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 ⟪
infix 0 THEN-ELSE';
infix 1 THEN-ALL-NEW-FWD;

structure Refine-Misc = struct
  (***********************)
  (* Tacticals *)
  (***********************)

  fun (tac1 THEN-ELSE' (tac2,tac3)) x = tac1 x THEN-ELSE (tac2 x,tac3 x);

  (* Fail if goal index out of bounds. *)
  fun wrap-nogoals tac i st = if nprems-of st < i then
    no-tac st
  else
    tac i st;

  (* Apply tactic to subgoals in interval, in a forward manner, skipping over
    emerging subgoals *)
  exception FWD of string;
  fun INTERVAL-FWD tac l u st =
    if l>u then all-tac st
    else (tac l THEN (fn st' => let
        val ofs = nprems-of st' − nprems-of st;
      in
        if ofs < ~1 then raise FWD (INTERVAL-FWD −− tac solved more than
```

```
one goal)
        else INTERVAL-FWD tac (l+1+ofs) (u+ofs) st'
      end)) st;

  (* Apply tac2 to all subgoals emerged from tac1, in forward manner. *)
  fun (tac1 THEN-ALL-NEW-FWD tac2) i st =
    (tac1 i
     THEN (fn st' => INTERVAL-FWD tac2 i (i+nprems-of st'-nprems-of st)
st')) st;

  (* Repeat tac on subgoal. Determinize each step.
     Stop if tac fails or subgoal is solved. *)
  fun REPEAT-DETERM' tac i st = let
    val n = nprems-of st
  in
    REPEAT-DETERM (COND (has-fewer-prems n) no-tac (tac i)) st
  end

  (* Apply tactic to all goals in a forward manner.
    Newly generated goals are ignored.
  *)
  fun ALL-GOALS-FWD' tac i st =
    (tac i THEN (fn st' =>
      let
        val i' = i + nprems-of st' + 1 - nprems-of st;
      in
        if i' <= nprems-of st' then
          ALL-GOALS-FWD' tac i' st'
        else
          all-tac st'
      end
    )) st;

  fun ALL-GOALS-FWD tac = ALL-GOALS-FWD' tac 1;

  (*******************)
  (* Tactics *)
  (*******************)

  (* Resolve with rule. Use first-order unification.
    From cookbook, added exception handling *)
  fun fo-rtac thm = Subgoal.FOCUS (fn {concl, ...} =>
  let
    val concl-pat = Drule.strip-imp-concl (cprop-of thm)
    val insts = Thm.first-order-match (concl-pat, concl)
  in
    rtac (Drule.instantiate-normalize insts thm) 1
  end handle Pattern.MATCH => no-tac )
```

```
(* Resolve with premises. From cookbook. *)
fun rprems-tac ctxt =
  Subgoal.FOCUS-PREMS (fn {prems,...} => resolve-tac prems 1) ctxt;


(*******************)
(* Monotonicity Prover *)
(*******************)

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

  structure refine-mono-trigger = Named-Thms
    ( val name = @{binding refine-mono-trigger}
      val description = Refinement Framework: ^
        Triggering rules for monotonicity prover )

  (* Monotonicity prover: Solve by removing function arguments *)
  fun solve-le-tac ctxt = SOLVED' (REPEAT-ALL-NEW (
    Method.assm-tac ctxt ORELSE' fo-rtac @{thm le-funD} ctxt));

  (* Monotonicity prover. TODO: A related thing is in the
    partial-function package, can we reuse (parts of) that?
  *)
  fun mono-prover-tac ctxt = REPEAT-ALL-NEW (FIRST' [
    Method.assm-tac ctxt,
    match-tac (refine-mono.get ctxt),
    solve-le-tac ctxt
  ]);

  (* Monotonicity prover: Match triggering rule, and solve
    resulting goals completely or fail.
  *)
  fun triggered-mono-tac ctxt =
    SOLVED' (
      match-tac (refine-mono-trigger.get ctxt)
      THEN-ALL-NEW mono-prover-tac ctxt);



end;
⟫
```

setup ⟪ *Refine-Misc.refine-mono.setup* ⟫
setup ⟪ *Refine-Misc.refine-mono-trigger.setup* ⟫

method-setup *refine-mono* =

⟪ *Scan.succeed (fn ctxt => SIMPLE-METHOD′ (*
Refine-Misc.mono-prover-tac ctxt
)) ⟫
Refinement framework: *Monotonicity prover*

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*]:
$\llbracket\bigwedge a\ b.\ p=(a,b) \Longrightarrow f\ a\ b \leq f'\ a\ b\rrbracket \Longrightarrow$ *prod-case f p* ≤ *prod-case f′ p*
by (*auto split*: *prod.split*)

lemma *if-mono*[*refine-mono*]:
assumes $b \Longrightarrow m1 \leq m1'$
assumes $\neg b \Longrightarrow m2 \leq m2'$
shows (*if b then m1 else m2*) ≤ (*if b then m1′ else m2′*)
using *assms* **by** *auto*

lemma *let-mono*[*refine-mono*]:
$f\ x \leq f'\ x' \Longrightarrow$ *Let x f* ≤ *Let x′ f′* **by** *auto*

2.1.2 Uncategorized Lemmas

lemma *all-nat-split-at*: $\forall\, i$::′*a*::*linorder*<*k*. *P i* ⟹ *P k* ⟹ ∀ *i*>*k*. *P i*
⟹ ∀ *i*. *P i*
by (*metis linorder-neq-iff*)

lemma *list-all2-induct*[*consumes 1*, *case-names Nil Cons*]:
assumes *list-all2 P l l′*
assumes *Q* [] []
assumes ⋀*x x′ ls ls′*. ⟦ *P x x′*; *list-all2 P ls ls′*; *Q ls ls′* ⟧
⟹ *Q* (*x*#*ls*) (*x′*#*ls′*)
shows *Q l l′*
using *list-all2-lengthD*[*OF assms*(*1*)] *assms*
apply (*induct rule*: *list-induct2*)
apply *auto*
done

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

lemma *comp-cong-right*: *x* = *y* ⟹ *f o x* = *f o y* **by** (*simp*)
lemma *comp-cong-left*: *x* = *y* ⟹ *x o f* = *y o f* **by** (*simp*)

lemma *nested-prod-case-simp*: (λ(*a*,*b*) *c*. *f a b c*) *x y* =
(*prod-case* (λ*a b*. *f a b y*) *x*)
by (*auto split*: *prod.split*)

2.1.3 Relations

Transitive Closure

lemma *rtrancl-apply-insert*: $R^* `` (insert\ x\ S) = insert\ x\ (R^* `` (S \cup R `` \{x\}))$
  apply (*auto*)
  apply (*erule converse-rtranclE*)
  apply *auto* [*2*]
  apply (*erule converse-rtranclE*)
  apply (*auto intro*: *converse-rtrancl-into-rtrancl*) [*2*]
  done

lemma *rtrancl-last-visit-node*:
  assumes $(s,s') \in R^*$
  shows $s \neq sh \wedge (s,s') \in (R \cap (UNIV \times (-\{sh\})))^* \vee$
    $(s,sh) \in R^* \wedge (sh,s') \in (R \cap (UNIV \times (-\{sh\})))^*$
  using *assms*
proof (*induct rule*: *converse-rtrancl-induct*)
  case *base* **thus** *?case* **by** *auto*
next
  case (*step s st*)
  moreover {
    assume *P*: $(st,s') \in (R \cap UNIV \times - \{sh\})^*$
    {
      assume $st = sh$ **with** *step* **have** *?case*
        by *auto*
    } **moreover** {
      assume $st \neq sh$
      with ‹$(s,st) \in R$› **have** $(s,st) \in (R \cap UNIV \times - \{sh\})^*$ **by** *auto*
      also note *P*
      finally have *?case* **by** *blast*
    } **ultimately have** *?case* **by** *blast*
  } **moreover** {
    assume *P*: $(st,\ sh) \in R^* \wedge (sh,\ s') \in (R \cap UNIV \times - \{sh\})^*$
    with *step*(*1*) **have** *?case*
      by (*auto dest*: *converse-rtrancl-into-rtrancl*)
  } **ultimately show** *?case* **by** *blast*
qed

Well-Foundedness

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

This lemma transfers well-foundedness over a simulation relation.

lemma *sim-wf*:
  assumes *WF*: $wf\ (S'^{-1})$
  assumes *STARTR*: $(x0,x0') \in R$

```
  assumes SIM: ⋀s s' t. ⟦ (s,s')∈R; (s,t)∈S; (x0',s')∈S'* ⟧
    ⟹ ∃t'. (s',t')∈S' ∧ (t,t')∈R
  assumes CLOSED: Domain S ⊆ S* ``{x0}
  shows wf (S⁻¹)
proof (rule wf-no-infinite-down-chainI, simp)
```

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

```
  fix f
  assume CHAIN: ⋀i. (f i, f (Suc i))∈S
```

Extend to start with *x0*

```
  obtain f' where CHAIN': ⋀i. (f' i, f' (Suc i))∈S and [simp]: f' 0 = x0
  proof –
    {
      fix x
      assume S: x = f 0
      from CHAIN have f 0 ∈ Domain S by auto
      with CLOSED have (x0,x)∈S* by (auto simp: S)
      then obtain g k where G0: g 0 = x0 and X: x = g k
        and CH: (∀i<k. (g i, g (Suc i))∈S)
      proof induct
        case base thus ?case by force
      next
        case (step y z) show ?case proof (rule step.hyps(3))
          fix g k
          assume g 0 = x0 and y = g k
            and ∀i<k. (g i, g (Suc i))∈S
          thus ?case using ‹(y,z)∈S›
            by (rule-tac step.prems[where g=g(Suc k := z) and k=Suc k])
              auto
        qed
      qed
      def f'≡λi. if i<k then g i else f (i–k)
      have ∃f'. f' 0 = x0 ∧ (∀i. (f' i, f' (Suc i))∈S)
        apply (rule-tac x=f' in exI)
        apply (unfold f'-def)
        apply (rule conjI)
        using S X G0 apply (auto) []
        apply (rule-tac k=k in all-nat-split-at)
        apply (auto)
        apply (simp add: CH)
        apply (subgoal-tac k = Suc i)
        apply (simp add: S[symmetric] CH X)
        apply simp
```

apply (*simp add*: *CHAIN*)
apply (*subgoal-tac Suc i* $-$ *k* $=$ *Suc* ($i-k$))
using *CHAIN* **apply** *simp*
apply *simp*
done
}
then obtain f' **where** $\forall i$. (f' i,f' (*Suc* i))$\in S$ **and** f' 0 $=$ *x0* **by** *blast*
thus *?thesis* **by** (*blast intro*!: *that*)
qed

Construct chain in S'

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

This contradicts well-foundedness

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

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

definition *finite-psupset* S $\equiv$ { (Q',Q). $Q\subset Q'$ $\wedge$ $Q'\subseteq S$ }
lemma *finite-psupset-wf*[*simp*, *intro*]: *finite* S $\Longrightarrow$ *wf* (*finite-psupset* S)
unfolding *finite-psupset-def* **by** (*blast intro*: *wf-bounded-supset*)

2.1.4 Monotonicity and Orderings

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

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

lemma *mono-infI'*:
mono f $\Longrightarrow$ *mono g* $\Longrightarrow$ *mono* (λx. *inf* (*f x*) (*g x*) :: *'b*::*lattice*)
by (*rule mono-infI*[*unfolded inf-fun-def*])

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

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

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

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

apply *simp*
apply (*erule monoD*)
apply (*erule Inf-lower*)
done

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

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

declare *SUP-insert*[*simp*]

lemma (**in** *semilattice-inf*) *le-infD1*:
$a \leq inf\ b\ c \Longrightarrow a \leq b$ **by** (*rule le-infE*)
lemma (**in** *semilattice-inf*) *le-infD2*:
$a \leq inf\ b\ c \Longrightarrow a \leq c$ **by** (*rule le-infE*)
lemma (**in** *semilattice-inf*) *inf-leI*:
$\llbracket \bigwedge x.\ \llbracket x{\leq}a;\ x{\leq}b \rrbracket \Longrightarrow x{\leq}c \rrbracket \Longrightarrow inf\ a\ b \leq c$
by (*metis inf-le1 inf-le2*)

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

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

Galois Connections

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

```
    have α x ≤ α x by simp
    with galois show x ≤ γ(α(x)) by blast
  qed

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

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

  lemma dist-γ[simp]:
    γ (inf a b) = inf (γ a) (γ b)
    apply (rule order-antisym)
    apply (rule mono-inf[OF γ-mono])
    apply (simp add: galois)
    apply (simp add: galois[symmetric])
    done

