Refinement for Monadic Programs

Peter Lammich

March 12, 2013

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.

This submission comes with a collection of examples and a user-guide, illustrating the usage of the framework.

Contents

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.

Th 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.

1.2 Outline of this Submission

We briefly outline the most important files of this submission

./Generic This directory contains some concepts that can be instantiated with various monads.

- **./Generic/RefineG_Assert.thy** Generic Assertions.
- **./Generic/RefineG_Recursion.thy** Generic Recursion Combinators on arbitrary complete lattices.
- **./Generic/RefineG_While.thy** Generic While-Loops
- **./Generic/RefineG_Transfer.thy**

./Refine_Misc.thy Miscellaneous lemmas.

./Refine_Basic.thy Definition of nondeterminism monad and refinement.

./Refine_While.thy Setup of WHILE-loops.

./Refine_Foreach.thy Setup of foreach-loops.

./Refine_Det.thy Deterministic result monad.

./Refine_Transfer.thy Configuration of transfer to deterministic program.

./Refine_Heuristics.thy Heuristics to guess refinement relations.

./Refine_Autoref.thy Automatic data refinement (prototype).

./Iterator_Locale.thy Generic description of iterators.

./Collection_Bindings.thy Integration with Isabelle Collection Framework.

./Autoref_Collection_Bindings.thy ICF integration for automatic refinement.

./Refine_Pfun.thy Partial function package setup.

./Refine.thy Main theory of refinement framework. This one should be imported by users.

./Refine_Userguide.thy Userguide.

./ex Examples subdirectory.

- **./ex/Automatic_Refinement.thy** Automatic data refinement example.

./ex/Breadth_First_Search.thy Breadth first search (VSTTE 2012 competition, Task 5)

./ex/Bfs_Impl.thy Efficient implementation of breadth first search.

./ex/Recursion.thy Example for recursion combinators.

./ex/Refine_Fold.thy Example for partial function package use.

./ex/WordRefine.thy Example for refinement to machine words.

Chapter 2

Refinement Framework

2.1 Miscellanneous Lemmas and Tools

theory *Refine-Misc*
imports *Main ../Collections/common/Misc*
begin

2.1.1 ML-level stuff

⟨*ML*⟩

Basic configuration for monotonicity prover:

lemmas [*refine-mono*, *refine-mono-trigger*] = *monoI monotoneI*[*of op* ≤ *op* ≤]
lemmas [*refine-mono*] = *TrueI le-funI order-refl*

lemma *prod-case-mono*[*refine-mono*]:
  $[\![\bigwedge a\ b.\ p{=}(a,b) \Longrightarrow f\ a\ b \le f'\ a\ b]\!] \Longrightarrow \mathit{prod\text{-}case}\ f\ p \le \mathit{prod\text{-}case}\ f'\ p$
  ⟨*proof*⟩

lemma *if-mono*[*refine-mono*]:
  assumes $b \Longrightarrow m1 \le m1'$
  assumes $\neg b \Longrightarrow m2 \le m2'$
  shows $(\mathit{if}\ b\ \mathit{then}\ m1\ \mathit{else}\ m2) \le (\mathit{if}\ b\ \mathit{then}\ m1'\ \mathit{else}\ m2')$
  ⟨*proof*⟩

lemma *let-mono*[*refine-mono*]:
  $f\ x \le f'\ x' \Longrightarrow \mathit{Let}\ x\ f \le \mathit{Let}\ x'\ f'$ ⟨*proof*⟩

2.1.2 Uncategorized Lemmas

lemma *all-nat-split-at*: $\forall\, i{::}'a{::}\mathit{linorder}{<}k.\ P\ i \Longrightarrow P\ k \Longrightarrow \forall\, i{>}k.\ P\ i$
  $\Longrightarrow \forall\, i.\ P\ i$
  ⟨*proof*⟩

lemma *list-all2-induct*[*consumes 1*, *case-names Nil Cons*]:
  assumes *list-all2 P l l′*

assumes Q [] []
assumes $\bigwedge x\ x'\ ls\ ls'.\ [\![\ P\ x\ x';\ list\text{-}all2\ P\ ls\ ls';\ Q\ ls\ ls'\ ]\!]$
$\Longrightarrow Q\ (x\#ls)\ (x'\#ls')$
shows $Q\ l\ l'$
⟨*proof*⟩

lemma *pair-set-inverse*[*simp*]: $\{(a,b).\ P\ a\ b\}^{-1} = \{(b,a).\ P\ a\ b\}$
⟨*proof*⟩

lemma *comp-cong-right*: $x = y \Longrightarrow f\ o\ x = f\ o\ y$ ⟨*proof*⟩
lemma *comp-cong-left*: $x = y \Longrightarrow x\ o\ f = y\ o\ f$ ⟨*proof*⟩

lemma *nested-prod-case-simp*: $(\lambda(a,b)\ c.\ f\ a\ b\ c)\ x\ y =$
$(prod\text{-}case\ (\lambda a\ b.\ f\ a\ b\ y)\ x)$
⟨*proof*⟩

2.1.3 Relations

Transitive Closure

lemma *rtrancl-apply-insert*: $R^{*}``(insert\ x\ S) = insert\ x\ (R^{*}``(S \cup R``\{x\}))$
⟨*proof*⟩

lemma *rtrancl-last-visit-node*:
assumes $(s,s') \in R^{*}$
shows $s \neq sh \wedge (s,s') \in (R \cap (UNIV \times (-\{sh\})))^{*} \vee$
$(s,sh) \in R^{*} \wedge (sh,s') \in (R \cap (UNIV \times (-\{sh\})))^{*}$
⟨*proof*⟩

Well-Foundedness

lemma *wf-no-infinite-down-chainI*:
assumes $\bigwedge f.\ [\![\bigwedge i.\ (f\ (Suc\ i),\ f\ i) \in r]\!] \Longrightarrow False$
shows $wf\ r$
⟨*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*]: $finite\ S \Longrightarrow wf\ (finite\text{-}psupset\ S)$
⟨*proof*⟩

2.1.4 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 $mono\ f \Longrightarrow Sup\ (f`S) \le f\ (Sup\ S)$
⟨*proof*⟩

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

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

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

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

declare *SUP-insert*[*simp*]

lemma (**in** *semilattice-inf*) *le-infD1*:
$a \le inf\ b\ c \Longrightarrow a \le b$ ⟨*proof*⟩
lemma (**in** *semilattice-inf*) *le-infD2*:
$a \le inf\ b\ c \Longrightarrow a \le c$ ⟨*proof*⟩

lemma (**in** *semilattice-inf*) *inf-leI*:
$[\![\bigwedge x.\ [\![x{\le}a;\ x{\le}b]\!] \Longrightarrow x{\le}c]\!] \Longrightarrow inf\ a\ b \le c$
⟨*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*: $mono\ f \Longrightarrow x{\le}y \Longrightarrow f\ o\ x \le f\ o\ y$
⟨*proof*⟩

Galois Connections

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

lemma *α-mono*: $mono\ \alpha$
⟨*proof*⟩

lemma *γ-mono*: $mono\ \gamma$
⟨*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*: $f\ a \le a$
assumes *LEAST*: $\bigwedge x.\ f\ x = x \Longrightarrow a{\le}x$
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*: $mono\ f \Longrightarrow lfp\ f \leq gfp\ f$
⟨*proof*⟩

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

Connecting Complete Lattices and Chain-Complete Partial Orders

lemma (**in** *complete-lattice*) *is-ccpo*: $class.ccpo\ Sup\ (op\ \leq)\ (op\ <)$
⟨*proof*⟩

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

lemma *ccpo-mono-simp*: $monotone\ (op\ \leq)\ (op\ \leq)\ f \longleftrightarrow mono\ f$
⟨*proof*⟩
lemma *ccpo-monoI*: $mono\ f \Longrightarrow monotone\ (op\ \leq)\ (op\ \leq)\ f$
⟨*proof*⟩
lemma *ccpo-monoD*: $monotone\ (op\ \leq)\ (op\ \leq)\ f \Longrightarrow mono\ f$
⟨*proof*⟩

lemma *dual-ccpo-mono-simp*: $monotone\ (op\ \geq)\ (op\ \geq)\ f \longleftrightarrow mono\ f$
⟨*proof*⟩
lemma *dual-ccpo-monoI*: $mono\ f \Longrightarrow monotone\ (op\ \geq)\ (op\ \geq)\ f$
⟨*proof*⟩
lemma *dual-ccpo-monoD*: $monotone\ (op\ \geq)\ (op\ \geq)\ f \Longrightarrow mono\ f$
⟨*proof*⟩

lemma *ccpo-lfp-simp*: $\bigwedge f.\ mono\ f \Longrightarrow ccpo.fixp\ Sup\ op \leq f = lfp\ f$
⟨*proof*⟩

lemma *ccpo-gfp-simp*: $\bigwedge f.\ mono\ f \Longrightarrow ccpo.fixp\ Inf\ op \geq f = gfp\ f$
⟨*proof*⟩

abbreviation *chain-admissible* $P \equiv ccpo.admissible\ Sup\ op \leq P$
abbreviation *is-chain* $\equiv$ *Complete-Partial-Order.chain* $(op\ \leq)$
lemmas *chain-admissibleI*[*intro?*] = *ccpo.admissibleI*[*OF is-ccpo*]

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

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* $M \Longrightarrow$ *is-chain* $((\lambda f.\ f\ x)\text{`}M)$
⟨*proof*⟩

We transfer the admissible induction lemmas to complete lattices.

lemma *lfp-cadm-induct*:
⟦*chain-admissible* P; *mono* f; $\bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)$⟧ $\Longrightarrow P$ (*lfp* f)
⟨*proof*⟩

lemma *gfp-cadm-induct*:
⟦*dual-chain-admissible* P; *mono* f; $\bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)$⟧ $\Longrightarrow P$ (*gfp* f)
⟨*proof*⟩

Continuity and Kleene Fixed Point Theorem

definition *cont* $f \equiv \forall C.\ C \neq \{\} \longrightarrow f$ (*Sup* C) = *Sup* $(f\text{`}C)$
definition *strict* $f \equiv f$ *bot* = *bot*
definition *inf-distrib* $f \equiv$ *strict* f $\wedge$ *cont* f

lemma *contI* [*intro?*]: ⟦$\bigwedge C.\ C \neq \{\} \Longrightarrow f$ (*Sup* C) = *Sup* $(f\text{`}C)$⟧ $\Longrightarrow$ *cont* f
⟨*proof*⟩
lemma *contD*: *cont* $f \Longrightarrow C \neq \{\} \Longrightarrow f$ (*Sup* C) = *Sup* $(f\text{`}C)$
⟨*proof*⟩

lemma *strictD* [*dest*]: *strict* $f \Longrightarrow f$ *bot* = *bot*
⟨*proof*⟩
lemma *strictD-simp* [*simp*]: *strict* $f \Longrightarrow f$ (*bot*::′*a*::*bot*) = (*bot*::′*a*)
⟨*proof*⟩

lemma *strictI* [*intro?*]: f *bot* = *bot* $\Longrightarrow$ *strict* f
⟨*proof*⟩

lemma *inf-distribD* [*simp*]:
inf-distrib $f \Longrightarrow$ *strict* f
inf-distrib $f \Longrightarrow$ *cont* f
⟨*proof*⟩

lemma *inf-distribI* [*intro?*]: ⟦*strict* f; *cont* f⟧ $\Longrightarrow$ *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 *kleene-lfp*:
fixes *f*:: *'a::complete-lattice* ⇒ *'a*
assumes *CONT*: *cont f*
shows *lfp f* = (*SUP i*. (*f^^i*) *bot*)
⟨*proof*⟩

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

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

2.1.5 Maps

Key-Value Set

lemma *map-to-set-simps*[*simp*]:
map-to-set Map.empty = {}
map-to-set [*a*↦*b*] = {(*a*,*b*)}
map-to-set (*m*|'*K*) = *map-to-set m* ∩ *K*×*UNIV*
map-to-set (*m*(*x*:=*None*)) = *map-to-set m* − {*x*}×*UNIV*
map-to-set (*m*(*x*↦*v*)) = *map-to-set m* − {*x*}×*UNIV* ∪ {(*x*,*v*)}
map-to-set m ∩ *dom m*×*UNIV* = *map-to-set m*
m k = *Some v* ⟹ (*k*,*v*)∈*map-to-set m*
single-valued (*map-to-set m*)
⟨*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.

⟨*ML*⟩

locale *transfer* = **fixes** α :: $'c \Rightarrow 'a$::*complete-lattice*
begin

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

lemma *transfer-if*[*refine-transfer*]:
assumes $b \Longrightarrow \alpha\ s1 \le S1$
assumes $\neg b \Longrightarrow \alpha\ s2 \le S2$
shows α (*if b then s1 else s2*) $\le$ (*if b then S1 else S2*)
⟨*proof*⟩

lemma *transfer-prod*[*refine-transfer*]:
assumes $\bigwedge a\ b.\ \alpha\ (f\ a\ b) \le F\ a\ b$
shows α (*prod-case f x*) $\le$ (*prod-case F x*)
⟨*proof*⟩

lemma *transfer-Let*[*refine-transfer*]:
assumes $\bigwedge x.\ \alpha\ (f\ x) \le F\ x$
shows α (*Let x f*) $\le$ *Let x F*
⟨*proof*⟩

lemma *transfer-option*[*refine-transfer*]:
assumes $\alpha\ fa \le Fa$
assumes $\bigwedge x.\ \alpha\ (fb\ x) \le Fb\ x$
shows α (*option-case fa fb x*) $\le$ *option-case Fa Fb x*
⟨*proof*⟩

end

Transfer into complete lattice structure

locale *ordered-transfer* = *transfer* +
constrains α :: $'c$::*complete-lattice* $\Rightarrow$ $'a$::*complete-lattice*

Transfer into complete lattice structure with distributive transfer function.

locale *dist-transfer* = *ordered-transfer* +
constrains α :: $'c$::*complete-lattice* $\Rightarrow$ $'a$::*complete-lattice*
assumes α-*dist*: $\bigwedge A.$ *is-chain* $A \Longrightarrow \alpha$ (*Sup* A) = *Sup* (α‘A)
begin
lemma α-*mono*[*simp*, *intro*!]: *mono* α
⟨*proof*⟩

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

Transfer into ccpo

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

Transfer into ccpo with distributive transfer function.

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

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

lemma α-*strict*[*simp*]: α (*Sup* {}) = *bot*
⟨*proof*⟩
end

end

2.3 Generic Recursion Combinator for Complete Lattice Structured Domains

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

We define a recursion combinator that asserts monotonicity.

definition *REC* **where** *REC B x* ≡ *if* (*mono B*) *then* (*lfp B x*) *else top*
definition *RECT* (REC_T) **where** *RECT B x* ≡ *if* (*mono B*) *then* (*gfp B x*) *else top*

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

lemma *RECT-unfold*: $mono\ B \Longrightarrow RECT\ B\ x = B\ (RECT\ B)\ x$
⟨*proof*⟩

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

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

The following lemma shows that greatest and least fixed point are equal, if we can provide a variant.

lemma *RECT-eq-REC*:
assumes *WF*: *wf V*
assumes *I0*: *I x*
assumes *IS*: $\bigwedge f\ x.\ I\ x \Longrightarrow$
$body\ (\lambda x'.\ if\ (I\ x' \wedge (x',x){\in}V)\ then\ f\ x'\ else\ top)\ x \leq body\ f\ x$
shows $RECT\ body\ x = REC\ body\ x$
⟨*proof*⟩

The following lemma is a stronger version of *RECT-eq-REC*, which allows to keep track of a function specification, that can be used to argue about nested recursive calls.

lemma *RECT-eq-REC'*:
assumes *MONO*: *mono body*
assumes *WF*: *wf R*
assumes *I0*: *I x*
assumes *IS*: $\bigwedge f\ x.\ [\![I\ x;$
$\bigwedge x'.\ [\![\ I\ x';\ (x',x){\in}R\ ]\!] \Longrightarrow f\ x' \leq M\ x'$
$]\!] \Longrightarrow$
$body\ (\lambda x'.\ if\ (I\ x' \wedge (x',x){\in}R)\ then\ f\ x'\ else\ top)\ x \leq body\ f\ x\ \wedge$
$body\ f\ x \leq M\ x$
shows $RECT\ body\ x = REC\ body\ x \wedge RECT\ body\ x \leq M\ x$
⟨*proof*⟩

lemma *REC-rule*:

fixes x::$'x$
assumes M: *mono body*
assumes $I0$: $\Phi\ x$
assumes IS: $\bigwedge f\ x.\ [\![\ \bigwedge x.\ \Phi\ x \Longrightarrow f\ x \leq M\ x;\ \Phi\ x\ ]\!]$
$\Longrightarrow$ *body* $f\ x \leq M\ x$
shows REC *body* $x \leq M\ x$
⟨*proof*⟩

lemma *RECT-rule*:
assumes M: *mono body*
assumes WF: *wf* $(V$::$('x \times 'x)$ *set*$)$
assumes $I0$: $\Phi\ (x$::$'x)$
assumes IS: $\bigwedge f\ x.\ [\![\ \bigwedge x'.\ [\![\Phi\ x';\ (x',x) \in V]\!] \Longrightarrow f\ x' \leq M\ x';\ \Phi\ x\ ]\!]$
$\Longrightarrow$ *body* $f\ x \leq M\ x$
shows $RECT$ *body* $x \leq M\ x$
⟨*proof*⟩

2.3.1 Transfer

lemma (**in** *transfer*) *transfer-RECT'*[*refine-transfer*]:
assumes $REC\text{-}EQ$: $\bigwedge x.\ fr\ x = b\ fr\ x$
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 *mono 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 *mono b*
shows $\alpha\ (REC\ b\ x) \leq REC\ B\ x$
⟨*proof*⟩

end

2.4 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* ::
$('mu::complete\text{-}lattice) \Rightarrow (unit \Rightarrow ('ma::complete\text{-}lattice)) \Rightarrow 'ma$
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*: $[\![\Phi \Longrightarrow M \le M']\!] \Longrightarrow M \le bind\ (ASSERT\ \Phi)\ (\lambda\text{-}.\ M')$
⟨*proof*⟩

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

lemma *ASSERT-leI*: $[\![\Phi; \Phi \Longrightarrow M \le M']\!] \Longrightarrow bind\ (ASSERT\ \Phi)\ (\lambda\text{-}.\ M) \le$
M'
⟨*proof*⟩

lemma *ASSUME-leI*: $[\![\Phi \Longrightarrow M \le M']\!] \Longrightarrow bind\ (ASSUME\ \Phi)\ (\lambda\text{-}.\ M) \le M'$
⟨*proof*⟩
lemma *ASSUME-leI-pres*: $[\![\Phi \Longrightarrow bind\ (ASSUME\ \Phi)\ (\lambda\text{-}.\ M) \le M']\!]$
$\Longrightarrow bind\ (ASSUME\ \Phi)\ (\lambda\text{-}.\ M) \le M'$
⟨*proof*⟩
lemma *le-ASSUMEI*: $[\![\Phi; \Phi \Longrightarrow M \le M']\!] \Longrightarrow M \le bind\ (ASSUME\ \Phi)\ (\lambda\text{-}.$
$M')$
⟨*proof*⟩

The order of these declarations does matter!

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

lemma *ASSERT-le-iff*:

$bind\ (ASSERT\ \Phi)\ (\lambda\text{-}.\ S) \leq S' \longleftrightarrow (S' \neq top \longrightarrow \Phi) \wedge S \leq S'$
$\langle proof \rangle$

lemma *ASSUME-le-iff*:
$bind\ (ASSUME\ \Phi)\ (\lambda\text{-}.\ S) \leq S' \longleftrightarrow (\Phi \longrightarrow S \leq S')$
$\langle proof \rangle$

lemma *le-ASSERT-iff*:
$S \leq bind\ (ASSERT\ \Phi)\ (\lambda\text{-}.\ S') \longleftrightarrow (\Phi \longrightarrow S \leq S')$
$\langle proof \rangle$

lemma *le-ASSUME-iff*:
$S \leq bind\ (ASSUME\ \Phi)\ (\lambda\text{-}.\ S') \longleftrightarrow (S \neq bot \longrightarrow \Phi) \wedge S \leq S'$
$\langle proof \rangle$
end

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

locale *transfer-generic-Assert* =
c!: *generic-Assert cbind creturn cASSERT cASSUME* +
a!: *generic-Assert abind areturn aASSERT aASSUME* +
ordered-transfer α
for *cbind* :: $('muc::complete\text{-}lattice)$
$\Rightarrow (unit \Rightarrow 'mac) \Rightarrow ('mac::complete\text{-}lattice)$
and *creturn* :: $unit \Rightarrow 'muc$ **and** *cASSERT* **and** *cASSUME*
and *abind* :: $('mua::complete\text{-}lattice)$
$\Rightarrow (unit \Rightarrow 'maa) \Rightarrow ('maa::complete\text{-}lattice)$
and *areturn* :: $unit \Rightarrow 'mua$ **and** *aASSERT* **and** *aASSUME*
and $\alpha :: 'mac \Rightarrow 'maa$
begin
lemma *transfer-ASSERT*[*refine-transfer*]:
$[\![\Phi \Longrightarrow \alpha\ M \leq M']\!]$
$\Longrightarrow \alpha\ (cbind\ (cASSERT\ \Phi)\ (\lambda\text{-}.\ M)) \leq (abind\ (aASSERT\ \Phi)\ (\lambda\text{-}.\ M'))$
$\langle proof \rangle$

lemma *transfer-ASSUME*[*refine-transfer*]:
$[\![\Phi;\ \Phi \Longrightarrow \alpha\ M \leq M']\!]$
$\Longrightarrow \alpha\ (cbind\ (cASSUME\ \Phi)\ (\lambda\text{-}.\ M)) \leq (abind\ (aASSUME\ \Phi)\ (\lambda\text{-}.\ M'))$
$\langle proof \rangle$

end

locale *transfer-generic-Assert-remove* =
a!: *generic-Assert abind areturn aASSERT aASSUME* +
transfer α
for *abind* :: $('mua::complete\text{-}lattice)$
$\Rightarrow (unit \Rightarrow 'maa) \Rightarrow ('maa::complete\text{-}lattice)$
and *areturn* :: $unit \Rightarrow 'mua$ **and** *aASSERT* **and** *aASSUME*

and $\alpha :: {}'mac \Rightarrow {}'maa$
begin
lemma *transfer-ASSERT-remove*[*refine-transfer*]:
$[\![\ \Phi \Longrightarrow \alpha\ M \leq M'\ ]\!] \Longrightarrow \alpha\ M \leq abind\ (aASSERT\ \Phi)\ (\lambda\text{-}.\ M')$
⟨*proof*⟩

lemma *transfer-ASSUME-remove*[*refine-transfer*]:
$[\![\ \Phi;\ \Phi \Longrightarrow \alpha\ M \leq M'\ ]\!] \Longrightarrow \alpha\ M \leq abind\ (aASSUME\ \Phi)\ (\lambda\text{-}.\ M')$
⟨*proof*⟩
end

end

2.5 Basic Concepts

theory *Refine-Basic*
imports *Main*
~~/src/HOL/Library/Monad-Syntax
Generic/RefineG-Recursion
Generic/RefineG-Assert
begin

