$
⟨*proof*⟩

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*⟩

2.8.1 Type Based Heuristics

This heuristics instantiates schematic data refinement relations based on their type. Both, the left hand side and right hand side type are considered.

The heuristics works by proving goals of the form *RELATES ?R*, thereby instantiating *?R*.

definition $RELATES :: ('a \times 'b)\ set \Rightarrow bool$ **where** $RELATES\ R \equiv True$
lemma *RELATESI*: $RELATES\ R$ ⟨*proof*⟩

⟨*ML*⟩

2.8.2 Patterns

This section defines the patterns that are recognized as data refinement goals.

lemma *RELATESI-memb*[*refine-dref-pattern*]:
$RELATES\ R \Longrightarrow (a,b)\in R \Longrightarrow (a,b)\in R\ \langle proof\rangle$
lemma *RELATESI-refspec*[*refine-dref-pattern*]:
$RELATES\ R \Longrightarrow S \leq\Downarrow R\ S' \Longrightarrow S \leq\Downarrow R\ S'\ \langle proof\rangle$

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

lemma *RELATES-pattern*[*refine-dref-pattern*]: $RELATES\ R \Longrightarrow RELATES\ R\ \langle proof\rangle$
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*]: $RELATES\ Id\ \langle proof\rangle$

lemma *prod-rel-RELATES*[*refine-dref-RELATES*]:
$RELATES\ Ra \Longrightarrow RELATES\ Rb \Longrightarrow RELATES\ (\langle Ra,Rb\rangle prod\text{-}rel)$
$\langle proof\rangle$

declare *prod-rel-sv*[*refine-hsimp*]
lemma *prod-rel-iff*[*refine-hsimp*]:
$((a,b),(a',b'))\in\langle A,B\rangle prod\text{-}rel \longleftrightarrow (a,a')\in A \wedge (b,b')\in B$
$\langle proof\rangle$

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

lemma *option-rel-RELATES*[*refine-dref-RELATES*]:
$RELATES\ Ra \Longrightarrow RELATES\ (\langle Ra\rangle option\text{-}rel)$
$\langle proof\rangle$

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*]:
$RELATES\ R \Longrightarrow RELATES\ (\langle R\rangle set\text{-}rel)\ \langle proof\rangle$

lemma *set-rel-empty-eq*: $(S,S')\in\langle X\rangle set\text{-}rel \Longrightarrow S=\{\} \longleftrightarrow S'=\{\}$
$\langle proof\rangle$

lemma *set-rel-sngD*: $(\{a\},\{b\})\in\langle R\rangle set\text{-}rel \Longrightarrow (a,b)\in R$
$\langle proof\rangle$

lemma *Image-insert*[*refine-hsimp*]:
$(a,b)\in R \Longrightarrow$ *single-valued* $R \Longrightarrow R``$*insert* $a\ A =$ *insert* $b\ (R``A)$
⟨*proof*⟩

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

lemma *Image-Diff*[*refine-hsimp*]:
single-valued (*converse* R) $\Longrightarrow R``(A-B) = R``A - R``B$
⟨*proof*⟩

lemma *Image-Inter*[*refine-hsimp*]:
single-valued (*converse* R) $\Longrightarrow R``(A\cap B) = R``A \cap R``B$
⟨*proof*⟩

lemma *list-rel-RELATES*[*refine-dref-RELATES*]:
RELATES $R \Longrightarrow$ *RELATES* (⟨R⟩*list-rel*) ⟨*proof*⟩

lemmas [*refine-hsimp*] = *list-rel-sv-iff list-rel-simp*

lemma *RELATES-nres-rel*[*refine-dref-RELATES*]: *RELATES* $R \Longrightarrow$ *RELATES* (⟨R⟩*nres-rel*)
⟨*proof*⟩

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) $\longleftrightarrow (\exists x.\ P\ x)$
⟨*proof*⟩
lemma *OBTAIN-inres*[*refine-pw-simps*]: *inres* (*OBTAIN* P) $x \longleftrightarrow (\forall x.\ \neg P\ x) \vee P\ x$

⟨*proof*⟩
lemma *OBTAIN-rule*[*refine-vcg*]: ⟦ ∃ *x*. *P x*; ⋀*x*. *P x* ⟹ *Q x* ⟧ ⟹ *OBTAIN P* ≤ *SPEC Q*
⟨*proof*⟩
lemma *OBTAIN-refine-iff*: *OBTAIN P* ≤⇓*R* (*OBTAIN Q*) ⟷ (*Ex Q* ⟶ *Ex P* ∧ *Collect P* ⊆ R^{-1} ‘‘*Collect Q*)
⟨*proof*⟩

lemma *OBTAIN-refine*[*refine*]:
assumes *RELATES R*
assumes ⋀*x*. *Q x* ⟹ *Ex P*
assumes ⋀*x y*. ⟦*Q x*; *P y*⟧ ⟹ ∃ *x'*. (*y*,*x'*)∈*R* ∧ *Q x'*
shows *OBTAIN P* ≤⇓*R* (*OBTAIN Q*)
⟨*proof*⟩

SELECT Combinator

Select some value with given property, or *None* if there is none.

definition *SELECT* :: (*'a* ⇒ *bool*) ⇒ *'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*
⟨*proof*⟩

lemma *SELECT-refine-iff*: (*SELECT P* ≤⇓(⟨*R*⟩*option-rel*) (*SELECT P'*))
⟷ (
(*Ex P'* ⟶ *Ex P*) ∧
(∀ *x*. *P x* ⟶ (∃ *x'*. (*x*,*x'*)∈*R* ∧ *P' x'*))
)
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *SELECT-as-SPEC*: *SELECT P* = *SPEC* (λ*None* ⇒ ∀ *x*. ¬*P x* | *Some x* ⇒ *P x*)
⟨*proof*⟩

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

x)
⟨*proof*⟩

lemma *SELECT-pw-simps*[*simp*]:
nofail (*SELECT P*)
inres (*SELECT P*) *None* ⟷ (∀ *x*. ¬*P x*)
inres (*SELECT P*) (*Some x*) ⟷ *P x*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemmas *WHILEI-mono* = *trimonoD-mono*[*OF WHILEI-body-trimono*]
lemmas *WHILEI-mono-ge* = *trimonoD-flatf-ge*[*OF WHILEI-body-trimono*]

lemma *WHILEI-unfold*: *WHILEI I b f x* = (
if (*I x*) *then* (*if b x then bind* (*f x*) (*WHILEI I b f*) *else return x*) *else top*)
⟨*proof*⟩

lemma *REC-mono-ref*[*refine-mono*]:
⟦*trimono B*; ⋀*D x*. *B D x* ≤ *B′ D x*⟧ ⟹ *REC B x* ≤ *REC B′ x*
⟨*proof*⟩

lemma *RECT-mono-ref*[*refine-mono*]:
⟦*trimono B*; ⋀*D x*. *B D x* ≤ *B′ D x*⟧ ⟹ *RECT B x* ≤ *RECT B′ x*
⟨*proof*⟩

lemma *WHILEI-weaken*:
assumes *IW*: ⋀*x*. *I x* ⟹ *I′ x*
shows *WHILEI I′ b f x* ≤ *WHILEI I b f x*
⟨*proof*⟩

lemma *WHILEIT-unfold*: *WHILEIT I b f x* = (
if (*I x*) *then*
(*if b x then bind* (*f x*) (*WHILEIT I b f*) *else return x*)
else top)
⟨*proof*⟩

lemma *WHILEIT-weaken*:

assumes *IW*: $\bigwedge x.\ I\ x \Longrightarrow I'\ x$
shows *WHILEIT I' b f x* $\leq$ *WHILEIT I b f x*
⟨*proof*⟩

lemma *WHILEI-le-WHILEIT*: *WHILEI I b f s* $\leq$ *WHILEIT I b f s*
⟨*proof*⟩

While without Annotated Invariant

lemma *WHILE-unfold*:
WHILE b f s = (*if b s then bind* (*f s*) (*WHILE b f*) *else return s*)
⟨*proof*⟩

lemma *WHILET-unfold*:
WHILET b f s = (*if b s then bind* (*f s*) (*WHILET b f*) *else return s*)
⟨*proof*⟩

lemma *transfer-WHILEIT-esc*[*refine-transfer*]:
assumes *REF*: $\bigwedge x.$ *return* (*f x*) $\leq$ *F x*
shows *return* (*while b f x*) $\leq$ *WHILEIT I b F x*
⟨*proof*⟩

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*
⟨*proof*⟩

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-ge* (*f x*) (*f' x*)$]\!] \Longrightarrow$ *flat-ge* (*WHILET b f s0*) (*WHILET b f' s0*)
$[\![\bigwedge x.$ *flat-ge* (*f x*) (*f' x*)$]\!] \Longrightarrow$ *flat-ge* (*WHILEIT I b f s0*) (*WHILEIT I b f' s0*)
⟨*proof*⟩

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 $\alpha\ (cWHILEIT\ I\ b\ f\ x) \leq aWHILEIT\ I\ b\ F\ x$
⟨*proof*⟩

lemma *transfer-WHILEI*[*refine-transfer*]:
assumes *REF*: $\bigwedge x.\ \alpha\ (f\ x) \leq F\ x$
shows $\alpha\ (cWHILEI\ I\ b\ f\ x) \leq aWHILEI\ I\ b\ F\ x$
⟨*proof*⟩

lemma *transfer-WHILE*[*refine-transfer*]:
assumes *REF*: $\bigwedge x.\ \alpha\ (f\ x) \leq F\ x$
shows $\alpha\ (cWHILE\ b\ f\ x) \leq aWHILE\ b\ F\ x$
⟨*proof*⟩

lemma *transfer-WHILET*[*refine-transfer*]:
assumes *REF*: $\bigwedge x.\ \alpha\ (f\ x) \leq F\ x$
shows $\alpha\ (cWHILET\ b\ f\ x) \leq aWHILET\ b\ F\ x$
⟨*proof*⟩

end

locale *generic-WHILE-rules* =
generic-WHILE bind return WHILEIT WHILEI WHILET WHILE
for *bind return SPEC WHILEIT WHILEI WHILET WHILE* +
assumes *iSPEC-eq*: $SPEC\ \Phi = Sup\ \{return\ x\ \mid x.\ \Phi\ x\}$
assumes *ibind-rule*: $[\![\ M \leq SPEC\ (\lambda x.\ f\ x \leq SPEC\ \Phi)\ ]\!] \Longrightarrow bind\ M\ f \leq SPEC\ \Phi$
begin

lemma *ireturn-eq*: $return\ x = SPEC\ ((=)\ x)$
⟨*proof*⟩

lemma *iSPEC-rule*: $(\bigwedge x.\ \Phi\ x \Longrightarrow \Psi\ x) \Longrightarrow SPEC\ \Phi \leq SPEC\ \Psi$
⟨*proof*⟩

lemma *ireturn-rule*: $\Phi\ x \Longrightarrow return\ x \leq SPEC\ \Phi$
⟨*proof*⟩

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
⟨*proof*⟩

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

⟨*proof*⟩

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* $\leq$ *SPEC* Φ
⟨*proof*⟩

lemma *WHILET-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 *WHILET b f s* $\leq$ *SPEC* Φ
⟨*proof*⟩

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
⟨*proof*⟩

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 $M \leq \Downarrow R\ (WHILEI\ I\ b\ f\ s)$
⟨*proof*⟩

lemma *ref-WHILEIT-invarI*:
assumes $I\ s \Longrightarrow M \leq \Downarrow R\ (WHILEIT\ I\ b\ f\ s)$
shows $M \leq \Downarrow R\ (WHILEIT\ I\ b\ f\ s)$
⟨*proof*⟩

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 $WHILEI\ I\ b\ f\ s \leq M\ s'$
⟨*proof*⟩

WHILE-refinement rule with invisible concrete steps. Intuitively, a concrete step may either refine an abstract step, or must not change the corresponding abstract state.