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

end
```

Fixed Points

```
lemma mono-lfp-eqI:
  assumes MONO: mono f
  assumes FIXP: f a ≤ a
  assumes LEAST: ⋀x. f x = x ⟹ a≤x
  shows lfp f = a
  apply (rule antisym)
  apply (rule lfp-lowerbound)
  apply (rule FIXP)
  by (metis LEAST MONO lfp-unfold)

lemma mono-gfp-eqI:
  assumes MONO: mono f
  assumes FIXP: a ≤ f a
  assumes GREATEST: ⋀x. f x = x ⟹ x≤a
  shows gfp f = a
  apply (rule antisym)
  apply (metis GREATEST MONO gfp-unfold)
```

apply (*rule gfp-upperbound*)
apply (*rule FIXP*)
done

lemma *lfp-le-gfp*: *mono f* $\Longrightarrow$ *lfp f* $\leq$ *gfp f*
by (*metis gfp-upperbound lfp-lemma3*)

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

Connecting Complete Lattices and Chain-Complete Partial Orders

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

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

lemma *ccpo-mono-simp*: *monotone* (*op* $\leq$) (*op* $\leq$) *f* $\longleftrightarrow$ *mono f*
unfolding *monotone-def mono-def* **by** *simp*
lemma *ccpo-monoI*: *mono f* $\Longrightarrow$ *monotone* (*op* $\leq$) (*op* $\leq$) *f*
by (*simp add*: *ccpo-mono-simp*)
lemma *ccpo-monoD*: *monotone* (*op* $\leq$) (*op* $\leq$) *f* $\Longrightarrow$ *mono f*
by (*simp add*: *ccpo-mono-simp*)

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

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

```
  apply simp

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

lemma ccpo-gfp-simp: ⋀f. mono f ⟹ ccpo.fixp Inf op ≥ f = gfp f
  apply (rule antisym)
  apply (rule gfp-upperbound)
  apply (drule ccpo.fixp-unfold[OF is-dual-ccpo dual-ccpo-monoI, symmetric])
  apply simp

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

abbreviation chain-admissible P ≡ ccpo.admissible Sup op ≤ P
abbreviation is-chain ≡ Complete-Partial-Order.chain (op ≤)
lemmas chain-admissibleI[intro?] = ccpo.admissibleI[OF is-ccpo]

abbreviation dual-chain-admissible P ≡ ccpo.admissible Inf (λx y. y≤x) P
abbreviation is-dual-chain ≡ Complete-Partial-Order.chain (λx y. y≤x)
lemmas dual-chain-admissibleI[intro?] = ccpo.admissibleI[OF is-dual-ccpo]

lemma dual-chain-iff[simp]: is-dual-chain C = is-chain C
  unfolding chain-def
  apply auto
  done

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

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

lemma point-chainI: is-chain M ⟹ is-chain ((λf. f x)`M)
  by (auto intro: chainI le-funI dest: chainD le-funD)
```

We transfer the admissible induction lemmas to complete lattices.

```
lemma lfp-cadm-induct:
  ⟦chain-admissible P; mono f; ⋀x. P x ⟹ P (f x)⟧ ⟹ P (lfp f)
  apply (simp only: ccpo-mono-simp[symmetric] ccpo-lfp-simp[symmetric])
  by (rule ccpo.fixp-induct[OF is-ccpo])

lemma gfp-cadm-induct:
  ⟦dual-chain-admissible P; mono f; ⋀x. P x ⟹ P (f x)⟧ ⟹ P (gfp f)
  apply (simp only: dual-ccpo-mono-simp[symmetric] ccpo-gfp-simp[symmetric])
```

by (*rule ccpo.fixp-induct*[*OF is-dual-ccpo*])

Continuity and Kleene Fixed Point Theorem

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

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

lemma *strictD*[*dest*]: $strict\ f \Longrightarrow f\ bot = bot$
  unfolding *strict-def* **by** *auto*
— We only add this lemma to the simpset for functions on the same type. Otherwise, the simplifier tries it much too often.
lemma *strictD-simp*[*simp*]: $strict\ f \Longrightarrow f\ (bot::'a::bot) = (bot::'a)$
  unfolding *strict-def* **by** *auto*

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

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

lemma *inf-distribI*[*intro?*]: $[\![strict\ f;\ cont\ f]\!] \Longrightarrow inf\text{-}distrib\ f$
  unfolding *inf-distrib-def* **by** *auto*

lemma *inf-distribD'*[*simp*]:
  fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice$
  shows $inf\text{-}distrib\ f \Longrightarrow f\ (Sup\ C) = Sup\ (f`C)$
  apply (*cases* $C=\{\}$)
  apply (*auto dest*: *inf-distribD contD*)
  done

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

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

lemma inf-distrib-is-mono[simp]:
  fixes f :: 'a::complete-lattice ⇒ 'b::complete-lattice
  shows inf-distrib f ⟹ mono f
  by simp
```

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

```
theorem kleene-lfp:
  fixes f:: 'a::complete-lattice ⇒ 'a
  assumes CONT: cont f
  shows lfp f = (SUP i. (f^^i) bot)
proof (rule antisym)
  let ?K={ (f^^i) bot | i . True}
  note MONO=cont-is-mono[OF CONT]
  {
    fix i
    have (f^^i) bot ≤ lfp f
      apply (induct i)
      apply simp
      apply simp
      by (metis MONO lfp-unfold monoD)
  } thus (SUP i. (f^^i) bot) ≤ lfp f by (blast intro: SUP-least)
next
  show lfp f ≤ (SUP i. (f^^i) bot)
    apply (rule lfp-lowerbound)
    unfolding SUP-def
    apply (simp add: contD[OF CONT])
    apply (rule Sup-subset-mono)
    apply (auto)
    apply (rule-tac x=Suc i in range-eqI)
    apply auto
    done
qed


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

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

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

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

  apply (rule Sup-least)
  apply clarify
  apply (rule monoD[OF α-mono])
  apply (rule Sup-upper)
  apply simp
  done
```

2.1.5 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)
    apply (simp-all)
    by (auto simp: map-to-set-def restrict-map-def split: split-if-asm
      intro: single-valuedI)

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

end
```

2.2 Transfer between Domains

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

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

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

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

end
⟫

setup ⟪ RefineG-Transfer.transfer.setup ⟫
method-setup refine-transfer =
  ⟪ Attrib.thms >> (fn thms => fn ctxt => SIMPLE-METHOD'
    (RefineG-Transfer.transfer-tac thms ctxt))
  ⟫ Invoke transfer rules
```

locale *transfer* = **fixes** $\alpha :: {'c} \Rightarrow {'a}{::}complete\text{-}lattice$
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 $\alpha\ (\mathit{if}\ b\ \mathit{then}\ s1\ \mathit{else}\ s2) \le (\mathit{if}\ b\ \mathit{then}\ S1\ \mathit{else}\ S2)$
  using *assms* **by** *auto*

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

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

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

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
    apply *rule*
    apply (*subgoal-tac is-chain* {*x*,*y*})
    apply (*drule* α-*dist*)
    apply (*auto simp*: *le-iff-sup*) []
    apply (*rule chainI*)
    apply *auto*
    done

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

lemma α-*strict*[*simp*]: α (*Sup* $\{\}$) $=$ *bot*
using α-*dist*[*of* $\{\}$] **by** *simp*
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* $\equiv$ *if* (*mono B*) *then* (*lfp B x*) *else top*
definition *RECT* (REC_T) **where** *RECT B x* $\equiv$ *if* (*mono B*) *then* (*gfp B x*) *else top*

lemma *REC-unfold*: *mono B* $\Longrightarrow$ *REC B x* $=$ *B* (*REC B*) *x*
unfolding *REC-def* [*abs-def*]
by (*simp add*: *lfp-unfold*[*symmetric*])

lemma *RECT-unfold*: *mono B* $\Longrightarrow$ *RECT B x* $=$ *B* (*RECT B*) *x*
unfolding *RECT-def* [*abs-def*]
by (*simp add*: *gfp-unfold*[*symmetric*])

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*
unfolding *REC-def*
apply *clarsimp*
apply (*rule lfp-mono*[*THEN le-funD*])
apply (*rule le-funI*[*OF LE*])
done

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*
unfolding *RECT-def*
apply *clarsimp*
apply (*rule gfp-mono*[*THEN le-funD*])
apply (*rule le-funI*[*OF LE*])
done

lemma *REC-le-RECT*: *REC body x* $\leq$ *RECT body x*
unfolding *REC-def RECT-def*
apply *clarsimp*
apply (*erule lfp-le-gfp′*)
done

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 (*cases mono body*)
assume ¬*mono body*
thus *?thesis* **unfolding** *REC-def RECT-def* **by** *simp*
next
assume *MONO*: *mono body*

from *lfp-le-gfp′ MONO* **have** *lfp body x* $\leq$ *gfp body x* **.**
moreover have *I x* $\longrightarrow$ *gfp body x* $\leq$ *lfp body x*
using *WF*
apply (*induct rule*: *wf-induct*[*consumes 1*])
apply (*rule impI*)
apply (*subst lfp-unfold*[*OF MONO*])
apply (*subst gfp-unfold*[*OF MONO*])
apply (*rule order-trans*[*OF - IS*])
apply (*rule monoD*[*OF MONO*,*THEN le-funD*])
apply (*rule le-funI*)

```
    apply simp
    apply simp
    done
  ultimately show ?thesis
    unfolding REC-def RECT-def
    apply (rule-tac antisym)
    using I0 MONO by auto
qed
```

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: ⋀f x. ⟦I x;
    ⋀x'. ⟦ I x'; (x',x)∈R ⟧ ⟹ f x' ≤ M x'
  ⟧ ⟹
    body (λx'. if (I x' ∧ (x',x)∈R) then f x' else top) x ≤ body f x ∧
    body f x ≤ M x
  shows RECT body x = REC body x ∧ RECT body x ≤ M x
proof −


  from lfp-le-gfp' MONO have lfp body x ≤ gfp body x .
  moreover have I x ⟶ gfp body x ≤ lfp body x ∧ lfp body x ≤ M x
    using WF
    apply (induct rule: wf-induct[consumes 1])
    apply (rule impI)
    apply (rule conjI)
    apply (subst lfp-unfold[OF MONO])
    apply (subst gfp-unfold[OF MONO])
    apply (rule order-trans[OF - conjunct1[OF IS]])
    apply (rule monoD[OF MONO,THEN le-funD])
    apply (rule le-funI)
    apply simp
    apply simp
    apply simp

    apply (subst lfp-unfold[OF MONO])
    apply (rule conjunct2[OF IS])
    apply simp
    apply simp
    done
  ultimately show ?thesis
    unfolding REC-def RECT-def
    using I0 MONO by auto
qed
```

```
lemma REC-rule:
  fixes x::'x
  assumes M: mono body
  assumes I0: Φ x
  assumes IS: ⋀f x. ⟦ ⋀x. Φ x ⟹ f x ≤ M x; Φ x ⟧
    ⟹ body f x ≤ M x
  shows REC body x ≤ M x
proof –
  have ∀ x. Φ x ⟶ lfp body x ≤ M x
    apply (rule lfp-cadm-induct[OF - M])
      apply rule
      apply rule
      apply rule
      unfolding Sup-fun-def
      apply (rule SUP-least)
      apply simp

      apply (rule)
      apply (rule)
      apply (rule IS)
      apply blast
      apply assumption
    done
  thus ?thesis
    unfolding REC-def
    by (auto simp: I0 M)
qed

lemma RECT-rule:
  assumes M: mono body
  assumes WF: wf (V::('x×'x) set)
  assumes I0: Φ (x::'x)
  assumes IS: ⋀f x. ⟦ ⋀x'. ⟦Φ x'; (x',x)∈V⟧ ⟹ f x' ≤ M x'; Φ x ⟧
    ⟹ body f x ≤ M x
  shows RECT body x ≤ M x
proof –
  have Φ x ⟶ gfp body x ≤ M x
    using WF
    apply (induct x rule: wf-induct[consumes 1])
    apply (rule impI)
    apply (subst gfp-unfold[OF M])
    apply (rule IS)
    apply simp
    apply simp
    done
  with I0 M show ?thesis unfolding RECT-def by auto
qed
```

2.3.1 Transfer