2.5.1 Setup

⟨*ML*⟩

2.5.2 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 $'a\ nres = FAILi \mid RES\ 'a\ set$

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 bottom element of the lattice.

instantiation *nres* :: (*type*) *complete-lattice*
begin
fun *less-eq-nres* **where**
  $- \leq FAILi \longleftrightarrow True$ |
  $(RES\ a) \leq (RES\ b) \longleftrightarrow a \subseteq b$ |
  $FAILi \leq (RES\ \text{-}) \longleftrightarrow False$

fun *less-nres* **where**
  $FAILi < - \longleftrightarrow False$ |
  $(RES\ \text{-}) < FAILi \longleftrightarrow True$ |
  $(RES\ a) < (RES\ b) \longleftrightarrow a \subset b$

fun *sup-nres* **where**
  $sup\ \text{-}\ FAILi = FAILi$ |
  $sup\ FAILi\ \text{-} = FAILi$ |
  $sup\ (RES\ a)\ (RES\ b) = RES\ (a \cup b)$

fun *inf-nres* **where**
  $inf\ x\ FAILi = x$ |
  $inf\ FAILi\ x = x$ |
  $inf\ (RES\ a)\ (RES\ b) = RES\ (a \cap b)$

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

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

instance
  ⟨*proof*⟩

end

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

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

lemma *nres-cases*[*case-names FAIL RES*, *cases type*]:
  obtains $M{=}FAIL$ | X **where** $M{=}RES\ X$
  ⟨*proof*⟩

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

lemma *nres-inequalities*[*simp*]:
$FAIL \neq RES\ X$
$RES\ X \neq FAIL$
$FAIL \neq SUCCEED$
$SUCCEED \neq FAIL$
⟨*proof*⟩

lemma *nres-more-simps*[*simp*]:
$RES\ X = SUCCEED \longleftrightarrow X=\{\}$
$SUCCEED = RES\ X \longleftrightarrow X=\{\}$
$RES\ X = RES\ Y \longleftrightarrow X=Y$
⟨*proof*⟩

lemma *nres-order-simps*[*simp*]:
$SUCCEED \leq M$
$M \leq SUCCEED \longleftrightarrow M=SUCCEED$
$M \leq FAIL$
$FAIL \leq M \longleftrightarrow M=FAIL$
$RES\ X \leq RES\ Y \longleftrightarrow X \leq Y$
$Sup\ X = FAIL \longleftrightarrow FAIL \in X$
$FAIL = Sup\ X \longleftrightarrow FAIL \in X$
$FAIL \in X \Longrightarrow Sup\ X = FAIL$
$Sup\ (RES`A) = RES\ (Sup\ A)$
$A \neq \{\} \Longrightarrow Inf\ (RES`A) = RES\ (Inf\ A)$
$Inf\ \{\} = FAIL$
$Inf\ UNIV = SUCCEED$
$Sup\ \{\} = SUCCEED$
$Sup\ UNIV = FAIL$
⟨*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

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

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

lemma *inres-simps*[*simp*, *refine-pw-simps*]:
$inres\ FAIL = (\lambda\text{-}.\ True)$
$inres\ (RES\ X) = (\lambda x.\ x{\in}X)$
$inres\ SUCCEED = (\lambda\text{-}.\ False)$
⟨*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*:
$S \le S' \longleftrightarrow (nofail\ S' \longrightarrow (nofail\ S \wedge (\forall x.\ inres\ S\ x \longrightarrow inres\ S'\ x)))$
⟨*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-leI*:
$(nofail\ S' \longrightarrow (nofail\ S \wedge (\forall x.\ inres\ S\ x \longrightarrow inres\ S'\ x))) \Longrightarrow S \le S'$
⟨*proof*⟩

lemma *pw-leI'*:
assumes $nofail\ S' \Longrightarrow nofail\ S$
assumes $\bigwedge x.\ [\![nofail\ S';\ inres\ S\ x]\!] \Longrightarrow inres\ S'\ x$
shows $S \le S'$
⟨*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 $S{\le}S'$ $nofail\ S'$
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*: ⟦ *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*]:
nofail (*sup a b*) $\longleftrightarrow$ *nofail a* $\wedge$ *nofail b*
⟨*proof*⟩

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

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

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

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

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

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

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

Monad Operators

definition *bind* **where** *bind M f* $\equiv$ *case M of*
FAILi $\Rightarrow$ *FAIL* |
RES X $\Rightarrow$ *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*⟩

⟨*ML*⟩

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

lemma *pw-bind-inres*[*refine-pw-simps*]:
$inres\ (bind\ M\ f) = (\lambda x.\ nofail\ M \longrightarrow (\exists y.\ (inres\ M\ y \wedge inres\ (f\ y)\ x)))$
⟨*proof*⟩

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

lemma *pw-bind-leI*: ⟦
$nofail\ S \Longrightarrow nofail\ M;\ \bigwedge x.\ [\![nofail\ M;\ inres\ M\ x]\!] \Longrightarrow f\ x \le S]\!]$
$\Longrightarrow bind\ M\ f \le S$
⟨*proof*⟩

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

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

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

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

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

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

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

2.5.3 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.

We define two lifting operations: $\Uparrow R$ yields a function from concrete to abstract values (abstraction function), and $\Downarrow R$ yields a function from abstract to concrete values (concretization function). The abstraction function fails if it cannot abstract its argument, i.e., if there is a value in the argument with no abstract counterpart. The concretization function, however, will just ignore abstract elements with no concrete counterpart. This matches the intuition that the concrete datastructure needs not to be able to represent any abstract value, while concrete values that have no corresponding abstract value make no sense. Also, the concretization relation will ignore abstract values that have a concretization whose abstractions are not all included in the result to be abstracted. Intuitively, this indicates an abstraction mismatch.

The concretization function results from the abstraction function by the natural postulate that concretization and abstraction function form a Galois connection, i.e., that abstraction and concretization can be exchanged:

$\Uparrow R\ M \leq M' \longleftrightarrow M \leq \Downarrow R\ M'$

Our approach is inspired by [16], that also uses the adjuncts of a Galois connection to express data refinement by program refinement.

definition *abs-fun* (⇑) **where**
  abs-fun R M ≡ *case M of*
    FAILi ⇒ *FAIL* |
    RES X ⇒ (
      if X⊆*Domain R then*
        RES (*R*“*X*)
      else
        FAIL
)

lemma
  abs-fun-FAIL[*simp*]: ⇑*R FAIL* = *FAIL* **and**

abs-fun-RES: $\Uparrow R$ (*RES X*) = (
  if $X \subseteq$*Domain R then*
    RES ($R``X$)
  else
    FAIL
)
⟨*proof*⟩

definition *conc-fun* ($\Downarrow$) **where**
  conc-fun R M $\equiv$ *case M of*
  FAILi $\Rightarrow$ *FAIL* |
  RES X $\Rightarrow$ *RES* $\{c.\ \exists\, a{\in}X.\ (c,a){\in}R \wedge (\forall\, a'.\ (c,a'){\in}R \longrightarrow a'{\in}X)\}$

lemma
  conc-fun-FAIL[*simp*]: $\Downarrow R$ *FAIL* = *FAIL* **and**
  conc-fun-RES: $\Downarrow R$ (*RES X*) = *RES* $\{c.\ \exists\, a{\in}X.\ (c,a){\in}R \wedge (\forall\, a'.\ (c,a'){\in}R \longrightarrow$
$a'{\in}X)\}$
⟨*proof*⟩

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

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

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*:
  inres ($\Downarrow R$ S') = ($\lambda s.$ *nofail* S'
  $\longrightarrow (\exists\, s'.\ (s,s'){\in}R \wedge$ *inres* $S'\ s' \wedge (\forall\, s''.\ (s,s''){\in}R \longrightarrow$ *inres* $S'\ s'')))$
⟨*proof*⟩

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

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

$\Uparrow R$ *SUCCEED* = *SUCCEED*
⟨*proof*⟩

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

lemmas [*simp*, *intro*!] = *ac*.α-*mono* *ac*.γ-*mono*

lemma *conc-fun-chain*: *single-valued* $R \Longrightarrow \Downarrow R\ (\Downarrow S\ M) = \Downarrow (R\ O\ S)\ M$
⟨*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: $C \le \Downarrow R\ B$ **and** B: $B \le \Downarrow R'\ A$
shows $C \le \Downarrow R\ (\Downarrow R'\ A)$
⟨*proof*⟩

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

Transitivity Reasoner Setup

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

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

$\bigwedge A\ B\ C.\ A \le \Downarrow Id\ B \Longrightarrow B \le \Downarrow Id\ C \Longrightarrow A \le \quad C$
$\bigwedge A\ B\ C.\ A \le \Downarrow Id\ B \Longrightarrow B \le \quad C \Longrightarrow A \le \quad C$
$\bigwedge A\ B\ C.\ A \le \quad B \Longrightarrow B \le \Downarrow Id\ C \Longrightarrow A \le \quad C$
⟨*proof*⟩

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

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

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

$\bigwedge A\ B\ C.\ [\![\Uparrow\ Id\ A \le B;\ \Uparrow\ Id\ B \le C]\!] \Longrightarrow A \le C$
$\bigwedge A\ B\ C.\ [\![\Uparrow\ Id\ A \le B;\ B \le C]\!] \Longrightarrow A \le C$
$\bigwedge A\ B\ C.\ [\![A \le B;\ \Uparrow\ Id\ B \le C]\!] \Longrightarrow A \le C$
⟨*proof*⟩

Invariant and Abstraction Functions

lemmas [*refine-post*] = *single-valued-Id*

Quite often, the abstraction relation can be described as combination of an abstraction function and an invariant, such that the invariant describes valid values on the concrete domain, and the abstraction function maps valid concrete values to its corresponding abstract value.

definition *build-rel* **where** [*simp*]: *build-rel* α *I* $\equiv$ $\{(c,a)\ .\ a{=}\alpha\ c \wedge I\ c\}$
abbreviation *br*$\equiv$*build-rel*
lemmas *br-def* = *build-rel-def*

lemma *br-id*[*simp*]: *br id* (λ-. *True*) = *Id*
⟨*proof*⟩

lemma *br-chain*:
(*build-rel* β *J*) *O* (*build-rel* α *I*) = *build-rel* ($\alpha \circ \beta$) ($\lambda s.\ J\ s \wedge I\ (\beta\ s)$)
⟨*proof*⟩

lemma *br-single-valued*[*simp*, *intro*!,*refine-post*]: *single-valued* (*build-rel* α *I*)
⟨*proof*⟩

lemma *converse-br-sv-iff*[*simp*]:
single-valued (*converse* (*br* α *I*)) $\longleftrightarrow$ *inj-on* α (*Collect I*)
⟨*proof*⟩

2.5.4 Derived Program Constructs

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

ASSUME and ASSERT

definition *ASSERT* **where** *ASSERT* $\equiv$ *iASSERT RETURN*
definition *ASSUME* **where** *ASSUME* $\equiv$ *iASSUME RETURN*
interpretation *assert?*: *generic-Assert bind RETURN ASSERT ASSUME*

⟨*proof*⟩

lemma *pw-ASSERT*[*refine-pw-simps*]:
nofail (*ASSERT* Φ) $\longleftrightarrow \Phi$
inres (*ASSERT* Φ) x
⟨*proof*⟩

lemma *pw-ASSUME*[*refine-pw-simps*]:
nofail (*ASSUME* Φ)
inres (*ASSUME* Φ) $x \longleftrightarrow \Phi$
⟨*proof*⟩

Assumptions and assertions are in particular useful with bindings, hence we show some handy rules here:

Recursion

lemma *pw-REC-nofail*: *nofail* (*REC B x*) $\longleftrightarrow$ *mono B* $\wedge$
($\exists F$. ($\forall x$.
nofail (*F x*) $\longrightarrow$ *nofail* (*B F x*)
$\wedge$ ($\forall x'$. *inres* (*B F x*) $x' \longrightarrow$ *inres* (*F x*) x')
) $\wedge$ *nofail* (*F x*))
⟨*proof*⟩

lemma *pw-REC-inres*: *inres* (*REC B x*) x' = (*mono B* $\longrightarrow$
($\forall F$. ($\forall x''$.
nofail (*F* x'') $\longrightarrow$ *nofail* (*B F* x'')
$\wedge$ ($\forall x$. *inres* (*B F* x'') $x \longrightarrow$ *inres* (*F* x'') x))
$\longrightarrow$ *inres* (*F x*) x'))
⟨*proof*⟩

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

2.5.5 Proof Rules

Proving Correctness

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

lemma *RETURN-rule*[*refine-vcg*]: $\Phi\ x \Longrightarrow$ *RETURN* $x \leq$ *SPEC* Φ
⟨*proof*⟩
lemma *RES-rule*[*refine-vcg*]: $[\![\bigwedge x.\ x{\in}S \Longrightarrow \Phi\ x]\!] \Longrightarrow$ *RES* $S \leq$ *SPEC* Φ
⟨*proof*⟩
lemma *SUCCEED-rule*[*refine-vcg*]: *SUCCEED* $\leq$ *SPEC* Φ ⟨*proof*⟩
lemma *FAIL-rule*: *False* $\Longrightarrow$ *FAIL* $\leq$ *SPEC* Φ ⟨*proof*⟩
lemma *SPEC-rule*[*refine-vcg*]: $[\![\bigwedge x.\ \Phi\ x \Longrightarrow \Phi'\ x]\!] \Longrightarrow$ *SPEC* $\Phi \leq$ *SPEC* Φ'
⟨*proof*⟩

lemma *Sup-img-rule-complete*:
$(\forall x.\ x{\in}S \longrightarrow f\,x \le SPEC\ \Phi) \longleftrightarrow Sup\ (f`S) \le SPEC\ \Phi$
⟨*proof*⟩
lemma *Sup-img-rule*[*refine-vcg*]:
$[\![\ \bigwedge x.\ x{\in}S \Longrightarrow f\,x \le SPEC\ \Phi\ ]\!] \Longrightarrow Sup(f`S) \le SPEC\ \Phi$
⟨*proof*⟩

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

lemma *bind-rule-complete*: $bind\ M\ f \le SPEC\ \Phi \longleftrightarrow M \le SPEC\ (\lambda x.\ f\ x \le SPEC\ \Phi)$
⟨*proof*⟩
lemma *bind-rule*[*refine-vcg*]:
$[\![\ M \le SPEC\ (\lambda x.\ f\,x \le SPEC\ \Phi)\ ]\!] \Longrightarrow bind\ M\,f \le SPEC\ \Phi$
⟨*proof*⟩

lemma *ASSUME-rule*[*refine-vcg*]: $[\![\Phi \Longrightarrow \Psi\ ()]\!] \Longrightarrow ASSUME\ \Phi \le SPEC\ \Psi$
⟨*proof*⟩
lemma *ASSERT-rule*[*refine-vcg*]: $[\![\Phi;\ \Phi \Longrightarrow \Psi\ ()]\!] \Longrightarrow ASSERT\ \Phi \le SPEC\ \Psi$
⟨*proof*⟩

lemma *prod-rule*[*refine-vcg*]:
$[\![\bigwedge a\ b.\ p{=}(a,b) \Longrightarrow S\ a\ b \le SPEC\ \Phi]\!] \Longrightarrow prod\text{-}case\ S\ p \le SPEC\ \Phi$
⟨*proof*⟩

lemma *prod2-rule*[*refine-vcg*]:
assumes $\bigwedge a\ b\ c\ d.\ [\![ab{=}(a,b);\ cd{=}(c,d)]\!] \Longrightarrow f\ a\ b\ c\ d \le SPEC\ \Phi$
shows $(\lambda(a,b)\ (c,d).\ f\ a\ b\ c\ d)\ ab\ cd \le SPEC\ \Phi$
⟨*proof*⟩

lemma *if-rule*[*refine-vcg*]:
$[\![\ b \Longrightarrow S1 \le SPEC\ \Phi;\ \neg b \Longrightarrow S2 \le SPEC\ \Phi]\!]$
$\Longrightarrow (if\ b\ then\ S1\ else\ S2) \le SPEC\ \Phi$
⟨*proof*⟩

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

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

The following lemma shows that greatest and least fixed point are equal, if we can provide a variant.

thm *RECT-eq-REC*
lemma *RECT-eq-REC-old*:
assumes *WF*: *wf V*

assumes *I0*: $I\ x$
assumes *IS*: $\bigwedge f\ x.\ I\ x \Longrightarrow$
$body\ (\lambda x'.\ do\ \{\ ASSERT\ (I\ x' \wedge (x',x) \in V);\ f\ x'\})\ x \leq body\ f\ x$
shows $REC_T\ body\ x = REC\ body\ x$
⟨*proof*⟩

lemma *REC-le-rule*:
assumes *M*: *mono 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*:
assumes $S \subseteq Domain\ R$
assumes $\bigwedge x\ x'.\ [\![x \in S;\ (x,x') \in R]\!] \Longrightarrow x' \in S'$
shows $RES\ S \leq \Downarrow R\ (RES\ S')$
⟨*proof*⟩

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

lemma *SPEC-refine*:
  assumes $S \leq SPEC\ (\lambda x.\ x{\in}Domain\ R \wedge (\forall (x,x'){\in}R.\ \Phi\ x'))$
  shows $S \leq \Downarrow R\ (SPEC\ \Phi)$
  $\langle proof \rangle$

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

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

lemma *RETURN-refine*:
  assumes $x{\in}Domain\ R$
  assumes $\bigwedge x''.\ (x,x''){\in}R \Longrightarrow x''{=}x'$
  shows $RETURN\ x \leq \Downarrow R\ (RETURN\ x')$
  $\langle proof \rangle$

lemma *RETURN-refine-sv*[*refine*]:
  assumes *single-valued R*
  assumes $(x,x'){\in}R$
  shows $RETURN\ x \leq \Downarrow R\ (RETURN\ x')$
  $\langle proof \rangle$

lemma *RETURN-SPEC-refine-sv*:
  assumes *SV*: *single-valued R*
  assumes $\exists\, x'.\ (x,x'){\in}R \wedge \Phi\ x'$
  shows $RETURN\ x \leq \Downarrow R\ (SPEC\ \Phi)$
  $\langle proof \rangle$

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

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

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

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 $bind\ M\ f \leq \Downarrow R\ (bind\ M'\ f')$
$\langle proof \rangle$

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

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

Assertions and assumptions are treated specially in bindings

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

lemma *ASSERT-refine-left*:
assumes Φ
assumes $\Phi \Longrightarrow S \leq \Downarrow R\ S'$
shows $do\{ASSERT\ \Phi;\ S\} \leq \Downarrow R\ S'$
$\langle proof \rangle$

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

lemma *ASSUME-refine-left*:
assumes $\Phi \Longrightarrow S \leq \Downarrow R\ S'$
shows $do\ \{ASSUME\ \Phi;\ S\} \leq \Downarrow R\ S'$
$\langle proof \rangle$

lemma *ASSUME-refine-left-pres*:
assumes $\Phi \Longrightarrow do\ \{ASSUME\ \Phi;\ S\} \leq \Downarrow R\ S'$
shows $do\ \{ASSUME\ \Phi;\ S\} \leq \Downarrow R\ S'$
$\langle proof \rangle$

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

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

lemma *REC-refine*[*refine*]:
  assumes M: *mono body*
  assumes $R0$: $(x,x')\in R$
  assumes RS: $\bigwedge f\, f'\, x\, x'.\ [\![\ \bigwedge x\, x'.\ (x,x')\in R \Longrightarrow f\, x \le\Downarrow S\ (f'\, x');\ (x,x')\in R\ ]\!]$
    $\Longrightarrow body\ f\ x \le\Downarrow S\ (body'\ f'\ x')$
  shows $REC\ body\ x \le\Downarrow S\ (REC\ body'\ x')$
  ⟨*proof*⟩

lemma *RECT-refine*[*refine*]:
  assumes M: *mono body*
  assumes $R0$: $(x,x')\in R$
  assumes RS: $\bigwedge f\, f'\, x\, x'.\ [\![\ \bigwedge x\, x'.\ (x,x')\in R \Longrightarrow f\, x \le\Downarrow S\ (f'\, x');\ (x,x')\in R\ ]\!]$
    $\Longrightarrow body\ f\ x \le\Downarrow S\ (body'\ f'\ x')$
  shows $RECT\ body\ x \le\Downarrow S\ (RECT\ body'\ x')$
  ⟨*proof*⟩

lemma *if-refine*[*refine*]:
  assumes $b \longleftrightarrow b'$
  assumes $[\![b;b']\!] \Longrightarrow S1 \le \Downarrow R\ S1'$
  assumes $[\![\neg b;\neg b']\!] \Longrightarrow S2 \le \Downarrow R\ S2'$
  shows $(if\ b\ then\ S1\ else\ S2) \le \Downarrow R\ (if\ b'\ then\ S1'\ else\ S2')$
  ⟨*proof*⟩

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

lemma *intro-Let-refine*[*refine2*]:
  assumes $f\ x \le \Downarrow R\ M'$
  shows $Let\ x\ f \le \Downarrow R\ M'$ ⟨*proof*⟩

lemma *bind2let-refine*[*refine2*]:
  assumes $RETURN\ x \le \Downarrow R'\ M'$
  assumes $\bigwedge x'.\ (x,x')\in R' \Longrightarrow f\ x \le \Downarrow R\ (f'\ x')$

shows $\mathit{Let}\ x\ f \le \Downarrow R\ (\mathit{bind}\ M'\ f')$
$\langle \mathit{proof} \rangle$

2.5.6 Convenience Rules

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

lemma *ref-two-step*: $A{\le}\Downarrow R\ \ B \Longrightarrow B{\le}\ \ \ \ C \Longrightarrow A{\le}\Downarrow R\ \ C$
$\langle \mathit{proof} \rangle$

lemma *build-rel-SPEC*:
$M \le \mathit{SPEC}\ (\ \lambda x.\ \Phi\ (\alpha\ x) \wedge I\ x) \Longrightarrow M \le \Downarrow(\mathit{build\text{-}rel}\ \alpha\ I)\ (\mathit{SPEC}\ \Phi)$
$\langle \mathit{proof} \rangle$

lemma *pw-ref-sv-iff*:
shows $\mathit{single\text{-}valued}\ R \Longrightarrow S \le \Downarrow R\ S'$
$\longleftrightarrow (\mathit{nofail}\ S'$
$\longrightarrow \mathit{nofail}\ S \wedge (\forall x.\ \mathit{inres}\ S\ x \longrightarrow (\exists s'.\ (x,\ s') \in R \wedge \mathit{inres}\ S'\ s')))$
$\langle \mathit{proof} \rangle$

lemma *pw-ref-svI*:
assumes $\mathit{single\text{-}valued}\ R$
assumes $\mathit{nofail}\ S'$
$\longrightarrow \mathit{nofail}\ S \wedge (\forall x.\ \mathit{inres}\ S\ x \longrightarrow (\exists s'.\ (x,\ s') \in R \wedge \mathit{inres}\ S'\ s'))$
shows $S \le \Downarrow R\ S'$
$\langle \mathit{proof} \rangle$