lemma *WHILEI-invisible-refine-genR*:
assumes *R0*: $I'\ s' \Longrightarrow (s,s')\in R'$
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*: $\bigwedge s\ s'.\ [\![\ (s,s')\in R';\ I'\ s';\ I\ s;\ b\ s\ ]\!]$
$\Longrightarrow f\ s \leq sup\ (\Downarrow R'\ (do\ \{ASSUME\ (b'\ s');\ f'\ s'\}))\ (\Downarrow R'\ (RETURN\ s'))$
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 $WHILEI\ I\ b\ f\ s \leq \Downarrow R\ (WHILEI\ I'\ b'\ f'\ s')$
⟨*proof*⟩

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*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R';\ b\ x;\ b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R'\ (f'\ x')$
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 $WHILEI\ I\ b\ f\ x \leq \Downarrow R\ (WHILEI\ I'\ b'\ f'\ x')$
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

WHILE-refinement rule with invisible concrete steps. Intuitively, a concrete step may either refine an abstract step, or must not change the corresponding abstract state.

lemma *WHILEI-invisible-refine*:
assumes *R0*: $I'\ s' \Longrightarrow (s,s')\in R$
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'*)
⟨*proof*⟩

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*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$

shows *WHILEI I b f x* $\leq \Downarrow R$ (*WHILEI I' b' f' x'*)
⟨*proof*⟩

lemma *WHILE-invisible-refine*:
assumes *R0*: $(s,s') \in R$
assumes *RB*: $\bigwedge s\ s'.\ [\![\ (s,s') \in R;\ b'\ s'\ ]\!] \Longrightarrow b\ s$
assumes *RS*: $\bigwedge s\ s'.\ [\![\ (s,s') \in R;\ b\ s\ ]\!]$
$\Longrightarrow f\ s \leq sup\ (\Downarrow R\ (do\ \{ASSUME\ (b'\ s');\ f'\ s'\}))\ (\Downarrow R\ (RETURN\ s'))$
shows *WHILE b f s* $\leq \Downarrow R$ (*WHILE b' f' s'*)
⟨*proof*⟩

lemma *WHILE-le-rule*:
assumes *R0*: $(s,s') \in R$
assumes *RS*: $\bigwedge W\ s\ s'.\ [\![\bigwedge s\ s'.\ (s,s') \in R \Longrightarrow W\ s \leq M\ s';\ (s,s') \in R]\!] \Longrightarrow$
do {*if b s then bind* (*f s*) *W else RETURN s*} $\leq M\ s'$
shows *WHILE b f s* $\leq M\ s'$
⟨*proof*⟩

lemma *WHILE-refine*[*refine*]:
assumes *R0*: $(x,x') \in R$
assumes *COND-REF*: $\bigwedge x\ x'.\ [\![\ (x,x') \in R\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x') \in R;\ b\ x;\ b'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
shows *WHILE b f x* $\leq \Downarrow R$ (*WHILE b' f' x'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *AIF-leI*: $[\![\Phi;\ \Phi \Longrightarrow S \leq S']\!] \Longrightarrow (if\ \Phi\ then\ S\ else\ FAIL) \leq S'$
⟨*proof*⟩
lemma *ref-AIFI*: $[\![\Phi \Longrightarrow S \leq \Downarrow R\ S']\!] \Longrightarrow S \leq \Downarrow R\ (if\ \Phi\ then\ S'\ else\ FAIL)$
⟨*proof*⟩

Refinement with generalized refinement relation. Required to exploit the fact that the condition does not hold at the end of the loop.

lemma *WHILEIT-refine-genR*:
assumes *R0*: $I'\ x' \Longrightarrow (x,x') \in R'$
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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

2.11.2 Autoref Setup

context begin interpretation *autoref-syn* ⟨*proof*⟩

lemma [*autoref-op-pat-def*]:
WHILEIT I ≡ *OP* (*WHILEIT I*)
WHILEI I ≡ *OP* (*WHILEI I*)
⟨*proof*⟩

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*:::$(R\to Id)\to(R\to\langle R\rangle nres\text{-}rel)\to R\to\langle R\rangle nres\text{-}rel)\$c'\$f'\$s'$)
$\in \langle R\rangle nres\text{-}rel$
⟨*proof*⟩

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*) ::: $(R\to Id)\to(R\to\langle R\rangle nres\text{-}rel)\to R\to\langle R\rangle nres\text{-}rel)\$c'\$f'\$s'$

$)\in\langle R\rangle nres\text{-}rel$
⟨*proof*⟩

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 $(WHILET\ c\ f,$
$(OP\ (WHILEIT\ I) ::: (R\to Id) \to (R\to\langle R\rangle nres\text{-}rel) \to R \to \langle R\rangle nres\text{-}rel)\$c'\$f'$
$)\in R \to \langle R\rangle nres\text{-}rel$
⟨*proof*⟩

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 $(WHILE\ c\ f\ s,$
$(OP\ WHILE ::: (R\to Id) \to (R\to\langle R\rangle nres\text{-}rel) \to R \to \langle R\rangle nres\text{-}rel)\$c'\$f'\$s'$
$)\in\langle R\rangle nres\text{-}rel$
⟨*proof*⟩

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 $(WHILE\ c\ f,$
$(OP\ WHILE ::: (R\to Id) \to (R\to\langle R\rangle nres\text{-}rel) \to R \to \langle R\rangle nres\text{-}rel)\$c'\$f'$
$)\in R \to \langle R\rangle nres\text{-}rel$
⟨*proof*⟩

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 $(WHILE\ c\ f\ s,$
$(OP\ (WHILEI\ I) ::: (R\to Id) \to (R\to\langle R\rangle nres\text{-}rel) \to R \to \langle R\rangle nres\text{-}rel)\$c'\$f'\$s'$
$)\in\langle R\rangle nres\text{-}rel$
⟨*proof*⟩

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 $(WHILE\ c\ f,$
$(OP\ (WHILEI\ I) ::: (R\to Id) \to (R\to\langle R\rangle nres\text{-}rel) \to R \to \langle R\rangle nres\text{-}rel)\$c'\$f'$
$)\in R \to \langle R\rangle nres\text{-}rel$

⟨*proof*⟩

end

2.11.3 Invariants

Tail Recursion

context begin
private lemma *tailrec-transform-aux1*:
assumes *RETURN s* ≤ *m*
shows *REC* (λ*D s*. *sup* (*a s* ⤜ *D*) (*b s*)) *s* ≤ *lfp* (λ*x*. *sup m* (*x*⤜*a*)) ⤜ *b*
(**is** *REC ?F s* ≤ *lfp ?f* ⤜ *b*)
⟨*proof*⟩ **corollary** *tailrec-transform1*:
fixes *m* :: *'a nres*
shows *m*⤜*REC* (λ*D s*. *sup* (*a s* ⤜ *D*) (*b s*)) ≤ *lfp* (λ*x*. *sup m* (*x*⤜*a*)) ⤜ *b*
⟨*proof*⟩ **lemma** *tailrec-transform-aux2*:
fixes *m* :: *'a nres*
shows *lfp* (λ*x*. *sup m* (*x*⤜*a*))
≤ *m* ⤜ *REC* (λ*D s*. *sup* (*a s* ⤜ *D*) (*RETURN s*))
(**is** *lfp ?f* ≤ *m* ⤜ *REC ?F*)
⟨*proof*⟩ **lemma** *tailrec-transform-aux3*: *REC* (λ*D s*. *sup* (*a s* ⤜ *D*) (*RETURN s*)) *s* ⤜ *b*
≤ *REC* (λ*D s*. *sup* (*a s* ⤜ *D*) (*b s*)) *s*
⟨*proof*⟩ **lemma** *tailrec-transform2*:
lfp (λ*x*. *sup m* (*bind x a*)) ⤜ *b* ≤ *m* ⤜ *REC* (λ*D s*. *sup* (*a s* ⤜ *D*) (*b s*))
⟨*proof*⟩

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

theorem *tailrec-transform*:
m ⤜ *REC* (λ*D s*. *sup* (*a s* ⤜ *D*) (*b s*)) = *lfp* (λ*x*. *sup m* (*bind x a*)) ⤜ *b*
⟨*proof*⟩

theorem *tailrec-transform'*:
m ⤜ *REC* (*tailrec-body a b*) = *lfp* (λ*x*. *sup m* (*bind x a*)) ⤜ *b*
⟨*proof*⟩

lemma *WHILE c f* =
REC (*tailrec-body*
(λ*s*. *do* {*ASSUME* (*c s*); *f s*})
(λ*s*. *do* {*ASSUME* (¬*c s*); *RETURN s*})
)
⟨*proof*⟩

lemma *WHILEI-tailrec-conv*: *WHILEI I c f* =

REC (*tailrec-body*
(λ*s*. *do* {*ASSERT* (*I s*); *ASSUME* (*c s*); *f s*})
(λ*s*. *do* {*ASSERT* (*I s*); *ASSUME* (¬*c s*); *RETURN s*})
)
⟨*proof*⟩

lemma *WHILEIT-tailrec-conv*: *WHILEIT I c f* =
RECT (*tailrec-body*
(λ*s*. *do* {*ASSERT* (*I s*); *ASSUME* (*c s*); *f s*})
(λ*s*. *do* {*ASSERT* (*I s*); *ASSUME* (¬*c s*); *RETURN s*})
)
⟨*proof*⟩

definition *WHILEI-lfp-body I m c f* ≡
(λ*x*. *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
}
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *msii-is-invar*:
$m \leq msii\ I\ m\ c\ f$
$m \leq msii\ I\ m\ c\ f \Longrightarrow bind\ (filter\text{-}ASSUME\ c\ (filter\text{-}ASSERT\ I\ m))\ f \leq msii\ I\ m\ c\ f$
⟨*proof*⟩

lemma *WHILE-msii-conv*: $m \gg\!= WHILEI\ I\ c\ f$
$= filter\text{-}ASSUME\ (Not\ o\ c)\ (filter\text{-}ASSERT\ I\ (msii\ I\ m\ c\ f))$
⟨*proof*⟩

lemma *msii-induct*:
assumes *I0*: $m0 \leq P$
assumes *IS*: $\bigwedge m.\ [\![m \leq msii\ I\ m0\ c\ f;\ m \leq P;$
$filter\text{-}ASSUME\ c\ (filter\text{-}ASSERT\ I\ m) \gg\!= f \leq msii\ I\ m0\ c\ f$
$]\!] \Longrightarrow filter\text{-}ASSUME\ c\ (filter\text{-}ASSERT\ I\ m) \gg\!= f \leq P$
shows $msii\ I\ m0\ c\ f \leq P$
⟨*proof*⟩

Reachable without fail

Reachable states in a while loop, ignoring failing states

inductive *rwof* :: $'a\ nres \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a\ nres) \Rightarrow 'a \Rightarrow bool$
for *m0 cond step*
where
init: $[\![\ m0{=}RES\ X;\ x{\in}X\ ]\!] \Longrightarrow rwof\ m0\ cond\ step\ x$
| *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*
⟨*proof*⟩

definition *is-rwof-invar m0 cond step I* ≡
$(m0 \leq_n SPEC\ I)$
$\wedge\ (\forall s.\ rwof\ m0\ cond\ step\ s \wedge I\ s \wedge cond\ s$
$\longrightarrow step\ s \leq_n SPEC\ I\ )$

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*
⟨*proof*⟩

lemma *rwof-cons*: ⟦*is-rwof-invar m0 cond step I*; *rwof m0 cond step s*⟧ ⟹ *I s*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *filter-nb-mono*: *m*≤*m*′ ⟹ *filter-nb cond m* ≤ *filter-nb cond m*′
⟨*proof*⟩

lemma *filter-nb-cont*:
filter-nb cond (*Sup M*) = *Sup* {*filter-nb cond m* | *m*. *m* ∈ *M*}
⟨*proof*⟩

lemma *filter-nb-FAIL*[*simp*]: *filter-nb cond FAIL* = *FAIL*
⟨*proof*⟩

lemma *filter-nb-RES*[*simp*]: *filter-nb cond* (*RES X*) = *RES* {*x*∈*X*. ¬*cond x*}
⟨*proof*⟩