```
lemma (in transfer) transfer-RECT'[refine-transfer]:
  assumes REC-EQ: ⋀x. fr x = b fr x
  assumes REF: ⋀F f x. ⟦⋀x. α (f x) ≤ F x ⟧ ⟹ α (b f x) ≤ B F x
  shows α (fr x) ≤ RECT B x
  unfolding RECT-def
proof clarsimp
  assume MONO: mono B
  show α (fr x) ≤ gfp B x
    apply (rule-tac x=x in spec)
    apply (rule gfp-cadm-induct[where f=B])
    apply rule
    apply (intro allI)

    apply (unfold Inf-fun-def INF-def)
    apply (drule chain-dualI)
    apply (drule-tac x=x in point-chainI)

    apply (case-tac A={})
    apply simp
    apply (rule Inf-greatest)
    apply (auto intro: le-funI) []

    apply fact

    using REF REC-EQ by force
qed

lemma (in ordered-transfer) transfer-RECT[refine-transfer]:
  assumes REF: ⋀F f x. ⟦⋀x. α (f x) ≤ F x ⟧ ⟹ α (b f x) ≤ B F x
  assumes mono b
  shows α (RECT b x) ≤ RECT B x
  apply (rule transfer-RECT')
  apply (rule RECT-unfold[OF ‹mono b›])
  by fact

lemma (in dist-transfer) transfer-REC[refine-transfer]:
  assumes REF: ⋀F f x. ⟦⋀x. α (f x) ≤ F x ⟧ ⟹ α (b f x) ≤ B F x
  assumes mono b
  shows α (REC b x) ≤ REC B x
  unfolding REC-def
proof (clarsimp simp add: ‹mono b›)
  assume mono B
  show α (lfp b x) ≤ lfp B x
    apply (rule-tac x=x in spec)
    apply (rule lfp-cadm-induct[OF - ‹mono b›])

    apply rule
    apply clarsimp
```

```
  apply (unfold Sup-fun-def SUP-def)
  apply (drule-tac x=x in point-chainI)
  apply (simp add: α-dist)
  apply (rule Sup-least)
  apply auto []

  apply clarsimp
  apply (subst lfp-unfold[OF ‹mono B›])
  apply (rule REF)
  apply simp
  done
qed

end
```

2.4 Assert and Assume

```
theory RefineG-Assert
imports RefineG-Transfer
begin

definition iASSERT return Φ ≡ if Φ then return () else top
definition iASSUME return Φ ≡ if Φ then return () else bot

 locale generic-Assert =
  fixes bind ::
   ('mu::complete-lattice) ⇒ (unit ⇒ ('ma::complete-lattice)) ⇒ 'ma
  fixes return :: unit ⇒ 'mu
  fixes ASSERT
  fixes ASSUME
  assumes ibind-strict:
   bind bot f = bot
   bind top f = top
  assumes imonad1: bind (return u) f = f u
  assumes ASSERT-eq: ASSERT ≡ iASSERT return
  assumes ASSUME-eq: ASSUME ≡ iASSUME return
 begin
   lemma ASSERT-simps[simp,code]:
    ASSERT True = return ()
    ASSERT False = top
    unfolding ASSERT-eq iASSERT-def by auto

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

lemma *le-ASSERTI*: $[\![\ \Phi \Longrightarrow M{\le}M'\ ]\!] \Longrightarrow M \le bind\ (ASSERT\ \Phi)\ (\lambda\text{-}.\ M')$
apply (*cases* Φ) **by** (*auto simp*: *ibind-strict imonad1*)

lemma *le-ASSERTI-pres*: $[\![\ \Phi \Longrightarrow M{\le}bind\ (ASSERT\ \Phi)\ (\lambda\text{-}.\ M')\ ]\!]$
$\Longrightarrow M \le bind\ (ASSERT\ \Phi)\ (\lambda\text{-}.\ M')$
apply (*cases* Φ) **by** (*auto simp*: *ibind-strict imonad1*)

lemma *ASSERT-leI*: $[\![\ \Phi;\ \Phi \Longrightarrow M{\le}M'\ ]\!] \Longrightarrow bind\ (ASSERT\ \Phi)\ (\lambda\text{-}.\ M) \le M'$
apply (*cases* Φ) **by** (*auto simp*: *ibind-strict imonad1*)

lemma *ASSUME-leI*: $[\![\ \Phi \Longrightarrow M{\le}M'\ ]\!] \Longrightarrow bind\ (ASSUME\ \Phi)\ (\lambda\text{-}.\ M) \le M'$
apply (*cases* Φ) **by** (*auto simp*: *ibind-strict imonad1*)
lemma *ASSUME-leI-pres*: $[\![\ \Phi \Longrightarrow bind\ (ASSUME\ \Phi)\ (\lambda\text{-}.\ M){\le}M'\ ]\!]$
$\Longrightarrow bind\ (ASSUME\ \Phi)\ (\lambda\text{-}.\ M) \le M'$
apply (*cases* Φ) **by** (*auto simp*: *ibind-strict imonad1*)
lemma *le-ASSUMEI*: $[\![\ \Phi;\ \Phi \Longrightarrow M{\le}M'\ ]\!] \Longrightarrow M \le bind\ (ASSUME\ \Phi)\ (\lambda\text{-}.\ M')$
apply (*cases* Φ) **by** (*auto simp*: *ibind-strict imonad1*)

The order of these declarations does matter!

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

lemma *ASSERT-le-iff*:
$bind\ (ASSERT\ \Phi)\ (\lambda\text{-}.\ S) \le S' \longleftrightarrow (S'{\neq}top \longrightarrow \Phi) \wedge S{\le}S'$
by (*cases* Φ) (*auto simp*: *ibind-strict imonad1 simp*: *top-unique*)

lemma *ASSUME-le-iff*:
$bind\ (ASSUME\ \Phi)\ (\lambda\text{-}.\ S) \le S' \longleftrightarrow (\Phi \longrightarrow S{\le}S')$
by (*cases* Φ) (*auto simp*: *ibind-strict imonad1*)

lemma *le-ASSERT-iff*:
$S \le bind\ (ASSERT\ \Phi)\ (\lambda\text{-}.\ S') \longleftrightarrow (\Phi \longrightarrow S{\le}S')$
by (*cases* Φ) (*auto simp*: *ibind-strict imonad1*)

lemma *le-ASSUME-iff*:
$S \le bind\ (ASSUME\ \Phi)\ (\lambda\text{-}.\ S') \longleftrightarrow (S{\neq}bot \longrightarrow \Phi) \wedge S{\le}S'$
by (*cases* Φ) (*auto simp*: *ibind-strict imonad1 simp*: *bot-unique*)
end

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

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

$\Rightarrow$ (*unit*$\Rightarrow$'*mac*) $\Rightarrow$ ('*mac*::*complete-lattice*)
and *creturn* :: *unit* $\Rightarrow$ '*muc* **and** *cASSERT* **and** *cASSUME*
and *abind* :: ('*mua*::*complete-lattice*)
$\Rightarrow$ (*unit*$\Rightarrow$'*maa*) $\Rightarrow$ ('*maa*::*complete-lattice*)
and *areturn* :: *unit* $\Rightarrow$ '*mua* **and** *aASSERT* **and** *aASSUME*
and α :: '*mac* $\Rightarrow$ '*maa*
begin
lemma *transfer-ASSERT*[*refine-transfer*]:
$[\![\Phi \Longrightarrow \alpha\ M \leq M']\!]$
$\Longrightarrow \alpha\ (cbind\ (cASSERT\ \Phi)\ (\lambda\text{-}.\ M)) \leq (abind\ (aASSERT\ \Phi)\ (\lambda\text{-}.\ M'))$
apply (*cases* Φ)
apply (*auto simp*: *c.ibind-strict a.ibind-strict c.imonad1 a.imonad1*)
done

lemma *transfer-ASSUME*[*refine-transfer*]:
$[\![\Phi; \Phi \Longrightarrow \alpha\ M \leq M']\!]$
$\Longrightarrow \alpha\ (cbind\ (cASSUME\ \Phi)\ (\lambda\text{-}.\ M)) \leq (abind\ (aASSUME\ \Phi)\ (\lambda\text{-}.\ M'))$
apply (*auto simp*: *c.ibind-strict a.ibind-strict c.imonad1 a.imonad1*)
done

end

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

lemma *transfer-ASSUME-remove*[*refine-transfer*]:
$[\![\ \Phi; \Phi \Longrightarrow \alpha\ M \leq M'\ ]\!] \Longrightarrow \alpha\ M \leq abind\ (aASSUME\ \Phi)\ (\lambda\text{-}.\ M')$
by (*rule a.le-ASSUMEI*)
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 ⟪
  structure Refine = struct

  open Refine-Misc;

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

  structure refine0 = Named-Thms
    ( val name = @{binding refine0}
      val description = Refinement Framework: ˆ
        Refinement rules applied first (intro) )

  structure refine = Named-Thms
    ( val name = @{binding refine}
      val description = Refinement Framework: Refinement rules (intro) )

  structure refine2 = Named-Thms
    ( val name = @{binding refine2}
      val description = Refinement Framework: ˆ
        Refinement rules 2nd stage (intro) )

  structure refine-pw-simps = Named-Thms
    ( val name = @{binding refine-pw-simps}
      val description = Refinement Framework: ˆ
        Simplifier rules for pointwise reasoning )

  structure refine-post = Named-Thms
    ( val name = @{binding refine-post}
      val description = Refinement Framework: ˆ
        Postprocessing rules )



  (* If set to true, the product splitter of refine-rcg is disabled. *)
  val no-prod-split =
    Attrib.setup-config-bool @{binding refine-no-prod-split} (K false);

  fun rcg-tac add-thms ctxt =
    let
      val ref-thms = (refine0.get ctxt
        @ add-thms @ refine.get ctxt @ refine2.get ctxt);
```

```
      val prod-ss = (Splitter.add-split @{thm prod.split} HOL-basic-ss);
      val prod-simp-tac =
        if Config.get ctxt no-prod-split then
          K no-tac
        else
          (simp-tac prod-ss THEN'
            REPEAT-ALL-NEW (resolve-tac @{thms impI allI}));
    in
    REPEAT-ALL-NEW (DETERM o (resolve-tac ref-thms ORELSE' prod-simp-tac))
    end;

    fun post-tac ctxt = let
      val thms = refine-post.get ctxt;
    in
      REPEAT-ALL-NEW (match-tac thms ORELSE' triggered-mono-tac ctxt)
    end;

  end;
⟫
setup ⟪ Refine.vcg.setup ⟫
setup ⟪ Refine.refine0.setup ⟫
setup ⟪ Refine.refine.setup ⟫
setup ⟪ Refine.refine2.setup ⟫
setup ⟪ Refine.refine-post.setup ⟫
setup ⟪ Refine.refine-pw-simps.setup ⟫

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

method-setup refine-post =
  ⟪ Scan.succeed (fn ctxt => SIMPLE-METHOD' (
    Refine.post-tac ctxt
  )) ⟫
  Refinement framework: Postprocessing of refinement goals
```

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

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

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

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

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

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

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

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

definition bot ≡ RES {}
definition top ≡ FAILi

instance
  apply (intro-classes)
  unfolding Sup-nres-def Inf-nres-def bot-nres-def top-nres-def
  apply (case-tac x, case-tac [!] y, auto) []
  apply (case-tac x, auto) []
  apply (case-tac x, case-tac [!] y, case-tac [!] z, auto) []
  apply (case-tac x, (case-tac [!] y)?, auto) []
  apply (case-tac x, (case-tac [!] y)?, simp-all) []
  apply (case-tac x, (case-tac [!] y)?, auto) []
```

apply (*case-tac* x, *case-tac* [!] y, *case-tac* [!] z, *auto*) []
apply (*case-tac* x, (*case-tac* [!] y)?, *auto*) []
apply (*case-tac* x, (*case-tac* [!] y)?, *auto*) []
apply (*case-tac* x, *case-tac* [!] y, *case-tac* [!] z, *auto*) []
apply (*case-tac* a, *auto*) []
apply (*case-tac* a, *auto*) []
apply (*case-tac* x, *auto*) []
apply (*case-tac* z, *fastforce*+) []
apply (*case-tac* x, *auto*) []
apply (*case-tac* z, *fastforce*+) []
done

end

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

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

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

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

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

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

lemma *nres-order-simps*[*simp*]:
$\mathit{SUCCEED} \leq M$
$M \leq \mathit{SUCCEED} \longleftrightarrow M{=}\mathit{SUCCEED}$
$M \leq \mathit{FAIL}$
$\mathit{FAIL} \leq M \longleftrightarrow M{=}\mathit{FAIL}$
$\mathit{RES}\ X \leq \mathit{RES}\ Y \longleftrightarrow X{\leq}Y$

Sup X = *FAIL* ⟷ *FAIL*∈*X*
FAIL = *Sup X* ⟷ *FAIL*∈*X*
FAIL∈*X* ⟹ *Sup X* = *FAIL*
Sup (*RES*‘*A*) = *RES* (*Sup A*)
A≠{} ⟹ *Inf* (*RES*‘*A*) = *RES* (*Inf A*)
Inf {} = *FAIL*
Inf UNIV = *SUCCEED*
Sup {} = *SUCCEED*
Sup UNIV = *FAIL*
unfolding *Sup-nres-def Inf-nres-def*
by (*auto simp add*: *bot-unique top-unique nres-simp-internals*)

lemma *Sup-eq-RESE*:
assumes *Sup A* = *RES B*
obtains *C* **where** *A*=*RES*‘*C* **and** *B*=*Sup C*
proof −
show *?thesis*
using *assms* **unfolding** *Sup-nres-def*
apply (*simp split*: *split-if-asm*)
apply (*rule-tac C*={*X*. *RES X* ∈ *A*} **in** *that*)
apply *auto* []
apply (*case-tac x*, *auto simp*: *nres-simp-internals*) []
apply (*auto simp*: *nres-simp-internals*) []
done
qed