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

lemma *introR*: $(a,a'){\in}R \Longrightarrow (a,a'){\in}R$ $\langle \mathit{proof} \rangle$

lemma *le-ASSERTI-pres*:
assumes $\Phi \Longrightarrow S \le \mathit{do}\ \{\mathit{ASSERT}\ \Phi;\ S'\}$
shows $S \le \mathit{do}\ \{\mathit{ASSERT}\ \Phi;\ S'\}$
$\langle \mathit{proof} \rangle$

lemma *RETURN-ref-SPECD*:
assumes $\mathit{RETURN}\ c \le \Downarrow R\ (\mathit{SPEC}\ \Phi)$
obtains a **where** $(c,a){\in}R\ \ \ \ \Phi\ a$
$\langle \mathit{proof} \rangle$

lemma *RETURN-ref-RETURND*:
assumes $\mathit{RETURN}\ c \le \Downarrow R\ (\mathit{RETURN}\ a)$
shows $(c,a){\in}R$
$\langle \mathit{proof} \rangle$

end

2.6 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*⟩

2.6.1 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** $RELATES\ R \equiv True$

⟨*ML*⟩

2.6.2 Patterns

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

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

2.6.3 Refinement Relations

In this section, we define some general purpose refinement relations, e.g., for product types and sets.

lemma *Id-RELATES* [*refine-dref-RELATES*]: $RELATES\ Id$ ⟨*proof*⟩

Component-wise refinement for product types:

definition [*simp*]: $rprod\ R1\ R2 \equiv \{\ ((a,b),(a',b'))\ .\ (a,a')\in R1 \wedge (b,b')\in R2\ \}$

lemma *rprod-RELATES*[*refine-dref-RELATES*]:
  $RELATES\ Ra \Longrightarrow RELATES\ Rb \Longrightarrow RELATES\ (rprod\ Ra\ Rb)$
  ⟨*proof*⟩

lemma *rprod-sv*[*refine-hsimp*, *refine-post*]:
$\llbracket$*single-valued R1*; *single-valued R2*$\rrbracket \Longrightarrow$ *single-valued* (*rprod R1 R2*)
⟨*proof*⟩

Pointwise refinement for set types:

definition [*simp*]: *map-set-rel R* $\equiv$ *build-rel* (*op* `` *R*) (λ-. *True*)

lemma *map-set-rel-sv*[*refine-hsimp*, *refine-post*]:
single-valued (*map-set-rel R*)
⟨*proof*⟩

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

lemma *prod-set-eq-is-Id*[*refine-hsimp*]:
$\{(a,b).\ a{=}b\} = Id$
$\{(a,b).\ b{=}a\} = Id$
⟨*proof*⟩

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*⟩

Pointwise refinement for list types:

definition [*simp*]: *map-list-rel R* $\equiv \{(l,l').\ list\text{-}all2\ (\lambda x\ x'.\ (x,x'){\in}R)\ l\ l'\}$

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

lemma *map-list-rel-sv-iff-raw*:
single-valued (*map-list-rel R*) $\longleftrightarrow$ *single-valued R*
⟨*proof*⟩

lemma *map-list-rel-sv-iff*[*simp*, *refine-hsimp*]:
single-valuedP (*list-all2* ($\lambda x\ x'.\ (x,\ x') \in R$)) = *single-valued R*
⟨*proof*⟩

lemma *map-list-rel-sv*[*refine*, *refine-post*]:
single-valued R $\Longrightarrow$ *single-valued* (*map-list-rel R*)