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*
⟨*proof*⟩

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⟩


lemma bounded-WHILE'-eq:
  bounded-WHILE' n cond step m = bounded-WHILE n cond step m
  ⟨proof⟩

lemma mWHILE-unfold: m ≫= WHILE cond step = do {
    x ← m;
    if cond x then step x ≫= WHILE cond step
    else RETURN x
  }
  ⟨proof⟩

lemma WHILE-bounded-aux1:
  filter-nb cond (bounded-WHILE n cond step m) ≤ m ≫= WHILE cond step
  ⟨proof⟩

lemma WHILE-bounded-aux2:
  m ≫= WHILE cond step
    ≤ filter-nb cond (Sup {bounded-WHILE n cond step m | n. True})
  ⟨proof⟩


lemma WHILE-bounded:
```

m ⤜ *WHILE cond step*
= *filter-nb cond* (*Sup* {*bounded-WHILE n cond step m* | *n*. *True*})
⟨*proof*⟩

Relation to rwof

lemma *rwof-in-bounded-WHILE*:
assumes *rwof m0 cond step s*
shows ∃ *n*. *RETURN s* ≤ (*bounded-WHILE n cond step m0*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bounded-WHILE-RES-rwof*:
assumes *bounded-WHILE n cond step m0* = *RES X*
assumes *x*∈*X*
shows *rwof m0 cond step x*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *WHILE-le-WHILEI*: *WHILE b f s* ≤ *WHILEI I b f s*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *pw-rwof-in-bounded-WHILE*:
rwof m0 cond step x ⟹ ∃ *n*. *inres* (*bounded-WHILE n cond step m0*) *x*
⟨*proof*⟩

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*⟩

lemma *WHILE-refine-rwof*:
assumes *nofail* (*m* ≫= *WHILE c f*) ⟹ *mi* ≤ *SPEC* (λ*s*. *rwof m c f s* ∧ ¬*c s*)
shows *mi* ≤ *m* ≫= *WHILE c f*
⟨*proof*⟩

lemma *pw-rwof-init*:
assumes *NF*: *nofail* (*m0* ≫= *WHILE cond step*)
shows *inres m0 s* ⟹ *rwof m0 cond step s* **and** *nofail m0*
⟨*proof*⟩

lemma *rwof-init*:
assumes *NF*: *nofail* (*m0* ≫= *WHILE cond step*)
shows *m0* ≤ *SPEC* (*rwof m0 cond step*)
⟨*proof*⟩

lemma *pw-rwof-step'*:
assumes *NF*: *nofail* (*step s*)
assumes *R*: *rwof m0 cond step s*
assumes *C*: *cond s*
shows *inres* (*step s*) *s'* $\Longrightarrow$ *rwof m0 cond step s'*
⟨*proof*⟩

lemma *rwof-step'*:
⟦ *nofail* (*step s*); *rwof m0 cond step s*; *cond s* ⟧
$\Longrightarrow$ *step s* $\leq$ *SPEC* (*rwof m0 cond step*)
⟨*proof*⟩

lemma *pw-rwof-step*:
assumes *NF*: *nofail* (*m0* $\gg=$ *WHILE cond step*)
assumes *R*: *rwof m0 cond step s*
assumes *C*: *cond s*
shows *inres* (*step s*) *s'* $\Longrightarrow$ *rwof m0 cond step s'*
and *nofail* (*step s*)
⟨*proof*⟩

lemma *rwof-step*:
⟦ *nofail* (*m0* $\gg=$ *WHILE cond step*); *rwof m0 cond step s*; *cond s* ⟧
$\Longrightarrow$ *step s* $\leq$ *SPEC* (*rwof m0 cond step*)
⟨*proof*⟩

lemma (**in** −) *rwof-leof-init*: *m* $\leq_n$ *SPEC* (*rwof m c f*)
⟨*proof*⟩

lemma (**in** −) *rwof-leof-step*: ⟦*rwof m c f s*; *c s*⟧ $\Longrightarrow$ *f s* $\leq_n$ *SPEC* (*rwof m c f*)
⟨*proof*⟩

lemma (**in** −) *rwof-step-refine*:
assumes *NF*: *nofail* (*m0* $\gg=$ *WHILE cond step*)
assumes *A*: *rwof m0 cond step' s*
assumes *FR*: $\bigwedge$*s*. ⟦ *rwof m0 cond step s*; *cond s* ⟧ $\Longrightarrow$ *step' s* $\leq_n$ *step s*
shows *rwof m0 cond step s*
⟨*proof*⟩

Adding Invariants

lemma *WHILE-eq-I-rwof*: *m* $\gg=$ *WHILE c f* = *m* $\gg=$ *WHILEI* (*rwof m c f*) *c f*
⟨*proof*⟩

lemma *WHILET-eq-I-rwof*: *m* $\gg=$ *WHILET c f* = *m* $\gg=$ *WHILEIT* (*rwof m c f*) *c f*
⟨*proof*⟩

Refinement

lemma *rwof-refine*:
assumes *RW*: *rwof m c f s*
assumes *NF*: *nofail* ($m' \gg\!= WHILE\ c'\ f'$)
assumes *M*: $m \leq_n \Downarrow R\ m'$
assumes *C*: $\bigwedge s\ s'.\ [\![(s,s')\in R;\ rwof\ m\ c\ f\ s;\ rwof\ m'\ c'\ f'\ s']\!] \Longrightarrow c\ s = c'\ s'$
assumes *S*: $\bigwedge s\ s'.\ [\![(s,s')\in R;\ rwof\ m\ c\ f\ s;\ rwof\ m'\ c'\ f'\ s';\ c\ s;\ c'\ s']\!] \Longrightarrow f\ s \leq_n \Downarrow R\ (f'\ s')$
shows $\exists s'.\ (s,s')\in R \wedge rwof\ m'\ c'\ f'\ s'$
⟨*proof*⟩

Total Correct Loops

In this theory, we show that every non-failing total-correct while loop gives rise to a compatible well-founded relation

definition *rwof-rel*
— Transitions in a while-loop as relation
where *rwof-rel init cond step*
$\equiv \{(s,s').\ rwof\ init\ cond\ step\ s \wedge cond\ s \wedge inres\ (step\ s)\ s'\}$

lemma *rwof-rel-spec*: $[\![rwof\ init\ cond\ step\ s;\ cond\ s]\!]$
$\Longrightarrow step\ s \leq_n SPEC\ (\lambda s'.\ (s,s')\in rwof\text{-}rel\ init\ cond\ step)$
⟨*proof*⟩

lemma *rwof-reachable*:
assumes *rwof init cond step s*
shows $\exists s0.\ inres\ init\ s0 \wedge (s0,s)\in(rwof\text{-}rel\ init\ cond\ step)^*$
⟨*proof*⟩

theorem *nofail-WHILEIT-wf-rel*:
assumes *NF*: *nofail* (*init* $\gg\!=$ *WHILEIT I cond step*)
shows *wf* $((rwof\text{-}rel\ init\ cond\ step)^{-1})$
⟨*proof*⟩

2.11.4 Convenience

Iterate over range of list

lemma *range-set-WHILE*:
assumes *CEQ*: $\bigwedge i\ s.\ c\ (i,s) \longleftrightarrow i<u$
assumes *F0*: $F\ \{\}\ s0 = s0$
assumes *Fs*: $\bigwedge s\ i\ X.\ [\![l\leq i;\ i<u]\!]$
$\Longrightarrow f\ (i,\ (F\ X\ s)) \leq SPEC\ (\lambda(i',r).\ i'=i+1 \wedge r=F\ (insert\ (list!i)\ X)\ s)$
shows *WHILET c f* $(l,s0)$
$\leq SPEC\ (\lambda(\text{-},r).\ r=F\ \{list!i \mid i.\ l\leq i \wedge i<u\}\ s0)$
⟨*proof*⟩

end

2.12 Deterministic Monad

theory *Refine-Det*
imports
HOL−Library.Monad-Syntax
Generic/RefineG-Assert
Generic/RefineG-While
begin

2.12.1 Deterministic Result Lattice

We define the flat complete lattice of deterministic program results:

datatype *'a dres* =
dSUCCEEDi — No result
| *dFAILi* — Failure
| *dRETURN 'a* — Regular result

instantiation *dres* :: (*type*) *complete-lattice*
begin
definition *top-dres* ≡ *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
⟨*proof*⟩

fun *inf-dres* **where**
inf dFAILi x = x |
inf x dFAILi = x |
inf (*dRETURN a*) (*dRETURN b*) = (*if a=b then dRETURN b else 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
⟨*proof*⟩

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
)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

instance
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

lemma *dres-chain-eq-res*:

is-chain M $\Longrightarrow$
dRETURN r $\in$ *M* $\Longrightarrow$ *dRETURN s* $\in$ *M* $\Longrightarrow$ *r=s*
⟨*proof*⟩

lemma *dres-Sup-chain-cases*:
assumes *CHAIN*: *is-chain M*
obtains *M* $\subseteq$ {*dSUCCEED*} *Sup M = dSUCCEED*
| *r* **where** *M* $\subseteq$ {*dSUCCEED*,*dRETURN r*} *dRETURN r*$\in$*M* *Sup M = dRETURN r*
| *dFAIL*$\in$*M* *Sup M = dFAIL*
⟨*proof*⟩

lemma *dres-Inf-chain-cases*:
assumes *CHAIN*: *is-chain M*
obtains *M* $\subseteq$ {*dFAIL*} *Inf M = dFAIL*
| *r* **where** *M* $\subseteq$ {*dFAIL*,*dRETURN r*} *dRETURN r*$\in$*M* *Inf M = dRETURN r*
| *dSUCCEED*$\in$*M* *Inf M = dSUCCEED*
⟨*proof*⟩

lemma *dres-internal-simps*[*simp*]:
dSUCCEEDi = dSUCCEED
dFAILi = dFAIL
⟨*proof*⟩

Monad Operations

function *dbind* **where**
dbind dFAIL - = dFAIL
| *dbind dSUCCEED - = dSUCCEED*
| *dbind* (*dRETURN x*) *f* = *f x*
⟨*proof*⟩
termination ⟨*proof*⟩

adhoc-overloading
Monad-Syntax.bind dbind

lemma [*code*]:
dbind (*dRETURN x*) *f* = *f x*
dbind (*dSUCCEEDi*) *f* = *dSUCCEEDi*
dbind (*dFAILi*) *f* = *dFAILi*
⟨*proof*⟩

lemma *dres-monad1*[*simp*]: *dbind* (*dRETURN x*) *f* = *f x*
⟨*proof*⟩
lemma *dres-monad2*[*simp*]: *dbind M dRETURN* = *M*
⟨*proof*⟩

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

⟨*proof*⟩

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

lemma *dbind-mono*[*refine-mono*]:
⟦ $M \leq M'$; $\bigwedge x.\ dRETURN\ x \leq M \Longrightarrow f\ x \leq f'\ x$ ⟧ $\Longrightarrow$ *dbind* $M\ f \leq$ *dbind* $M'\ f'$
⟦ *flat-ge* $M\ M'$; $\bigwedge x.$ *flat-ge* $(f\ x)\ (f'\ x)$ ⟧ $\Longrightarrow$ *flat-ge* (*dbind* $M\ f$) (*dbind* $M'\ f'$)
⟨*proof*⟩

lemma *dbind-mono1*[*simp*, *intro*!]: *mono dbind*
⟨*proof*⟩

lemma *dbind-mono2*[*simp*, *intro*!]: *mono* (*dbind* M)
⟨*proof*⟩

lemma *dr-mono-bind*:
assumes *MA*: *mono* A **and** *MB*: $\bigwedge s.$ *mono* $(B\ s)$
shows *mono* $(\lambda F\ s.$ *dbind* $(A\ F\ s)\ (\lambda s'.\ B\ s\ F\ s'))$
⟨*proof*⟩

lemma *dr-mono-bind'*: *mono* $(\lambda F\ s.$ *dbind* $(f\ s)\ F)$
⟨*proof*⟩

lemmas *dr-mono* = *mono-if dr-mono-bind dr-mono-bind' mono-const mono-id*

lemma [*refine-mono*]:
dbind dSUCCEED f = *dSUCCEED*
dbind dFAIL f = *dFAIL*
⟨*proof*⟩

definition *dASSERT* ≡ *iASSERT dRETURN*
definition *dASSUME* ≡ *iASSUME dRETURN*
interpretation *dres-assert*: *generic-Assert dbind dRETURN dASSERT dASSUME*
⟨*proof*⟩

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
⟨*proof*⟩

lemmas [*code*] =
dres-while.WHILEIT-unfold
dres-while.WHILEI-unfold

dres-while.WHILET-unfold
dres-while.WHILE-unfold

Syntactic criteria to prove $s \neq dSUCCEED$

lemma *dres-ne-bot-basic*[*refine-transfer*]:
$dFAIL \neq dSUCCEED$
$\bigwedge x.\ dRETURN\ x \neq dSUCCEED$
$\bigwedge m\ f.\ [\![\ m \neq dSUCCEED;\ \bigwedge x.\ f\ x \neq dSUCCEED\ ]\!] \Longrightarrow dbind\ m\ f \neq dSUCCEED$
$\bigwedge \Phi.\ dASSERT\ \Phi \neq dSUCCEED$