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

Pointwise Reasoning

definition *nofail S* ≡ *S*≠*FAIL*
definition *inres S x* ≡ *RETURN x* ≤ *S*

lemma *nofail-simps*[*simp*, *refine-pw-simps*]:
nofail FAIL ⟷ *False*
nofail (*RES X*) ⟷ *True*
nofail SUCCEED ⟷ *True*
unfolding *nofail-def*
by *simp-all*

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

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

lemma *not-nofail-inres*[*simp*, *refine-pw-simps*]:
$\neg$*nofail* $S \Longrightarrow$ *inres* S x
apply (*cases* S) **by** *auto*

lemma *intro-nofail*[*refine-pw-simps*]:
$S{\neq}FAIL \longleftrightarrow$ *nofail* S
$FAIL{\neq}S \longleftrightarrow$ *nofail* S
by (*cases* S, *simp-all*)+

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

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

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

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

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

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

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

lemma *pwD2*:

assumes $S \le S'$ *inres* S x
shows *inres* S' x
using *assms*
by (*auto simp add*: *pw-le-iff*)

lemmas *pwD* = *pwD1 pwD2*

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

lemma *le-nofailI*: $[\![$ *nofail* $M' \Longrightarrow M \le M'$ $]\!] \Longrightarrow M \le M'$
by (*cases* M') *auto*

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

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

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

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

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

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

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

lemma *pw-Inf-nofail*[*refine-pw-simps*]: *nofail* (*Inf C*) $\longleftrightarrow$ ($\exists$ *x*$\in$*C*. *nofail x*)
  apply (*cases C*={})
  apply *simp*
  apply (*cases Inf C*)
  apply (*subgoal-tac C*={*FAIL*})
  apply *simp*
  apply *auto* []
  apply (*subgoal-tac C*$\neq${*FAIL*})
  apply (*auto simp*: *not-nofail-iff*) []
  apply *auto* []
  done

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

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*
  unfolding *bind-def* **by** (*auto split*: *nres.split*)

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

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

setup ⟪
  Adhoc-Overloading.add-variant
  @{*const-name Monad-Syntax.bind*} @{*const-name bind*}

$\rangle\rangle$

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

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

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

lemma *pw-bind-leI*: $[\![$
  nofail S $\Longrightarrow$ *nofail M*; $\bigwedge$*x*. $[\![$*nofail M*; *inres M x*$]\!]$ $\Longrightarrow$ *f x* $\leq$ *S*$]\!]$
  $\Longrightarrow$ *bind M f* $\leq$ *S*
  by (*simp add*: *pw-bind-le-iff*)

lemma *nres-monad1*[*simp*]: *bind* (*RETURN x*) *f* = *f x*
  by (*rule pw-eqI*) (*auto simp*: *refine-pw-simps*)
lemma *nres-monad2*[*simp*]: *bind M RETURN* = *M*
  by (*rule pw-eqI*) (*auto simp*: *refine-pw-simps*)
lemma *nres-monad3*[*simp*]: *bind* (*bind M f*) *g* = *bind M* (λ*x*. *bind* (*f x*) *g*)
  by (*rule pw-eqI*) (*auto simp*: *refine-pw-simps*)
lemmas *nres-monad-laws* = *nres-monad1 nres-monad2 nres-monad3*

lemma *bind-mono*[*refine-mono*]:
  $[\![$ *M* $\leq$ *M'*; $\bigwedge$*x*. *RETURN x* $\leq$ *M* $\Longrightarrow$ *f x* $\leq$ *f' x* $]\!]$ $\Longrightarrow$
  bind M f $\leq$ *bind M' f'*
  by (*auto simp*: *refine-pw-simps pw-le-iff*)

lemma *bind-mono1*[*simp*, *intro*!]: *mono* (λ*M*. *bind M f*)
  apply (*rule monoI*)
  apply (*rule bind-mono*)
  by *auto*

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

by *auto*

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

lemma *bind-distrib-sup*: *bind* (*sup M N*) *f* = *sup* (*bind M f*) (*bind N f*)
by (*auto simp add*: *pw-eq-iff refine-pw-simps*)

lemma *RES-Sup-RETURN*: *Sup* (*RETURN*'*X*) = *RES X*
by (*rule pw-eqI*) (*auto simp add*: *refine-pw-simps*)

2.5.3 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: ⇑*R* (*RES X*) = (
if X⊆*Domain R then*
RES (*R*‘‘*X*)
else
FAIL
)
unfolding *abs-fun-def*
by (*auto split*: *nres.split*)

definition *conc-fun* (⇓) **where**
conc-fun R M ≡ *case M of*
FAILi ⇒ *FAIL* |
RES X ⇒ *RES* {*c*. ∃ *a*∈*X*. (*c*,*a*)∈*R* ∧ (∀ *a*′. (*c*,*a*′)∈*R* ⟶ *a*′∈*X*)}

lemma
conc-fun-FAIL[*simp*]: ⇓*R FAIL* = *FAIL* **and**
conc-fun-RES: ⇓*R* (*RES X*) = *RES* {*c*. ∃ *a*∈*X*. (*c*,*a*)∈*R* ∧ (∀ *a*′. (*c*,*a*′)∈*R* ⟶ *a*′∈*X*)}
unfolding *conc-fun-def*
by (*auto split*: *nres.split*)

lemma *ac-galois*: *galois-connection* (⇑*R*) (⇓*R*)
apply (*unfold-locales*)
apply *rule*
unfolding *conc-fun-def abs-fun-def*
apply (*force split*: *nres.split nres.split-asm split-if-asm*
simp add: *top-nres-def*[*symmetric*]) []

apply (*force split*: *nres.split nres.split-asm split-if-asm*
simp add: *top-nres-def*[*symmetric*]) []
done

interpretation *ac*!: *galois-connection* (⇑*R*) (⇓*R*)
by (*rule ac-galois*)

lemma *pw-abs-nofail*[*refine-pw-simps*]:
nofail (⇑*R M*) ⟷ (*nofail M* ∧ (∀ *x*. *inres M x* ⟶ *x*∈*Domain R*))
apply (*cases M*)
apply *simp*
apply (*auto simp*: *abs-fun-RES*)
done

lemma *pw-abs-inres*[*refine-pw-simps*]:
inres (⇑*R M*) *a* ⟷ (*nofail* (⇑*R M*) ⟶ (∃ *c*. *inres M c* ∧ (*c*,*a*)∈*R*))
apply (*cases M*)

apply *simp*
apply (*auto simp*: *abs-fun-RES*)
done

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

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

lemma *pw-conc-inres-sv*[*refine-pw-simps*]:
$inres\ (\Downarrow R\ S') = (\lambda s.\ nofail\ S'$
$\longrightarrow (\exists\, s'.\ (s,s')\in R \wedge inres\ S'\ s' \wedge ($
$\neg\ single\text{-}valued\ R \longrightarrow (\forall\, s''.\ (s,s'')\in R \longrightarrow inres\ S'\ s''))))$
apply (*rule ext*)
apply (*cases S′*)
apply (*auto simp*: *conc-fun-RES dest*: *single-valuedD*)
done

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

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

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

lemma *conc-fun-chain*: $single\text{-}valued\ R \Longrightarrow \Downarrow R\ (\Downarrow S\ M) = \Downarrow(R\ O\ S)\ M$
unfolding *conc-fun-def*
by (*auto split*: *nres.split dest*: *single-valuedD*)

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

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

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

lemma *conc-trans*[*trans*]:
  assumes A: $C \leq \Downarrow R\ B$ **and** B: $B \leq \Downarrow R'\ A$
  shows $C \leq \Downarrow R\ (\Downarrow R'\ A)$
proof –
  from A **have** $\Uparrow R\ C \leq B$ **by** (*simp add*: *ac.galois*)
  also note B
  finally show *?thesis*
    by (*simp add*: *ac.galois*)
qed

lemma *abs-trans*[*trans*]:
  assumes A: $\Uparrow R\ C \leq B$ **and** B: $\Uparrow R'\ B \leq A$
  shows $\Uparrow R'\ (\Uparrow R\ C) \leq A$
  using *assms* **unfolding** *ac.galois*[*symmetric*]
  by (*rule conc-trans*)

Transitivity Reasoner Setup

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

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

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

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

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

  $\bigwedge A\ B\ C.\ [\![\Uparrow Id\ A \leq B;\ \Uparrow Id\ B \leq C]\!] \Longrightarrow A \leq C$
  $\bigwedge A\ B\ C.\ [\![\Uparrow Id\ A \leq B;\ B \leq C]\!] \Longrightarrow A \leq C$
  $\bigwedge A\ B\ C.\ [\![A \leq B;\ \Uparrow Id\ B \leq C]\!] \Longrightarrow A \leq C$
  using *conc-trans-additional*[*unfolded ac.galois*]
    abs-trans[**where** R=Id **and** R'=R]
  by *auto*

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*=α *c* $\wedge$ *I c*}
abbreviation *br*$\equiv$*build-rel*
lemmas *br-def* = *build-rel-def*

lemma *br-id*[*simp*]: *br id* (λ-. *True*) = *Id*
  unfolding *build-rel-def* **by** *auto*

lemma *br-chain*:
  (*build-rel* β *J*) *O* (*build-rel* α *I*) = *build-rel* ($\alpha\circ\beta$) (λ*s*. *J s* $\wedge$ *I* (β *s*))
  unfolding *build-rel-def* **by** *auto*

lemma *br-single-valued*[*simp*, *intro*!,*refine-post*]: *single-valued* (*build-rel* α *I*)
  unfolding *build-rel-def*
  apply (*rule single-valuedI*)
  apply *auto*
  done

lemma *converse-br-sv-iff*[*simp*]:
  single-valued (*converse* (*br* α *I*)) $\longleftrightarrow$ *inj-on* α (*Collect I*)
  by (*auto intro*!: *inj-onI single-valuedI dest*: *single-valuedD inj-onD*) []

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

lemma *pw-ASSERT*[*refine-pw-simps*]:
  nofail (*ASSERT* Φ) $\longleftrightarrow$ Φ
  inres (*ASSERT* Φ) *x*
  by (*cases* Φ, *simp-all*)+

lemma *pw-ASSUME*[*refine-pw-simps*]:
  nofail (*ASSUME* Φ)

inres $(ASSUME\ \Phi)\ x \longleftrightarrow \Phi$
by $(cases\ \Phi,\ simp\text{-}all)+$

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 −
have $nofail\ (REC\ B\ x) \longleftrightarrow mono\ B\ \wedge$
$(\exists F.\ (\forall x.\ B\ F\ x \leq F\ x)\ \wedge\ nofail\ (F\ x))$
unfolding *REC-def lfp-def*
by (*auto simp*: *refine-pw-simps Inf-fun-def INF-def*
intro: *le-funI dest*: *le-funD*)
thus *?thesis*
unfolding *pw-le-iff* **.**
qed

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 −
have $inres\ (REC\ B\ x)\ x'$
$\longleftrightarrow (mono\ B \longrightarrow (\forall F.\ (\forall x''.\ B\ F\ x'' \leq F\ x'') \longrightarrow inres\ (F\ x)\ x'))$
unfolding *REC-def lfp-def*
by (*auto simp*: *refine-pw-simps Inf-fun-def INF-def*
intro: *le-funI dest*: *le-funD*)
thus *?thesis* **unfolding** *pw-le-iff* **.**
qed

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*]: $\Phi\ x \Longrightarrow RETURN\ x \leq SPEC\ \Phi$
by *auto*

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

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

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

lemma *bind-rule-complete*: $bind\ M\ f \le SPEC\ \Phi \longleftrightarrow M \le SPEC\ (\lambda x.\ f\ x \le SPEC\ \Phi)$
by (*auto simp*: *pw-le-iff refine-pw-simps*)
lemma *bind-rule*[*refine-vcg*]:
$[\![\ M \le SPEC\ (\lambda x.\ f\ x \le SPEC\ \Phi)\ ]\!] \Longrightarrow bind\ M\ f \le SPEC\ \Phi$
by (*auto simp*: *bind-rule-complete*)

lemma *ASSUME-rule*[*refine-vcg*]: $[\![\Phi \Longrightarrow \Psi\ ()]\!] \Longrightarrow ASSUME\ \Phi \le SPEC\ \Psi$
by (*cases* Φ) *auto*
lemma *ASSERT-rule*[*refine-vcg*]: $[\![\Phi;\ \Phi \Longrightarrow \Psi\ ()]\!] \Longrightarrow ASSERT\ \Phi \le SPEC\ \Psi$ **by** *auto*