⟨*proof*⟩

end

2.7 Generic While-Combinator

```
theory RefineG-While
imports
  RefineG-Recursion
  ~~/src/HOL/Library/While-Combinator
begin

definition
  WHILEI-body bind return I b f ≡
  (λW 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)


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 (λx. bind (f x) g)
  assumes ibind-mono1: mono bind
  assumes ibind-mono2: mono (bind M)
  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*

lemma *ibind-mono*: $m \le m' \Longrightarrow f \le f' \Longrightarrow bind\ m\ f \le bind\ m'\ f'$
⟨*proof*⟩

lemma *WHILEI-mono*: *mono* (*WHILEI-body bind return I b f*)
⟨*proof*⟩

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 *WHILEI-weaken*:
assumes *IW*: $\bigwedge x.\ I\ x \Longrightarrow I'\ x$
shows $WHILEI\ I'\ b\ f\ x \le WHILEI\ I\ b\ f\ x$
⟨*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 $WHILEIT\ I'\ b\ f\ x \le WHILEIT\ I\ b\ f\ x$
⟨*proof*⟩

lemma *WHILEI-le-WHILEIT*: $WHILEI\ I\ b\ f\ s \le WHILEIT\ I\ b\ f\ s$
⟨*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*: $\bigwedge x.\ return\ (f\ x) \le F\ x$
shows $return\ (while\ b\ f\ x) \le WHILEIT\ I\ b\ F\ x$
⟨*proof*⟩

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

lemma *WHILE-mono-prover-rule*[*refine-mono*]:
  notes [*refine-mono*] = *ibind-mono*
  assumes $\bigwedge x.\ f\ x \le f'\ x$
  shows $WHILE\ b\ f\ s0 \le WHILE\ b\ f'\ s0$
  ⟨*proof*⟩

lemma *WHILEI-mono-prover-rule*[*refine-mono*]:
  notes [*refine-mono*] = *ibind-mono*
  assumes $\bigwedge x.\ f\ x \le f'\ x$
  shows $WHILEI\ I\ b\ f\ s0 \le WHILEI\ I\ b\ f'\ s0$
  ⟨*proof*⟩

lemma *WHILET-mono-prover-rule*[*refine-mono*]:
  notes [*refine-mono*] = *ibind-mono*
  assumes $\bigwedge x.\ f\ x \le f'\ x$
  shows $WHILET\ b\ f\ s0 \le WHILET\ b\ f'\ s0$
  ⟨*proof*⟩

lemma *WHILEIT-mono-prover-rule*[*refine-mono*]:
  notes [*refine-mono*] = *ibind-mono*
  assumes $\bigwedge x.\ f\ x \le f'\ x$
  shows $WHILEIT\ I\ b\ f\ s0 \le WHILEIT\ I\ b\ f'\ s0$
  ⟨*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*: $[\![\alpha\ m \le M;\ \bigwedge x.\ \alpha\ (f\ x) \le F\ x\ ]\!]$
    $\Longrightarrow \alpha\ (cbind\ m\ f) \le abind\ M\ F$
  assumes *transfer-return*: $\alpha\ (creturn\ x) \le areturn\ x$
begin

lemma *transfer-WHILEIT*[*refine-transfer*]:
  assumes *REF*: $\bigwedge x.\ \alpha\ (f\ x) \le F\ x$
  shows $\alpha\ (cWHILEIT\ I\ b\ f\ x) \le aWHILEIT\ I\ b\ F\ x$

$\langle proof \rangle$

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

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

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

end

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

lemma *ireturn-eq*: $return\ x = SPEC\ (op = x)$
$\langle proof \rangle$

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

lemma *ireturn-rule*: $\Phi\ x \Longrightarrow return\ x \leq SPEC\ \Phi$
$\langle proof \rangle$

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 $WHILEI\ I\ b\ f\ s \leq SPEC\ \Phi$
$\langle proof \rangle$

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 $WHILEIT\ I\ b\ f\ s \leq SPEC\ \Phi$

⟨*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 $WHILE\ b\ f\ s \leq SPEC\ \Phi$
⟨*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 $WHILET\ b\ f\ s \leq SPEC\ \Phi$
⟨*proof*⟩

end

end

2.8 While-Loops

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

definition *WHILEIT* ($WHILE_T$⁻) **where**
$WHILEIT \equiv iWHILEIT\ bind\ RETURN$
definition *WHILEI* ($WHILE$⁻) **where** $WHILEI \equiv iWHILEI\ bind\ RETURN$
definition *WHILET* ($WHILE_T$) **where** $WHILET \equiv iWHILET\ bind\ RETURN$
definition *WHILE* **where** $WHILE \equiv iWHILE\ bind\ RETURN$

interpretation *while?*: *generic-WHILE-rules bind RETURN SPEC*
WHILEIT WHILEI WHILET WHILE
⟨*proof*⟩

lemmas [*refine-vcg*] = *WHILEI-rule*
lemmas [*refine-vcg*] = *WHILEIT-rule*

2.8.1 Data Refinement Rules

lemma *ref-WHILEI-invarI*:
assumes $I\ s \Longrightarrow M \leq \Downarrow R\ (WHILEI\ I\ b\ f\ s)$

shows $M \le \Downarrow R\ (WHILEI\ I\ b\ f\ s)$
$\langle proof \rangle$

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

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 $R0$: $(s,s')\in R$
assumes SV: $single\text{-}valued\ R$
assumes RI: $\bigwedge s\ s'.\ [\![\ (s,s')\in R;\ I'\ s'\ ]\!] \Longrightarrow I\ s$
assumes RB: $\bigwedge s\ s'.\ [\![\ (s,s')\in R;\ I'\ s';\ I\ s;\ b'\ s'\ ]\!] \Longrightarrow b\ s$
assumes RS: $\bigwedge s\ s'.\ [\![\ (s,s')\in R;\ I'\ s';\ I\ s;\ b\ s\ ]\!]$
$\Longrightarrow f\ s \le sup\ (\Downarrow R\ (do\ \{ASSUME\ (b'\ s');\ f'\ s'\}))\ (\Downarrow R\ (RETURN\ s'))$
shows $WHILEI\ I\ b\ f\ s \le \Downarrow R\ (WHILEI\ I'\ b'\ f'\ s')$
$\langle proof \rangle$

lemma *WHILEI-refine*[*refine*]:
assumes $R0$: $(x,x')\in R$
assumes SV: $single\text{-}valued\ R$
assumes $IREF$: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I'\ x'\ ]\!] \Longrightarrow I\ x$
assumes $COND\text{-}REF$: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes $STEP\text{-}REF$:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow f\ x \le \Downarrow R\ (f'\ x')$
shows $WHILEI\ I\ b\ f\ x \le \Downarrow R\ (WHILEI\ I'\ b'\ f'\ x')$
$\langle proof \rangle$

lemma *WHILE-invisible-refine*:
assumes $R0$: $(s,s')\in R$
assumes SV: $single\text{-}valued\ R$
assumes RB: $\bigwedge s\ s'.\ [\![\ (s,s')\in R;\ b'\ s'\ ]\!] \Longrightarrow b\ s$
assumes RS: $\bigwedge s\ s'.\ [\![\ (s,s')\in R;\ b\ s\ ]\!]$
$\Longrightarrow f\ s \le sup\ (\Downarrow R\ (do\ \{ASSUME\ (b'\ s');\ f'\ s'\}))\ (\Downarrow R\ (RETURN\ s'))$
shows $WHILE\ b\ f\ s \le \Downarrow R\ (WHILE\ b'\ f'\ s')$
$\langle proof \rangle$

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

lemma *WHILE-refine*[*refine*]:
assumes *R0*: $(x,x')\in R$
assumes *SV*: *single-valued* 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 $\mathit{WHILE}\ b\ f\ x \leq\Downarrow R\ (\mathit{WHILE}\ b'\ f'\ x')$
⟨*proof*⟩

lemma *WHILE-refine'*[*refine*]:
assumes *R0*: $(x,x')\in R$
assumes *SV*: *single-valued* 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 $\mathit{WHILE}\ b\ f\ x \leq\Downarrow R\ (\mathit{WHILEI}\ I'\ b'\ f'\ x')$
⟨*proof*⟩

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

lemma *WHILEIT-refine*[*refine*]:
assumes *R0*: $(x,x')\in R$
assumes *SV*: *single-valued* 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 $\mathit{WHILEIT}\ I\ b\ f\ x \leq\Downarrow R\ (\mathit{WHILEIT}\ I'\ b'\ f'\ x')$
⟨*proof*⟩

lemma *WHILET-refine*[*refine*]:
assumes *R0*: $(x,x')\in R$
assumes *SV*: *single-valued* 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 $\mathit{WHILET}\ b\ f\ x \leq\Downarrow R\ (\mathit{WHILET}\ b'\ f'\ x')$
⟨*proof*⟩

lemma *WHILET-refine'*[*refine*]:
assumes *R0*: $(x,x')\in R$
assumes *SV*: *single-valued* 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 $\mathit{WHILET}\ b\ f\ x \leq\Downarrow R\ (\mathit{WHILEIT}\ I'\ b'\ f'\ x')$

⟨*proof*⟩

end

2.9 Foreach Loops

theory *Refine-Foreach*
imports *Refine-While ../Collections/iterator/SetIterator*
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.

2.9.1 Auxilliary Lemmas

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

lemma *it-step-insert-iff*:
$it \subseteq S \Longrightarrow x{\in}it \Longrightarrow S-(it-\{x\}) = insert\ x\ (S-it)$ ⟨*proof*⟩

2.9.2 Definition

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

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

definition *FOREACH-body f* $\equiv \lambda(xs,\ \sigma).\ do\ \{$
$let\ x = hd\ xs;\ \sigma'{\leftarrow}f\ x\ \sigma;\ RETURN\ (tl\ xs,\sigma')$
$\}$

definition *FOREACH-cond* **where** *FOREACH-cond c* $\equiv (\lambda(xs,\sigma).\ xs{\neq}[] \wedge c\ \sigma)$

Foreach with continuation condition, order and annotated invariant:

definition *FOREACHoci* ($FOREACH_{OC}$⁻) **where** *FOREACHoci* $\Phi\ S\ R\ c\ f\ \sigma 0$
$\equiv do\ \{$
$ASSERT\ (finite\ S);$
$xs \leftarrow SPEC\ (\lambda xs.\ distinct\ xs \wedge S = set\ xs \wedge sorted\text{-}by\text{-}rel\ R\ xs);$
$(\text{-},\sigma) \leftarrow WHILEIT$
$(\lambda(it,\sigma).\ \exists xs'.\ xs = xs'\ @\ it \wedge \Phi\ (set\ it)\ \sigma)\ (FOREACH\text{-}cond\ c)\ (FOREACH\text{-}body\ f)\ (xs,\sigma 0);$

RETURN σ }

Foreach with continuation condition and annotated invariant:

definition *FOREACHci* ($FOREACH_C$ˆ) **where** *FOREACHci* Φ *S* $\equiv$ *FOREACHoci* Φ *S* (λ- -. *True*)

Foreach with continuation condition:

definition *FOREACHc* ($FOREACH_C$) **where** *FOREACHc* $\equiv$ *FOREACHci* (λ- -. *True*)

Foreach with annotated invariant:

definition *FOREACHi* (*FOREACH*ˆ) **where**
FOREACHi Φ *S f* $\sigma 0$ $\equiv$ *FOREACHci* Φ *S* (λ-. *True*) *f* $\sigma 0$

Foreach with annotated invariant and order:

definition *FOREACHoi* ($FOREACH_O$ˆ) **where**
FOREACHoi Φ *S R f* $\sigma 0$ $\equiv$ *FOREACHoci* Φ *S R* (λ-. *True*) *f* $\sigma 0$

Basic foreach

definition *FOREACH S f* $\sigma 0$ $\equiv$ *FOREACHc S* (λ-. *True*) *f* $\sigma 0$

2.9.3 Proof Rules

lemma *FOREACHoci-rule*[*refine-vcg*]:
assumes *FIN*: *finite S*
assumes *I0*: *I S* $\sigma 0$
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ [\![\ c\ \sigma;\ x \in it;\ it \subseteq S;\ I\ it\ \sigma;\ \forall y \in it - \{x\}.\ R\ x\ y;$
$\forall y \in S - it.\ R\ y\ x\ ]\!] \Longrightarrow f\ x\ \sigma \le SPEC\ (I\ (it - \{x\}))$
assumes *II1*: $\bigwedge \sigma.\ [\![I\ \{\}\ \sigma]\!] \Longrightarrow P\ \sigma$
assumes *II2*: $\bigwedge it\ \sigma.\ [\![\ it \neq \{\};\ it \subseteq S;\ I\ it\ \sigma;\ \neg c\ \sigma;$
$\forall x \in it.\ \forall y \in S - it.\ R\ y\ x\ ]\!] \Longrightarrow P\ \sigma$
shows *FOREACHoci I S R c f* $\sigma 0 \le$ *SPEC P*
⟨*proof*⟩

lemma *FOREACHoi-rule*[*refine-vcg*]:
assumes *FIN*: *finite S*
assumes *I0*: *I S* $\sigma 0$
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ [\![\ x \in it;\ it \subseteq S;\ I\ it\ \sigma;\ \forall y \in it - \{x\}.\ R\ x\ y;$
$\forall y \in S - it.\ R\ y\ x\ ]\!] \Longrightarrow f\ x\ \sigma \le SPEC\ (I\ (it - \{x\}))$
assumes *II1*: $\bigwedge \sigma.\ [\![I\ \{\}\ \sigma]\!] \Longrightarrow P\ \sigma$
shows *FOREACHoi I S R f* $\sigma 0 \le$ *SPEC P*
⟨*proof*⟩

lemma *FOREACHci-rule*[*refine-vcg*]:
assumes *FIN*: *finite S*
assumes *I0*: *I S* $\sigma 0$

assumes *IP*:
$\bigwedge x\ it\ \sigma.\ [\![\ x{\in}it;\ it{\subseteq}S;\ I\ it\ \sigma;\ c\ \sigma\ ]\!] \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it{-}\{x\}))$
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 $FOREACHci\ I\ S\ c\ f\ \sigma 0 \leq SPEC\ P$
⟨*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*: $S' = \alpha`S$
assumes *REF0*: $(\sigma 0,\sigma 0')\in R$
assumes *SV*: *single-valued* 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 *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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![\forall y{\in}it{-}\{x\}.\ RR\ x\ y;$
$x'{=}\alpha\ x;\ x{\in}it;\ x'{\in}it';\ it'{=}\alpha`it;\ it{\subseteq}S;\ it'{\subseteq}S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ \ \Phi''\ it\ \sigma\ it'\ \sigma';\ c\ \sigma;\ c'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\ \sigma').\ (\sigma,\ \sigma') \in R \wedge \Phi''\ (it\ -\ \{x\})\ \sigma\ (it'\ -\ \{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows $FOREACHoci\ \Phi\ S\ RR\ c\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHoci\ \Phi'\ S'\ RR'\ c'\ f'\ \sigma 0')$
⟨*proof*⟩

lemma *FOREACHoci-refine-rcg*[*refine*]:
fixes $\alpha :: {'S} \Rightarrow {'Sa}$
fixes $S :: {'S}\ set$
fixes $S' :: {'Sa}\ set$
assumes *INJ*: *inj-on* $\alpha\ S$
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 *SV*: *single-valued* 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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![\ \forall y{\in}it{-}\{x\}.\ RR\ x\ y;$
$x'{=}\alpha\ x;\ x{\in}it;\ x'{\in}it';\ it'{=}\alpha`it;\ it{\subseteq}S;\ it'{\subseteq}S';$

$\Phi\ it\ \sigma$; $\Phi'\ it'\ \sigma'$; $c\ \sigma$; $c'\ \sigma'$;
$(\sigma,\sigma')\in R$
$\rrbracket \Longrightarrow f\ x\ \sigma \leq \Downarrow R\ (f'\ x'\ \sigma')$
shows $FOREACHoci\ \Phi\ S\ RR\ c\ f\ \sigma0 \leq \Downarrow R\ (FOREACHoci\ \Phi'\ S'\ RR'\ c'\ f'\ \sigma0')$
$\langle proof\rangle$

lemma *FOREACHoci-weaken*:
assumes $IREF$: $\bigwedge it\ \sigma.\ it\subseteq S \Longrightarrow I\ it\ \sigma \Longrightarrow I'\ it\ \sigma$
shows $FOREACHoci\ I'\ S\ RR\ c\ f\ \sigma0 \leq FOREACHoci\ I\ S\ RR\ c\ f\ \sigma0$
$\langle proof\rangle$

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 $FOREACHoci\ I\ S\ RR\ c\ f\ \sigma0 \leq FOREACHoci\ I\ S\ RR'\ c\ f\ \sigma0$
$\langle proof\rangle$

Rules for Derived Constructs

lemma *FOREACHoi-refine*:
fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S\ set$
fixes $S' :: 'Sa\ set$
assumes INJ: $inj\text{-}on\ \alpha\ S$
assumes $REFS$: $S' = \alpha`S$
assumes $REF0$: $(\sigma0,\sigma0')\in R$
assumes SV: $single\text{-}valued\ R$
assumes $RR\text{-}OK$: $\bigwedge x\ y.\ \llbracket x \in S;\ y \in S;\ RR\ x\ y\rrbracket \Longrightarrow RR'\ (\alpha\ x)\ (\alpha\ y)$
assumes $REFPHI0$: $\Phi''\ S\ \sigma0\ (\alpha`S)\ \sigma0'$
assumes $REFPHI$: $\bigwedge it\ \sigma\ it'\ \sigma'.\ \llbracket$
$it'=\alpha`it$; $it\subseteq S$; $it'\subseteq S'$; $\Phi'\ it'\ \sigma'$; $\Phi''\ it\ \sigma\ it'\ \sigma'$; $(\sigma,\sigma')\in R$
$\rrbracket \Longrightarrow \Phi\ it\ \sigma$
assumes $REFSTEP$: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ \llbracket\forall y\in it-\{x\}.\ RR\ x\ y;$
$x'=\alpha\ x$; $x\in it$; $x'\in it'$; $it'=\alpha`it$; $it\subseteq S$; $it'\subseteq S'$;
$\Phi\ it\ \sigma$; $\Phi'\ it'\ \sigma'$; $\Phi''\ it\ \sigma\ it'\ \sigma'$; $(\sigma,\sigma')\in R$
$\rrbracket \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma, \sigma').\ (\sigma, \sigma') \in R \wedge \Phi''\ (it - \{x\})\ \sigma\ (it' - \{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows $FOREACHoi\ \Phi\ S\ RR\ f\ \sigma0 \leq \Downarrow R\ (FOREACHoi\ \Phi'\ S'\ RR'\ f'\ \sigma0')$
$\langle proof\rangle$

lemma *FOREACHoi-refine-rcg*[*refine*]:
fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S\ set$
fixes $S' :: 'Sa\ set$
assumes INJ: $inj\text{-}on\ \alpha\ S$
assumes $REFS$: $S' = \alpha`S$
assumes $REF0$: $(\sigma0,\sigma0')\in R$
assumes $RR\text{-}OK$: $\bigwedge x\ y.\ \llbracket x \in S;\ y \in S;\ RR\ x\ y\rrbracket \Longrightarrow RR'\ (\alpha\ x)\ (\alpha\ y)$
assumes SV: $single\text{-}valued\ R$
assumes $REFPHI$: $\bigwedge it\ \sigma\ it'\ \sigma'.\ \llbracket$

$it'{=}\alpha`it$; $it{\subseteq}S$; $it'{\subseteq}S'$; $\Phi'\ it'\ \sigma'$; $(\sigma,\sigma'){\in}R$
$\rrbracket \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ \llbracket\ \forall y{\in}it{-}\{x\}.\ RR\ x\ y$;
$x'{=}\alpha\ x$; $x{\in}it$; $x'{\in}it'$; $it'{=}\alpha`it$; $it{\subseteq}S$; $it'{\subseteq}S'$;
$\Phi\ it\ \sigma$; $\Phi'\ it'\ \sigma'$; $(\sigma,\sigma'){\in}R$
$\rrbracket \Longrightarrow f\ x\ \sigma \leq \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACHoi* $\Phi\ S\ RR\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHoi\ \Phi'\ S'\ RR'\ f'\ \sigma 0')$
$\langle proof \rangle$

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 *SV*: *single-valued* R
assumes *REFPHI0*: $\Phi''\ S\ \sigma 0\ (\alpha`S)\ \sigma 0'$
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ \llbracket$
$it'{=}\alpha`it$; $it{\subseteq}S$; $it'{\subseteq}S'$; $\Phi'\ it'\ \sigma'$; $\Phi''\ it\ \sigma\ it'\ \sigma'$; $\Phi\ it\ \sigma$; $(\sigma,\sigma'){\in}R$
$\rrbracket \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ \llbracket$
$it'{=}\alpha`it$; $it{\subseteq}S$; $it'{\subseteq}S'$; $\Phi'\ it'\ \sigma'$; $\Phi''\ it\ \sigma\ it'\ \sigma'$; $(\sigma,\sigma'){\in}R$
$\rrbracket \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ \llbracket$
$x'{=}\alpha\ x$; $x{\in}it$; $x'{\in}it'$; $it'{=}\alpha`it$; $it{\subseteq}S$; $it'{\subseteq}S'$;
$\Phi\ it\ \sigma$; $\Phi'\ it'\ \sigma'$; $\Phi''\ it\ \sigma\ it'\ \sigma'$; $c\ \sigma$; $c'\ \sigma'$;
$(\sigma,\sigma'){\in}R$
$\rrbracket \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\ \sigma').\ (\sigma,\ \sigma') \in R \wedge \Phi''\ (it - \{x\})\ \sigma\ (it' - \{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows *FOREACHci* $\Phi\ S\ c\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHci\ \Phi'\ S'\ c'\ f'\ \sigma 0')$
$\langle proof \rangle$

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 *SV*: *single-valued* R
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ \llbracket$
$it'{=}\alpha`it$; $it{\subseteq}S$; $it'{\subseteq}S'$; $\Phi'\ it'\ \sigma'$; $\Phi\ it\ \sigma$; $(\sigma,\sigma'){\in}R$
$\rrbracket \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes *REFPHI*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ \llbracket$
$it'{=}\alpha`it$; $it{\subseteq}S$; $it'{\subseteq}S'$; $\Phi'\ it'\ \sigma'$; $(\sigma,\sigma'){\in}R$
$\rrbracket \Longrightarrow \Phi\ it\ \sigma$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ \llbracket$
$x'{=}\alpha\ x$; $x{\in}it$; $x'{\in}it'$; $it'{=}\alpha`it$; $it{\subseteq}S$; $it'{\subseteq}S'$;

Φ *it* σ; Φ' *it'* σ'; *c* σ; *c'* σ';
$(\sigma,\sigma')\in R$
$\rrbracket \Longrightarrow f\ x\ \sigma \le \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACHci* Φ *S c f* $\sigma 0 \le \Downarrow R$ (*FOREACHci* Φ' *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* $\sigma 0 \le$ *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*:
$\bigwedge x\ it\ \sigma.\ \llbracket\ x\in it;\ it\subseteq S;\ I\ it\ \sigma\ \rrbracket \Longrightarrow f\ x\ \sigma \le SPEC\ (I\ (it-\{x\}))$
assumes *II*: $\bigwedge\sigma.\ \llbracket I\ \{\}\ \sigma\rrbracket \Longrightarrow P\ \sigma$
shows *FOREACHi I S f* $\sigma 0 \le$ *SPEC P*
⟨*proof*⟩

lemma *FOREACHc-rule*:
assumes *FIN*: *finite S*
assumes *I0*: *I S* $\sigma 0$
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ \llbracket\ x\in it;\ it\subseteq S;\ I\ it\ \sigma\ \rrbracket \Longrightarrow f\ x\ \sigma \le SPEC\ (I\ (it-\{x\}))$
assumes *II1*: $\bigwedge\sigma.\ \llbracket I\ \{\}\ \sigma\rrbracket \Longrightarrow P\ \sigma$
assumes *II2*: $\bigwedge it\ \sigma.\ \llbracket\ it\neq\{\};\ it\subseteq S;\ I\ it\ \sigma;\ \neg c\ \sigma\ \rrbracket \Longrightarrow P\ \sigma$
shows *FOREACHc S c f* $\sigma 0 \le$ *SPEC P*
⟨*proof*⟩

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

lemma *FOREACHc-refine*:
fixes α :: *'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 *SV*: *single-valued R*
assumes *REFPHI0*: Φ'' *S* $\sigma 0$ (α‘*S*) $\sigma 0'$
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ \llbracket$

$it'{=}\alpha`it;\ it{\subseteq}S;\ it'{\subseteq}S';\ \Phi''\ it\ \sigma\ it'\ \sigma';\ (\sigma,\sigma')\in R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
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 $FOREACHc\ S\ c\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHc\ S'\ c'\ f'\ \sigma 0')$
$\langle proof \rangle$

lemma *FOREACHc-refine-rcg*[*refine*]:
fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S\ set$
fixes $S' :: 'Sa\ set$
assumes *INJ*: $inj\text{-}on\ \alpha\ S$
assumes *REFS*: $S' = \alpha`S$
assumes *REF0*: $(\sigma 0,\sigma 0')\in R$
assumes *SV*: $single\text{-}valued\ R$
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 $FOREACHc\ S\ c\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHc\ S'\ c'\ f'\ \sigma 0')$
$\langle proof \rangle$

lemma *FOREACHi-refine*:
fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S\ set$
fixes $S' :: 'Sa\ set$
assumes *INJ*: $inj\text{-}on\ \alpha\ S$
assumes *REFS*: $S' = \alpha`S$
assumes *REF0*: $(\sigma 0,\sigma 0')\in R$
assumes *SV*: $single\text{-}valued\ R$
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 $FOREACHi\ \Phi\ S\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHi\ \Phi'\ S'\ f'\ \sigma 0')$
$\langle proof \rangle$

lemma *FOREACHi-refine-rcg*[*refine*]:
fixes α :: *'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 *SV*: *single-valued 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* Φ *S f* $\sigma 0 \le \Downarrow R$ (*FOREACHi* Φ' *S' f'* $\sigma 0'$)
$\langle proof \rangle$

lemma *FOREACH-refine*:
fixes α :: *'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 *SV*: *single-valued 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'* $\sigma 0'$)
$\langle proof \rangle$

lemma *FOREACH-refine-rcg*[*refine*]:
fixes α :: *'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 *SV*: *single-valued R*
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \le \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACH S f* $\sigma 0 \le \Downarrow R$ (*FOREACH S' f'* $\sigma 0'$)

$\langle proof \rangle$

lemma *FOREACHci-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 *SV*: *single-valued* 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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi'\ it'\ \sigma';\ c\ \sigma;\ c'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \le \Downarrow R\ (f'\ x'\ \sigma')$
shows $FOREACHc\ S\ c\ f\ \sigma 0 \le \Downarrow R\ (FOREACHci\ \Phi'\ S'\ c'\ f'\ \sigma 0')$
$\langle proof \rangle$

lemma *FOREACHi-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 *SV*: *single-valued* R
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi'\ it'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \le \Downarrow R\ (f'\ x'\ \sigma')$
shows $FOREACH\ S\ f\ \sigma 0 \le \Downarrow R\ (FOREACHi\ \Phi'\ S'\ f'\ \sigma 0')$
$\langle proof \rangle$

2.9.4 FOREACH with empty sets

lemma *FOREACHoci-emp* [*simp*] :
$FOREACHoci\ \Phi\ \{\}\ R\ c\ f\ \sigma = do\ \{ASSERT\ (\Phi\ \{\}\ \sigma);\ RETURN\ \sigma\}$
$\langle proof \rangle$

lemma *FOREACHoi-emp* [*simp*] :
$FOREACHoi\ \Phi\ \{\}\ R\ f\ \sigma = do\ \{ASSERT\ (\Phi\ \{\}\ \sigma);\ RETURN\ \sigma\}$
$\langle proof \rangle$

lemma *FOREACHci-emp* [*simp*] :
$FOREACHci\ \Phi\ \{\}\ c\ f\ \sigma = do\ \{ASSERT\ (\Phi\ \{\}\ \sigma);\ RETURN\ \sigma\}$

⟨*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*⟩

locale *transfer-FOREACH* = *transfer* +
constrains α :: *'mc* ⇒ *'s nres*
fixes *creturn* :: *'s* ⇒ *'mc*
and *cbind* :: *'mc* ⇒ (*'s*⇒*'mc*) ⇒ *'mc*
and *liftc* :: (*'s* ⇒ *bool*) ⇒ *'mc* ⇒ *bool*

assumes *transfer-bind*: ⟦α *m* ≤ *M*; ⋀*x*. α (*f x*) ≤ *F x* ⟧
⟹ α (*cbind m f*) ≤ *bind M F*
assumes *transfer-return*: α (*creturn x*) ≤ *RETURN x*

assumes *liftc*: ⟦*RETURN x* ≤ α *m*; α *m* ≠ *FAIL*⟧ ⟹ *liftc c m* ⟷ *c x*
begin

abbreviation *lift-it* :: (*'x*,*'mc*) *set-iterator* ⇒
(*'s*⇒*bool*) ⇒ (*'x* ⇒ *'s* ⇒ *'mc*) ⇒ *'s* ⇒ *'mc*
where
lift-it it c f s0 ≡ *it*
(*liftc c*)
(λ*x s*. *cbind s* (*f x*))
(*creturn s0*)

lemma *transfer-FOREACHoci*[*refine-transfer*]:
fixes *X* :: *'x set* **and** *s0* :: *'s*
assumes *IT*: *set-iterator-genord iterate X R*
assumes *RS*: ⋀*x s*. α (*f x s*) ≤ *F x s*
shows α (*lift-it iterate c f s0*) ≤ *FOREACHoci I X R c F s0*
⟨*proof*⟩

lemma *transfer-FOREACHoi*[*refine-transfer*]:
fixes *X* :: *'x set* **and** *s0* :: *'s*
assumes *IT*: *set-iterator-genord iterate X R*
assumes *RS*: ⋀*x s*. α (*f x s*) ≤ *F x s*
shows α (*lift-it iterate* (λ-. *True*) *f s0*) ≤ *FOREACHoi I X R F s0*
⟨*proof*⟩

lemma *transfer-FOREACHci*[*refine-transfer*]:
  fixes X :: $'x$ *set* **and** *s0* :: $'s$
  assumes *IT*: *set-iterator iterate X*
  assumes *RS*: $\bigwedge x\ s.\ \alpha\ (f\ x\ s) \le F\ x\ s$
  shows α (*lift-it iterate c f s0*) $\le$ *FOREACHci I X c F s0*
⟨*proof*⟩

lemma *transfer-FOREACHc*[*refine-transfer*]:
  fixes X :: $'x$ *set* **and** *s0* :: $'s$
  assumes *IT*: *set-iterator iterate X*
  assumes *RS*: $\bigwedge x\ s.\ \alpha\ (f\ x\ s) \le F\ x\ s$
  shows α (*lift-it iterate c f s0*) $\le$ *FOREACHc X c F s0*
  ⟨*proof*⟩

lemma *transfer-FOREACHI*[*refine-transfer*]:
  fixes X :: $'x$ *set* **and** *s0* :: $'s$
  assumes *IT*: *set-iterator iterate X*
  assumes *RS*: $\bigwedge x\ s.\ \alpha\ (f\ x\ s) \le F\ x\ s$
  shows α (*lift-it iterate* (λ-. *True*) *f s0*) $\le$ *FOREACHi I X F s0*
  ⟨*proof*⟩

lemma *det-FOREACH*[*refine-transfer*]:
  fixes X :: $'x$ *set* **and** *s0* :: $'s$
  assumes *IT*: *set-iterator iterate X*
  assumes *RS*: $\bigwedge x\ s.\ \alpha\ (f\ x\ s) \le F\ x\ s$
  shows α (*lift-it iterate* (λ-. *True*) *f s0*) $\le$ *FOREACH X F s0*
  ⟨*proof*⟩

end

lemma *FOREACHoci-mono*[*refine-mono*]:
  assumes $\bigwedge x.\ f\ x \le f'\ x$
  shows *FOREACHoci I S R c f s0* $\le$ *FOREACHoci I S R c* f' *s0*
  ⟨*proof*⟩
lemma *FOREACHoi-mono*[*refine-mono*]:
  assumes $\bigwedge x.\ f\ x \le f'\ x$
  shows *FOREACHoi I S R f s0* $\le$ *FOREACHoi I S R* f' *s0*
  ⟨*proof*⟩
lemma *FOREACHci-mono*[*refine-mono*]:
  assumes $\bigwedge x.\ f\ x \le f'\ x$
  shows *FOREACHci I S c f s0* $\le$ *FOREACHci I S c* f' *s0*
  ⟨*proof*⟩
lemma *FOREACHc-mono*[*refine-mono*]:
  assumes $\bigwedge x.\ f\ x \le f'\ x$
  shows *FOREACHc S c f s0* $\le$ *FOREACHc S c* f' *s0*
  ⟨*proof*⟩
lemma *FOREACHi-mono*[*refine-mono*]:
  assumes $\bigwedge x.\ f\ x \le f'\ x$

shows *FOREACHi I S f s0* $\leq$ *FOREACHi I S f' s0*
⟨*proof*⟩
lemma *FOREACH-mono*[*refine-mono*]:
assumes $\bigwedge x.\ f\ x \leq f'\ x$
shows *FOREACH S f s0* $\leq$ *FOREACH S f' s0*
⟨*proof*⟩

end

2.10 Deterministic Monad

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

2.10.1 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 x dFAILi = x
⟨*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*

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.cases*(*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*

$dFAIL \neq dRETURN\ r$
$dSUCCEED \neq dRETURN\ r$
$dFAIL{\neq}dSUCCEED$
$dSUCCEED{\neq}dFAIL$
$x{=}y \Longrightarrow inf\ (dRETURN\ x)\ (dRETURN\ y) = dRETURN\ y$
$x{\neq}y \Longrightarrow inf\ (dRETURN\ x)\ (dRETURN\ y) = dSUCCEED$
$x{=}y \Longrightarrow sup\ (dRETURN\ x)\ (dRETURN\ y) = dRETURN\ y$
$x{\neq}y \Longrightarrow sup\ (dRETURN\ x)\ (dRETURN\ y) = dFAIL$
$\langle proof \rangle$

lemma *dres-Sup-cases*:
obtains $S{\subseteq}\{dSUCCEED\}$ **and** $Sup\ S = dSUCCEED$
| $dFAIL{\in}S$ **and** $Sup\ S = dFAIL$
| $a\ b$ **where** $a{\neq}b$ $dRETURN\ a{\in}S$ $dRETURN\ b{\in}S$ $dFAIL{\notin}S$ $Sup\ S = dFAIL$
| a **where** $S \subseteq \{dSUCCEED,\ dRETURN\ a\}$ $dRETURN\ a{\in}S$ $Sup\ S = dRETURN\ a$
$\langle proof \rangle$

lemma *dres-Inf-cases*:
obtains $S{\subseteq}\{dFAIL\}$ **and** $Inf\ S = dFAIL$
| $dSUCCEED{\in}S$ **and** $Inf\ S = dSUCCEED$
| $a\ b$ **where** $a{\neq}b$ $dRETURN\ a{\in}S$ $dRETURN\ b{\in}S$ $dSUCCEED{\notin}S$ $Inf\ S = dSUCCEED$
| a **where** $S \subseteq \{dFAIL,\ dRETURN\ a\}$ $dRETURN\ a{\in}S$ $Inf\ S = dRETURN\ a$
$\langle proof \rangle$

lemma *dres-chain-eq-res*:
$is\text{-}chain\ M \Longrightarrow$
$dRETURN\ r \in M \Longrightarrow dRETURN\ s \in M \Longrightarrow r{=}s$
$\langle proof \rangle$

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

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

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*⟩

⟨*ML*⟩

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 (λx. dbind (f x) g)*
⟨*proof*⟩

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

lemma *dbind-mono*[*refine-mono*]:
$\llbracket\ M \le M';\ \bigwedge x.\ dRETURN\ x \le M \Longrightarrow f\ x \le f'\ x\ \rrbracket \Longrightarrow$
$dbind\ M\ f \le dbind\ M'\ f'$
⟨*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*]:
$dFAIL \neq dSUCCEED$
$dRETURN\ x \neq dSUCCEED$
$[\![\ m{\neq}dSUCCEED;\ \bigwedge x.\ f\ x \neq dSUCCEED\ ]\!] \Longrightarrow dbind\ m\ f \neq dSUCCEED$
$dASSERT\ \Phi \neq dSUCCEED$
$[\![\ m1{\neq}dSUCCEED;\ m2{\neq}dSUCCEED\ ]\!] \Longrightarrow If\ b\ m1\ m2 \neq dSUCCEED$
$[\![\ \bigwedge x.\ f\ x{\neq}dSUCCEED\ ]\!] \Longrightarrow Let\ x\ f \neq dSUCCEED$
$[\![\ \bigwedge x1\ x2.\ g\ x1\ x2 \neq dSUCCEED\ ]\!] \Longrightarrow prod\text{-}case\ g\ p \neq dSUCCEED$
⟨*proof*⟩

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

2.11 Transfer Setup

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

2.11.1 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 $nres\text{-}of\ r \equiv case\ r\ of$
$dSUCCEEDi \Rightarrow SUCCEED$
$|\ dFAILi \Rightarrow FAIL$
$|\ dRETURN\ x \Rightarrow RETURN\ x$

lemma *nres-of-simps*[*simp*]:
$nres\text{-}of\ dSUCCEED = SUCCEED$
$nres\text{-}of\ dFAIL = FAIL$
$nres\text{-}of\ (dRETURN\ x) = RETURN\ x$
⟨*proof*⟩

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

lemma *nres-transfer*:
$nres\text{-}of\ dSUCCEED = SUCCEED$
$nres\text{-}of\ dFAIL = FAIL$
$nres\text{-}of\ a \leq nres\text{-}of\ b \longleftrightarrow a \leq b$
$nres\text{-}of\ a < nres\text{-}of\ b \longleftrightarrow a < b$

is-chain A $\Longrightarrow$ *nres-of* (*Sup A*) = *Sup* (*nres-of*'*A*)
is-chain A $\Longrightarrow$ *nres-of* (*Inf A*) = *Inf* (*nres-of*'*A*)
⟨*proof*⟩

lemma *nres-correctD*:
assumes *nres-of S* $\leq$ *SPEC* Φ
shows
S=*dRETURN x* $\Longrightarrow$ Φ *x*
S$\neq$*dFAIL*
⟨*proof*⟩

Transfer Theorems Setup

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

lemma *det-FAIL*[*refine-transfer*]: *nres-of* (*dFAIL*) $\leq$ *FAIL* ⟨*proof*⟩
lemma *det-SUCCEED*[*refine-transfer*]: *nres-of* (*dSUCCEED*) $\leq$ *SUCCEED* ⟨*proof*⟩
lemma *det-SPEC*: Φ *x* $\Longrightarrow$ *nres-of* (*dRETURN x*) $\leq$ *SPEC* Φ ⟨*proof*⟩
lemma *det-RETURN*[*refine-transfer*]:
nres-of (*dRETURN x*) $\leq$ *RETURN x* ⟨*proof*⟩
lemma *det-bind*[*refine-transfer*]:
assumes *nres-of m* $\leq$ *M*
assumes $\bigwedge$*x*. *nres-of* (*f x*) $\leq$ *F x*
shows *nres-of* (*dbind m f*) $\leq$ *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*⟩

interpretation *det-foreach*!:
transfer-FOREACH nres-of dRETURN dbind dres-case True True
⟨*proof*⟩

2.11.2 Transfer to Plain Function

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

lemma *plain-RETURN*[*refine-transfer*]: *RETURN a* $\leq$ *RETURN a* ⟨*proof*⟩
lemma *plain-bind*[*refine-transfer*]:
$\llbracket$*RETURN x* $\leq$ *M*; $\bigwedge$*x*. *RETURN* (*f x*) $\leq$ *F x*$\rrbracket$ $\Longrightarrow$ *RETURN* (*Let x f*) $\leq$ *bind M F*
⟨*proof*⟩

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

interpretation *plain*!: *transfer-FOREACH RETURN* $(\lambda x.\ x)$ *Let* $(\lambda x.\ x)$
⟨*proof*⟩

2.11.3 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) $\leq$ S'. Here, *the-res* extracts the result from a deterministic monad. As *the-res* is executable, the above shows that (*the-res* S) is always a correct result.

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

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

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

lemma *the-resI*:
assumes *nres-of* $S \leq S'$
assumes $S \neq$ *dSUCCEED*
shows *RETURN* (*the-res* S) $\leq S'$
⟨*proof*⟩

end

2.12 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.

2.12.1 Nondeterministic Result Monad

interpretation *nrec*!:

partial-function-definitions op $\leq$ *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*]:
$[\![$ *nrec.mono-body* B; $\bigwedge y.$ *nrec.mono-body* $(\lambda f.\ C\ y\ f)$ $]\!] \Longrightarrow$
nrec.mono-body $(\lambda f.\ bind\ (B\ f)\ (\lambda y.\ C\ y\ f))$
⟨*proof*⟩

2.12.2 Deterministic Result Monad

interpretation *drec*!:
partial-function-definitions op $\leq$ *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*]:
$[\![$ *drec.mono-body* B; $\bigwedge y.$ *drec.mono-body* $(\lambda f.\ C\ y\ f)$ $]\!] \Longrightarrow$
drec.mono-body $(\lambda f.\ dbind\ (B\ f)\ (\lambda y.\ C\ y\ f))$
⟨*proof*⟩

end

2.13 Automatic Data Refinement

theory *Refine-Autoref*
imports
Refine-Basic
Refine-While
Refine-Foreach
Refine-Heuristics

begin

This theory demonstrates a possible approach to automatic data refinement according to some type-based and tag-based rules.

Note that this is still a prototype.

2.13.1 Preliminaries

Preliminaries for idtrans-tac, that transfers operations along identity relation.

lemma *idtrans0*: $(f,f)\in Id$ ⟨*proof*⟩
lemma *idtrans-arg*: $(f,f')\in Id \Longrightarrow (a,a')\in Id \Longrightarrow (f\ a,f'\ a')\in Id$ ⟨*proof*⟩

2.13.2 Setup

⟨*ML*⟩

2.13.3 Configuration

We now add the default configuration for the automatic refinement tactic, including decomposition statements for most program constructs, and some default setup for product types in expressions.

Program Constructs

lemma *Let-autoref*[*autoref-prg*]:
assumes $(x,x')\in R'$
assumes $\bigwedge x\ x'.\ (x,x')\in R' \Longrightarrow f\ x \le\Downarrow R\ (f'\ x')$
shows $Let\ x\ f \le\Downarrow R\ (Let\ x'\ f')$
⟨*proof*⟩

lemmas *std-constructs-autoref-aux* =
bind-refine ASSERT-refine-right if-refine
WHILET-refine WHILE-refine WHILET-refine′ WHILE-refine′

lemmas [*autoref-prg*] = *std-constructs-autoref-aux*[*folded pair-in-Id-conv*]

lemma *REC-autoref*[*autoref-prg*]:
assumes *R0*: $(x,x')\in R$
assumes *RS*: $\bigwedge f\ f'\ x\ x'.\ [\![\ \bigwedge x\ x'.\ (x,x')\in R \Longrightarrow f\ x \le\Downarrow S\ (f'\ x');\ (x,x')\in R\ ]\!]$
$\Longrightarrow body\ f\ x \le\Downarrow S\ (body'\ f'\ x')$
assumes *M*: *mono body*
shows $REC\ body\ x \le\Downarrow S\ (REC\ body'\ x')$
⟨*proof*⟩

lemma *RECT-autoref* [*autoref-prg*]:
assumes *R0*: $(x,x')\in R$
assumes *RS*: $\bigwedge f\, f'\, x\, x'.\ [\![\ \bigwedge x\, x'.\ (x,x')\in R \Longrightarrow f\ x \le\Downarrow S\ (f'\ x');\ (x,x')\in R\ ]\!]$
$\Longrightarrow body\ f\ x \le\Downarrow S\ (body'\ f'\ x')$
assumes *M*: *mono body*
shows $RECT\ body\ x \le\Downarrow S\ (RECT\ body'\ x')$
⟨*proof*⟩

lemma *RETURN-autoref* [*autoref-prg*]:
$[\![(x,\ x') \in R;\ single\text{-}valued\ R]\!] \Longrightarrow RETURN\ x \le \Downarrow R\ (RETURN\ x')$
⟨*proof*⟩

lemma *FOREACH-autoref* [*autoref-prg*]:
fixes $S :: 'S\ set$
assumes *REF0*: $(\sigma 0,\sigma 0')\in R$
assumes *REFS*: $(S,S')\in Id$
assumes *SV*: *single-valued R*
assumes *REFSTEP*: $\bigwedge\sigma\ \sigma'\ x\ x'.\ [\![\ (\sigma,\sigma')\in R;\ (x,x')\in Id\ ]\!] \Longrightarrow f\ x\ \sigma$
$\le \Downarrow R\ (f'\ x'\ \sigma')$
shows $FOREACH\ S\ f\ \sigma 0 \le \Downarrow R\ (FOREACH\ S'\ f'\ \sigma 0')$
⟨*proof*⟩

lemma *FOREACHi-autoref* [*autoref-prg*]:
fixes $S :: 'S\ set$
assumes *REF0*: $(\sigma 0,\sigma 0')\in R$
assumes *REFS*: $(S,S')\in Id$
assumes *SV*: *single-valued R*
assumes *REFSTEP*: $\bigwedge\sigma\ \sigma'\ x\ x'.\ [\![\ (\sigma,\sigma')\in R;\ (x,x')\in Id\ ]\!] \Longrightarrow f\ x\ \sigma$
$\le \Downarrow R\ (f'\ x'\ \sigma')$
shows $FOREACH\ S\ f\ \sigma 0 \le \Downarrow R\ (FOREACHi\ I\ S'\ f'\ \sigma 0')$
⟨*proof*⟩

lemma *FOREACHc-autoref* [*autoref-prg*]:
fixes $S :: 'S\ set$
assumes *REF0*: $(\sigma 0,\sigma 0')\in R$
assumes *REFS*: $(S,S')\in Id$
assumes *SV*: *single-valued R*
assumes *REFC*: $\bigwedge\sigma\ \sigma'.\ (\sigma,\sigma')\in R \Longrightarrow (c\ \sigma,\ c'\ \sigma')\in Id$
assumes *REFSTEP*: $\bigwedge\sigma\ \sigma'\ x\ x'.\ [\![\ (\sigma,\sigma')\in R;\ (x,x')\in Id\ ]\!] \Longrightarrow f\ x\ \sigma$
$\le \Downarrow R\ (f'\ x'\ \sigma')$
shows $FOREACHc\ S\ c\ f\ \sigma 0 \le \Downarrow R\ (FOREACHc\ S'\ c'\ f'\ \sigma 0')$
⟨*proof*⟩

lemma *FOREACHci-autoref* [*autoref-prg*]:
fixes $S :: 'S\ set$
assumes *REF0*: $(\sigma 0,\sigma 0')\in R$
assumes *REFS*: $(S,S')\in Id$
assumes *SV*: *single-valued R*
assumes *REFC*: $\bigwedge\sigma\ \sigma'.\ (\sigma,\sigma')\in R \Longrightarrow (c\ \sigma,\ c'\ \sigma')\in Id$

assumes *REFSTEP*: $\bigwedge\sigma\ \sigma'\ x\ x'.\ [\![\ (\sigma,\sigma')\in R;\ (x,x')\in Id\ ]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow R\ (f'\ x'\ \sigma')$
shows $FOREACHc\ S\ c\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHci\ I\ S'\ c'\ f'\ \sigma 0')$
⟨*proof*⟩

Product splitting

Product splitting has not yet been solved properly. Below, you see the current hack that works in many cases.

lemma *rprod-iff* [*autoref-simp*]:
$((a,b),(a',b'))\in rprod\ R1\ R2 \longleftrightarrow (a,a')\in R1 \wedge (b,b')\in R2$ ⟨*proof*⟩

lemmas [*autoref-elim*] = *conjE impE*

lemma *prod-id-split*[*autoref-simp*]: *Id* = *rprod Id Id* ⟨*proof*⟩

declare *nested-prod-case-simp*[*autoref-simp*]

lemma *prod-split-elim*[*autoref-elim*]:
$[\![(x,x')\in rprod\ Ra\ Rb;$
$\bigwedge a\ b\ a'\ b'.\ [\![x=(a,b);\ x'=(a',b');\ (a,a')\in Ra;\ (b,b')\in Rb]\!]$
$\Longrightarrow (f\ a\ b,f'\ (a',b'))\in R$
$]\!]$
$\Longrightarrow$ (*prod-case* $f\ x,f'\ x')\in R$ ⟨*proof*⟩

lemma *prod-split-elim-prg*[*autoref-elim*]:
$[\![(x,x')\in rprod\ Ra\ Rb;$
$\bigwedge a\ b\ a'\ b'.\ [\![x=(a,b);\ x'=(a',b');\ (a,a')\in Ra;\ (b,b')\in Rb]\!]$
$\Longrightarrow f\ a\ b \leq \Downarrow R\ (f'\ (a',b'))$
$]\!]$
$\Longrightarrow$ *prod-case* $f\ x \leq\Downarrow R\ (f'\ x')$ ⟨*proof*⟩

lemma *rprod-spec*:
fixes $ca :: 'ca$ **and** $cb::'cb$ **and** $aa::'aa$ **and** $ab::'ab$
assumes $(ca,aa)\in Ra$
assumes $(cb,ab)\in Rb$
shows $((ca,cb),(aa,ab))\in rprod\ Ra\ Rb$
⟨*proof*⟩

This will split all products by default. In most cases, this is the desired behaviour, otherwise the lemma should be removed in your local theory by *declare rprod-spec*[*autoref-spec del*]

declare *rprod-spec*[*autoref-spec*]

lemma *prod-case-autoref-ex*[*autoref-ex*]:
assumes $(f,\ f'\ a'\ b')\in R$
shows $(f,$ *prod-case* $f'\ (a',b'))\in R$
⟨*proof*⟩

lemma *prod-case-autoref-prg*[*autoref-prg*]:
  assumes $f \leq \Downarrow R\ (f'\ a'\ b')$
  shows $f \leq \Downarrow R\ (\mathit{prod\text{-}case}\ f'\ (a',b'))$
  ⟨*proof*⟩

lemma *pair-autoref*[*autoref-ex*]:
  $\llbracket\ (a,a')\in Ra;\ (b,b')\in Rb\ \rrbracket \Longrightarrow ((a,b),(a',b'))\in \mathit{rprod}\ Ra\ Rb$
  ⟨*proof*⟩

Discharging single-valued goals

lemmas [*autoref-other*] = *br-single-valued rprod-sv single-valued-Id map-list-rel-sv map-set-rel-sv*

Tagging

Tags can be used to stop the processing at the tagged term, or to apply some special rules to the tagged translation.

Named tag to be placed around a term

definition $TAG :: \mathit{string} \Rightarrow 'a \Rightarrow 'a$ **where** [*simp*]: $TAG\ s = \mathit{id}$

To be used as *simp only*: *TAG-remove* or *unfold TAG-remove*, to safely remove tags from program.

lemma *TAG-remove*[*simp*]: $TAG\ s\ t = t$ ⟨*proof*⟩

Remove tag. To be used partially instantiated as spec-rule

lemma *TAG-dest*:
  assumes $(c::'c,a::'a)\in R$
  shows $(c,TAG\ \mathit{name}\ a)\in R$
  ⟨*proof*⟩

Specify refinement relation. To be used partially instantiated as spec-rule.

lemma *spec-R*:
  fixes $c::'c$ **and** $a::'a$
  assumes $(c,a)\in R$
  shows $(c,a)\in R$
  ⟨*proof*⟩

Specify identity translation. To be used partially instantiated as spec-rule.

lemma *spec-Id*:
  fixes $c::'a$ **and** $a::'a$
  assumes $(c,a)\in Id$
  shows $(c,a)\in Id$
  ⟨*proof*⟩

2.13.4 Convenience Lemmas

Introduce autoref, determinize, optimize

lemma *autoref-det-optI*:
  assumes $m \leq \Downarrow R\ a$
  assumes $\alpha\ c' \leq m$
  assumes $c = c'$
  shows $\alpha\ c \leq \Downarrow R\ a$
  ⟨*proof*⟩

Introduce autoref, determinize

lemma *autoref-detI*:
  assumes $m \leq \Downarrow R\ a$
  assumes $\alpha\ c \leq m$
  shows $\alpha\ c \leq \Downarrow R\ a$
  ⟨*proof*⟩

To be used in optimize phase:

lemma *prod-case-rename*:
  $\textit{prod-case}\ (\lambda a\ \text{-}.\ f\ a) = f\ o\ \textit{fst}$
  $\textit{prod-case}\ (\lambda\text{-}\ b.\ f\ b) = f\ o\ \textit{snd}$
  ⟨*proof*⟩

end

2.14 Refinement Framework

theory *Refine*
imports
  Refine-Chapter
  Refine-Basic
  Refine-Heuristics
  Refine-While
  Refine-Foreach
  Refine-Transfer
  Refine-Pfun
  Refine-Autoref
begin

This theory summarizes all default theories of the refinement framework.

end

2.15 ICF Bindings

theory *Collection-Bindings*
imports *../Collections/Collections Refine*
begin

This theory sets up some lemmas that automate refinement proofs using the Isabelle Collection Framework (ICF).

lemma (**in** *set*) *drh*[*refine-dref-RELATES*]:
  RELATES (*build-rel* α *invar*) $\langle proof \rangle$
lemma (**in** *map*) *drh*[*refine-dref-RELATES*]:
  RELATES (*build-rel* α *invar*) $\langle proof \rangle$

lemma (**in** *uprio*) *drh*[*refine-dref-RELATES*]: *RELATES* (*build-rel* α *invar*)
  $\langle proof \rangle$
lemma (**in** *prio*) *drh*[*refine-dref-RELATES*]: *RELATES* (*build-rel* α *invar*)
  $\langle proof \rangle$

lemmas (**in** *StdSet*) [*refine-hsimp*] = *correct*
lemmas (**in** *StdMap*) [*refine-hsimp*] = *correct*

lemma (**in** *set-sel'*) *pick-ref*[*refine-hsimp*]:
  $[\![$ *invar s*; $\alpha\ s \neq \{\}]\!] \Longrightarrow$ *the* (*sel' s* (λ-. *True*)) $\in \alpha\ s$
  $\langle proof \rangle$

Wrapper to prevent higher-order unification problems

definition [*simp*, *code-unfold*]: *IT-tag x* $\equiv$ *x*

lemma (**in** *set-iteratei*) *it-is-iterator*[*refine-transfer*]:
  invar s $\Longrightarrow$ *set-iterator* (*IT-tag iteratei s*) (α *s*)
  $\langle proof \rangle$

lemma (**in** *map-iteratei*) *it-is-iterator*[*refine-transfer*]:
  invar m $\Longrightarrow$ *set-iterator* (*IT-tag iteratei m*) (*map-to-set* (α *m*))
  $\langle proof \rangle$

This definition is handy to be used on the abstract level.

definition *prio-pop-min q* $\equiv$ *do* {
    ASSERT (*dom q* $\neq$ {});
    SPEC (λ(*e*,*w*,*q'*).
      q'=*q*(*e*:=*None*) $\wedge$
      q e = *Some w* $\wedge$
      ($\forall$ *e' w'*. *q e'* = *Some w'* $\longrightarrow$ *w*$\leq$*w'*)
)
  }

lemma (**in** *uprio-pop*) *prio-pop-min-refine*[*refine*]:
  (*q*,*q'*)$\in$*build-rel* α *invar* $\Longrightarrow$ *RETURN* (*pop q*)

$\le \Downarrow (rprod\ Id\ (rprod\ Id\ (build\text{-}rel\ \alpha\ invar)))$
$(prio\text{-}pop\text{-}min\ q')$
$\langle proof \rangle$

lemma *dres-ne-bot-iterate*[*refine-transfer*]:
assumes B: $set\text{-}iterator\ (IT\text{-}tag\ it\ r)\ S$
assumes A: $\bigwedge x\ s.\ f\ x\ s \neq dSUCCEED$
shows $IT\text{-}tag\ it\ r\ c\ (\lambda x\ s.\ dbind\ s\ (f\ x))\ (dRETURN\ s) \neq dSUCCEED$
$\langle proof \rangle$

Monotonicity for Iterators

lemma *it-mono-aux*:
assumes $COND$: $\bigwedge \sigma\ \sigma'.\ \sigma \le \sigma' \Longrightarrow c\ \sigma \neq c\ \sigma' \Longrightarrow \sigma = bot \vee \sigma' = top$
assumes $STRICT$: $\bigwedge x.\ f\ x\ bot = bot \quad \bigwedge x.\ f'\ x\ top = top$
assumes B: $\sigma \le \sigma'$
assumes A: $\bigwedge a\ x\ x'.\ x \le x' \Longrightarrow f\ a\ x \le f'\ a\ x'$
shows $foldli\ l\ c\ f\ \sigma \le foldli\ l\ c\ f'\ \sigma'$
$\langle proof \rangle$

lemma *it-mono-aux-dres'*:
assumes $STRICT$: $\bigwedge x.\ f\ x\ bot = bot \quad \bigwedge x.\ f'\ x\ top = top$
assumes A: $\bigwedge a\ x\ x'.\ x \le x' \Longrightarrow f\ a\ x \le f'\ a\ x'$
shows $foldli\ l\ (dres\text{-}case\ True\ True\ c)\ f\ \sigma$
$\le foldli\ l\ (dres\text{-}case\ True\ True\ c)\ f'\ \sigma$
$\langle proof \rangle$

lemma *it-mono-aux-dres*:
assumes A: $\bigwedge a\ x.\ f\ a\ x \le f'\ a\ x$
shows $foldli\ l\ (dres\text{-}case\ True\ True\ c)\ (\lambda x\ s.\ dbind\ s\ (f\ x))\ \sigma$
$\le foldli\ l\ (dres\text{-}case\ True\ True\ c)\ (\lambda x\ s.\ dbind\ s\ (f'\ x))\ \sigma$
$\langle proof \rangle$

lemma *iteratei-mono'*:
assumes L: $set\text{-}iteratei\ \alpha\ invar\ it$
assumes $STRICT$: $\bigwedge x.\ f\ x\ bot = bot \quad \bigwedge x.\ f'\ x\ top = top$
assumes A: $\bigwedge a\ x\ x'.\ x \le x' \Longrightarrow f\ a\ x \le f'\ a\ x'$
assumes I: $invar\ s$
shows $IT\text{-}tag\ it\ s\ (dres\text{-}case\ True\ True\ c)\ f\ \sigma$
$\le IT\text{-}tag\ it\ s\ (dres\text{-}case\ True\ True\ c)\ f'\ \sigma$
$\langle proof \rangle$

lemma *iteratei-mono*:
assumes L: $set\text{-}iteratei\ \alpha\ invar\ it$
assumes A: $\bigwedge a\ x.\ f\ a\ x \le f'\ a\ x$
assumes I: $invar\ s$
shows $IT\text{-}tag\ it\ s\ (dres\text{-}case\ True\ True\ c)\ (\lambda x\ s.\ dbind\ s\ (f\ x))\ \sigma$

$\leq$ *IT-tag it s (dres-case True True c) (λx s. dbind s (f' x))* σ
⟨*proof*⟩

lemmas [*refine-mono*] = *iteratei-mono*[*OF ls-iteratei-impl*]
lemmas [*refine-mono*] = *iteratei-mono*[*OF lsi-iteratei-impl*]
lemmas [*refine-mono*] = *iteratei-mono*[*OF rs-iteratei-impl*]
lemmas [*refine-mono*] = *iteratei-mono*[*OF ahs-iteratei-impl*]
lemmas [*refine-mono*] = *iteratei-mono*[*OF ias-iteratei-impl*]
lemmas [*refine-mono*] = *iteratei-mono*[*OF ts-iteratei-impl*]

end

2.16 Automatic Refinement: Collection Bindings

theory *Autoref-Collection-Bindings*
imports *Refine-Autoref Collection-Bindings ../Collections/Collections*
begin

This theory provides an (incomplete) automatic refinement setup for the Isabelle Collection Framework.

Note that the quality of the generated refinement depends on the order in which the lemmas are set up here: Lemmas for special operations must be set up after lemmas for general operations, e.g., the singleton set should come after insert and empty set. Otherwise, the specialized operation will never be tried, because the general operation yields a suitable translation, too.

abbreviation (*input*) *DETREFe* **where** *DETREFe c r a* $\equiv$ *(c,a)*$\in$*r*

2.16.1 Set

lemma (**in** *set-empty*) *set-empty-et*[*autoref-ex*]:
(*empty* (),{})$\in$(*build-rel* (α::'s $\Rightarrow$ 'x set) *invar*)
⟨*proof*⟩

lemma (**in** *set-memb*) *set-memb-et*[*autoref-ex*]:
assumes *DETREFe x Id* x'
assumes *DETREFe s* (*build-rel* α *invar*) s'
shows *DETREFe* (*memb x s*) *Id* ($x'$$\in$$s'$)
⟨*proof*⟩

lemma (**in** *set-ball*) *set-ball-et*[*autoref-ex*]:
DETREFe s (*build-rel* α *invar*) s' $\Longrightarrow$ *DETREFe P Id* P'
$\Longrightarrow$ *DETREFe* (*ball s P*) *Id* ($\forall x$$\in$$s'$. P' *x*)
⟨*proof*⟩

lemma (**in** *set-bexists*) *set-bexists-et*[*autoref-ex*]:

DETREFe s (*build-rel* α *invar*) $s' \Longrightarrow$ *DETREFe P Id* P'
$\Longrightarrow$ *DETREFe* (*bexists s P*) *Id* ($\exists x \in s'.\ P'\ x$)
⟨*proof*⟩

lemma (**in** *set-size*) *set-size-et*[*autoref-ex*]:
DETREFe s (*build-rel* α *invar*) s'
$\Longrightarrow$ *DETREFe* (*size s*) *Id* (*card* s')
⟨*proof*⟩

lemma (**in** *set-size-abort*) *set-size-abort-et*[*autoref-ex*]:
DETREFe m Id $m' \Longrightarrow$ *DETREFe s* (*build-rel* α *invar*) s'
$\Longrightarrow$ *DETREFe* (*size-abort m s*) *Id* (*min* m' (*card* s'))
⟨*proof*⟩

lemma (**in** *set-union*) *set-union-et*[*autoref-ex*]:
assumes *DETREFe s1* (*build-rel* $\alpha 1$ *invar1*) $s1'$
assumes *DETREFe s2* (*build-rel* $\alpha 2$ *invar2*) $s2'$
shows *DETREFe* (*union s1 s2*) (*build-rel* $\alpha 3$ *invar3*) ($s1' \cup s2'$)
⟨*proof*⟩

lemma (**in** *set-union-dj*) *set-union-dj-et*[*autoref-ex*]:
assumes $s1' \cap s2' = \{\}$
assumes *DETREFe s1* (*build-rel* $\alpha 1$ *invar1*) $s1'$
assumes *DETREFe s2* (*build-rel* $\alpha 2$ *invar2*) $s2'$
shows *DETREFe* (*union-dj s1 s2*) (*build-rel* $\alpha 3$ *invar3*) ($s1' \cup s2'$)
⟨*proof*⟩

lemma (**in** *set-diff*) *set-diff-et*[*autoref-ex*]:
assumes *DETREFe s1* (*build-rel* $\alpha 1$ *invar1*) $s1'$
assumes *DETREFe s2* (*build-rel* $\alpha 2$ *invar2*) $s2'$
shows *DETREFe* (*diff s1 s2*) (*build-rel* $\alpha 1$ *invar1*) ($s1' - s2'$)
⟨*proof*⟩

lemma (**in** *set-filter*) *set-filter-et*[*autoref-ex*]:
assumes *DETREFe P Id* P'
assumes *DETREFe s* (*build-rel* $\alpha 1$ *invar1*) s'
shows *DETREFe* (*filter P s*) (*build-rel* $\alpha 2$ *invar2*) ($\{e \in s'.\ P'\ e\}$)
⟨*proof*⟩

lemma (**in** *set-inter*) *set-inter-et*[*autoref-ex*]:
assumes *DETREFe s1* (*build-rel* $\alpha 1$ *invar1*) $s1'$
assumes *DETREFe s2* (*build-rel* $\alpha 2$ *invar2*) $s2'$
shows *DETREFe* (*inter s1 s2*) (*build-rel* $\alpha 3$ *invar3*) ($s1' \cap s2'$)
⟨*proof*⟩

lemma (**in** *set-ins*) *set-ins-et*[*autoref-ex*]:
DETREFe x Id $x' \Longrightarrow$ *DETREFe s* (*build-rel* α *invar*) s'
$\Longrightarrow$ *DETREFe* (*ins x s*) (*build-rel* α *invar*) (*insert* x' s')
⟨*proof*⟩

lemma (**in** *set-ins-dj*) *set-ins-dj-et*[*autoref-ex*]:
$x' \notin s' \Longrightarrow$
$DETREFe\ x\ Id\ x' \Longrightarrow DETREFe\ s\ (build\text{-}rel\ \alpha\ invar)\ s'$
$\Longrightarrow DETREFe\ (ins\text{-}dj\ x\ s)\ (build\text{-}rel\ \alpha\ invar)\ (insert\ x'\ s')$
⟨*proof*⟩

lemma (**in** *set-sng*) *set-sng-et*[*autoref-ex*]:
$DETREFe\ x\ Id\ x'$
$\Longrightarrow DETREFe\ (sng\ x)\ (build\text{-}rel\ \alpha\ invar)\ (\{x'\})$
⟨*proof*⟩

lemma (**in** *set-delete*) *set-delete-et*[*autoref-ex*]:
$DETREFe\ x\ Id\ x' \Longrightarrow DETREFe\ s\ (build\text{-}rel\ \alpha\ invar)\ s'$
$\Longrightarrow DETREFe\ (delete\ x\ s)\ (build\text{-}rel\ \alpha\ invar)\ (s'-\{x'\})$
⟨*proof*⟩

lemma (**in** *set-subset*) *set-subset-et*[*autoref-ex*]:
assumes $DETREFe\ s1\ (build\text{-}rel\ \alpha 1\ invar1)\ s1'$
assumes $DETREFe\ s2\ (build\text{-}rel\ \alpha 2\ invar2)\ s2'$
shows $DETREFe\ (subset\ s1\ s2)\ Id\ (s1' \subseteq s2')$
⟨*proof*⟩

lemma (**in** *set-equal*) *set-equal-et*[*autoref-ex*]:
assumes $DETREFe\ s1\ (build\text{-}rel\ \alpha 1\ invar1)\ s1'$
assumes $DETREFe\ s2\ (build\text{-}rel\ \alpha 2\ invar2)\ s2'$
shows $DETREFe\ (equal\ s1\ s2)\ Id\ (s1'=s2')$
⟨*proof*⟩

lemma (**in** *set-isEmpty*) *set-isEmpty-et*[*autoref-ex*]:
$DETREFe\ s\ (build\text{-}rel\ \alpha\ invar)\ s' \Longrightarrow DETREFe\ (isEmpty\ s)\ Id\ (s'=\{\})$
⟨*proof*⟩

lemma (**in** *set-isSng*) *set-isSng-et*[*autoref-ex*]:
$DETREFe\ s\ (build\text{-}rel\ \alpha\ invar)\ s' \Longrightarrow DETREFe\ (isSng\ s)\ Id\ (\exists\, e.\ s'=\{e\})$
⟨*proof*⟩

lemma (**in** *set-disjoint*) *set-disjoint-et*[*autoref-ex*]:
assumes $DETREFe\ s1\ (build\text{-}rel\ \alpha 1\ invar1)\ s1'$
assumes $DETREFe\ s2\ (build\text{-}rel\ \alpha 2\ invar2)\ s2'$
shows $DETREFe\ (disjoint\ s1\ s2)\ Id\ (s1' \cap s2'=\{\})$
⟨*proof*⟩

thm *set-disjoint-witness.disjoint-witness-correct*
lemma (**in** *set-disjoint-witness*) *set-disjoint-witness-et*[*autoref-ex*]:
assumes $DETREFe\ s1\ (build\text{-}rel\ \alpha 1\ invar1)\ s1'$
assumes $DETREFe\ s2\ (build\text{-}rel\ \alpha 2\ invar2)\ s2'$
shows $RETURN\ (disjoint\text{-}witness\ s1\ s2) \le \Downarrow Id\ (SPEC$
$(\lambda r.\ if\ (s1' \cap s2'=\{\})\ then\ r=None\ else\ (\exists\, e \in s1' \cap s2'.\ r=Some\ e)))$

⟨*proof*⟩

lemma (**in** *set-sel'*) *set-sel'-et*[*autoref-ex*]:
assumes $DETREFe\ P\ Id\ P'$
assumes $DETREFe\ s\ (build\text{-}rel\ \alpha\ invar)\ s'$
shows $RETURN\ (sel'\ s\ P) \leq\Downarrow Id$
$(SPEC\ (\lambda r.\ if\ (\forall x{\in}s'.\ \neg\ P'\ x)\ then\ r{=}None\ else\ (\exists x{\in}s'.\ P'\ x \wedge r{=}Some\ x)))$
⟨*proof*⟩

lemma (**in** *set-to-list*) *set-to-list-et*[*autoref-ex*]:
assumes $DETREFe\ s\ (build\text{-}rel\ \alpha\ invar)\ s'$
shows $RETURN\ (to\text{-}list\ s) \leq\Downarrow Id\ (SPEC\ (\lambda l.\ set\ l = s' \wedge distinct\ l))$
⟨*proof*⟩

lemma (**in** *list-to-set*) *list-to-set-et*[*autoref-ex*]:
assumes $DETREFe\ l\ Id\ l'$
shows $DETREFe\ (to\text{-}set\ l)\ (build\text{-}rel\ \alpha\ invar)\ (set\ l')$
⟨*proof*⟩

lemma (**in** *set-sel'*) *pick-et*[*autoref-ex*]:
assumes $s'{\neq}\{\}$
assumes $DETREFe\ s\ (build\text{-}rel\ \alpha\ invar)\ s'$
shows $RETURN\ (the\ (sel'\ s\ (\lambda\text{-}.\ True))) \leq\Downarrow Id\ (SPEC\ (\lambda x.\ x{\in}s'))$
⟨*proof*⟩

2.16.2 Map

TODO: Still incomplete

lemma (**in** *map-empty*) *map-empty-t*[*autoref-ex*]:
$DETREFe\ (empty\ ())\ (build\text{-}rel\ \alpha\ invar)\ Map.empty$
⟨*proof*⟩

lemma (**in** *map-lookup*) *map-lookup-t*:
assumes $DETREFe\ k\ Id\ k'$
assumes $DETREFe\ m\ (build\text{-}rel\ \alpha\ invar)\ m'$
shows $DETREFe\ (lookup\ k\ m)\ Id\ (m'\ k')$
⟨*proof*⟩

lemma (**in** *map-update*) *map-update-t*[*autoref-ex*]:
assumes $DETREFe\ k\ Id\ k'$
assumes $DETREFe\ v\ Id\ v'$
assumes $DETREFe\ m\ (build\text{-}rel\ \alpha\ invar)\ m'$
shows $DETREFe\ (update\ k\ v\ m)\ (build\text{-}rel\ \alpha\ invar)\ (m'(k'{\mapsto}v'))$
⟨*proof*⟩

lemma (**in** *map-sng*) *map-sng-t*[*autoref-ex*]:
assumes $DETREFe\ k\ Id\ k'$

assumes $DETREFe\ v\ Id\ v'$
shows $DETREFe\ (sng\ k\ v)\ (build\text{-}rel\ \alpha\ invar)\ [k' \mapsto v']$
$\langle proof \rangle$

2.16.3 Unique Priority Queue

TODO: Still incomplete

lemma (**in** *uprio-empty*) *uprio-empty-t*[*autoref-ex*]:
$DETREFe\ (empty\ ())\ (build\text{-}rel\ \alpha\ invar)\ Map.empty$
$\langle proof \rangle$

lemma (**in** *uprio-isEmpty*) *uprio-isEmpty-t*[*autoref-ex*]:
assumes $DETREFe\ s\ (build\text{-}rel\ \alpha\ invar)\ s'$
shows
$DETREFe\ (isEmpty\ s)\ Id\ (s'{=}Map.empty)$
$DETREFe\ (isEmpty\ s)\ Id\ (dom\ s'{=}\{\})$
$\langle proof \rangle$

lemma (**in** *uprio-insert*) *uprio-insert-t*[*autoref-ex*]:
assumes $DETREFe\ s\ (build\text{-}rel\ \alpha\ invar)\ s'$
assumes $DETREFe\ e\ Id\ e'$
assumes $DETREFe\ a\ Id\ a'$
shows $DETREFe\ (insert\ s\ e\ a)\ (build\text{-}rel\ \alpha\ invar)\ (s'(e' \mapsto a'))$
$\langle proof \rangle$

lemma (**in** *uprio-pop*) *uprio-pop-t*[*autoref-ex*]:
assumes $dom\ q'{\neq}\{\}$
assumes $DETREFe\ q\ (build\text{-}rel\ \alpha\ invar)\ q'$
shows $RETURN\ (pop\ q) \leq \Downarrow(rprod\ Id\ (rprod\ Id\ (build\text{-}rel\ \alpha\ invar)))$
$(SPEC\ (\lambda(e,\ w,\ rq').$
$rq' = q'(e := None)\ \wedge$
$q'\ e = Some\ w \wedge (\forall\ e'\ w'.\ q'\ e' = Some\ w' \longrightarrow w \leq w')))$
$\langle proof \rangle$

hide-const (**open**) *DETREFe*
end

Chapter 3

Userguide

3.1 Introduction

The Isabelle/HOL refinement framework is a library that supports program and data refinement.

Programs are specified using a nondeterminism monad: An element of the monad type is either a set of results, or the special element *FAIL*, that indicates a failed assertion.

The bind-operation of the monad applies a function to all elements of the result-set, and joins all possible results.

On the monad type, an ordering $\leq$ is defined, that is lifted subset ordering, where *FAIL* is the greatest element. Intuitively, $S \leq S'$ means that program S refines program S', i.e., all results of S are also results of S', and S may only fail if S' also fails.

3.2 Guided Tour

In this section, we provide a small example program development in our framework. All steps of the development are heavily commented.

3.2.1 Defining Programs

A program is defined using the Haskell-like do-notation, that is provided by the Isabelle/HOL library. We start with a simple example, that iterates over a set of numbers, and computes the maximum value and the sum of all elements.

definition *sum-max* :: *nat set* ⇒ (*nat*×*nat*) *nres* **where**
  sum-max $V \equiv$ *do* {
    (-,s,m) ← *WHILE* (λ(V,s,m). $V \neq \{\}$) (λ(V,s,m). *do* {
      x←*SPEC* ($\lambda x.\ x \in V$);