$\bigwedge b\ m1\ m2.\ [\![\ m1 \neq dSUCCEED;\ m2 \neq dSUCCEED\ ]\!] \Longrightarrow If\ b\ m1\ m2 \neq dSUCCEED$
$\bigwedge x\ f.\ [\![\ \bigwedge x.\ f\ x \neq dSUCCEED\ ]\!] \Longrightarrow Let\ x\ f \neq dSUCCEED$
$\bigwedge g\ p.\ [\![\ \bigwedge x1\ x2.\ g\ x1\ x2 \neq dSUCCEED\ ]\!] \Longrightarrow case\text{-}prod\ g\ p \neq dSUCCEED$
$\bigwedge fn\ fs\ x.$
$[\![\ fn \neq dSUCCEED;\ \bigwedge v.\ fs\ v \neq dSUCCEED\ ]\!] \Longrightarrow case\text{-}option\ fn\ fs\ x \neq dSUCCEED$
$\bigwedge fn\ fc\ x.$
$[\![\ fn \neq dSUCCEED;\ \bigwedge x\ xs.\ fc\ x\ xs \neq dSUCCEED\ ]\!] \Longrightarrow case\text{-}list\ fn\ fc\ x \neq dSUCCEED$
⟨*proof*⟩

lemma *dres-ne-bot-RECT*[*refine-transfer*]:
assumes *A*: $\bigwedge f\ x.\ [\![\ \bigwedge x.\ f\ x \neq dSUCCEED\ ]\!] \Longrightarrow B\ f\ x \neq dSUCCEED$
shows $RECT\ B\ x \neq dSUCCEED$
⟨*proof*⟩

lemma *dres-ne-bot-dWHILEIT*[*refine-transfer*]:
assumes $\bigwedge x.\ f\ x \neq dSUCCEED$
shows $dWHILEIT\ I\ b\ f\ s \neq dSUCCEED$ ⟨*proof*⟩

lemma *dres-ne-bot-dWHILET*[*refine-transfer*]:
assumes $\bigwedge x.\ f\ x \neq dSUCCEED$
shows $dWHILET\ b\ f\ s \neq dSUCCEED$ ⟨*proof*⟩

end

2.13 Partial Function Package Setup

theory *Refine-Pfun*
imports *Refine-Basic Refine-Det*
begin

In this theory, we set up the partial function package to be used with our refinement framework.

2.13.1 Nondeterministic Result Monad

interpretation *nrec*:
partial-function-definitions $(\leq)$ *Sup*::$'a$ *nres set* $\Rightarrow$ $'a$ *nres*
⟨*proof*⟩

lemma *nrec-admissible*: *nrec.admissible* $(\lambda f{::}'a \Rightarrow 'b\ nres).$
$(\forall x0.\ f\ x0 \leq SPEC\ (P\ x0)))$
⟨*proof*⟩

⟨*ML*⟩

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* $(\leq)$) $(\leq)$ $(\lambda f.\ bind\ (B\ f)\ (\lambda y.\ C\ y\ f))$
⟨*proof*⟩

2.13.2 Deterministic Result Monad

interpretation *drec*:
partial-function-definitions $(\leq)$ *Sup*::$'a$ *dres set* $\Rightarrow$ $'a$ *dres*
⟨*proof*⟩

lemma *drec-admissible*: *drec.admissible* $(\lambda f{::}'a \Rightarrow 'b\ dres).$
$(\forall x.\ P\ x \longrightarrow$
$(f\ x \neq dFAIL \wedge$
$(\forall r.\ f\ x = dRETURN\ r \longrightarrow Q\ x\ r))))$
⟨*proof*⟩

⟨*ML*⟩

lemma *drec-bind-mono-pfun*[*partial-function-mono*]:
fixes $C :: 'a \Rightarrow ('b \Rightarrow 'c\ dres) \Rightarrow ('d\ dres)$
shows
$[\![$ *monotone* (*fun-ord* $(\leq)$) $(\leq)$ B;
$\bigwedge y.$ *monotone* (*fun-ord* $(\leq)$) $(\leq)$ $(\lambda f.\ C\ y\ f)$ $]\!] \Longrightarrow$
monotone (*fun-ord* $(\leq)$) $(\leq)$ $(\lambda f.\ dbind\ (B\ f)\ (\lambda y.\ C\ y\ f))$
⟨*proof*⟩

end

2.14 Transfer Setup

theory *Refine-Transfer*
imports
Refine-Basic
Refine-While
Refine-Det
Generic/RefineG-Transfer
begin

2.14.1 Transfer to Deterministic Result Lattice

TODO: Once lattice and ccpo are connected, also transfer to option monad, that is a ccpo, but no complete lattice!

Connecting Deterministic and Non-Deterministic Result Lattices

definition *nres-of r* ≡ *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*
⟨*proof*⟩

lemma *nres-of-mono*: *mono nres-of*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *nres-correctD*:
assumes *nres-of S* ≤ *SPEC* Φ
shows
S=*dRETURN x* ⟹ Φ *x*
S≠*dFAIL*
⟨*proof*⟩

Transfer Theorems Setup

interpretation *dres*: *dist-transfer nres-of*
⟨*proof*⟩

lemma *nres-of-transfer*[*refine-transfer*]: *nres-of x* ≤ *nres-of x* ⟨*proof*⟩

lemma *det-FAIL*[*refine-transfer*]: *nres-of* (*dFAIL*) ≤ *FAIL* ⟨*proof*⟩
lemma *det-SUCCEED*[*refine-transfer*]: *nres-of* (*dSUCCEED*) ≤ *SUCCEED* ⟨*proof*⟩
lemma *det-SPEC*: Φ *x* ⟹ *nres-of* (*dRETURN x*) ≤ *SPEC* Φ ⟨*proof*⟩
lemma *det-RETURN*[*refine-transfer*]:
nres-of (*dRETURN x*) ≤ *RETURN x* ⟨*proof*⟩
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*
⟨*proof*⟩

interpretation *det-assert*: *transfer-generic-Assert-remove*
bind RETURN ASSERT ASSUME
nres-of
⟨*proof*⟩

interpretation *det-while*: *transfer-WHILE*
dbind dRETURN dWHILEIT dWHILEI dWHILET dWHILE
bind RETURN WHILEIT WHILEI WHILET WHILE nres-of
⟨*proof*⟩

2.14.2 Transfer to Plain Function

interpretation *plain*: *transfer RETURN* ⟨*proof*⟩

lemma *plain-RETURN*[*refine-transfer*]: *RETURN a* ≤ *RETURN a* ⟨*proof*⟩
lemma *plain-bind*[*refine-transfer*]:
⟦*RETURN x* ≤ *M*; ⋀*x*. *RETURN* (*f x*) ≤ *F x*⟧ ⟹ *RETURN* (*Let x f*) ≤ *bind M F*
⟨*proof*⟩

interpretation *plain-assert*: *transfer-generic-Assert-remove*
bind RETURN ASSERT ASSUME
RETURN
⟨*proof*⟩

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'*
⟨*proof*⟩

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* ⟨*proof*⟩
lemma *autoref-detI*:
assumes (*b*,*a*)∈⟨*R*⟩*nres-rel*
assumes *RETURN c* ≤ *b*
assumes *c* = *detTAG d*
shows (*RETURN d*, *a*)∈⟨*R*⟩*nres-rel*
⟨*proof*⟩

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*}
⟨*proof*⟩

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

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

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

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*}

⟨*proof*⟩

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

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

lemma *plain-nres-rel-as-br-conv*:
⟨*R*⟩*plain-nres-rel* = *br RETURN* (λ-. *True*) *O* ⟨*R*⟩*nres-rel*
⟨*proof*⟩

2.14.5 Post-Simplification Setup

lemma *dres-unit-simps*[*refine-transfer-post-simp*]:
dbind (*dRETURN* (*u*::*unit*)) *f* = *f* ()
⟨*proof*⟩

lemma *Let-dRETURN-simp*[*refine-transfer-post-simp*]:
Let m dRETURN = *dRETURN m* ⟨*proof*⟩

lemmas [*refine-transfer-post-simp*] = *dres-monad-laws*

end

2.15 Foreach Loops

theory *Refine-Foreach*
imports
Refine-While
Refine-Pfun
Refine-Transfer
Refine-Heuristics

begin

A common pattern for loop usage is iteration over the elements of a set. This theory provides the *FOREACH*-combinator, that iterates over each element of a set.

2.15.1 Auxilliary Lemmas

The following lemma is commonly used when reasoning about iterator invariants. It helps converting the set of elements that remain to be iterated over to the set of elements already iterated over.

lemma *it-step-insert-iff*:
it ⊆ *S* ⟹ *x*∈*it* ⟹ *S*−(*it*−{*x*}) = *insert x* (*S*−*it*) ⟨*proof*⟩

2.15.2 Definition

Foreach-loops come in different versions, depending on whether they have an annotated invariant (I), a termination condition (C), and an order (O).

Note that asserting that the set is finite is not necessary to guarantee termination. However, we currently provide only iteration over finite sets, as this also matches the ICF concept of iterators.

definition *FOREACH-body f* ≡ 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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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 *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 *RR-OK*: ⋀*x y*. ⟦*x* ∈ *S*; *y* ∈ *S*; *RR x y*⟧ ⟹ *RR'* (*α x*) (*α y*)
assumes *REF0*: ((*S*,*σ0*),(*α*‘*S*,*σ0'*)) ∈ *R*

assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';\ \Phi'\ it'\ \sigma';\ \Phi\ it\ \sigma;$
$\forall x{\in}S{-}it.\ \forall y{\in}it.\ RR\ x\ y;\ \forall x{\in}S'{-}it'.\ \forall y{\in}it'.\ RR'\ x\ y;$
$it'{=}\alpha\text{`}it;\ ((it,\sigma),(it',\sigma')){\in}R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';\ \Phi'\ it'\ \sigma';$
$\forall x{\in}S{-}it.\ \forall y{\in}it.\ RR\ x\ y;\ \forall x{\in}S'{-}it'.\ \forall y{\in}it'.\ RR'\ x\ y;$
$it'{=}\alpha\text{`}it;\ ((it,\sigma),(it',\sigma')){\in}R$
$]\!] \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';\ \Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';$
$\forall x{\in}S{-}it.\ \forall y{\in}it.\ RR\ x\ y;\ \forall x{\in}S'{-}it'.\ \forall y{\in}it'.\ RR'\ x\ y;$
$x'{=}\alpha\ x;\ it'{=}\alpha\text{`}it;\ ((it,\sigma),(it',\sigma')){\in}R;$
$x{\in}it;\ \forall y{\in}it{-}\{x\}.\ RR\ x\ y;$
$x'{\in}it';\ \forall y'{\in}it'{-}\{x'\}.\ RR'\ x'\ y';$
$c\ \sigma;\ c'\ \sigma'$
$]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\sigma').\ ((it{-}\{x\},\sigma),(it'{-}\{x'\},\sigma')){\in}R\})\ (f'\ x'\ \sigma')$
assumes *REF-R-DONE*: $\bigwedge\sigma\ \sigma'.\ [\![\ \Phi\ \{\}\ \sigma;\ \Phi'\ \{\}\ \sigma';\ ((\{\},\sigma),(\{\},\sigma')){\in}R\ ]\!]$
$\Longrightarrow (\sigma,\sigma'){\in}R'$
assumes *REF-R-BRK*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';\ \Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';$
$\forall x{\in}S{-}it.\ \forall y{\in}it.\ RR\ x\ y;\ \forall x{\in}S'{-}it'.\ \forall y{\in}it'.\ RR'\ x\ y;$
$it'{=}\alpha\text{`}it;\ ((it,\sigma),(it',\sigma')){\in}R;$
$it{\neq}\{\};\ it'{\neq}\{\};$
$\neg c\ \sigma;\ \neg c'\ \sigma'$
$]\!] \Longrightarrow (\sigma,\sigma'){\in}R'$
shows *FOREACHoci RR* Φ *S c f* $\sigma 0 \leq \Downarrow R'$ (*FOREACHoci RR'* Φ' *S' c' f'* $\sigma 0'$)

⟨*proof*⟩

lemma *FOREACHoci-refine*:
fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S\ set$
fixes $S' :: 'Sa\ set$
assumes *INJ*: *inj-on* $\alpha\ S$
assumes *REFS*: $S' = \alpha\text{`}S$