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

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 $(\lambda(a,b)\ (c,d).\ f\ a\ b\ c\ d)\ ab\ cd \le SPEC\ \Phi$
using *assms*
by (*auto split*: *prod.split*)

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

lemma *option-rule*[*refine-vcg*]:
  $[\![$ *v=None* $\Longrightarrow$ *S1* $\leq$ *SPEC* Φ; $\bigwedge x$. *v=Some x* $\Longrightarrow$ *f2 x* $\leq$ *SPEC* $\Phi]\!]$
  $\Longrightarrow$ *option-case S1 f2 v* $\leq$ *SPEC* Φ
  by (*auto split*: *option.split*)

lemma *Let-rule*[*refine-vcg*]:
  f x $\leq$ *SPEC* Φ $\Longrightarrow$ *Let x f* $\leq$ *SPEC* Φ **by** *auto*

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*
  apply (*rule RECT-eq-REC*)
  apply (*rule WF*)
  apply (*rule I0*)
  apply (*rule order-trans*[*OF - IS*])
  apply (*subgoal-tac* ($\lambda x'$. *if I x'* $\wedge$ (x', x) $\in$ *V then f x' else FAIL*) =
    ($\lambda x'$. *ASSERT* (*I x'* $\wedge$ (x', x) $\in$ *V*) $\gg=$ (λ-. *f x'*)))
  apply *simp*
  apply (*rule ext*)
  apply (*rule pw-eqI*)
  apply (*auto simp add*: *refine-pw-simps*)
  done

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 −
  have $\forall\, x\ x'$. (x,x')$\in$*R* $\longrightarrow$ *lfp body x* $\leq$ *M x'*
    apply (*rule lfp-cadm-induct*[*OF - M*])
      apply *rule+*
      unfolding *Sup-fun-def*
      apply (*rule SUP-least*)
      apply *simp*

      using *IS*
      apply *blast*
    done
  with *I0* **show** *?thesis*
    unfolding *REC-def*

by (*auto simp*: *M*)
qed

Proving Monotonicity

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

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

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

Proving Refinement

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

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

lemma *Id-refine*[*refine0*]: $S \le \Downarrow Id\ S$ **by** *auto*

lemma *RES-refine*:
assumes $S \subseteq Domain\ R$
assumes $\bigwedge x\ x'.\ [\![x \in S;\ (x,x') \in R]\!] \Longrightarrow x' \in S'$
shows $RES\ S \le \Downarrow R\ (RES\ S')$
using *assms*
by (*force simp*: *conc-fun-RES*)

lemma *RES-refine-sv*:
$[\![$*single-valued R*;
$\bigwedge s.\ s \in S \Longrightarrow \exists s' \in S'.\ (s,s') \in R]\!] \Longrightarrow RES\ S \le \Downarrow R\ (RES\ S')$
by (*auto simp*: *conc-fun-RES dest*: *single-valuedD*)

lemma *SPEC-refine*:

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

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

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

lemma *RETURN-refine*:
assumes $x{\in}Domain\ R$
assumes $\bigwedge x''.\ (x,x''){\in}R \Longrightarrow x''{=}x'$
shows $RETURN\ x \leq \Downarrow R\ (RETURN\ x')$
using *assms* **by** (*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *RETURN-refine-sv*[*refine*]:
assumes *single-valued R*
assumes $(x,x'){\in}R$
shows $RETURN\ x \leq \Downarrow R\ (RETURN\ x')$
apply (*rule RETURN-refine*)
using *assms*
by (*auto dest*: *single-valuedD*)

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

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

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

lemma *bind-refine'*:
fixes $R' :: ('a{\times}'b)\ set$ **and** $R{::}('c{\times}'d)\ set$
assumes *R1*: $M \leq \Downarrow R'\ M'$
assumes *R2*: $\bigwedge x\ x'.\ [\![\ (x,x'){\in}R';\ inres\ M\ x;\ inres\ M'\ x';$
$nofail\ M;\ nofail\ M'$
$]\!] \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
shows $bind\ M\ f \leq \Downarrow R\ (bind\ M'\ f')$
using *assms*

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

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

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

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

Assertions and assumptions are treated specially in bindings

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

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

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

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

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

shows *do* $\{ASSUME\ \Phi;\ S\} \leq \Downarrow R\ S'$
using *assms* **by** (*cases* Φ) *auto*

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

lemmas [*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: $\bigwedge f\ f'\ x\ x'.\ [\![\ \bigwedge x\ x'.\ (x,x')\in R \Longrightarrow f\ x \leq\Downarrow S\ (f'\ x');\ (x,x')\in R\ ]\!]$
    $\Longrightarrow body\ f\ x \leq\Downarrow S\ (body'\ f'\ x')$
  shows $REC\ body\ x \leq\Downarrow S\ (REC\ body'\ x')$
  unfolding *REC-def*
  apply (*clarsimp simp add*: M)
proof –
  assume M': *mono body'*
  have $\forall x\ x'.\ (x,x')\in R \longrightarrow lfp\ body\ x \leq \Downarrow S\ (lfp\ body'\ x')$
    apply (*rule lfp-cadm-induct*[*OF - M*])
      apply *rule+*
      unfolding *Sup-fun-def*
      apply (*rule SUP-least*)
      apply *simp*

      apply (*rule*)+
      apply (*subst lfp-unfold*[*OF M'*])
      apply (*rule RS*)
      apply *blast*
      apply *assumption*
    done
  thus $lfp\ body\ x \leq \Downarrow\ S\ (lfp\ body'\ x')$ **by** (*blast intro*: $R0$)
qed

lemma *RECT-refine*[*refine*]:
  assumes M: *mono body*
  assumes $R0$: $(x,x')\in R$
  assumes RS: $\bigwedge f\ f'\ x\ x'.\ [\![\ \bigwedge x\ x'.\ (x,x')\in R \Longrightarrow f\ x \leq\Downarrow S\ (f'\ x');\ (x,x')\in R\ ]\!]$
    $\Longrightarrow body\ f\ x \leq\Downarrow S\ (body'\ f'\ x')$
  shows $RECT\ body\ x \leq\Downarrow S\ (RECT\ body'\ x')$
  unfolding *RECT-def*
  apply (*clarsimp simp add*: M)
proof –
  assume M': *mono body'*
  have $\forall x\ x'.\ (x,x')\in R \longrightarrow \Uparrow S\ (gfp\ body\ x) \leq (gfp\ body'\ x')$
    apply (*rule gfp-cadm-induct*[*OF - M'*])
      apply *rule+*

unfolding *Inf-fun-def*
apply (*rule INF-greatest*)
apply *simp*

apply (*rule*)+
apply (*subst gfp-unfold*[*OF M*])
apply (*rule RS*[*unfolded ac.galois*])
apply *blast*
apply *assumption*
done
thus *gfp body x* $\leq \Downarrow$ *S* (*gfp body' x'*)
unfolding *ac.galois* **by** (*blast intro*: *R0*)
qed

lemma *if-refine*[*refine*]:
assumes $b \longleftrightarrow b'$
assumes $[\![b;b']\!] \Longrightarrow S1 \leq \Downarrow R\ S1'$
assumes $[\![\neg b;\neg b']\!] \Longrightarrow S2 \leq \Downarrow R\ S2'$
shows (*if b then S1 else S2*) $\leq \Downarrow R$ (*if b' then S1' else S2'*)
using *assms* **by** *auto*

lemma *Let-unfold-refine*[*refine*]:
assumes $f\ x \leq \Downarrow R\ (f'\ x')$
shows *Let x f* $\leq \Downarrow R$ (*Let x' f'*)
using *assms* **by** *auto*

It is safe to split conjunctions in refinement goals.

declare *conjI*[*refine*]

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

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

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

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

2.5.6 Convenience Rules

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

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

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

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

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

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

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

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

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

lemma *RETURN-ref-RETURND*:
  assumes $RETURN\ c \le \Downarrow R\ (RETURN\ a)$
  shows $(c,a) \in R$
  using *assms*
  apply (*auto simp*: *pw-le-iff refine-pw-simps*)
  done

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 ⟪
  structure refine-heuristics-simps = Named-Thms
    ( val name = @{binding refine-hsimp}
      val description = Refinement Framework: ˆ
        Data refinement heuristics simp rules );
⟫
```

setup ⟪ *refine-heuristics-simps.setup* ⟫

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* ⇒ *bool* **where** *RELATES R* ≡ *True*

```
ML ⟪
structure Refine-dref-type = struct
  open Refine-Misc;

  structure pattern-rules = Named-Thms
    ( val name = @{binding refine-dref-pattern}
      val description = Refinement Framework: ˆ
        Pattern rules to recognize refinement goal );

  structure RELATES-rules = Named-Thms (
    val name = @{binding refine-dref-RELATES}
    val description = Refinement Framework: ˆ
        Type based heuristics introduction rules
  );

  val tracing =
    Attrib.setup-config-bool @{binding refine-dref-tracing} (K false);

  (* Check whether term contains schematic variable *)
```

```
fun
  has-schematic (Var -) = true |
  has-schematic (Abs (-,-,t)) = has-schematic t |
  has-schematic (t1$t2) = has-schematic t1 orelse has-schematic t2 |
  has-schematic - = false;

(* Match proof states where the conclusion of some goal has the specified
   shape *)
fun match-goal-shape-tac (shape:term->bool) (ctxt:Proof.context) i thm =
  if Thm.nprems-of thm >= i then
    let
      val t = HOLogic.dest-Trueprop (Logic.concl-of-goal (Thm.prop-of thm) i);
    in
      (if shape t then all-tac thm else no-tac thm)
    end
  else
    no-tac thm;

fun output-failed-msg ctxt failed-t = let
  val failed-t-str = Pretty.string-of
    (Syntax.pretty-term (Config.put show-types true ctxt) failed-t);
  val msg = Failed to resolve refinement goal \n   ^ failed-t-str;
  (*val - = Output.urgent-message (msg);*)
  val - = if Config.get ctxt tracing then Output.tracing msg else ();
  in () end;

(* Try to apply patternI-rules, ensure that produced first subgoal
   contains a schematic variable, and then solve it using
   refine-dref-RELATES-rules. *)
fun type-tac ctxt =
  ALL-GOALS-FWD (TRY o (
    resolve-tac (pattern-rules.get ctxt) THEN'
    match-goal-shape-tac has-schematic ctxt THEN'
    (SOLVED' (REPEAT-ALL-NEW (resolve-tac (RELATES-rules.get ctxt)))
      ORELSE' (fn i => fn st => let
        val failed-t =
          HOLogic.dest-Trueprop (Logic.concl-of-goal (Thm.prop-of st) i);
        val - = output-failed-msg ctxt failed-t;
        in no-tac st end)
    )
  ));


end;
»
```

setup « *Refine-dref-type.RELATES-rules.setup* »
setup « *Refine-dref-type.pattern-rules.setup* »

```
method-setup refine-dref-type =
  ⟨⟨ Args.mode trace -- Args.mode nopost >> (fn (tracing,nopost) =>
    fn ctxt => (let
      val ctxt =
        if tracing then Config.put Refine-dref-type.tracing true ctxt else ctxt;
    in
      SIMPLE-METHOD (CHANGED (
        Refine-dref-type.type-tac ctxt
          THEN (if nopost then all-tac else ALLGOALS (TRY o Refine.post-tac
ctxt))))
    end))
  ⟩⟩
  Use type-based heuristics to instantiate data refinement relations
```

2.6.2 Patterns

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

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

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* **by** (*simp add*: *RELATES-def*)

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*)
  by (*simp add*: *RELATES-def*)

lemma *rprod-sv*[*refine-hsimp*, *refine-post*]:
  ⟦*single-valued R1*; *single-valued R2*⟧ $\Longrightarrow$ *single-valued* (*rprod R1 R2*)
  by (*auto intro*: *single-valuedI dest*: *single-valuedD*)

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*)
  by (*auto intro*: *single-valuedI dest*: *single-valuedD*) []

```
lemma map-set-rel-RELATES[refine-dref-RELATES]:
  RELATES R ⟹ RELATES (map-set-rel R) by (simp add: RELATES-def)

lemma prod-set-eq-is-Id[refine-hsimp]:
  {(a,b). a=b} = Id
  {(a,b). b=a} = Id
  by auto