```
    let V=V-{x};
    let s=s+x;
    let m=max m x;
    RETURN (V,s,m)
  }) (V,0,0);
  RETURN (s,m)
}
```

The type of the nondeterminism monad is $'a\ nres$, where $'a$ is the type of the results. Note that this program has only one possible result, however, the order in which we iterate over the elements of the set is unspecified.

This program uses the following statements provided by our framework: While-loops, bindings, return, and specification. We briefly explain the statements here. A complete reference can be found in Section 3.5.1.

A while-loop has the form $\mathit{WHILE}\ b\ f\ \sigma_0$, where b is the continuation condition, f is the loop body, and σ_0 is the initial state. In our case, the state used for the loop is a triple $(V,\ s,\ m)$, where V is the set of remaining elements, s is the sum of the elements seen so far, and m is the maximum of the elements seen so far. The $\mathit{WHILE}\ b\ f\ \sigma_0$ construct describes a partially correct loop, i.e., it describes only those results that can be reached by finitely many iterations, and ignores infinite paths of the loop. In order to prove total correctness, the construct $\mathit{WHILE}_T\ b\ f\ \sigma_0$ is used. It fails if there exists an infinite execution of the loop.

A binding $do\ \{x \leftarrow (S_1{::}'a\ nres);\ S_2\}$ nondeterministically chooses a result of S_1, binds it to variable x, and then continues with S_2. If S_1 is FAIL, the bind statement also fails.

The syntactic form $do\ \{\ let\ x{=}V;\ (S{::}'a \Rightarrow 'b\ nres)\}$ assigns the value V to variable x, and continues with S.

The return statement $\mathit{RETURN}\ x$ specifies precisely the result x.

The specification statement $\mathit{SPEC}\ \Phi$ describes all results that satisfy the predicate Φ. This is the source of nondeterminism in programs, as there may be more than one such result. In our case, we describe any element of set V.

Note that these statement are shallowly embedded into Isabelle/HOL, i.e., they are ordinary Isabelle/HOL constants. The main advantage is, that any other construct and datatype from Isabelle/HOL may be used inside programs. In our case, we use Isabelle/HOL's predefined operations on sets and natural numbers. Another advantage is that extending the framework with new commands becomes fairly easy.

3.2.2 Proving Programs Correct

The next step in the program development is to prove the program correct w.r.t. a specification. In refinement notion, we have to prove that the program S refines a specification Φ if the precondition Ψ holds, i.e., $\Psi \Longrightarrow S \leq \mathit{SPEC}\ \Phi$.

For our purposes, we prove that *sum-max* really computes the sum and the maximum.

As usual, we have to think of a loop invariant first. In our case, this is rather straightforward. The main complication is introduced by the partially defined *Max*-operator of the Isabelle/HOL standard library.

definition $\mathit{sum\text{-}max\text{-}invar}\ V_0 \equiv \lambda(V,s{::}\mathit{nat},m).$
$\quad V \subseteq V_0$
$\quad \wedge\ s = \sum (V_0 - V)$
$\quad \wedge\ m = (\mathit{if}\ (V_0 - V) = \{\}\ \mathit{then}\ 0\ \mathit{else}\ \mathit{Max}\ (V_0 - V))$
$\quad \wedge\ \mathit{finite}\ (V_0 - V)$

We have extracted the most complex verification condition — that the invariant is preserved by the loop body — to an own lemma. For complex proofs, it is always a good idea to do that, as it makes the proof more readable.

lemma *sum-max-invar-step*:
assumes $x \in V \quad \mathit{sum\text{-}max\text{-}invar}\ V_0\ (V,s,m)$
shows $\mathit{sum\text{-}max\text{-}invar}\ V_0\ (V - \{x\}, s+x, \mathit{max}\ m\ x)\langle \mathit{proof}\rangle$

The correctness is now proved by first invoking the verification condition generator, and then discharging the verification conditions by *auto*. Note that we have to apply the *sum-max-invar-step* lemma, *before* we unfold the definition of the invariant to discharge the remaining verification conditions.

theorem *sum-max-correct*:
assumes *PRE*: $V \neq \{\}$
shows $\mathit{sum\text{-}max}\ V \leq \mathit{SPEC}\ (\lambda(s,m).\ s = \sum V \wedge m = \mathit{Max}\ V)\langle \mathit{proof}\rangle$

In this proof, we specified the invariant explicitely. Alternatively, we may annotate the invariant at the while loop, using the syntax $\mathit{WHILE}^I\ b\ f\ \sigma_0$. Then, the verification condition generator will use the annotated invariant automatically.

Total Correctness Now, we reformulate our program to use a total correct while loop, and annotate the invariant at the loop. The invariant is strengthened by stating that the set of elements is finite.

definition $\mathit{sum\text{-}max'\text{-}invar}\ V_0\ \sigma \equiv$
$\quad \mathit{sum\text{-}max\text{-}invar}\ V_0\ \sigma$
$\quad \wedge\ (\mathit{let}\ (V,\text{-},\text{-}) = \sigma\ \mathit{in}\ \mathit{finite}\ (V_0 - V))$

definition $\mathit{sum\text{-}max'} :: \mathit{nat}\ \mathit{set} \Rightarrow (\mathit{nat} \times \mathit{nat})\ \mathit{nres}$ **where**
$\quad \mathit{sum\text{-}max'}\ V \equiv \mathit{do}\ \{$
$\quad\quad (\text{-},s,m) \leftarrow \mathit{WHILE_T}^{\mathit{sum\text{-}max'\text{-}invar}\ V}\ (\lambda(V,s,m).\ V \neq \{\})\ (\lambda(V,s,m).\ \mathit{do}\ \{$