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*: $\Phi''\ S\ \sigma 0\ (\alpha\text{`}S)\ \sigma 0'$
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'{=}\alpha\text{`}it;\ it{\subseteq}S;\ it'{\subseteq}S';\ \Phi'\ it'\ \sigma';\ \ \Phi''\ it\ \sigma\ it'\ \sigma';\ \Phi\ it\ \sigma;\ (\sigma,\sigma'){\in}R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'{=}\alpha\text{`}it;\ it{\subseteq}S;\ it'{\subseteq}S';\ \Phi'\ it'\ \sigma';\ \Phi''\ it\ \sigma\ it'\ \sigma';\ (\sigma,\sigma'){\in}R$
$]\!] \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![\forall y{\in}it{-}\{x\}.\ RR\ x\ y;$
$x'{=}\alpha\ x;\ x{\in}it;\ x'{\in}it';\ it'{=}\alpha\text{`}it;\ it{\subseteq}S;\ it'{\subseteq}S';$

$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ \Phi''\ it\ \sigma\ it'\ \sigma';\ c\ \sigma;\ c'\ \sigma';$
$(\sigma,\sigma')\in R$
$\rrbracket \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\ \sigma').\ (\sigma,\ \sigma') \in R \wedge \Phi''\ (it - \{x\})\ \sigma\ (it' - \{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows *FOREACHoci RR* Φ *S c f* $\sigma 0 \leq \Downarrow R$ (*FOREACHoci RR'* Φ' *S' c' f'* $\sigma 0'$)

⟨*proof*⟩

lemma *FOREACHoci-refine-rcg*[*refine*]:
fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S\ set$
fixes $S' :: 'Sa\ set$
assumes *INJ*: *inj-on* α *S*
assumes *REFS*: $S' = \alpha`S$
assumes *REF0*: $(\sigma 0,\sigma 0')\in R$
assumes *RR-OK*: $\bigwedge x\ y.\ [\![x \in S;\ y \in S;\ RR\ x\ y]\!] \Longrightarrow RR'\ (\alpha\ x)\ (\alpha\ y)$
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';\ \Phi'\ it'\ \sigma';\ \Phi\ it\ \sigma;\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![\ \forall y\in it-\{x\}.\ RR\ x\ y;$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ c\ \sigma;\ c'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \leq \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACHoci RR* Φ *S c f* $\sigma 0 \leq \Downarrow R$ (*FOREACHoci RR'* Φ' *S' c' f'* $\sigma 0'$)
⟨*proof*⟩

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* $\sigma 0$
⟨*proof*⟩

lemma *FOREACHoci-weaken-order*:
assumes *RRREF*: $\bigwedge x\ y.\ x \in S \Longrightarrow y \in S \Longrightarrow RR\ x\ y \Longrightarrow RR'\ x\ y$
shows *FOREACHoci RR I S c f* $\sigma 0 \leq$ *FOREACHoci RR' I S c f* $\sigma 0$
⟨*proof*⟩

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* $\Phi\ S\ f\ \sigma 0 \leq \Downarrow R'\ (FOREACHoi\ RR'\ \Phi'\ S'\ f'\ \sigma 0')$
⟨*proof*⟩

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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![\forall y\in it-\{x\}.\ RR\ x\ y;$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ \Phi''\ it\ \sigma\ it'\ \sigma';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\sigma').\ (\sigma,\sigma')\in R \wedge \Phi''\ (it-\{x\})\ \sigma\ (it'-\{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows *FOREACHoi RR* $\Phi\ S\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHoi\ RR'\ \Phi'\ S'\ f'\ \sigma 0')$
⟨*proof*⟩

lemma *FOREACHoi-refine-rcg*[*refine*]:
fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S\ set$
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*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'{=}\alpha\text{`}it;\ it{\subseteq}S;\ it'{\subseteq}S';\ \Phi'\ it'\ \sigma';\ (\sigma,\sigma'){\in}R$
$]\!] \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![\ \forall y{\in}it{-}\{x\}.\ RR\ x\ y;$
$x'{=}\alpha\ x;\ x{\in}it;\ x'{\in}it';\ it'{=}\alpha\text{`}it;\ it{\subseteq}S;\ it'{\subseteq}S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ (\sigma,\sigma'){\in}R$
$]\!] \Longrightarrow f\ x\ \sigma \le \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACHoi RR* $\Phi\ S\ f\ \sigma 0 \le \Downarrow R\ (FOREACHoi\ RR'\ \Phi'\ S'\ f'\ \sigma 0')$
⟨*proof*⟩

lemma *FOREACHci-refine-genR*:
fixes $\alpha :: 'S \Rightarrow 'Sa$ — Abstraction mapping of elements
fixes $S :: 'S\ set$ — Concrete set
fixes $S' :: 'Sa\ set$ — Abstract set
fixes $\sigma 0 :: '\sigma$
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*]: $S' = \alpha\text{`}S$
assumes *REF0*: $((S,\sigma 0),(\alpha\text{`}S,\sigma 0')) \in R$
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';\ \Phi'\ it'\ \sigma';\ \Phi\ it\ \sigma;$
$it'{=}\alpha\text{`}it;\ ((it,\sigma),(it',\sigma')){\in}R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';\ \Phi'\ it'\ \sigma';$
$it'{=}\alpha\text{`}it;\ ((it,\sigma),(it',\sigma')){\in}R$
$]\!] \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';\ \Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';$
$x'{=}\alpha\ x;\ it'{=}\alpha\text{`}it;\ ((it,\sigma),(it',\sigma')){\in}R;$
$x{\in}it;\ x'{\in}it';$
$c\ \sigma;\ c'\ \sigma'$
$]\!] \Longrightarrow f\ x\ \sigma$
$\le \Downarrow(\{(\sigma,\ \sigma').\ ((it{-}\{x\},\sigma),(it'{-}\{x'\},\sigma')){\in}R\})\ (f'\ x'\ \sigma')$
assumes *REF-R-DONE*: $\bigwedge \sigma\ \sigma'.\ [\![\ \Phi\ \{\}\ \sigma;\ \Phi'\ \{\}\ \sigma';\ ((\{\},\sigma),(\{\},\sigma')){\in}R\ ]\!]$
$\Longrightarrow (\sigma,\sigma'){\in}R'$
assumes *REF-R-BRK*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';\ \Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';$
$it'{=}\alpha\text{`}it;\ ((it,\sigma),(it',\sigma')){\in}R;$
$it{\neq}\{\};\ it'{\neq}\{\};$
$\neg c\ \sigma;\ \neg c'\ \sigma'$
$]\!] \Longrightarrow (\sigma,\sigma'){\in}R'$
shows *FOREACHci* $\Phi\ S\ c\ f\ \sigma 0 \le \Downarrow R'\ (FOREACHci\ \Phi'\ S'\ c'\ f'\ \sigma 0')$
⟨*proof*⟩

lemma *FOREACHci-refine*:
fixes $\alpha :: 'S \Rightarrow 'Sa$

fixes $S :: 'S\ set$
fixes $S' :: 'Sa\ set$
assumes *INJ*: *inj-on* $\alpha\ S$
assumes *REFS*: $S' = \alpha`S$
assumes *REF0*: $(\sigma 0,\sigma 0')\in R$
assumes *REFPHI0*: $\Phi''\ S\ \sigma 0\ (\alpha`S)\ \sigma 0'$
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';\ \Phi'\ it'\ \sigma';\ \Phi''\ it\ \sigma\ it'\ \sigma';\ \Phi\ it\ \sigma;\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ \Phi''\ it\ \sigma\ it'\ \sigma';\ c\ \sigma;\ c'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\ \sigma').\ (\sigma,\ \sigma')\in R\wedge\Phi''\ (it-\{x\})\ \sigma\ (it'-\{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows $FOREACHci\ \Phi\ S\ c\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHci\ \Phi'\ S'\ c'\ f'\ \sigma 0')$
⟨*proof*⟩

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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ c\ \sigma;\ c'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \leq \Downarrow R\ (f'\ x'\ \sigma')$
shows $FOREACHci\ \Phi\ S\ c\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHci\ \Phi'\ S'\ c'\ f'\ \sigma 0')$
⟨*proof*⟩

lemma *FOREACHci-weaken*:
assumes *IREF*: $\bigwedge it\ \sigma.\ it\subseteq S \Longrightarrow I\ it\ \sigma \Longrightarrow I'\ it\ \sigma$
shows $FOREACHci\ I'\ S\ c\ f\ \sigma 0 \leq FOREACHci\ I\ S\ c\ f\ \sigma 0$
⟨*proof*⟩

lemma *FOREACHi-rule*[*refine-vcg*]:
assumes *FIN*: *finite* S

assumes *I0*: $I\ S\ \sigma 0$
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ [\![\ x{\in}it;\ it{\subseteq}S;\ I\ it\ \sigma\ ]\!] \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it{-}\{x\}))$
assumes *II*: $\bigwedge\sigma.\ [\![I\ \{\}\ \sigma]\!] \Longrightarrow P\ \sigma$
shows *FOREACHi I S f* $\sigma 0 \leq SPEC\ P$
⟨*proof*⟩

lemma *FOREACHc-rule*:
assumes *FIN*: *finite S*
assumes *I0*: $I\ S\ \sigma 0$
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ [\![\ x{\in}it;\ it{\subseteq}S;\ I\ it\ \sigma;\ c\ \sigma\ ]\!] \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it{-}\{x\}))$
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* $\sigma 0 \leq SPEC\ P$
⟨*proof*⟩

lemma *FOREACH-rule*:
assumes *FIN*: *finite S*
assumes *I0*: $I\ S\ \sigma 0$
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ [\![\ x{\in}it;\ it{\subseteq}S;\ I\ it\ \sigma\ ]\!] \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it{-}\{x\}))$
assumes *II*: $\bigwedge\sigma.\ [\![I\ \{\}\ \sigma]\!] \Longrightarrow P\ \sigma$
shows *FOREACH S f* $\sigma 0 \leq SPEC\ P$
⟨*proof*⟩

lemma *FOREACHc-refine-genR*:
fixes $\alpha :: 'S \Rightarrow 'Sa$ — Abstraction mapping of elements
fixes $S :: 'S\ set$ — Concrete set
fixes $S' :: 'Sa\ set$ — Abstract set
fixes $\sigma 0 :: '\sigma$
fixes $\sigma 0' :: '\sigma a$
fixes $R :: (('S\ set \times '\sigma) \times ('Sa\ set \times '\sigma a))\ set$
assumes *INJ*: *inj-on* $\alpha\ S$
assumes *REFS*[*simp*]: $S' = \alpha\text{`}S$
assumes *REF0*: $((S,\sigma 0),(\alpha\text{`}S,\sigma 0')) \in R$
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';$
$it'{=}\alpha\text{`}it;\ ((it,\sigma),(it',\sigma')){\in}R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$it{\subseteq}S;\ it'{\subseteq}S';$
$x'{=}\alpha\ x;\ it'{=}\alpha\text{`}it;\ ((it,\sigma),(it',\sigma')){\in}R;$
$x{\in}it;\ x'{\in}it';$
$c\ \sigma;\ c'\ \sigma'$