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

lemmas [refine-hsimp] = Image-Un

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

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

Pointwise refinement for list types:

```
definition [simp]: map-list-rel R ≡ {(l,l'). list-all2 (λx x'. (x,x')∈R) l l'}

lemma map-list-rel-RELATES[refine-dref-RELATES]:
  RELATES R ⟹ RELATES (map-list-rel R) by (simp add: RELATES-def)

lemma map-list-rel-sv-iff-raw:
  single-valued (map-list-rel R) ⟷ single-valued R
  apply (intro iffI[rotated] single-valuedI allI impI)
  apply clarsimp
proof −
  fix x y z
  assume SV: single-valued R
  assume list-all2 (λx x'. (x, x') ∈ R) x y and
    list-all2 (λx x'. (x, x') ∈ R) x z
  thus y=z
    apply (induct arbitrary: z rule: list-all2-induct)
    apply simp
    apply (case-tac z)
    apply force
    apply (force intro: single-valuedD[OF SV])
    done
next
  fix x y z
  assume SV: single-valued (map-list-rel R)
  assume (x,y)∈R    (x,z)∈R
  hence ([x],[y])∈map-list-rel R and ([x],[z])∈map-list-rel R
```

```
    by auto
  with single-valuedD[OF SV] show y=z by (auto simp del: map-list-rel-def)
qed

lemma map-list-rel-sv-iff[simp, refine-hsimp]:
  single-valuedP (list-all2 (λx x'. (x, x') ∈ R)) = single-valued R
  by (rule map-list-rel-sv-iff-raw[simplified])

lemma map-list-rel-sv[refine, refine-post]:
  single-valued R ⟹ single-valued (map-list-rel R)
  by (simp)



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 ≤ m' ⟹ f ≤ f' ⟹ bind m f ≤ bind m' f'
    by (metis (no-types) ibind-mono1 ibind-mono2 le-funD monoD order-trans)


lemma WHILEI-mono: mono (WHILEI-body bind return I b f)
  apply rule
  unfolding WHILEI-body-def
  apply (rule le-funI)
  apply (clarsimp)
  apply (rule-tac x=x and y=y in monoD)
  apply (auto intro: ibind-mono2)
  done


lemma WHILEI-unfold: WHILEI I b f x = (
  if (I x) then (if b x then bind (f x) (WHILEI I b f) else return x) else top)
  unfolding WHILEI-def
  apply (subst REC-unfold[OF WHILEI-mono])
  unfolding WHILEI-body-def
  apply (rule refl)
  done

lemma WHILEI-weaken:
  assumes IW: ⋀x. I x ⟹ I' x
  shows WHILEI I' b f x ≤ WHILEI I b f x
  unfolding WHILEI-def
  apply (rule REC-mono[OF WHILEI-mono])
  apply (auto simp add: WHILEI-body-def dest: IW)
  done

lemma WHILEIT-unfold: WHILEIT I b f x = (
  if (I x) then
    (if b x then bind (f x) (WHILEIT I b f) else return x)
  else top)
  unfolding WHILEIT-def
  apply (subst RECT-unfold[OF WHILEI-mono])
```

unfolding *WHILEI-body-def*
apply (*rule refl*)
done

lemma *WHILEIT-weaken*:
assumes *IW*: $\bigwedge x.\ I\ x \Longrightarrow I'\ x$
shows *WHILEIT* I' *b f x* $\leq$ *WHILEIT I b f x*
unfolding *WHILEIT-def*
apply (*rule RECT-mono*[*OF WHILEI-mono*])
apply (*auto simp add*: *WHILEI-body-def dest*: *IW*)
done

lemma *WHILEI-le-WHILEIT*: *WHILEI I b f s* $\leq$ *WHILEIT I b f s*
unfolding *WHILEI-def WHILEIT-def*
by (*rule REC-le-RECT*)

While without Annotated Invariant

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

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

lemma *transfer-WHILEIT-esc*[*refine-transfer*]:
assumes *REF*: $\bigwedge x.$ *return* (*f x*) $\leq$ *F x*
shows *return* (*while b f x*) $\leq$ *WHILEIT I b F x*
proof −
interpret *transfer return* **.**
show *?thesis*
unfolding *WHILEIT-def*
apply (*rule transfer-RECT'*[**where** *fr=while b f*])
apply (*rule while-unfold*)
unfolding *WHILEI-body-def*
apply (*split split-if*, *intro allI impI conjI*)+
apply *simp-all*
apply (*rule order-trans*[*OF - monoD*[*OF ibind-mono1 REF*, *THEN le-funD*]])
apply (*simp add*: *imonad-laws*)
done
qed

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

lemma *WHILE-mono-prover-rule*[*refine-mono*]:
  notes [*refine-mono*] = *ibind-mono*
  assumes $\bigwedge x$. *f x* $\leq$ *f' x*
  shows *WHILE b f s0* $\leq$ *WHILE b f' s0*
  unfolding *WHILE-def WHILEI-def WHILEI-body-def*
  using *assms* **apply** −
  apply *refine-mono*
  done

lemma *WHILEI-mono-prover-rule*[*refine-mono*]:
  notes [*refine-mono*] = *ibind-mono*
  assumes $\bigwedge x$. *f x* $\leq$ *f' x*
  shows *WHILEI I b f s0* $\leq$ *WHILEI I b f' s0*
  unfolding *WHILE-def WHILEI-def WHILEI-body-def*
  using *assms* **apply** −
  apply *refine-mono*
  done

lemma *WHILET-mono-prover-rule*[*refine-mono*]:
  notes [*refine-mono*] = *ibind-mono*
  assumes $\bigwedge x$. *f x* $\leq$ *f' x*
  shows *WHILET b f s0* $\leq$ *WHILET b f' s0*
  unfolding *WHILET-def WHILEIT-def WHILEI-body-def*
  using *assms* **apply** −
  apply *refine-mono*
  done

lemma *WHILEIT-mono-prover-rule*[*refine-mono*]:
  notes [*refine-mono*] = *ibind-mono*
  assumes $\bigwedge x$. *f x* $\leq$ *f' x*
  shows *WHILEIT I b f s0* $\leq$ *WHILEIT I b f' s0*
  unfolding *WHILET-def WHILEIT-def WHILEI-body-def*
  using *assms* **apply** −
  apply *refine-mono*
  done

end

locale *transfer-WHILE* =
  c!: *generic-WHILE cbind creturn cWHILEIT cWHILEI cWHILET cWHILE* +
  a!: *generic-WHILE abind areturn aWHILEIT aWHILEI aWHILET aWHILE* +
  dist-transfer α

for *cbind* **and** *creturn*::$'a \Rightarrow 'mc$::*complete-lattice*
and *cWHILEIT cWHILEI cWHILET cWHILE*
and *abind* **and** *areturn*::$'a \Rightarrow 'ma$::*complete-lattice*
and *aWHILEIT aWHILEI aWHILET aWHILE*
and $\alpha :: 'mc \Rightarrow 'ma$ +
assumes *transfer-bind*: $[\![\alpha\ m \leq M;\ \bigwedge x.\ \alpha\ (f\ x) \leq F\ x\ ]\!]$
$\Longrightarrow \alpha\ (cbind\ m\ f) \leq abind\ M\ F$
assumes *transfer-return*: $\alpha\ (creturn\ x) \leq areturn\ x$
begin

lemma *transfer-WHILEIT*[*refine-transfer*]:
assumes *REF*: $\bigwedge x.\ \alpha\ (f\ x) \leq F\ x$
shows α (*cWHILEIT I b f x*) $\leq$ *aWHILEIT I b F x*
unfolding *c.WHILEIT-def a.WHILEIT-def*
apply (*rule transfer-RECT*[*OF - c.WHILEI-mono*])
unfolding *WHILEI-body-def*
apply *auto*
apply (*rule transfer-bind*)
apply (*rule REF*)
apply *assumption*
apply (*rule transfer-return*)
done

lemma *transfer-WHILEI*[*refine-transfer*]:
assumes *REF*: $\bigwedge x.\ \alpha\ (f\ x) \leq F\ x$
shows α (*cWHILEI I b f x*) $\leq$ *aWHILEI I b F x*
unfolding *c.WHILEI-def a.WHILEI-def*
apply (*rule transfer-REC*[*OF - c.WHILEI-mono*])
unfolding *WHILEI-body-def*
apply *auto*
apply (*rule transfer-bind*)
apply (*rule REF*)
apply *assumption*
apply (*rule transfer-return*)
done

lemma *transfer-WHILE*[*refine-transfer*]:
assumes *REF*: $\bigwedge x.\ \alpha\ (f\ x) \leq F\ x$
shows α (*cWHILE b f x*) $\leq$ *aWHILE b F x*
unfolding *c.WHILE-def a.WHILE-def*
using *assms* **by** (*rule transfer-WHILEI*)

lemma *transfer-WHILET*[*refine-transfer*]:
assumes *REF*: $\bigwedge x.\ \alpha\ (f\ x) \leq F\ x$
shows α (*cWHILET b f x*) $\leq$ *aWHILET b F x*
unfolding *c.WHILET-def a.WHILET-def*
using *assms* **by** (*rule transfer-WHILEIT*)

end

```
locale generic-WHILE-rules =
  generic-WHILE bind return WHILEIT WHILEI WHILET WHILE
  for bind return SPEC WHILEIT WHILEI WHILET WHILE +
  assumes iSPEC-eq: SPEC Φ = Sup {return x | x. Φ x}
  assumes ibind-rule: ⟦ M ≤ SPEC (λx. f x ≤ SPEC Φ) ⟧ ⟹ bind M f ≤ SPEC
Φ
begin

  lemma ireturn-eq: return x = SPEC (op = x)
    unfolding iSPEC-eq by auto

  lemma iSPEC-rule: (⋀x. Φ x ⟹ Ψ x) ⟹ SPEC Φ ≤ SPEC Ψ
    unfolding iSPEC-eq
    by (auto intro: Sup-mono)

  lemma ireturn-rule: Φ x ⟹ return x ≤ SPEC Φ
    unfolding ireturn-eq
    by (auto intro: iSPEC-rule)

lemma WHILEI-rule:
  assumes I0: I s
  assumes ISTEP: ⋀s. ⟦I s; b s⟧ ⟹ f s ≤ SPEC I
  assumes CONS: ⋀s. ⟦ I s; ¬ b s ⟧ ⟹ Φ s
  shows WHILEI I b f s ≤ SPEC Φ
  apply (rule order-trans[where y=SPEC (λs. I s ∧ ¬ b s)])
    apply (unfold WHILEI-def)
    apply (rule REC-rule[OF WHILEI-mono])
      apply (rule I0)

      unfolding WHILEI-body-def
      apply (split split-if)+
      apply (intro impI conjI)
      apply simp-all
      apply (rule ibind-rule)
        apply (erule (1) order-trans[OF ISTEP])
        apply (rule iSPEC-rule, assumption)

        apply (rule ireturn-rule)
        apply simp

    apply (rule iSPEC-rule)
    apply (simp add: CONS)
  done

lemma WHILEIT-rule:
  assumes WF: wf R
  assumes I0: I s
  assumes IS: ⋀s. ⟦ I s; b s ⟧ ⟹ f s ≤ SPEC (λs'. I s' ∧ (s',s)∈R)
```

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

unfolding *WHILEIT-def*
apply (*rule RECT-rule*[*OF WHILEI-mono WF*, **where** Φ=*I*,*OF I0*])
unfolding *WHILEI-body-def*
apply (*split split-if*)+
apply (*intro impI conjI*)
apply *simp-all*

apply (*rule ibind-rule*)
apply (*rule order-trans*[*OF IS*], *assumption*+)
apply (*rule iSPEC-rule*)
apply *simp*

apply (*rule ireturn-rule*)
apply (*simp add*: *PHI*)
done

lemma *WHILE-rule*:
assumes *I0*: *I s*
assumes *ISTEP*: $\bigwedge s.\ [\![I\ s;\ b\ s]\!] \Longrightarrow f\ s \leq SPEC\ I$
assumes *CONS*: $\bigwedge s.\ [\![\ I\ s;\ \neg\ b\ s\ ]\!] \Longrightarrow \Phi\ s$
shows *WHILE b f s* $\leq$ *SPEC* Φ
unfolding *WHILE-def*
apply (*rule order-trans*[*OF WHILEI-weaken WHILEI-rule*])
apply (*rule TrueI*)
apply (*rule I0*)
using *assms* **by** *auto*

lemma *WHILET-rule*:
assumes *WF*: *wf R*
assumes *I0*: *I s*
assumes *IS*: $\bigwedge s.\ [\![\ I\ s;\ b\ s\ ]\!] \Longrightarrow f\ s \leq SPEC\ (\lambda s'.\ I\ s' \wedge (s',s) \in R)$
assumes *PHI*: $\bigwedge s.\ [\![\ I\ s;\ \neg\ b\ s\ ]\!] \Longrightarrow \Phi\ s$
shows *WHILET b f s* $\leq$ *SPEC* Φ
unfolding *WHILET-def*
apply (*rule order-trans*[*OF WHILEIT-weaken WHILEIT-rule*])
apply (*rule TrueI*)
apply (*rule WF*)
apply (*rule I0*)
using *assms* **by** *auto*

end

end

2.8 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
apply *unfold-locales*
apply (*simp-all add*: *WHILEIT-def WHILEI-def WHILET-def WHILE-def*)
apply (*subst RES-Sup-RETURN*[*symmetric*])
apply (*rule-tac f=Sup* **in** *arg-cong*) **apply** *auto* []
apply (*erule bind-rule*)
done