```
    x←SPEC (λx. x∈V);
    let V=V−{x};
    let s=s+x;
    let m=max m x;
    RETURN (V,s,m)
  }) (V,0,0);
  RETURN (s,m)
}
```

theorem *sum-max′-correct*:
  assumes *NE*: $V \neq \{\}$ **and** *FIN*: *finite V*
  shows *sum-max′ V* $\leq$ *SPEC* $(\lambda(s,m).\ s=\sum V \wedge m=Max\ V)$
  ⟨*proof*⟩

3.2.3 Refinement

The next step in the program development is to refine the initial program towards an executable program. This usually involves both, program refinement and data refinement. Program refinement means changing the structure of the program. Usually, some specification statements are replaced by more concrete implementations. Data refinement means changing the used data types towards implementable data types.

In our example, we implement the set V with a distinct list, and replace the specification statement *SPEC* $(\lambda x.\ x \in V)$ by the head operation on distinct lists. For the lists, we use the list-set data structure provided by the Isabelle Collection Framework [11, 13].

For this example, we write the refined program ourselves. An automation of this task can be achieved with the automatic refinement tool, which is available as a prototype in Refine-Autoref. Usage examples are in ex/Automatic-Refinement.

definition *sum-max-impl* :: *nat ls* ⇒ (*nat*×*nat*) *nres* **where**

```
  sum-max-impl V ≡ do {
    (-,s,m) ← WHILE (λ(V,s,m). ¬ls-isEmpty V) (λ(V,s,m). do {
      x←RETURN (the (ls-sel′ V (λx. True)));
      let V=ls-delete x V;
      let s=s+x;
      let m=max m x;
      RETURN (V,s,m)
    }) (V,0,0);
    RETURN (s,m)
  }
```

Note that we replaced the operations on sets by the respective operations on lists (with the naming scheme *ls-xxx*). The specification statement was replaced by *the* (*ls-sel′ V* $(\lambda x.$ *True*)), i.e., selection of an element that satisfies the predicate $\lambda x.$ *True*. As *ls-sel′* returns an option datatype, we extract the value with *the*. Moreover, we omitted the loop invariant, as we don't need it any more.

Next, we have to show that our concrete pogram actually refines the abstract one.

theorem *sum-max-impl-refine*:
  assumes $(V, V') \in$ *build-rel ls-α ls-invar*
  shows *sum-max-impl* $V \le \Downarrow Id$ (*sum-max* V')⟨*proof*⟩

Refinement is transitive, so it is easy to show that the concrete program meets the specification.

theorem *sum-max-impl-correct*:
  assumes $(V, V') \in$ *build-rel ls-α ls-invar* **and** $V' \neq \{\}$
  shows *sum-max-impl* $V \le SPEC\ (\lambda(s,m).\ s = \sum V' \wedge m = Max\ V')$
⟨*proof*⟩

Just for completeness, we also refine the total correct program in the same way.

definition *sum-max′-impl* :: *nat ls* ⇒ (*nat*×*nat*) *nres* **where**
  sum-max′-impl V ≡ *do* {
    (-,*s*,*m*) ← $WHILE_T$ (λ(*V*,*s*,*m*). ¬*ls-isEmpty V*) (λ(*V*,*s*,*m*). *do* {
      x←*RETURN* (*the* (*ls-sel′ V* (λ*x*. *True*)));
      let V=*ls-delete x V*;
      let s=*s*+*x*;
      let m=*max m x*;
      RETURN (*V*,*s*,*m*)
    }) (*V*,*0*,*0*);
    RETURN (*s*,*m*)
  }

theorem *sum-max′-impl-refine*:
  $(V, V') \in$ *build-rel ls-α ls-invar* ⟹ *sum-max′-impl* $V \le \Downarrow Id$ (*sum-max′* V')
  ⟨*proof*⟩

theorem *sum-max′-impl-correct*:
  assumes $(V, V') \in$ *build-rel ls-α ls-invar* **and** $V' \neq \{\}$
  shows *sum-max′-impl* $V \le SPEC\ (\lambda(s,m).\ s = \sum V' \wedge m = Max\ V')$
  ⟨*proof*⟩

3.2.4 Code Generation

In order to generate code from the above definitions, we convert the function defined in our monad to an ordinary, deterministic function, for that the Isabelle/HOL code generator can generate code.

For partial correct algorithms, we can generate code inside a deterministic result monad. The domain of this monad is a flat complete lattice, where top means a failed assertion and bottom means nontermination. (Note that executing a function in this monad will never return bottom, but just diverge). The construct *nres-of x* embeds the deterministic into the nondeterministic monad.

Thus, we have to construct a function *?sum-max-code* such that:

schematic-lemma *sum-max-code-aux*: *nres-of ?sum-max-code* $\leq$ *sum-max-impl V* ⟨*proof*⟩

This generated the function as a lemma. In order to define it, in our (prototype) framework, we have to copy the function and make a definition of it. We use the output of the following command:

```
thm sum-max-code-aux[no-vars]
definition sum-max-code :: nat ls ⇒ (nat×nat) dres  where
  sum-max-code V ≡
 (dWHILE (λ(V, s, m). ¬ ls-isEmpty V)
   (λ(a, b).
       case b of
       (aa, ba) ⇒
         dRETURN (the (ls-sel' a (λx. True))) ≫=
         (λxa. let xb = ls-delete xa a; xc = aa + xa; xd = max ba xa
               in dRETURN (xb, xc, xd)))
   (V, 0, 0) ≫=
  (λ(a, b). case b of (aa, ba) ⇒ dRETURN (aa, ba)))
```

A simple folding gives us the desired refinement lemma

theorem *sum-max-code-refine*: *nres-of* (*sum-max-code V*) $\leq$ *sum-max-impl V* ⟨*proof*⟩

Finally, we can prove a correctness statement that is independent from our refinement framework:

theorem *sum-max-code-correct*:
  assumes *ls-α V* $\neq \{\}$
  shows *sum-max-code V* = *dRETURN* $(s,m) \Longrightarrow s=\sum$(*ls-α V*) $\wedge$ *m=Max* (*ls-α V*)
    and *sum-max-code V* $\neq$ *dFAIL*⟨*proof*⟩

For total correctness, the approach is the same. The only difference is, that we use *RETURN* instead of *nres-of*:

schematic-lemma *sum-max′-code-aux*:
  RETURN ?sum-max′-code $\leq$ *sum-max′-impl V*
  ⟨*proof*⟩