$]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\ \sigma').\ ((it{-}\{x\},\sigma),(it'{-}\{x'\},\sigma')){\in}R\})\ (f'\ x'\ \sigma')$
assumes *REF-R-DONE*: $\bigwedge\sigma\ \sigma'.\ [\![\ ((\{\},\sigma),(\{\},\sigma')){\in}R\ ]\!]$

$\Longrightarrow (\sigma,\sigma')\in R'$
assumes *REF-R-BRK*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it\subseteq S;\ it'\subseteq S';$
$it'=\alpha`it;\ ((it,\sigma),(it',\sigma'))\in R;$
$it\neq\{\};\ it'\neq\{\};$
$\neg c\ \sigma;\ \neg c'\ \sigma'$
$]\!] \Longrightarrow (\sigma,\sigma')\in R'$
shows *FOREACHc S c f* $\sigma 0 \leq \Downarrow R'$ (*FOREACHc S' c' f'* $\sigma 0'$)
⟨*proof*⟩

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*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';\ \Phi''\ it\ \sigma\ it'\ \sigma';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi''\ it\ \sigma\ it'\ \sigma';\ c\ \sigma;\ c'\ \sigma';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\sigma').\ (\sigma,\sigma')\in R \wedge \Phi''\ (it-\{x\})\ \sigma\ (it'-\{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows *FOREACHc S c f* $\sigma 0 \leq \Downarrow R$ (*FOREACHc S' c' f'* $\sigma 0'$)
⟨*proof*⟩

lemma *FOREACHc-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*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';\ c\ \sigma;\ c'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \leq \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACHc S c f* $\sigma 0 \leq \Downarrow R$ (*FOREACHc S' c' f'* $\sigma 0'$)
⟨*proof*⟩

lemma *FOREACHi-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 *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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$it \subseteq S;\ it' \subseteq S';\ \Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';$
$x' = \alpha\ x;\ it' = \alpha`it;\ ((it,\sigma),(it',\sigma')) \in R;$
$x \in it;\ x' \in it'$
$]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\ \sigma').\ ((it - \{x\},\sigma),(it' - \{x'\},\sigma')) \in R\})\ (f'\ x'\ \sigma')$
assumes *REF-R-DONE*: $\bigwedge \sigma\ \sigma'.\ [\![\ \Phi\ \{\}\ \sigma;\ \Phi'\ \{\}\ \sigma';\ ((\{\},\sigma),(\{\},\sigma')) \in R\ ]\!]$
$\Longrightarrow (\sigma,\sigma') \in R'$
shows *FOREACHi* $\Phi\ S\ f\ \sigma 0 \leq \Downarrow R'\ (FOREACHi\ \Phi'\ S'\ f'\ \sigma 0')$
⟨*proof*⟩

lemma *FOREACHi-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 *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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x' = \alpha\ x;\ x \in it;\ x' \in it';\ it' = \alpha`it;\ it \subseteq S;\ it' \subseteq S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ \Phi''\ it\ \sigma\ it'\ \sigma';$
$(\sigma,\sigma') \in R$
$]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\ \sigma').\ (\sigma,\ \sigma') \in R \wedge \Phi''\ (it - \{x\})\ \sigma\ (it' - \{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows *FOREACHi* $\Phi\ S\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHi\ \Phi'\ S'\ f'\ \sigma 0')$
⟨*proof*⟩

lemma *FOREACHi-refine-rcg*[*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 *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 $FOREACHi\ \Phi\ S\ f\ \sigma 0 \le \Downarrow R\ (FOREACHi\ \Phi'\ S'\ f'\ \sigma 0')$
⟨*proof*⟩

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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$it\subseteq S;\ it'\subseteq S';$
$x'=\alpha\ x;\ it'=\alpha`it;\ ((it,\sigma),(it',\sigma'))\in R;$
$x\in it;\ x'\in it'$
$]\!] \Longrightarrow f\ x\ \sigma$
$\le \Downarrow(\{(\sigma,\ \sigma').\ ((it-\{x\},\sigma),(it'-\{x'\},\sigma'))\in R\})\ (f'\ x'\ \sigma')$
assumes *REF-R-DONE*: $\bigwedge\sigma\ \sigma'.\ [\![\ ((\{\},\sigma),(\{\},\sigma'))\in R\ ]\!]$
$\Longrightarrow (\sigma,\sigma')\in R'$
shows $FOREACH\ S\ f\ \sigma 0 \le \Downarrow R'\ (FOREACH\ S'\ f'\ \sigma 0')$
⟨*proof*⟩

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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi''\ it\ \sigma\ it'\ \sigma';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma$
$\le \Downarrow(\{(\sigma,\ \sigma').\ (\sigma,\ \sigma')\in R \wedge \Phi''\ (it-\{x\})\ \sigma\ (it'-\{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows $FOREACH\ S\ f\ \sigma 0 \le \Downarrow R\ (FOREACH\ S'\ f'\ \sigma 0')$
⟨*proof*⟩

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 $FOREACH\ S\ f\ \sigma 0 \leq \Downarrow R\ (FOREACH\ S'\ f'\ \sigma 0')$
⟨*proof*⟩

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 $FOREACHc\ S\ c\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHci\ \Phi'\ S'\ c'\ f'\ \sigma 0')$
⟨*proof*⟩

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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi'\ it'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \leq \Downarrow R\ (f'\ x'\ \sigma')$
shows $FOREACH\ S\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHi\ \Phi'\ S'\ f'\ \sigma 0')$
⟨*proof*⟩

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 σ0*
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ [\![\ c\ \sigma;\ x{\in}it;\ it{\subseteq}S;\ I\ it\ \sigma;\ \forall\, y{\in}it - \{x\}.\ R\ x\ y;$
$\forall\, y{\in}S - it.\ R\ y\ x\ ]\!] \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it{-}\{x\}))$
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 σ0* $\leq$ *SPEC P*
⟨*proof*⟩

lemma *FOREACHci-rule'*[*refine-vcg*]:
assumes *FIN*: *finite S*
assumes *I0*: *I S σ0*
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ [\![\ x{\in}it;\ it{\subseteq}S;\ I\ it\ \sigma;\ c\ \sigma\ ]\!] \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it{-}\{x\}))$
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 σ0* $\leq$ *SPEC P*
⟨*proof*⟩

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*: $\bigwedge\sigma.\ [\![I\ \{\}\ \sigma;\ c\ \sigma]\!] \Longrightarrow P\ \sigma$
assumes *II2*: $\bigwedge it\ \sigma.\ [\![\ it{\subseteq}S;\ I\ it\ \sigma;\ \neg c\ \sigma\ ]\!] \Longrightarrow P\ \sigma$
shows *FOREACHc S c f σ0* $\leq$ *SPEC P*
⟨*proof*⟩

2.15.4 FOREACH with empty sets

lemma *FOREACHoci-emp* [*simp*] :
FOREACHoci R Φ {} c f σ = *do* {*ASSERT* (*Φ {} σ*); *RETURN σ*}
⟨*proof*⟩

lemma *FOREACHoi-emp* [*simp*] :
FOREACHoi R Φ {} f σ = *do* {*ASSERT* (*Φ {} σ*); *RETURN σ*}
⟨*proof*⟩

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

⟨*proof*⟩

lemma *FOREACHc-emp* [*simp*] :
FOREACHc {} *c f σ* = *RETURN σ*
⟨*proof*⟩

lemma *FOREACH-emp* [*simp*] :
FOREACH {} *f σ* = *RETURN σ*
⟨*proof*⟩

lemma *FOREACHi-emp* [*simp*] :
FOREACHi Φ {} *f σ* = *do* {*ASSERT* (Φ {} *σ*); *RETURN σ*}
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *FOREACHoci-mono*[*unfolded trimono-spec-defs*,*refine-mono*]:
trimono-spec (*R o R o R o R o M2 o R*) *FOREACHoci*
trimono-spec (*R o R o R o M2 o R*) *FOREACHoi*
trimono-spec (*R o R o R o M2 o R*) *FOREACHci*
trimono-spec (*R o R o M2 o R*) *FOREACHc*
trimono-spec (*R o R o M2 o R*) *FOREACHi*
trimono-spec (*R o M2 o R*) *FOREACH*
⟨*proof*⟩

end

2.15.6 Nres-Fold with Interruption (nfoldli)

A foreach-loop can be conveniently expressed as an operation that converts the set to a list, followed by folding over the list.

This representation is handy for automatic refinement, as the complex foreach-operation is expressed by two relatively simple operations.

We first define a fold-function in the nres-monad

partial-function (*nrec*) *nfoldli* **where**
nfoldli l c f s = (*case l of*
[] ⇒ *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*)
⟨*proof*⟩

lemma *param-nfoldli*[*param*]:
shows (*nfoldli*,*nfoldli*) ∈
⟨*Ra*⟩*list-rel* → (*Rb*→*Id*) → (*Ra*→*Rb*→⟨*Rb*⟩*nres-rel*) → *Rb* → ⟨*Rb*⟩*nres-rel*
⟨*proof*⟩

lemma *nfoldli-no-ctd*[*simp*]: ¬*ctd s* ⟹ *nfoldli l ctd f s* = *RETURN s*
⟨*proof*⟩

lemma *nfoldli-append*[*simp*]: *nfoldli* (*l1*@*l2*) *ctd f s* = *nfoldli l1 ctd f s* ⤜ *nfoldli l2 ctd f*
⟨*proof*⟩

lemma *nfoldli-map*: *nfoldli* (*map f l*) *ctd g s* = *nfoldli l ctd* (*g o f*) *s*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *nfoldli-transfer-dres*[*refine-transfer*]:
fixes $l :: 'a\ list$ **and** $c :: 'b \Rightarrow bool$
assumes *FR*: $\bigwedge x\ s.\ nres\text{-}of\ (f\ x\ s) \le f'\ x\ s$
shows *nres-of*
$(foldli\ l\ (case\text{-}dres\ False\ False\ c)\ (\lambda x\ s.\ s \gg\!= f\ x)\ (dRETURN\ s))$
$\le (nfoldli\ l\ c\ f'\ s)$
⟨*proof*⟩

lemma *nfoldli-mono*[*refine-mono*]:
$[\![\bigwedge x\ s.\ f\ x\ s \le f'\ x\ s]\!] \Longrightarrow nfoldli\ l\ c\ f\ \sigma \le nfoldli\ l\ c\ f'\ \sigma$
$[\![\bigwedge x\ s.\ flat\text{-}ge\ (f\ x\ s)\ (f'\ x\ s)]\!] \Longrightarrow flat\text{-}ge\ (nfoldli\ l\ c\ f\ \sigma)\ (nfoldli\ l\ c\ f'\ \sigma)$
⟨*proof*⟩

We relate our fold-function to the while-loop that we used in the original definition of the foreach-loop

lemma *nfoldli-while*: $nfoldli\ l\ c\ f\ \sigma$
$\le$
$(WHILE_T{}^I$
$(FOREACH\text{-}cond\ c)\ (FOREACH\text{-}body\ f)\ (l,\ \sigma) \gg\!=$
$(\lambda(\text{-},\ \sigma).\ RETURN\ \sigma))$