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

2.8.1 Data Refinement Rules

lemma *ref-WHILEI-invarI*:
assumes $I\ s \Longrightarrow M \le \Downarrow R\ (WHILEI\ I\ b\ f\ s)$
shows $M \le \Downarrow R\ (WHILEI\ I\ b\ f\ s)$
apply (*subst WHILEI-unfold*)
apply (*cases I s*)
apply (*subst WHILEI-unfold*[*symmetric*])
apply (*erule assms*)
apply *simp*
done

lemma *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 $WHILEI\ I\ b\ f\ s \le M\ s'$
unfolding *WHILEI-def*
apply (*rule REC-le-rule*[**where** *M=M*])
apply (*rule WHILEI-mono*)
apply (*rule R0*)
apply (*erule* (*1*) *order-trans*[*rotated*,*OF RS*])
unfolding *WHILEI-body-def*
apply *auto*
done

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

lemma *WHILEI-invisible-refine*:
  assumes *R0*: $(s,s')\in R$
  assumes *SV*: *single-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 $WHILEI\ I\ b\ f\ s \le \Downarrow R\ (WHILEI\ I'\ b'\ f'\ s')$
  apply (*rule WHILEI-le-rule*[**where** *R=R*])
  apply (*rule R0*)
  apply (*rule ref-WHILEI-invarI*)
  apply (*frule* (*1*) *RI*)
  apply (*case-tac b s=False*)
  apply (*subst WHILEI-unfold*)
  apply (*auto dest*: *RB intro*: *RETURN-refine-sv*[*OF SV*]) []

  apply *simp*
  apply (*rule order-trans*[*OF monoD*[*OF bind-mono1 RS*]], *assumption+*)
  apply (*simp only*: *bind-distrib-sup*)
  apply (*rule sup-least*)
    apply (*subst WHILEI-unfold*)
    apply (*simp add*: *RB*, *intro impI*)
    apply (*rule bind-refine*)
    apply (*rule order-refl*)
    apply *simp*

    apply (*simp add*: *pw-le-iff refine-pw-simps*, *blast*)
  done

lemma *WHILEI-refine*[*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*:
    $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow f\ x \le \Downarrow R\ (f'\ x')$
  shows $WHILEI\ I\ b\ f\ x \le\Downarrow R\ (WHILEI\ I'\ b'\ f'\ x')$
  apply (*rule WHILEI-invisible-refine*[*OF R0 SV*])
  apply (*erule* (*1*) *IREF*)
  apply (*simp add*: *COND-REF*)
  apply (*rule le-supI1*)
  apply (*simp add*: *COND-REF STEP-REF*)
  done

lemma *WHILE-invisible-refine*:
  assumes *R0*: $(s,s')\in R$

assumes SV: *single-valued* R
assumes RB: $\bigwedge s\ s'.\ [\![\ (s,s')\in R;\ b'\ s'\ ]\!] \Longrightarrow b\ s$
assumes RS: $\bigwedge s\ s'.\ [\![\ (s,s')\in R;\ b\ s\ ]\!]$
$\Longrightarrow f\ s \leq sup\ (\Downarrow R\ (do\ \{ASSUME\ (b'\ s');\ f'\ s'\}))\ (\Downarrow R\ (RETURN\ s'))$
shows $WHILE\ b\ f\ s \leq \Downarrow R\ (WHILE\ b'\ f'\ s')$
unfolding *WHILE-def*
apply (*rule WHILEI-invisible-refine*)
using *assms* **by** *auto*

lemma *WHILE-le-rule*:
assumes $R0$: $(s,s')\in R$
assumes RS: $\bigwedge W\ s\ s'.\ [\![\bigwedge s\ s'.\ (s,s')\in R \Longrightarrow W\ s \leq M\ s';\ (s,s')\in R]\!] \Longrightarrow$
$do\ \{if\ b\ s\ then\ bind\ (f\ s)\ W\ else\ RETURN\ s\} \leq M\ s'$
shows $WHILE\ b\ f\ s \leq M\ s'$
unfolding *WHILE-def*
apply (*rule WHILEI-le-rule*[*OF R0*])
apply (*simp add*: *RS*)
done

lemma *WHILE-refine*[*refine*]:
assumes $R0$: $(x,x')\in R$
assumes SV: *single-valued* R
assumes $COND\text{-}REF$: $\bigwedge x\ x'.\ [\![\ (x,x')\in R\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes $STEP\text{-}REF$:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
shows $WHILE\ b\ f\ x \leq\Downarrow R\ (WHILE\ b'\ f'\ x')$
unfolding *WHILE-def*
apply (*rule WHILEI-refine*)
using *assms* **by** *auto*

lemma *WHILE-refine'*[*refine*]:
assumes $R0$: $(x,x')\in R$
assumes SV: *single-valued* R
assumes $COND\text{-}REF$: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I'\ x'\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes $STEP\text{-}REF$:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
shows $WHILE\ b\ f\ x \leq\Downarrow R\ (WHILEI\ I'\ b'\ f'\ x')$
unfolding *WHILE-def*
apply (*rule WHILEI-refine*)
using *assms* **by** *auto*

lemma *AIF-leI*: $[\![\Phi;\ \Phi \Longrightarrow S\leq S']\!] \Longrightarrow (if\ \Phi\ then\ S\ else\ FAIL) \leq S'$
by *auto*
lemma *ref-AIFI*: $[\![\Phi \Longrightarrow S\leq\Downarrow R\ S']\!] \Longrightarrow S \leq \Downarrow R\ (if\ \Phi\ then\ S'\ else\ FAIL)$
by (*cases* Φ) *auto*

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*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow f\ x \le \Downarrow R\ (f'\ x')$
shows *WHILEIT I b f x* $\le\Downarrow R$ (*WHILEIT I' b' f' x'*)
unfolding *WHILEIT-def*
apply (*rule RECT-refine*[*OF WHILEI-mono R0*])
unfolding *WHILEI-body-def*
apply (*rule ref-AIFI*)
apply (*rule AIF-leI*)
apply (*blast intro*: *IREF*)
apply (*rule if-refine*)
apply (*simp add*: *COND-REF*)
apply (*rule bind-refine*)
apply (*rule STEP-REF*, *assumption+*) []
apply *assumption*

apply (*simp add*: *RETURN-refine-sv*[*OF SV*])
done

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*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x'\ ]\!] \Longrightarrow f\ x \le \Downarrow R\ (f'\ x')$
shows *WHILET b f x* $\le\Downarrow R$ (*WHILET b' f' x'*)
unfolding *WHILET-def*
apply (*rule WHILEIT-refine*)
using *assms* **by** *auto*

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*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I'\ x'\ ]\!] \Longrightarrow f\ x \le \Downarrow R\ (f'\ x')$
shows *WHILET b f x* $\le\Downarrow R$ (*WHILEIT I' b' f' x'*)
unfolding *WHILET-def*
apply (*rule WHILEIT-refine*)
using *assms* **by** *auto*

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*:
$it \subseteq S \Longrightarrow x{\in}it \Longrightarrow S-(it-\{x\}) = insert\ x\ (S-it)$ **by** *auto*

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

Foreach with continuation condition and annotated invariant:

definition *FOREACHci* ($FOREACH_{C}$ˆ) **where** *FOREACHci* $\Phi\ S \equiv$ *FOREACHoci* $\Phi\ S\ (\lambda\text{-}\ \text{-}.\ True)$

Foreach with continuation condition:

definition *FOREACHc* ($FOREACH_{C}$) **where** *FOREACHc* $\equiv$ *FOREACHci* $(\lambda\text{-}.\ True)$

Foreach with annotated invariant:

definition *FOREACHi* (*FOREACH*$^{-}$) **where**
  $FOREACHi\ \Phi\ S\ f\ \sigma 0 \equiv FOREACHci\ \Phi\ S\ (\lambda\text{-}.\ True)\ f\ \sigma 0$

Foreach with annotated invariant and order:

definition *FOREACHoi* ($FOREACH_O{}^{-}$) **where**
  $FOREACHoi\ \Phi\ S\ R\ f\ \sigma 0 \equiv FOREACHoci\ \Phi\ S\ R\ (\lambda\text{-}.\ True)\ f\ \sigma 0$

Basic foreach

definition $FOREACH\ S\ f\ \sigma 0 \equiv FOREACHc\ S\ (\lambda\text{-}.\ True)\ f\ \sigma 0$

2.9.3 Proof Rules

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

  apply (*rule FIN*)

  apply (*subgoal-tac wf* (*measure* (λ(*xs*, -). *length xs*)))
    apply *assumption*
    apply *simp*

  apply (*insert I0*, *simp add*: *I0*) []
  unfolding *FOREACH-body-def FOREACH-cond-def*
  apply (*rule refine-vcg*)+
  apply (*simp*, *elim conjE exE*)+
  apply (*rename-tac* $xs''\ s\ xs'\ \sigma\ xs$)
defer
  apply (*simp*, *elim conjE exE*)+
  apply (*rename-tac* $x\ s\ xs'\ \sigma\ \sigma'\ xs$)
defer
proof −
  fix $xs'\ \sigma\ xs$

  assume *I-xs'*: $I\ (set\ xs')\ \sigma$
    and *sorted-xs-xs'*: $sorted\text{-}by\text{-}rel\ R\ (xs\ @\ xs')$
    and *dist*: $distinct\ xs \quad distinct\ xs' \quad set\ xs \cap set\ xs' = \{\}$
    and *S-eq*: $S = set\ xs \cup set\ xs'$

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

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

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

  from *IP*[*of σ x set xs′*, *OF* ‹*c σ*› *x-in-xs′* ‹*set xs′* ⊆ *S*› ‹*I* (*set xs′*) *σ*›] *x-nin-xs″*
      sorted-xs-xs′ S-diff
  show *f* (*hd xs′*) *σ* ≤ *SPEC*
        (λ*x*. (∃*xs′a*. *xs* @ *xs′* = *xs′a* @ *tl xs′*) ∧
            I (*set* (*tl xs′*)) *x*)
    apply (*simp add*: *xs′-eq*)
    apply (*simp add*: *sorted-by-rel-append*)
  done
}

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

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

lemma *FOREACHoi-rule*[*refine-vcg*]:
  assumes *FIN*: *finite S*
  assumes *I0*: *I S σ0*
  assumes *IP*:
    ⋀*x it σ*. ⟦ *x*∈*it*; *it*⊆*S*; *I it σ*; ∀*y*∈*it* − {*x*}. *R x y*;
            ∀*y*∈*S* − *it*. *R y x* ⟧ ⟹ *f x σ* ≤ *SPEC* (*I* (*it*−{*x*}))
  assumes *II1*: ⋀*σ*. ⟦*I* {} *σ*⟧ ⟹ *P σ*
  shows *FOREACHoi I S R f σ0* ≤ *SPEC P*
  unfolding *FOREACHoi-def*
  by (*rule FOREACHoci-rule*) (*simp-all add*: *assms*)