```
thm sum-max'-code-aux[no-vars]
definition sum-max'-code :: nat ls ⇒ (nat×nat)  where
  sum-max'-code V ≡
 (let (a, b) =
```

```
    while (λ(V, s, m). ¬ ls-isEmpty V)
     (λ(a, b).
         case b of
         (aa, ba) ⇒
           let xa = the (ls-sel' a (λx. True)); xb = ls-delete xa a;
               xc = aa + xa; xd = max ba xa
           in (xb, xc, xd))
     (V, 0, 0)
  in case b of (aa, ba) ⇒ (aa, ba))
```

theorem *sum-max′-code-refine*: *RETURN* (*sum-max′-code V*) ≤ *sum-max′-impl V*
⟨*proof*⟩

theorem *sum-max′-code-correct*:
⟦*ls-α V* ≠ {}⟧ ⟹ *sum-max′-code V* = (∑ (*ls-α V*), *Max* (*ls-α V*))
⟨*proof*⟩

If we use recursion combinators, a plain function can only be generated, if the recursion combinators can be defined. Alternatively, for total correct programs, we may generate a (plain) function that internally uses the deterministic monad, and then extracts the result.

schematic-lemma *sum-max″-code-aux*:
RETURN ?sum-max″-code ≤ *sum-max′-impl V*
⟨*proof*⟩

thm *sum-max″-code-aux*[*no-vars*]
definition *sum-max″-code* :: *nat ls* ⇒ (*nat*×*nat*) **where**

```
  sum-max''-code V ≡
 (the-res
   (dWHILET (λ(V, s, m). ¬ ls-isEmpty V)
     (λ(a, b).
         case b of
         (aa, ba) ⇒
           dRETURN (the (ls-sel' a (λx. True))) ≫=
           (λxa. let xb = ls-delete xa a; xc = aa + xa; xd = max ba xa
                 in dRETURN (xb, xc, xd)))
     (V, 0, 0) ≫=
    (λ(a, b). case b of (aa, ba) ⇒ dRETURN (aa, ba))))
```

theorem *sum-max″-code-refine*: *RETURN* (*sum-max″-code V*) ≤ *sum-max′-impl V*
⟨*proof*⟩

theorem *sum-max″-code-correct*:
⟦*ls-α V* ≠ {}⟧ ⟹ *sum-max″-code V* = (∑ (*ls-α V*), *Max* (*ls-α V*))
⟨*proof*⟩

Now, we can generate verified code with the Isabelle/HOL code generator:

export-code *sum-max-code sum-max′-code sum-max″-code* **in** *SML* **file** –
export-code *sum-max-code sum-max′-code sum-max″-code* **in** *OCaml* **file** –
export-code *sum-max-code sum-max′-code sum-max″-code* **in** *Haskell* **file** –
export-code *sum-max-code sum-max′-code sum-max″-code* **in** *Scala* **file** –

3.2.5 Foreach-Loops

In the *sum-max* example above, we used a while-loop to iterate over the elements of a set. As this pattern is used commonly, there is an abbreviation for it in the refinement framework. The construct $FOREACH\ S\ f\ \sigma_0$ iterates $f{::}'x{\Rightarrow}'s{\Rightarrow}'s$ for each element in $S{::}'x\ set$, starting with state $\sigma_0{::}'s$.

With foreach-loops, we could have written our example as follows:

definition *sum-max-it* :: *nat set* ⇒ (*nat*×*nat*) *nres* **where**
sum-max-it V ≡ *FOREACH V* (λ*x* (*s*,*m*). *RETURN* (*s*+*x*,*max m x*)) (*0*,*0*)

theorem *sum-max-it-correct*:
assumes *PRE*: *V*≠{} **and** *FIN*: *finite V*
shows *sum-max-it V* ≤ *SPEC* (λ(*s*,*m*). *s*=∑ *V* ∧ *m*=*Max V*)
⟨*proof*⟩

definition *sum-max-it-impl* :: *nat ls* ⇒ (*nat*×*nat*) *nres* **where**
sum-max-it-impl V ≡ *FOREACH* (*ls-α V*) (λ*x* (*s*,*m*). *RETURN* (*s*+*x*,*max m x*)) (*0*,*0*)

Note: The nondeterminism for iterators is currently resolved at code-generation phase, where they are replaced by iterators from the ICF.

lemma *sum-max-it-impl-refine*:
notes [*refine*] = *inj-on-id*
assumes (*V*,*V′*)∈*build-rel ls-α ls-invar*
shows *sum-max-it-impl V* ≤ ⇓*Id* (*sum-max-it V′*)
⟨*proof*⟩

schematic-lemma *sum-max-it-code-aux*:
nres-of ?sum-max-it-code ≤ *sum-max-it-impl V*
⟨*proof*⟩

Note that the code generator has replaced the iterator by an iterator from the Isabelle Collection Framework.

thm *sum-max-it-code-aux*[*no-vars*]
definition *sum-max-it-code* :: *nat ls* ⇒ (*nat*×*nat*) *dres* **where**
sum-max-it-code V ≡
(*IT-tag ls-iteratei V* (*dres-case True True* (λ-. *True*))
(λ*x s*. *s* ≫= (λ(*a*, *b*). *dRETURN* (*a* + *x*, *max b x*))) (*dRETURN* (*0*, *0*)))

theorem *sum-max-it-code-refine*:

nres-of (*sum-max-it-code V*) $\leq$ *sum-max-it-impl V*
⟨*proof*⟩

theorem *sum-max-it-code-correct*:
assumes *ls-α V* $\neq$ {}
shows
sum-max-it-code V = *dRETURN* (*s*,*m*) $\Longrightarrow$ *s*=$\sum$(*ls-α V*) $\wedge$ *m*=*Max* (*ls-α V*)
(**is** *?P1* $\Longrightarrow$ *?G1*)
and *sum-max-it-code V* $\neq$ *dFAIL* (**is** *?G2*)
⟨*proof*⟩

export-code *sum-max-it-code* **in** *SML* **file** –
export-code *sum-max-it-code* **in** *OCaml* **file** –
export-code *sum-max-it-code* **in** *Haskell* **file** –
export-code *sum-max-it-code* **in** *Scala* **file** –

3.3 Pointwise Reasoning

In this section, we describe how to use pointwise reasoning to prove refinement statements and other relations between element of the nondeterminism monad.

Pointwise reasoning is often a powerful tool to show refinement between structurally different program fragments.

The refinement framework defines the predicates *nofail* and *inres*. *nofail S* states that S does not fail, and *inres S x* states that one possible result of S is x (Note that this includes the case that S fails).

Equality and refinement can be stated using *nofail* and *inres*:

$$(?S = ?S') = (\mathit{nofail}\ ?S = \mathit{nofail}\ ?S' \wedge (\forall x.\ \mathit{inres}\ ?S\ x = \mathit{inres}\ ?S'\ x))$$

$$(?S \leq ?S') = (\mathit{nofail}\ ?S' \longrightarrow \mathit{nofail}\ ?S \wedge (\forall x.\ \mathit{inres}\ ?S\ x \longrightarrow \mathit{inres}\ ?S'\ x))$$

Useful corollaries of this lemma are *pw-leI*, *pw-eqI*, and *pwD*.

Once a refinement has been expressed via nofail/inres, the simplifier can be used to propagate the nofail and inres predicates inwards over the structure of the program. The relevant lemmas are contained in the named theorem collection *refine-pw-simps*.

As an example, we show refinement of two structurally different programs here, both returning some value in a certain range:

lemma *do* { *ASSERT* (*fst p* > *2*); *SPEC* (λ*x*. *x*≤(*2*::*nat*)∗(*fst p* + *snd p*)) }
≤ *do* { *let* (*x*,*y*)=*p*; *z*←*SPEC* (λ*z*. *z*≤*x*+*y*);
a←*SPEC* (λ*a*. *a*≤*x*+*y*); *ASSERT* (*x*>*2*); *RETURN* (*a*+*z*)}
⟨*proof*⟩

3.4 Arbitrary Recursion (TBD)

Explain *REC* and REC_T.
See examples/Recursion.
To Be Done

3.5 Reference

3.5.1 Statements

SUCCEED The empty set of results. Least element of the refinement ordering.

FAIL Result that indicates a failing assertion. Greatest element of the refinement ordering.

RES X All results from set *X*.

RETURN x Return single result *x*. Defined in terms of *RES*: *RETURN x* = *RETURN x*.

EMBED r Embed partial-correctness option type, i.e., succeed if r=*None*, otherwise return value of *r*.

SPEC Φ Specification. All results that satisfy predicate Φ. Defined in terms of *RES*: *SPEC* Φ = *SPEC* Φ

bind M f Binding. Nondeterministically choose a result from *M* and apply *f* to it. Note that usually the *do*-notation is used, i.e., *do* $\{x \leftarrow M;\ f\ x\}$ or *do* $\{M; f\}$ if the result of *M* is not important. If *M* fails, *bind M f* also fails.

ASSERT Φ Assertion. Fails if Φ does not hold, otherwise returns (). Note that the default usage with the do-notation is: *do* $\{ASSERT\ \Phi;\ f\}$.

ASSUME Φ Assumption. Succeeds if Φ does not hold, otherwise returns (). Note that the default usage with the do-notation is: *do* $\{ASSUME\ \Phi;\ f\}$.

REC body Recursion for partial correctness. May be used to express arbitrary recursion. Returns *SUCCEED* on nontermination.

REC_T *body* Recursion for total correctness. Returns *FAIL* on nontermination.

WHILE b f σ_0 Partial correct while-loop. Start with state σ_0, and repeatedly apply *f* as long as *b* holds for the current state. Non-terminating paths are ignored, i.e., they do not contribute a result.

$WHILE_T\ b\ f\ \sigma_0$ Total correct while-loop. If there is a non-terminating path, the result is *FAIL*.

$WHILE^I\ b\ f\ \sigma_0$, $WHILE_T{}^I\ b\ f\ \sigma_0$ While-loop with annotated invariant. It is asserted that the invariant holds.

$FOREACH\ S\ f\ \sigma_0$ Foreach loop. Start with state σ_0, and transform the state with $f\ x$ for each element $x{\in}S$. Asserts that S is finite.

$FOREACH^I\ S\ f\ \sigma_0$ Foreach-loop with annotated invariant.

Alternative syntax: $FOREACH^I\ S\ f\ \sigma_0$.

The invariant is a predicate of type $I{::}'a\ set \Rightarrow\ 'b \Rightarrow bool$, where $I\ it\ \sigma$ means, that the invariant holds for the remaining set of elements it and current state σ.

$FOREACH_C\ S\ c\ f\ \sigma_0$ Foreach-loop with explicit continuation condition.

Alternative syntax: $FOREACH_C\ S\ c\ f\ \sigma_0$.

If $c{::}'\sigma{\Rightarrow}bool$ becomes false for the current state, the iteration immediately terminates.

$FOREACH_C{}^I\ S\ c\ f\ \sigma_0$ Foreach-loop with explicit continuation condition and annotated invariant.

Alternative syntax: $FOREACH_C{}^I\ S\ c\ f\ \sigma_0$.

partial-function (*nrec*) Mode of the partial function package for the nondeterminism monad.

3.5.2 Refinement

$op \le{::}'a\ nres \Rightarrow\ 'a\ nres \Rightarrow bool$ Refinement ordering. $S \le S'$ means, that every result in S is also a result in S'. Moreover, S may only fail if S' fails. $\le$ forms a complete lattice, with least element *SUCCEED* and greatest element *FAIL*.

$\Downarrow R$ Concretization. Takes a refinement relation $R{::}('c{\times}'a)\ set$ that relates concrete to abstract values, and returns a concretization function $\Downarrow R$.

$\Uparrow R$ Abstraction. Takes a refinement relation and returns an abstraction function. The functions $\Downarrow R$ and $\Uparrow R$ form a Galois-connection, i.e., we have: $S \le \Downarrow R\ S' \longleftrightarrow \Uparrow R\ S \le S'$.

$br\ \alpha\ I$ Builds a refinement relation from an abstraction function and an invariant. Those refinement relations are always single-valued.

$nofail\ S$ Predicate that states that S does not fail.

inres S x Predicate that states that S includes result x. Note that a failing program includes all results.

3.5.3 Proof Tools

Verification Condition Generator:

Method: *intro refine-vcg*

Attributes: *refine-vcg*

Transforms a subgoal of the form $S \leq SPEC\ \Phi$ into verification conditions by decomposing the structure of S. Invariants for loops without annotation must be specified explicitely by instantiating the respective proof-rule for the loop construct, e.g., *intro WHILE-rule*[*where I*=...] *refine-vcg*.

refine-vcg is a named theorems collection that contains the rules that are used by default.

Refinement Condition Generator:

Method: *refine-rcg* [thms].

Attributes: *refine0*, *refine*, *refine2*.

Flags: *refine-no-prod-split*.

Tries to prove a subgoal of the form $S \leq \Downarrow R\ S'$ by decomposing the structure of S and S'. The rules to be used are contained in the theorem collection *refine*. More rules may be passed as argument to the method. Rules contained in *refine0* are always tried first, and rules in *refine2* are tried last. Usually, rules that decompose both programs equally should be put into *refine*. Rules that may make big steps, without decomposing the program further, should be put into *refine0* (e.g., *Id-refine*). Rules that decompose the programs differently and shall be used as last resort before giving up should be put into *refine2*, e.g., *remove-Let-refine*.

By default, this procedure will invoke the splitter to split product types in the goals. This behaviour can be disabled by setting the flag *refine-no-prod-split*.

Refinement Relation Heuristics:

Method: *refine-dref-type* [(trace)].

Attributes: *refine-dref-RELATES*, *refine-dref-pattern*.

Flags: *refine-dref-tracing*.

Tries to instantiate schematic refinement relations based on their type. By default, this rule is applied to all subgoals. Internally, it uses the rules declared as *refine-dref-pattern* to introduce a goal of the form *RELATES ?R*, that is then solved by exhaustively applying rules declared as *refine-dref-RELATES*.

The flag *refine-dref-tracing* controls tracing of resolving *RELATES*-goals. Tracing may also be enabled by passing (trace) as argument.

Pointwise Reasoning Simplification Rules:

Attributes: *refine-pw-simps*

A theorem collection that contains simplification lemmas to push inwards *nofail* and *inres* predicates into program constructs.

Refinement Simp Rules:

Attributes: *refine-hsimp*

A theorem collection that contains some simplification lemmas that are useful to prove membership in refinement relations.

Transfer:

Method: *refine-transfer* [thms]
Attribute: *refine-transfer*

Tries to prove a subgoal of the form $\alpha\ f \leq S$ by decomposing the structure of f and S. This is usually used in connection with a schematic lemma, to generate f from the structure of S.

The theorems declared as *refine-transfer* are used to do the transfer. More theorems may be passed as arguments to the method. Moreover, some simplification for nested abstraction over product types ($\lambda(a,b)$ $(c,d).\ \ldots$) is done, and the monotonicity prover is used on monotonicity goals.

There is a standard setup for α=*RETURN* (transfer to plain function for total correct code generation), and α=*nres-of* (transfer to deterministic result monad, for partial correct code generation).

Automatic Refinement:

Method: *refine-autoref* [(trace)] [(ss)] [thms]
Attributes: ...

Prototype method for automatic data refinement. Works well for simple examples. See ex/Automatic-Refinement for examples and preliminary documentation.

3.5.4 Packages

The following parts of the refinement framework are not included by default, but can be imported if necessary:

Collection-Bindings: Sets up refinement rules for the Isabelle Collection Framework. With this theory loaded, the refinement condition generator will discharge most data refinements using the ICF automatically. Moreover, the transfer procedure will replace *FOREACH*-statements by the corresponding ICF-iterators.

Autoref-Collection-Bindings: Automatic refinement for ICF data structures. Almost complete for sets, unique priority queues. Partial setup for maps.

end

Chapter 4

Examples

4.1 Breadth First Search

theory *Breadth-First-Search*
imports *../Refine*
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

4.1.1 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 $\mathit{min\text{-}dist}\ v\ v' = (\mathit{LEAST}\ d.\ \mathit{dist}\ v\ d\ v')$
definition $\mathit{conn}\ v\ v' = (\exists\, d.\ \mathit{dist}\ v\ d\ v')$

Properties

In this subsection, we prove some properties of paths.

lemma
shows $\mathit{connI}[\mathit{intro}]$: $\mathit{dist}\ v\ d\ v' \Longrightarrow \mathit{conn}\ v\ v'$
and $\mathit{connI\text{-}id}[\mathit{intro}]$: $\mathit{conn}\ v\ v$
and $\mathit{connI\text{-}succ}[\mathit{intro}]$: $\mathit{conn}\ v\ v' \Longrightarrow v'' \in \mathit{succ}\ v' \Longrightarrow \mathit{conn}\ v\ v''$
$\langle \mathit{proof} \rangle$

lemma $\mathit{min\text{-}distI2}$:
$[\![\ \mathit{conn}\ v\ v'\ ;\ \bigwedge d.\ [\![\ \mathit{dist}\ v\ d\ v';\ \bigwedge d'.\ \mathit{dist}\ v\ d'\ v' \Longrightarrow d \le d'\ ]\!] \Longrightarrow Q\ d\ ]\!]$
$\Longrightarrow Q\ (\mathit{min\text{-}dist}\ v\ v')$
$\langle \mathit{proof} \rangle$

lemma $\mathit{min\text{-}distI\text{-}eq}$:
$[\![\ \mathit{dist}\ v\ d\ v';\ \bigwedge d'.\ \mathit{dist}\ v\ d'\ v' \Longrightarrow d \le d'\ ]\!] \Longrightarrow \mathit{min\text{-}dist}\ v\ v' = d$
$\langle \mathit{proof} \rangle$

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

lemma $\mathit{dist\text{-}z\text{-}iff}[\mathit{simp}]$: $\mathit{dist}\ v\ 0\ v' \longleftrightarrow v'{=}v$
$\langle \mathit{proof} \rangle$

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

lemma $\mathit{min\text{-}dist\text{-}z}[\mathit{simp}]$: $\mathit{min\text{-}dist}\ v\ v = 0$
$\langle \mathit{proof} \rangle$

lemma $\mathit{min\text{-}dist\text{-}z\text{-}iff}[\mathit{simp}]$: $\mathit{conn}\ v\ v' \Longrightarrow \mathit{min\text{-}dist}\ v\ v' = 0 \longleftrightarrow v'{=}v$
$\langle \mathit{proof} \rangle$

lemma $\mathit{min\text{-}dist\text{-}is\text{-}dist}$: $\mathit{conn}\ v\ v' \Longrightarrow \mathit{dist}\ v\ (\mathit{min\text{-}dist}\ v\ v')\ v'$
$\langle \mathit{proof} \rangle$
lemma $\mathit{min\text{-}dist\text{-}minD}$: $\mathit{dist}\ v\ d\ v' \Longrightarrow \mathit{min\text{-}dist}\ v\ v' \le d$
$\langle \mathit{proof} \rangle$

We also provide introduction and destruction rules for the pattern $\mathit{min\text{-}dist}\ v\ v' = \mathit{Suc}\ d$.

lemma $\mathit{min\text{-}dist\text{-}succ}$:
$[\![\ \mathit{conn}\ v\ v';\ v'' \in \mathit{succ}\ v'\ ]\!] \Longrightarrow \mathit{min\text{-}dist}\ v\ v'' \le \mathit{Suc}\ (\mathit{min\text{-}dist}\ v\ v')$
$\langle \mathit{proof} \rangle$

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

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

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

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

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

4.1.2 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*$) \wedge$

$(\forall v{\in}V.\ \mathit{conn\ src\ v} \wedge \mathit{min\text{-}dist\ src\ v} \leq \mathit{Suc\ d}) \wedge$
$(\forall v.\ \mathit{conn\ src\ v} \wedge \mathit{min\text{-}dist\ src\ v} \leq d \longrightarrow v \in V) \wedge$
$(\forall v.\ \mathit{conn\ src\ v} \wedge \mathit{min\text{-}dist\ src\ v} = \mathit{Suc\ d} \longrightarrow v \in N \cup ((\bigcup \mathit{succ}`C) - V))$
$)) \wedge ($
$f \longrightarrow \mathit{conn\ src\ dst} \wedge \mathit{min\text{-}dist\ src\ dst} = d$
$)$

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

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

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

definition $\mathit{bfs\text{-}invar\ src\ dst}\ \sigma \equiv \mathit{bfs\text{-}invar}'\ \mathit{src\ dst}\ \sigma \wedge$
$\mathit{empty\text{-}assm}\ \sigma$

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

definition $\mathit{FE\text{-}invar}\ V\ N\ v\ \mathit{it}\ \sigma \equiv \mathit{let}\ (V',N'){=}\sigma\ \mathit{in}$
$V'{=}V \cup (\mathit{succ}\ v - \mathit{it}) \wedge$
$N'{=}N \cup ((\mathit{succ}\ v - \mathit{it}) - V)$

4.1.3 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 $\mathit{WHILEI}\ I\ c\ f\ \sigma_0$, where σ_0 is the initial state, and I is the loop invariant;

- We translated the textual specification *remove one vertex v from C* as accurately as possible: The statement $v \leftarrow \mathit{SPEC}\ (\lambda v.\ v{\in}C)$ 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* (λ*v*. *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*) (λ*w* (*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*
}

4.1.4 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 $bfs\text{-}invar'\ src\ dst\ (False,\ V,\ \{\},\ N,\ d)$
shows $bfs\text{-}invar'\ src\ dst\ (False,\ V,\ N,\ \{\},\ Suc\ d)$
⟨*proof*⟩

The invariant holds initially.

lemma *invar-init*: $bfs\text{-}invar\ src\ dst\ (False,\ \{src\},\ \{src\},\ \{\},\ 0)$
⟨*proof*⟩

The invariant is preserved if we break the loop.

lemma *invar-break*:
assumes $bfs\text{-}invar\ src\ dst\ (False,\ V,\ C,\ N,\ d)$
assumes $dst \in C$
shows $bfs\text{-}invar\ src\ dst\ (True,\ \{\},\ \{\},\ \{\},\ d)$
⟨*proof*⟩

If we have *break*ed the loop, the invariant implies that we, indeed, returned the shortest path.

lemma *invar-final-succeed*:
assumes $bfs\text{-}invar'\ src\ dst\ (True,\ V,\ C,\ N,\ d)$
shows $min\text{-}dist\ src\ dst = d$
⟨*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: $bfs\text{-}invar'\ src\ dst\ (False,\ V,\ \{\},\ \{\},\ d)$
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* $\equiv$ *SPEC* (
  $\lambda r.$ *case r of None* $\Rightarrow \neg$ *conn src dst*
          | *Some d* $\Rightarrow$ *conn src dst* $\wedge$ *min-dist src dst* = *d*)
theorem *bfs-correct*: *bfs src dst* $\leq$ *bfs-spec src dst*
  ⟨*proof*⟩

end

end

4.2 Verified BFS Implementation in ML

theory *Bfs-Impl*
imports
  Breadth-First-Search
  ../Collection-Bindings *../Refine*
  ~~/src/HOL/Library/Code-Target-Numeral
begin

Originally, this was part of our submission to the VSTTE 2010 Verification Competition. Some slight changes have been applied since the submitted version.

In the *Breadth-First-Search*-theory, we verified an abstract version of the algorithm. This abstract version tried to reflect the given pseudocode specification as precisely as possible.

However, it was not executable, as it was nondeterministic. Hence, we now refine our algorithm to an executable specification, and use Isabelle/HOLs code generator to generate ML-code.

The implementation uses the Isabelle Collection Framework (ICF) (Available at http://afp.sourceforge.net/entries/Collections.shtml), to provide efficient set implementations. We choose a hashset (backed by a Red Black Tree) for the visited set, and lists for all other sets. Moreover, we fix the node type to natural numbers.

The following algorithm is a straightforward rewriting of the original algorithm. We only exchanged the abstract set operations by concrete operations on the data structures provided by the ICF.

The operations of the list-set implementation are named *ls-xxx*, the ones of the hashset are named *hs-xxx*.