⟨*proof*⟩

lemma *while-nfoldli*:
do {
$(\text{-},\sigma) \leftarrow WHILE_T\ (FOREACH\text{-}cond\ c)\ (FOREACH\text{-}body\ f)\ (l,\sigma)$;
$RETURN\ \sigma$
} $\le nfoldli\ l\ c\ f\ \sigma$
⟨*proof*⟩

lemma *while-eq-nfoldli*: *do* {
$(\text{-},\sigma) \leftarrow WHILE_T\ (FOREACH\text{-}cond\ c)\ (FOREACH\text{-}body\ f)\ (l,\sigma)$;
$RETURN\ \sigma$
} $= nfoldli\ l\ c\ f\ \sigma$
⟨*proof*⟩

lemma *nfoldli-rule*:
assumes *I0*: $I\ []\ l0\ \sigma 0$
assumes *IS*: $\bigwedge x\ l1\ l2\ \sigma.\ [\![l0 = l1 @ x \# l2;\ I\ l1\ (x \# l2)\ \sigma;\ c\ \sigma]\!] \Longrightarrow f\ x\ \sigma \le SPEC\ (I\ (l1 @ [x])\ l2)$
assumes *FNC*: $\bigwedge l1\ l2\ \sigma.\ [\![l0 = l1 @ l2;\ I\ l1\ l2\ \sigma;\ \neg c\ \sigma]\!] \Longrightarrow P\ \sigma$
assumes *FC*: $\bigwedge \sigma.\ [\![I\ l0\ []\ \sigma;\ c\ \sigma]\!] \Longrightarrow P\ \sigma$
shows $nfoldli\ l0\ c\ f\ \sigma 0 \le SPEC\ P$
⟨*proof*⟩

lemma *nfoldli-leof-rule*:
assumes *I0*: $I\ []\ l0\ \sigma 0$
assumes *IS*: $\bigwedge x\ l1\ l2\ \sigma.\ [\![l0 = l1 @ x \# l2;\ I\ l1\ (x \# l2)\ \sigma;\ c\ \sigma]\!] \Longrightarrow f\ x\ \sigma \le_n SPEC\ (I\ (l1 @ [x])\ l2)$
assumes *FNC*: $\bigwedge l1\ l2\ \sigma.\ [\![l0 = l1 @ l2;\ I\ l1\ l2\ \sigma;\ \neg c\ \sigma]\!] \Longrightarrow P\ \sigma$

assumes *FC*: $\bigwedge\sigma.\ [\![\ I\ l0\ []\ \sigma;\ c\ \sigma\ ]\!] \Longrightarrow P\ \sigma$
shows *nfoldli l0 c f σ0* $\leq_n$ *SPEC P*
⟨*proof*⟩

lemma *nfoldli-refine*[*refine*]:
assumes $(li,\ l) \in \langle S\rangle$*list-rel*
and $(si,\ s) \in R$
and *CR*: $(ci,\ c) \in R \to$ *bool-rel*
and [*refine*]: $\bigwedge xi\ x\ si\ s.\ [\![\ (xi,x)\in S;\ (si,s)\in R;\ c\ s\ ]\!] \Longrightarrow fi\ xi\ si \leq \Downarrow R\ (f\ x\ s)$
shows *nfoldli li ci fi si* $\leq \Downarrow$ *R* (*nfoldli l c f s*)
⟨*proof*⟩

lemma *nfoldli-invar-refine*:
assumes $(li,l)\in\langle S\rangle$*list-rel*
assumes $(si,s)\in R$
assumes *I* [] *li si*
assumes *COND*: $\bigwedge l1i\ l2i\ l1\ l2\ si\ s.\ [\![$
$li=l1i@l2i;\ l=l1@l2;\ (l1i,l1)\in\langle S\rangle$*list-rel*; $(l2i,l2)\in\langle S\rangle$*list-rel*;
I l1i l2i si; $(si,s)\in R]\!] \Longrightarrow (ci\ si,\ c\ s)\in$*bool-rel*
assumes *INV*: $\bigwedge l1i\ xi\ l2i\ si.\ [\![li=l1i@xi\#l2i;\ I\ l1i\ (xi\#l2i)\ si]\!] \Longrightarrow fi\ xi\ si \leq_n$
SPEC (*I* (*l1i*@[*xi*]) *l2i*)
assumes *STEP*: $\bigwedge l1i\ xi\ l2i\ l1\ x\ l2\ si\ s.\ [\![$
$li=l1i@xi\#l2i;\ l=l1@x\#l2;\ (l1i,l1)\in\langle S\rangle$*list-rel*; $(xi,x)\in S$; $(l2i,l2)\in\langle S\rangle$*list-rel*;
I l1i (*xi*#*l2i*) *si*; $(si,s)\in R]\!] \Longrightarrow fi\ xi\ si \leq \Downarrow R\ (f\ x\ s)$
shows *nfoldli li ci fi si* $\leq \Downarrow R$ (*nfoldli l c f s*)
⟨*proof*⟩

lemma *foldli-mono-dres-aux1*:
fixes $\sigma :: 'a ::$ {*order-bot*, *order-top*}
assumes *COND*: $\bigwedge\sigma\ \sigma'.\ \sigma\leq\sigma' \Longrightarrow c\ \sigma \neq c\ \sigma' \Longrightarrow \sigma=bot \vee \sigma'=top$
assumes *STRICT*: $\bigwedge x.\ f\ x\ bot = bot \quad \bigwedge x.\ f'\ x\ top = top$
assumes *B*: $\sigma\leq\sigma'$
assumes *A*: $\bigwedge a\ x\ x'.\ x\leq x' \Longrightarrow f\ a\ x \leq f'\ a\ x'$
shows *foldli l c f σ* $\leq$ *foldli l c f' σ'*
⟨*proof*⟩

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*
⟨*proof*⟩

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*) (λ*x s*. *dbind s* (*f x*)) σ
≤ *foldli l* (*case-dres False False c*) (λ*x s*. *dbind s* (*f′ x*)) σ
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *dfoldli-mono*[*refine-mono*]:
⟦ ⋀*x s*. *f x s* ≤ *f′ x s* ⟧ ⟹ *dfoldli l c f* σ ≤ *dfoldli l c f′* σ
⟦ ⋀*x s*. *flat-ge* (*f x s*) (*f′ x s*) ⟧ ⟹ *flat-ge* (*dfoldli l c f* σ) (*dfoldli l c f′* σ)
⟨*proof*⟩

lemma *foldli-dres-pres-FAIL*[*simp*]:
foldli l (*case-dres False False c*) (λ*x s*. *dbind s* (*f x*)) *dFAIL* = *dFAIL*
⟨*proof*⟩

lemma *foldli-dres-pres-SUCCEED*[*simp*]:
foldli l (*case-dres False False c*) (λ*x s*. *dbind s* (*f x*)) *dSUCCEED* = *dSUCCEED*
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
}
⟨*proof*⟩

Patterns that convert FOREACH-constructs to *LIST-FOREACH*

context begin interpretation *autoref-syn* ⟨*proof*⟩

lemma *FOREACH-patterns*[*autoref-op-pat-def*]:
FOREACH^*I* *s f* ≡ *FOREACH*$_{OC}$^λ- -. *True*,*I* *s* (λ-. *True*) *f*
FOREACHci I s c f ≡ *FOREACHoci* (λ- -. *True*) *I s c f*
FOREACH$_{OC}$^*R*,Φ *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
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

context begin interpretation *trimono-spec* ⟨*proof*⟩

lemma *LIST-FOREACH′-mono*[*unfolded trimono-spec-defs*,*refine-mono*]:
trimono-spec (*R o R o M2 o R*) *LIST-FOREACH′*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

Simplification rules to summarize iterators

lemma [*refine-transfer-post-simp*]:
do {
xs ← *dRETURN tsl*;

foldli xs c f σ
} = *foldli tsl c f σ*
⟨*proof*⟩

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

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*⟩

lemma *LFOci-refine*:
assumes (*li*,*l*)∈⟨*A*⟩*list-set-rel*
assumes ⋀*s si*. (*si*,*s*)∈*R* ⟹ *ci si* ⟷ *c s*
assumes ⋀*x xi s si*. ⟦(*xi*,*x*)∈*A*; (*si*,*s*)∈*R*⟧ ⟹ *fi xi si* ≤ ⇓*R* (*f x s*)
assumes (*s0i*,*s0*)∈*R*
shows *nfoldli li ci fi s0i* ≤ ⇓*R* (*FOREACHci I l c f s0*)
⟨*proof*⟩

lemma *LFOc-refine*:
assumes (*li*,*l*)∈⟨*A*⟩*list-set-rel*
assumes ⋀*s si*. (*si*,*s*)∈*R* ⟹ *ci si* ⟷ *c s*
assumes ⋀*x xi s si*. ⟦(*xi*,*x*)∈*A*; (*si*,*s*)∈*R*⟧ ⟹ *fi xi si* ≤ ⇓*R* (*f x s*)
assumes (*s0i*,*s0*)∈*R*
shows *nfoldli li ci fi s0i* ≤ ⇓*R* (*FOREACHc l c f s0*)
⟨*proof*⟩

lemma *LFO-refine*:
assumes (*li*,*l*)∈⟨*A*⟩*list-set-rel*
assumes ⋀*x xi s si*. ⟦(*xi*,*x*)∈*A*; (*si*,*s*)∈*R*⟧ ⟹ *fi xi si* ≤ ⇓*R* (*f x s*)
assumes (*s0i*,*s0*)∈*R*
shows *nfoldli li* (λ-. *True*) *fi s0i* ≤ ⇓*R* (*FOREACH l f s0*)
⟨*proof*⟩

lemma *LFOi-refine*:
assumes (*li*,*l*)∈⟨*A*⟩*list-set-rel*
assumes ⋀*x xi s si*. ⟦(*xi*,*x*)∈*A*; (*si*,*s*)∈*R*⟧ ⟹ *fi xi si* ≤ ⇓*R* (*f x s*)
assumes (*s0i*,*s0*)∈*R*
shows *nfoldli li* (λ-. *True*) *fi s0i* ≤ ⇓*R* (*FOREACHi I l f s0*)
⟨*proof*⟩

lemma *LIST-FOREACH'-refine*: *LIST-FOREACH' tsl' c' f' σ' ≤ LIST-FOREACH Φ tsl' c' f' σ'*
⟨*proof*⟩

lemma *LIST-FOREACH'-eq*: *LIST-FOREACH* (λ- -. *True*) *tsl' c' f' σ' = (LIST-FOREACH' tsl' c' f' σ')*
⟨*proof*⟩

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
⟨*proof*⟩

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* 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 *FOREACHcdi I* S_0 *c f* σ_0 ≤ *SPEC* Φ
⟨*proof*⟩

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*⟧

$\Longrightarrow f'\ x'\ \sigma' \leq \Downarrow\ R\ (f\ x\ \sigma)$
shows *FOREACHcd* $s'\ c'\ f'\ \sigma' \leq \Downarrow R\ (FOREACHcdi\ I\ s\ c\ f\ \sigma)$
⟨*proof*⟩

lemma *FOREACHc-refines-FOREACHcd-aux*:
shows $FOREACHc\ s\ c\ f\ \sigma \leq FOREACHcd\ s\ c\ f\ \sigma$
⟨*proof*⟩

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 $FOREACH\ S\ (\lambda x\ \sigma.\ RETURN\ (insert\ (f\ x)\ \sigma))\ R0 \leq SPEC\ ((=)\ (R0 \cup f`S))$
⟨*proof*⟩

lemma *map-sigma-foreach*:
fixes $f :: 'a \times 'b \Rightarrow 'c$
assumes *finite A*
assumes $\bigwedge x.\ x{\in}A \Longrightarrow finite\ (B\ x)$
shows $FOREACH\ A\ (\lambda a\ \sigma.$
$FOREACH\ (B\ a)\ (\lambda b\ \sigma.\ RETURN\ (insert\ (f\ (a,b))\ \sigma))\ \sigma$
$)\ R0 \leq SPEC\ ((=)\ (R0 \cup f`Sigma\ A\ B))$
⟨*proof*⟩

lemma *map-sigma-sigma-foreach*:
fixes $f :: 'a \times ('b \times 'c) \Rightarrow 'd$
assumes *finite A*
assumes $\bigwedge a.\ a{\in}A \Longrightarrow finite\ (B\ a)$
assumes $\bigwedge a\ b.\ [\![a{\in}A;\ b{\in}B\ a]\!] \Longrightarrow finite\ (C\ a\ b)$
shows $FOREACH\ A\ (\lambda a\ \sigma.$
$FOREACH\ (B\ a)\ (\lambda b\ \sigma.$
$FOREACH\ (C\ a\ b)\ (\lambda c\ \sigma.$
$RETURN\ (insert\ (f\ (a,(b,c)))\ \sigma))\ \sigma)\ \sigma$
$)\ R0 \leq SPEC\ ((=)\ (R0 \cup f`Sigma\ A\ (\lambda a.\ Sigma\ (B\ a)\ (C\ a))))$
⟨*proof*⟩

lemma *bij-set-rel-for-inj*:
fixes *R*
defines $\alpha \equiv fun\text{-}of\text{-}rel\ R$
assumes *bijective R* $(s,s')\in\langle R\rangle set\text{-}rel$
shows *inj-on* $\alpha\ s$ $s' = \alpha`s$
— To be used when generating refinement conditions for foreach-loops

⟨*proof*⟩

lemma *nfoldli-by-idx-gen*:
shows *nfoldli* (*drop k l*) *c f s* = *nfoldli* [*k*..<*length l*] *c* (λ*i s*. *do* {
ASSERT (*i*<*length l*);
let x = *l*!*i*;
f x s
}) *s*
⟨*proof*⟩

lemma *nfoldli-by-idx*:
nfoldli l c f s = *nfoldli* [*0*..<*length l*] *c* (λ*i s*. *do* {
ASSERT (*i*<*length l*);
let x = *l*!*i*;
f x s
}) *s*
⟨*proof*⟩

lemma *nfoldli-map-inv*:
assumes *inj g*
shows *nfoldli l c f* = *nfoldli* (*map g l*) *c* (λ*x s*. *f* (*the-inv g x*) *s*)
⟨*proof*⟩

lemma *nfoldli-shift*:
fixes *ofs* :: *nat*
shows *nfoldli l c f* = *nfoldli* (*map* (λ*i*. *i*+*ofs*) *l*) *c* (λ*x s*. *do* {*ASSERT* (*x*≥*ofs*);
f (*x* − *ofs*) *s*})
⟨*proof*⟩

lemma *nfoldli-foreach-shift*:
shows *nfoldli* [*a*..<*b*] *c f* = *nfoldli* [*a*+*ofs*..<*b*+*ofs*] *c* (λ*x s*. *do*{*ASSERT*(*x*≥*ofs*);
f (*x* − *ofs*) *s*})
⟨*proof*⟩

lemma *member-by-nfoldli*: *nfoldli l* (λ*f*. ¬*f*) (λ*y* -. *RETURN* (*y*=*x*)) *False* ≤
SPEC (λ*r*. *r* ⟷ *x*∈*set l*)
⟨*proof*⟩

definition *sum-impl* :: (′*a* ⇒ ′*b*::*comm-monoid-add nres*) ⇒ ′*a set* ⇒ ′*b nres*
where
sum-impl g S ≡ *FOREACH S* (λ*x a*. *do* { *b* ← *g x*; *RETURN* (*a*+*b*)}) *0*

lemma *sum-impl-correct*:
assumes [*simp*]: *finite S*

assumes [*refine-vcg*]: $\bigwedge x.\ x{\in}S \Longrightarrow gi\ x \leq SPEC\ (\lambda r.\ r{=}g\ x)$
shows *sum-impl gi S* $\leq SPEC\ (\lambda r.\ r{=}sum\ g\ S)$
⟨*proof*⟩

end

2.16 More Automation

theory *Refine-Automation*
imports *Refine-Basic Refine-Transfer*
keywords *concrete-definition* :: *thy-decl*
and *prepare-code-thms* :: *thy-decl*
and *uses*
begin

This theory provides a tool for extracting definitions from terms, and for generating code equations for recursion combinators.

⟨*ML*⟩

Command: *concrete-definition name* [*attribs*] *for params uses thm is patterns* where *attribs*, *for*, and *is*-parts are optional.

Declares a new constant *name* by matching the theorem *thm* against a pattern.

If the *for* clause is given, it lists variables in the theorem, and thus determines the order of parameters of the defined constant. Otherwise, parameters will be in order of occurrence.

If the *is* clause is given, it lists patterns. The conclusion of the theorem will be matched against each of these patterns. For the first matching pattern, the constant will be declared to be the term that matches the first non-dummy variable of the pattern. If no *is*-clause is specified, the default patterns will be tried.

Attribute: *cd-patterns pats*. Declaration attribute. Declares default patterns for the *concrete-definition* command.

declare [[*cd-patterns* (*?f*,-)∈-]]
declare [[*cd-patterns RETURN ?f* ≤ - *nres-of ?f* ≤ -]]
declare [[*cd-patterns* (*RETURN ?f*,-)∈- (*nres-of ?f*,-)∈-]]
declare [[*cd-patterns* - = *?f* - == *?f*]]

⟨*ML*⟩

Command: *prepare-code-thms* (*modes*) *thm* where the (*mode*)-part is optional.

Set up code-equations for recursions in constant defined by *thm*. The optional *modes* is a comma-separated list of extraction modes.

lemma *gen-code-thm-RECT*:
fixes x
assumes *D*: $f \equiv RECT\ B$
assumes *M*: *trimono* B
shows $f\ x \equiv B\ f\ x$
⟨*proof*⟩

lemma *gen-code-thm-REC*:
fixes x
assumes *D*: $f \equiv REC\ B$
assumes *M*: *trimono* B
shows $f\ x \equiv B\ f\ x$
⟨*proof*⟩

⟨*ML*⟩

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.

⟨*ML*⟩

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 $(RETURN\ c,\ a)\in\langle R\rangle nres\text{-}rel \quad RETURN\ c \le \Downarrow R\ a$
⟨*proof*⟩

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

lemmas *autoref-monadicI = autoref-monadicI1 autoref-monadicI2*

⟨*ML*⟩

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*⟩

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″*
⟨*proof*⟩

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′*)
⟨*proof*⟩

lemma *min-distI-eq*:
⟦ *dist v d v′*; ⋀*d′*. *dist v d′ v′* ⟹ *d* ≤ *d′* ⟧ ⟹ *min-dist v v′* = *d*
⟨*proof*⟩

Two nodes are connected by a path of length *0*, iff they are equal.

lemma *dist-z-iff*[*simp*]: *dist v 0 v′* ⟷ *v′*=*v*
⟨*proof*⟩

The same holds for *min-dist*, i.e., the shortest path between two nodes has length *0*, iff these nodes are equal.

lemma *min-dist-z*[*simp*]: *min-dist v v* = *0*
⟨*proof*⟩

lemma *min-dist-z-iff*[*simp*]: *conn v v′* ⟹ *min-dist v v′* = *0* ⟷ *v′*=*v*
⟨*proof*⟩

lemma *min-dist-is-dist*: *conn v v′* ⟹ *dist v* (*min-dist v v′*) *v′*
⟨*proof*⟩
lemma *min-dist-minD*: *dist v d v′* ⟹ *min-dist v v′* ≤ *d*
⟨*proof*⟩

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

lemma *min-dist-succ*:

$[\![\mathit{conn}\ v\ v'; \ v'' \in \mathit{succ}\ v']\!] \Longrightarrow \mathit{min\text{-}dist}\ v\ v'' \leq \mathit{Suc}\ (\mathit{min\text{-}dist}\ v\ v')$
⟨*proof*⟩

lemma *min-dist-suc*:
assumes *c*: $\mathit{conn}\ v\ v' \quad \mathit{min\text{-}dist}\ v\ v' = \mathit{Suc}\ d$
shows $\exists v''.\ \mathit{conn}\ v\ v'' \wedge v' \in \mathit{succ}\ v'' \wedge \mathit{min\text{-}dist}\ v\ v'' = d$
⟨*proof*⟩

If there is a node with a shortest path of length d, then, for any $d'<d$, there is also a node with a shortest path of length d'.

lemma *min-dist-less*:
assumes $\mathit{conn}\ \mathit{src}\ v \quad \mathit{min\text{-}dist}\ \mathit{src}\ v = d$ **and** $d' < d$
shows $\exists v'.\ \mathit{conn}\ \mathit{src}\ v' \wedge \mathit{min\text{-}dist}\ \mathit{src}\ v' = d'$
⟨*proof*⟩

Lemma *min-dist-less* can be weakened to $d'\leq d$.

corollary *min-dist-le*:
assumes *c*: $\mathit{conn}\ \mathit{src}\ v$ **and** d': $d' \leq \mathit{min\text{-}dist}\ \mathit{src}\ v$
shows $\exists v'.\ \mathit{conn}\ \mathit{src}\ v' \wedge \mathit{min\text{-}dist}\ \mathit{src}\ v' = d'$
⟨*proof*⟩

3.1.2 Invariants

In our framework, it is convenient to annotate the invariants and auxiliary assertions into the program. Thus, we have to define the invariants first.

The invariant for the outer loop is split into two parts: The first part already holds before the *if* $C=\{\}$ check, the second part only holds again at the end of the loop body.

The first part of the invariant, *bfs-invar′*, intuitively states the following: If the loop is not *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* $\sigma \equiv$ *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-dist src* $v = d) \wedge$
$(\forall\, v{\in}N.$ *conn src* $v \wedge$ *min-dist src* $v =$ *Suc* $d) \wedge$
$(\forall\, v{\in}V.$ *conn src* $v \wedge$ *min-dist src* $v \leq$ *Suc* $d) \wedge$
$(\forall\, v.$ *conn src* $v \wedge$ *min-dist src* $v \leq d \longrightarrow v \in V) \wedge$
$(\forall\, v.$ *conn src* $v \wedge$ *min-dist src* $v =$ *Suc* $d \longrightarrow v \in N \cup ((\bigcup(succ`C)) - V))$
$)) \wedge ($
$f \longrightarrow$ *conn src dst* $\wedge$ *min-dist src dst* $= d$
$)$

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

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

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

definition *bfs-invar src dst* $\sigma \equiv$ *bfs-invar′ src dst* $\sigma \wedge$
empty-assm σ

The invariant of the inner foreach-loop states that the successors that have already been processed (*succ* v − *it*), have been added to V and have also been added to N' if they are not in V.

definition *FE-invar* V N v *it* $\sigma \equiv$ *let* $(V',N')=\sigma$ *in*
$V'=V \cup (succ\ v - it) \wedge$
$N'=N \cup ((succ\ v - it) - V)$

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*, $V \cup succ\ v$, $C - \{v\}$, $N \cup (succ\ v - V)$, d)
⟨*proof*⟩

The first part of the invariant is preserved if the *if* C={}-statement is executed. This is the verification condition for the loop-invariant. Note that preservation of the second part of the invariant is proven easily inside the main proof.

lemma *invar-empty-step*:
assumes *bfs-invar'* *src dst* (*False*, V, {}, N, d)
shows *bfs-invar'* *src dst* (*False*, V, N, {}, *Suc* d)
⟨*proof*⟩

The invariant holds initially.

lemma *invar-init*: *bfs-invar src dst* (*False*, {*src*}, {*src*}, {}, *0*)
⟨*proof*⟩

The invariant is preserved if we break the loop.

lemma *invar-break*:
assumes *bfs-invar src dst* (*False*, V, C, N, d)
assumes $dst \in C$
shows *bfs-invar src dst* (*True*, {}, {}, {}, d)
⟨*proof*⟩

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
⟨*proof*⟩

If the loop terminated normally, there is no path between *src* and *dst*.

The lemma is formulated as deriving a contradiction from the fact that there is a path and the loop terminated normally.

Note the proof language *Isar* that was used here. It allows one to write human-readable proofs in a theorem prover.

lemma *invar-final-fail*:
assumes C: *conn src dst* — There is a path between *src* and *dst*.
assumes *INV*: *bfs-invar'* *src dst* (*False*, V, {}, {}, d)
shows *False*
⟨*proof*⟩

Finally, we prove our algorithm correct: The following theorem solves both verification tasks.

Note that a proposition of the form $S \sqsubseteq SPEC\ \Phi$ 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* (
  λ*r*. *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*
  ⟨*proof*⟩

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 ⟨*proof*⟩

lemma [*simp*, *relator-props*]:
  single-valued word-nat-rel ⟨*proof*⟩

lemma [*simp*]: *single-valuedp* (λ*c a*. *a* = *unat c*)
  ⟨*proof*⟩

lemma [*simp*, *relator-props*]: *single-valued* (*converse word-nat-rel*)
  ⟨*proof*⟩

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 (λS. S={y0 .. y0 + x0 − 1})*
— Choosen pre-condition to get least trouble when proving
⟨*proof*⟩

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*⟩

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.