```
definition bfs-impl :: (nat ⇒ nat ls) ⇒ nat ⇒ nat ⇒ (nat option nres)
  where bfs-impl succ src dst ≡ do {
  (f,-,-,-,d) ← WHILE
    (λ(f,V,C,N,d). f=False ∧ ¬ ls-isEmpty C)
    (λ(f,V,C,N,d). do {
      v ← RETURN (the (ls-sel′ C (λ-. True))); let C = ls-delete v C;
      if v=dst then RETURN (True,hs-empty (),ls-empty (),ls-empty (),d)
      else do {
        (V,N) ← FOREACH (ls-α (succ v)) (λw (V,N).
          if (¬ hs-memb w V) then
             RETURN (hs-ins w V, ls-ins-dj w N)
          else RETURN (V,N)
        ) (V,N);
        if (ls-isEmpty C) then do {
          let C=N;
          let N=ls-empty ();
          let d=d+1;
          RETURN (f,V,C,N,d)
        } else RETURN (f,V,C,N,d)
      }
    })
    (False,hs-sng src,ls-sng src, ls-empty (),0::nat);
  if f then RETURN (Some d) else RETURN None
  }
```

Auxilliary lemma to initialize the refinement prover.

It is quite easy to show that the implementation respects the specification, as most work is done by the refinement framework.

```
theorem bfs-impl-correct:
  shows bfs-impl succ src dst ≤ Graph.bfs-spec (ls-α∘succ) src dst
⟨proof⟩
```

The last step is to actually generate executable ML-code.

We first use the partial-correctness code generator of our framework to automatically turn the algorithm described in our framework into a function that is independent from our framework. This step also removes the last nondeterminism, that has remained in the iteration order of the inner loop.

The result of the function is an option type, returning *None* for nontermination. Inside this option-type, there is the option type that encodes whether we return with failure or a distance.

```
schematic-lemma bfs-code-refine-aux:
```

nres-of ?bfs-code $\leq$ *bfs-impl succ src dst*
⟨*proof*⟩

thm *bfs-code-refine-aux*[*no-vars*]

Currently, our (prototype) framework cannot yet generate the definition itself. The following definition was copy-pasted from the output of the above *thm* command:

```
definition
  bfs-code :: (nat ⇒ nat ls) ⇒ nat ⇒ nat ⇒ nat option dres where
  bfs-code succ src dst ≡
(dWHILE (λ(f, V, C, N, d). f = False ∧ ¬ ls-isEmpty C)
 (λ(a, b).
    case b of
    (aa, ab, ac, bc) ⇒
      dRETURN (the (ls-sel′ ab (λ-. True))) ≫=
      (λxa. let xb = ls-delete xa ab
            in if xa = dst
               then dRETURN
                     (True, hs-empty (), ls-empty (), ls-empty (), bc)
               else IT-tag ls-iteratei (succ xa)
                     (dres-case True True (λ-. True))
                     (λx s. s ≫=
                            (λ(ad, bd).
                                if ¬ hs-memb x ad
                                then dRETURN
 (hs-ins x ad, ls-ins-dj x bd)
                                else dRETURN (ad, bd)))
                     (dRETURN (aa, ac)) ≫=
                    (λ(ad, bd).
                        if ls-isEmpty xb
                        then let xd = bd; xe = ls-empty (); xf = bc + 1
                             in dRETURN (a, ad, xd, xe, xf)
                        else dRETURN (a, ad, xb, bd, bc))))
 (False, hs-sng src, ls-sng src, ls-empty (), 0) ≫=
(λ(a, b).
    case b of
    (aa, ab, ac, bc) ⇒ if a then dRETURN (Some bc) else dRETURN None))
```

Finally, we get the desired lemma by folding the schematic lemma with the definition:

lemma *bfs-code-refine*:
nres-of (*bfs-code succ src dst*) $\leq$ *bfs-impl succ src dst*
⟨*proof*⟩

As a last step, we make the correctness property independent of our refinement framework. This step drastically decreases the trusted code base, as it completely eliminates the specifications made in the refinement framework from the trusted code base.

The following theorem solves both verification tasks, without depending on any concepts of the refinement framework, except the deterministic result monad.

theorem *bfs-code-correct*:
  $bfs\text{-}code\ succ\ src\ dst = dRETURN\ None$
    $\Longrightarrow \neg(Graph.conn\ (ls\text{-}\alpha \circ succ)\ src\ dst)$
  $bfs\text{-}code\ succ\ src\ dst = dRETURN\ (Some\ d)$
    $\Longrightarrow Graph.conn\ (ls\text{-}\alpha \circ succ)\ src\ dst$
      $\wedge\ Graph.min\text{-}dist\ (ls\text{-}\alpha \circ succ)\ src\ dst = d$
  $bfs\text{-}code\ succ\ src\ dst \neq dFAIL$
$\langle proof \rangle$

Now we can use the code-generator of Isabelle/HOL to generate code into various target languages:

export-code *bfs-code* **in** *SML* **file** −
export-code *bfs-code* **in** *OCaml* **file** −
export-code *bfs-code* **in** *Haskell* **file** −
export-code *bfs-code* **in** *Scala* **file** −

The generated code is most conveniently executed within Isabelle/HOL itself. We use a small test graph here:

definition *nat-list*:: *nat list* ⇒ *nat dlist*
where
  nat-list ≡ *dlist-of-list*

definition *succ* :: *nat* ⇒ *nat dlist*
where
  $succ\ n = nat\text{-}list\ (if\ n = 1\ then\ [2,\ 3]$
    $else\ if\ n = 2\ then\ [4]$
    $else\ if\ n = 4\ then\ [5]$
    $else\ if\ n = 5\ then\ [2]$
    $else\ if\ n = 3\ then\ [6]$
    $else\ [])$

definition *bfs-code-succ* :: *integer* ⇒ *integer* ⇒ *nat option dres*
where
  $bfs\text{-}code\text{-}succ\ m\ n = bfs\text{-}code\ succ\ (nat\text{-}of\text{-}integer\ m)\ (nat\text{-}of\text{-}integer\ n)$

$\langle ML \rangle$

end

4.3 Machine Words

theory *WordRefine*
imports *../Refine* *~~/src/HOL/Word/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.

4.3.1 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*]: *single-valued word-nat-rel* ⟨*proof*⟩

lemma [*simp*]: *single-valuedP* (λ*c a*. *a* = *unat c*)
  ⟨*proof*⟩

lemma [*simp*]: *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*]

4.3.2 Example

type-synonym *word32* = *32 word*

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

lemma *y0*>*0* ⟹ *test x0 y0* ≤ *SPEC* (λ*S*. *S*={*y0* .. *y0* + *x0* − *1*})
  — Choosen pre-condition to get least trouble when proving
  ⟨*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 $x'+y'<2\hat{}len\text{-}of\ TYPE(32)$
  assumes $(x,x')\in word\text{-}nat\text{-}rel$
  assumes $(y,y')\in word\text{-}nat\text{-}rel$
  shows $test\text{-}impl\ x\ y \le \Downarrow(map\text{-}set\text{-}rel\ word\text{-}nat\text{-}rel)\ (test\ x'\ y')$
⟨*proof*⟩

end

4.4 Fold-Combinator

theory *Refine-Fold*
imports *../Refine* *../Collection-Bindings*
begin

In this theory, we explore the usage of the partial-function package, and define a function with a higher-order argument. As example, we choose a nondeterministic fold-operation on lists.

```
partial-function (nrec) rfoldl where
  rfoldl f s l = (case l of
      [] ⇒ RETURN s
    | x#ls ⇒ do { s←f s x; rfoldl f s ls}
  )
```

Currently, we have to manually state the standard simplification lemmas:

```
lemma rfoldl-simps[simp]:
  rfoldl f s [] = RETURN s
  rfoldl f s (x#ls) = do { s←f s x; rfoldl f s ls}
  ⟨proof⟩
```

lemma *rfoldl-refines*[*refine*]:
  assumes *SV*: *single-valued Rs*
  assumes *REFF*: $\bigwedge x\ x'\ s\ s'.\ [\![\ (s,s')\in Rs;\ (x,x')\in Rl\ ]\!]$
    $\Longrightarrow f\ s\ x \le \Downarrow Rs\ (f'\ s'\ x')$
  assumes *REF0*: $(s0,s0')\in Rs$
  assumes *REFL*: $(l,l')\in map\text{-}list\text{-}rel\ Rl$
  shows $rfoldl\ f\ s0\ l \le \Downarrow Rs\ (rfoldl\ f'\ s0'\ l')$

⟨*proof*⟩

lemma *transfer-rfoldl*[*refine-transfer*]:
assumes $\bigwedge s\ x.\ RETURN\ (f\ s\ x) \leq F\ s\ x$
shows $RETURN\ (foldl\ f\ s\ l) \leq rfoldl\ F\ s\ l$
⟨*proof*⟩

4.4.1 Example

As example application, we define a program that takes as input a list of non-empty sets of natural numbers, picks some number of each list, and adds up the picked numbers.

definition *pick-sum* (*s0*::*nat*) *l* ≡
rfoldl (λ*s x*. *do* {
ASSERT (*x*≠{});
y←*SPEC* (λ*y*. *y*∈*x*);
RETURN (*s*+*y*)
}) *s0 l*

lemma [*simp*]:
pick-sum s [] = *RETURN s*
pick-sum s (*x*#*l*) = *do* {
ASSERT (*x*≠{}); *y*←*SPEC* (λ*y*. *y*∈*x*); *pick-sum* (*s*+*y*) *l*
}
⟨*proof*⟩

lemma *foldl-mono*:
assumes $\bigwedge x.\ mono\ (\lambda s.\ f\ s\ x)$
shows $mono\ (\lambda s.\ foldl\ f\ s\ l)$
⟨*proof*⟩

lemma *pick-sum-correct*:
assumes *NE*: {}∉*set l*
assumes *FIN*: ∀ *x*∈*set l*. *finite x*
shows *pick-sum s0 l* ≤ *SPEC* (λ*s*. *s* ≤ *foldl* (λ*s x*. *s*+*Max x*) *s0 l*)
⟨*proof*⟩

definition *pick-sum-impl s0 l* ≡
rfoldl (λ*s x*. *do* {
y←*RETURN* (*the* (*ls-sel′ x* (λ-. *True*)));
RETURN (*s*+*y*)
}) (*s0*::*nat*) *l*

lemma *pick-sum-impl-refine*:
assumes *A*: *list-all2* (λ*x x′*. (*x*,*x′*)∈*build-rel ls-α ls-invar*) *l l′*
shows *pick-sum-impl s0 l* ≤ ⇓*Id* (*pick-sum s0 l′*)
⟨*proof*⟩

schematic-lemma *pick-sum-code-aux*: *RETURN ?f* ≤ *pick-sum-impl s0 l*
⟨*proof*⟩

thm *pick-sum-code-aux*[*no-vars*]
definition *pick-sum-code s0 l* ≡
foldl (λ*s x*. *Let* (*the* (*ls-sel′ x* (λ-. *True*))) (*op* + *s*)) (*s0*::*nat*) *l*
lemma *pick-sum-code-refines*:
RETURN (*pick-sum-code s l*) ≤ *pick-sum-impl s l*
⟨*proof*⟩

value
pick-sum-code 0 [*list-to-ls* [*3*,*2*,*1*], *list-to-ls* [*1*,*2*,*3*], *list-to-ls*[*2*,*1*]]

end

4.5 Automatic Refinement Examples

theory *Automatic-Refinement*
imports ../*Refine* ../*Autoref-Collection-Bindings*
begin

This theory demonstrates the usage of the autodet-tool for automatic determinization.

We start with a worklist-algorithm

```
definition exploresl succ s0 st ≡
  do {
    (-,-,f)←WHILET (λ(wl,es,f). wl≠[] ∧ ¬f) (λ(wl,es,f). do {
      let s=hd wl; let wl=tl wl;
      if s∈es then
        RETURN (wl,es,f)
      else if s=st then
        RETURN ([],{},True)
      else
        RETURN (wl @ succ s,insert s es,f)
    }) ([s0],{},False);
    RETURN f
  }
```

The following is a typical approach for usage of the automatic refinement method. The desired refinements are specified by the type, declaring partially instantiated *spec-R*-lemmas as [*autoref-spec*]. Here, *′c* is the concrete type, and *′a* is the abstract type that shall be translated to *′c*.

Note that *spec-Id* is a shortcut for *spec-R* with *′c*=*′a*.

In general, variables that occur in the lemma, e.g., parameters of the function to be refined, will not be translated automatically. Hence, the assumptions of the lemma must specify appropriate relations. In our example, we use the same parameters for the abstract and concrete algorithm.

schematic-lemma
  fixes *s0*::*'V*::*hashable*
  notes [*autoref-spec*] =
    spec-Id[**where** *'a*=*bool*]
    spec-Id[**where** *'a*=*'V*]
    spec-Id[**where** *'a*=*'V list*]
    spec-R[**where** *'c*=*'V hs* **and** *'a*=*'V set*]
  shows ⟦(*succ*,*succ*)∈*Id*; (*s0*,*s0*)∈*Id*; (*st*,*st*)∈*Id*⟧ ⟹
    (*?f*::*?'a nres*) ≤⇓(*?R*) (*exploresl succ s0 st*)
  ⟨*proof*⟩

Unfortunately, Isabelle/HOL has some pitfalls here:

- Make sure that all type variables refered to in a *notes*-section occur in a fixes-section *before* the notes-section. Otherwise, Isabelle forgets to assign the typeclass-constraints to the noted theorems. Alternatively, you may pass the specification theorems as arguments to the autoref-method.
- If you use a schematic refinement relation at the toplevel (*?R* in this example), make sure to also explicitely specify a schematic type for it (*?f*::*?'a nres* in this example). Otherwise, Isabelle just assigns it a fixed type variable that cannot be further instantiated.

Autoref also accept spec-theorems as arguments:

schematic-lemma
  fixes *s0*::*'V*::*hashable*
  shows ⟦(*succ*,*succ*)∈*Id*; (*s0*,*s0*)∈*Id*; (*st*,*st*)∈*Id*⟧ ⟹
    (*?f*::*?'a nres*) ≤⇓(*?R*) (*exploresl succ s0 st*)
  ⟨*proof*⟩

If you forget something, in this case, a specification to translate *bool*, the method *refine-autoref* will stop at the expression that it could not completely translate. Here, single-step mode (*refine-autoref* (*ss*)) helps to identify the problem: It stops after each step applied to the subgoal. Note that it may have result sequences with more than one element. In this case, use *back* to switch through the alternatives.

schematic-lemma
  fixes *s0*::*'V*::*hashable*
  notes [*autoref-spec*] =

    spec-Id[**where** *'a*=*'V*]
    spec-Id[**where** *'a*=*'V list*]
    spec-R[**where** *'c*=*'V hs* **and** *'a*=*'V set*]

```
  shows ⟦(succ,succ)∈Id; (s0,s0)∈Id; (st,st)∈Id⟧ ⟹
    (?f::?'a nres) ≤⇓(?R) (exploresl succ s0 st)
  ⟨proof⟩
```

In the next example, we have two sets of the same type: Both, the workset and the visited set are sets of nodes.

```
definition explores1 succ s0 st ≡
  do {
    (-,-,f)←WHILET (λ(ws,es,f). ws≠{} ∧ ¬f) (λ(ws,es,f). do {
      ASSERT (ws≠{}); s←SPEC (λs. s∈ws);
      let ws=ws−{s};
      let es=es∪{s};
      if s=st then
        RETURN ({},{},True)
      else
        RETURN (ws ∪ (set (succ s) − es),es,f)
    }) ({s0},{},False);
    RETURN f
  }
```

First, we translate both sets to hashsets.

```
schematic-lemma
  fixes s0::'V::hashable
  notes [autoref-spec] =
    spec-Id[where 'a=bool]
    spec-Id[where 'a='V]
    spec-Id[where 'a='V list]
    spec-R[where 'c='V hs and 'a='V set]
  shows ⟦(succ,succ)∈Id; (s0,s0)∈Id; (st,st)∈Id⟧ ⟹
    (?f::?'a nres) ≤⇓(?R) (explores1 succ s0 st)
  ⟨proof⟩
```

As a second alternative, we use tagging to translate the sets to different concrete sets. Here, tags are used to indicate the desired concretization.

```
definition explores1' succ s0 st ≡
  do {
    (-,-,f)←WHILET (λ(ws,es,f). ws≠{} ∧ ¬f) (λ(ws,es,f). do {
      ASSERT (ws≠{}); s←SPEC (λs. s∈ws);
      let ws=ws−{s};
      let es=es∪{s};
      if s=st then
        RETURN ({},{},True)
      else
        RETURN (ws ∪ (set (succ s) − es),es,f)
    }) (TAG ''workset'' {s0},TAG ''visited−set'' {},False);
    RETURN f
  }
```

```
schematic-lemma test:
  fixes s0::'V::hashable
  notes [autoref-spec] =
    spec-Id[where 'a='V]
    spec-Id[where 'a='V list]
    spec-Id[where 'a=bool]
    spec-R[where 'a='V set and R=br ls-α ls-invar]
    spec-R[where 'a='V set and R=br hs-α hs-invar]
    TAG-dest[where name=''workset'' and R=br ls-α ls-invar]
    TAG-dest[where name=''visited−set'' and R=br hs-α hs-invar]

  shows ⟦(succ,succ)∈Id; (s0,s0)∈Id; (st,st)∈Id⟧
    ⟹ (?f::?'a nres) ≤⇓?R (explores1' succ (s0::'V) st)
  ⟨proof⟩

end
```

4.6 Example for Recursion Combinator

```
theory Recursion
imports
  ../Refine
  ../Refine-Autoref    ../Autoref-Collection-Bindings
  ../Collection-Bindings
  ../../Collections/Collections
begin
```

This example presents the usage of the recursion combinator *RECT*. The usage of the partial correct version *REC* is similar.

We define a simple DFS-algorithm, prove a simple correctness property, and do data refinement to an efficient implementation.

4.6.1 Definition

Recursive DFS-Algorithm. *E* is the edge relation of the graph, *vd* the node to search for, and *v0* the start node. Already explored nodes are stored in *V*.

```
definition dfs :: ('a×'a) set ⇒ 'a ⇒ 'a ⇒ bool nres where
  dfs E vd v0 ≡ do {
   REC_T (λD (V,v).
    if v=vd then RETURN True
    else if v∈V then RETURN False
    else do {
      let V=insert v V;
      FOREACH_C (E``{v}) (op = False) (λv' -. D (V,v')) False
    }
```

) ({},*v0*)
}

4.6.2 Correctness

As simple correctness property, we show: If the algorithm returns true, then *vd* is reachable from *v0*.

lemma *dfs-correct*:
fixes *v0 E*
assumes [*simp*,*intro*]: *finite* (E^* ``{*v0*})
shows *dfs E vd v0* $\leq$ *SPEC* ($\lambda r.\ r \longrightarrow (v0,vd) \in E^*$)
⟨*proof*⟩

4.6.3 Data Refinement and Determinization

Next, we use automatic data refinement and transfer to generate an executable algorithm using a hashset. The edges function is refined to a successor function returning a list-set.

schematic-lemma *dfs-impl-refine-aux*:
fixes *v0*::'*v*::*hashable* **and** *succi*
defines $E \equiv \{(v,v').\ v' \in ls\text{-}\alpha\ (succi\ v)\}$
shows *RETURN* (*?f*) $\leq \Downarrow$*Id* (*dfs E vd v0*)
⟨*proof*⟩

Executable Code

In our prototype, the code equations for recursion have to be set up manually, as done below:

thm *dfs-impl-refine-aux*[*no-vars*]

definition *dfs-rec-body succi t* $\equiv$
(λf (*a*, *b*).
if b = *t then dRETURN True*
else if hs-memb b a then dRETURN False
else let xa = *hs-ins-dj b a*
in IT-tag ls-iteratei (*succi b*)
(*dres-case True True* (*op* = *False*))
($\lambda x\ s.\ s \gg= (\lambda s.\ f\ (xa,\ x))$) (*dRETURN False*))

definition *dfs-rec succi t* $\equiv$ REC_T (*dfs-rec-body succi t*)
definition *dfs-impl* **where** *dfs-impl succi t s0* $\equiv$
the-res (*dfs-rec succi t* (*hs-empty* (), *s0*))

Code equation for recursion

lemma [*code*]: *dfs-rec succi t x* = *dfs-rec-body succi t* (*dfs-rec succi t*) *x*

⟨*proof*⟩

We fold the definitions, to get a nice refinement lemma

lemma *dfs-impl-refine*:
RETURN (*dfs-impl succi t s0*)
$\leq$ ⇓*Id* (*dfs* $\{(s,s').\ s' \in$ *ls-*α (*succi s*)$\}$ *t s0*)
⟨*proof*⟩

Combining refinement and the correctness lemma, we get a correctness property of the plain function.

lemma *dfs-impl-correct*:
fixes *succi*
defines $E \equiv \{(s,\ s').\ s' \in$ *ls-*α (*succi s*)$\}$
assumes *F*: *finite* (E^* ‘‘$\{v0\}$)
assumes *R*: *dfs-impl succi vd v0*
shows (*v0*,*vd*)$\in E^*$
⟨*proof*⟩

Finally, we can generate code

export-code *dfs-impl* **in** *SML* **file** −
export-code *dfs-impl* **in** *OCaml* **file** −
export-code *dfs-impl* **in** *Haskell* **file** −
export-code *dfs-impl* **in** *Scala* **file** −

end

Chapter 5

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://afp.sf.net/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://afp.sf.net/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.