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

assumes *I0*: $I\ S\ \sigma 0$
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ [\![\ x{\in}it;\ it{\subseteq}S;\ I\ it\ \sigma;\ c\ \sigma\ ]\!] \Longrightarrow f\ x\ \sigma \le SPEC\ (I\ (it{-}\{x\}))$
assumes *II1*: $\bigwedge\sigma.\ [\![I\ \{\}\ \sigma]\!] \Longrightarrow P\ \sigma$
assumes *II2*: $\bigwedge it\ \sigma.\ [\![\ it{\neq}\{\};\ it{\subseteq}S;\ I\ it\ \sigma;\ \neg c\ \sigma\ ]\!] \Longrightarrow P\ \sigma$
shows $FOREACHci\ I\ S\ c\ f\ \sigma 0 \le SPEC\ P$
unfolding *FOREACHci-def*
by (*rule FOREACHoci-rule*) (*simp-all add*: *assms*)

lemma *FOREACHoci-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 *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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![\forall y{\in}it{-}\{x\}.\ RR\ x\ y;$
$x'{=}\alpha\ x;\ x{\in}it;\ x'{\in}it';\ it'{=}\alpha`it;\ it{\subseteq}S;\ it'{\subseteq}S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ \ \Phi''\ it\ \sigma\ it'\ \sigma';\ c\ \sigma;\ c'\ \sigma';$
$(\sigma,\sigma'){\in}R$
$]\!] \Longrightarrow f\ x\ \sigma$
$\le \Downarrow(\{(\sigma,\ \sigma').\ (\sigma,\ \sigma') \in R \wedge \Phi''\ (it - \{x\})\ \sigma\ (it' - \{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows $FOREACHoci\ \Phi\ S\ RR\ c\ f\ \sigma 0 \le \Downarrow R\ (FOREACHoci\ \Phi'\ S'\ RR'\ c'\ f'\ \sigma 0')$
unfolding *FOREACHoci-def*
apply (*rule ASSERT-refine-right ASSERT-refine-left*)+
using *REFS INJ* **apply** (*auto dest*: *finite-imageD*) []

apply (*rule bind-refine*)
apply (*subgoal-tac*
$SPEC\ (\lambda xs.\ distinct\ xs \wedge S = set\ xs \wedge sorted\text{-}by\text{-}rel\ RR\ xs) \le$
$\Downarrow \{(xs,\ xsa).\ xsa = map\ \alpha\ xs \wedge sorted\text{-}by\text{-}rel\ RR\ xs \wedge distinct\ xs \wedge distinct\ xsa \wedge set\ xs = S \wedge set\ xsa = S'\}$
$(SPEC\ (\lambda xs.\ distinct\ xs \wedge S' = set\ xs \wedge sorted\text{-}by\text{-}rel\ RR'\ xs)))$
apply *assumption*
apply (*rule SPEC-refine-sv*)
apply (*simp add*: *single-valued-def*)
apply (*rule SPEC-rule*)
apply (*insert INJ*, *simp add*: *REFS distinct-map sorted-by-rel-map*)[]
apply (*rule sorted-by-rel-weaken*[*of* - *RR*])
apply (*simp add*: *RR-OK*)

apply (*simp*)

apply (*rule bind-refine*)
apply (*rule-tac WHILEIT-refine*)
apply (*subgoal-tac* ((x, $\sigma 0$), x', $\sigma 0'$)$\in${((xs,σ),(xs',σ')). *sorted-by-rel RR xs* $\wedge$ xs'=*map* α *xs* $\wedge$ *set xs* $\subseteq$ S $\wedge$ *set* xs' $\subseteq$ S' $\wedge$ (σ,σ')$\in$$R$ $\wedge$ Φ'' (*set xs*) σ (*set* xs') σ'}, *assumption*)
apply (*simp add*: *REFS REF0 REFPHI0*)

apply (*auto intro*: *single-valuedI single-valuedD*[*OF SV*]) []

using *REFPHI*
apply (*simp*)
apply (*clarify*)
apply (*rule conjI*)
prefer *2*
apply *auto*[]
defer
using *REFC* **apply** (*clarify*) **apply** (*auto simp add*: *FOREACH-cond-def*) []

apply (*simp add*: *FOREACH-body-def Let-def split*: *prod.splits*) []
apply (*rule bind-refine*)
apply (*case-tac a*, *simp add*: *FOREACH-cond-def*) []
apply (*rule REFSTEP*)
apply (*subgoal-tac* $\forall y \in$*set* $a - \{$*hd* $a\}$. *RR* (*hd a*) *y*, *assumption*)
apply *simp*
apply *simp*
apply *simp*
apply (*subgoal-tac hd* (*map* α *a*)$\in$*set aa*, *assumption*)
apply *simp-all*[*3*]
apply *fast*
apply *simp*
apply *simp*
apply *fast*
apply (*simp add*: *FOREACH-cond-def*)
apply (*simp add*: *FOREACH-cond-def*)
apply *simp*
apply (*rule RETURN-refine-sv*)
apply (*rule single-valuedI*)
apply (*auto simp add*: *single-valuedD*[*OF SV*]) []
apply (*case-tac a*, *simp*)
apply (*clarify*, *simp*)

apply (*split prod.split*, *intro allI impI*)+
apply (*rule RETURN-refine-sv*[*OF SV*])
apply (*auto*)[]
proof −
fix *xs* axs' xs''
assume *map-eq*: *map* α *xs* = axs' @ *map* α xs''

```
    and xs''-subset: set xs'' ⊆ set xs
    and dist-xs: distinct (map α xs)
  hence axs' = take (length axs') (map α xs) by (metis append-eq-conv-conj)
  hence axs'-eq: axs' = map α (take (length axs') xs) by (simp add: take-map)

  from map-eq axs'-eq have map-eq: map α xs = map α ((take (length axs') xs)
@ xs'')
    by (metis map-append)

  have inj-α: inj-on α (set xs ∪ set ((take (length axs') xs) @ xs''))
  proof -
    from xs''-subset
    have (set xs ∪ set ((take (length axs') xs) @ xs'')) = set xs
      by (auto elim: in-set-takeD)

    with dist-xs show ?thesis
      by (simp add: distinct-map)
  qed

  from inj-on-map-eq-map [OF inj-α] map-eq
  have xs = (take (length axs') xs) @ xs'' by blast
  thus ∃xs'. xs = xs' @ xs'' by blast
qed

lemma FOREACHoci-refine-rcg[refine]:
  fixes α :: 'S ⇒ 'Sa
  fixes S :: 'S set
  fixes S' :: 'Sa set
  assumes INJ: inj-on α S
  assumes REFS: S' = α`S
  assumes REF0: (σ0,σ0')∈R
  assumes RR-OK: ⋀x y. ⟦x ∈ S; y ∈ S; RR x y⟧ ⟹ RR' (α x) (α y)
  assumes SV: single-valued R
  assumes REFC: ⋀it σ it' σ'. ⟦
    it'=α`it; it⊆S; it'⊆S'; Φ' it' σ'; Φ it σ; (σ,σ')∈R
  ⟧ ⟹ c σ ⟷ c' σ'
  assumes REFPHI: ⋀it σ it' σ'. ⟦
    it'=α`it; it⊆S; it'⊆S'; Φ' it' σ'; (σ,σ')∈R
  ⟧ ⟹ Φ it σ
  assumes REFSTEP: ⋀x it σ x' it' σ'. ⟦ ∀y∈it−{x}. RR x y;
    x'=α x; x∈it; x'∈it'; it'=α`it; it⊆S; it'⊆S';
    Φ it σ; Φ' it' σ'; c σ; c' σ';
    (σ,σ')∈R
  ⟧ ⟹ f x σ ≤ ⇓R (f' x' σ')
  shows FOREACHoci Φ S RR c f σ0 ≤ ⇓R (FOREACHoci Φ' S' RR' c' f' σ0')
  apply (rule FOREACHoci-refine[where Φ''=λ- - - -. True])
  apply (rule assms)+
  using assms by simp-all
```

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

lemma *FOREACHoci-weaken-order*:
assumes *RRREF*: $\bigwedge x\ y.\ x \in S \Longrightarrow y \in S \Longrightarrow RR\ x\ y \Longrightarrow RR'\ x\ y$
shows *FOREACHoci* $I\ S\ RR\ c\ f\ \sigma 0 \leq$ *FOREACHoci* $I\ S\ RR'\ c\ f\ \sigma 0$
apply (*rule FOREACHoci-refine-rcg*[**where** $\alpha = id$ **and** $R = Id$, *simplified*])
apply (*auto intro*: *RRREF*)
done

Rules for Derived Constructs

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

lemma *FOREACHoi-refine-rcg*[*refine*]:
fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S\ set$
fixes $S' :: 'Sa\ set$
assumes *INJ*: *inj-on* $\alpha\ S$
assumes *REFS*: $S' = \alpha`S$
assumes *REF0*: $(\sigma 0, \sigma 0') \in R$
assumes *RR-OK*: $\bigwedge x\ y.\ [\![x \in S;\ y \in S;\ RR\ x\ y]\!] \Longrightarrow RR'\ (\alpha\ x)\ (\alpha\ y)$
assumes *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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![\ \forall y{\in}it{-}\{x\}.\ RR\ x\ y;$
$x'{=}\alpha\ x;\ x{\in}it;\ x'{\in}it';\ it'{=}\alpha`it;\ it{\subseteq}S;\ it'{\subseteq}S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';\ (\sigma{,}\sigma'){\in}R$
$]\!] \Longrightarrow f\ x\ \sigma \le \Downarrow R\ (f'\ x'\ \sigma')$
shows $FOREACHoi\ \Phi\ S\ RR\ f\ \sigma 0 \le \Downarrow R\ (FOREACHoi\ \Phi'\ S'\ RR'\ f'\ \sigma 0')$
apply (*rule FOREACHoi-refine*[**where** $\Phi''{=}\lambda\text{- - - -}.\ True$])
apply (*rule assms*)+
using *assms* **by** *simp-all*

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

lemma *FOREACHci-refine-rcg*[*refine*]:
fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S\ set$
fixes $S' :: 'Sa\ set$
assumes *INJ*: $inj\text{-}on\ \alpha\ S$
assumes *REFS*: $S' = \alpha`S$
assumes *REF0*: $(\sigma 0{,}\sigma 0'){\in}R$
assumes *SV*: $single\text{-}valued\ R$
assumes *REFC*: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'{=}\alpha`it;\ it{\subseteq}S;\ it'{\subseteq}S';\ \Phi'\ it'\ \sigma';\ \Phi\ it\ \sigma;\ (\sigma{,}\sigma'){\in}R$

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

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

lemma *FOREACHi-rule*[*refine-vcg*]:
assumes *FIN*: *finite S*
assumes *I0*: $I\ S\ \sigma 0$
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 $FOREACHi\ I\ S\ f\ \sigma 0 \le SPEC\ P$
unfolding *FOREACHi-def*
apply (*rule FOREACHci-rule*[*of S I*])
using *assms* **by** *auto*

lemma *FOREACHc-rule*:
assumes *FIN*: *finite S*
assumes *I0*: $I\ S\ \sigma 0$
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ \llbracket\ x{\in}it;\ it{\subseteq}S;\ I\ it\ \sigma\ \rrbracket \Longrightarrow f\ x\ \sigma \le SPEC\ (I\ (it{-}\{x\}))$
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 $FOREACHc\ S\ c\ f\ \sigma 0 \le SPEC\ P$
unfolding *FOREACHc-def*
apply (*rule order-trans*[*OF FOREACHci-weaken*], *rule TrueI*)
apply (*rule FOREACHci-rule*[**where** $I{=}I$])
using *assms* **by** *auto*

lemma *FOREACH-rule*:
assumes *FIN*: *finite S*
assumes *I0*: $I\ S\ \sigma 0$
assumes *IP*:

$\bigwedge x\ it\ \sigma.\ [\![\ x{\in}it;\ it{\subseteq}S;\ I\ it\ \sigma\ ]\!] \Longrightarrow f\ x\ \sigma \leq SPEC\ (I\ (it{-}\{x\}))$
assumes II: $\bigwedge\sigma.\ [\![I\ \{\}\ \sigma]\!] \Longrightarrow P\ \sigma$
shows $FOREACH\ S\ f\ \sigma 0 \leq SPEC\ P$
unfolding *FOREACH-def FOREACHc-def*
apply (*rule order-trans*[*OF FOREACHci-weaken*], *rule TrueI*)
apply (*rule FOREACHci-rule*[**where** $I{=}I$])
using *assms* **by** *auto*

lemma *FOREACHc-refine*:
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 $REFC$: $\bigwedge it\ \sigma\ it'\ \sigma'.\ [\![$
$it'{=}\alpha`it;\ it{\subseteq}S;\ it'{\subseteq}S';\ \Phi''\ it\ \sigma\ it'\ \sigma';\ (\sigma,\sigma'){\in}R$
$]\!] \Longrightarrow c\ \sigma \longleftrightarrow c'\ \sigma'$
assumes $REFSTEP$: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'{=}\alpha\ x;\ x{\in}it;\ x'{\in}it';\ it'{=}\alpha`it;\ it{\subseteq}S;\ it'{\subseteq}S';$
$\Phi''\ it\ \sigma\ it'\ \sigma';\ c\ \sigma;\ c'\ \sigma';\ (\sigma,\sigma'){\in}R$
$]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma,\ \sigma').\ (\sigma,\ \sigma') \in R \wedge \Phi''\ (it - \{x\})\ \sigma\ (it' - \{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows $FOREACHc\ S\ c\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHc\ S'\ c'\ f'\ \sigma 0')$
unfolding *FOREACHc-def*
apply (*rule FOREACHci-refine*[**where** $\Phi''{=}\Phi''$, *OF INJ REFS REF0 SV REF-PHI0*])
apply (*erule* (*4*) *REFC*)
apply (*rule TrueI*)
apply (*erule* (*9*) *REFSTEP*)
done

lemma *FOREACHc-refine-rcg*[*refine*]:
fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S\ set$
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 $FOREACHc\ S\ c\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHc\ S'\ c'\ f'\ \sigma 0')$
unfolding *FOREACHc-def*
apply (*rule FOREACHci-refine-rcg*)
apply (*rule assms*)+
using *assms* **by** *auto*

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 $FOREACHi\ \Phi\ S\ f\ \sigma 0 \leq \Downarrow R\ (FOREACHi\ \Phi'\ S'\ f'\ \sigma 0')$
unfolding *FOREACHi-def*
apply (*rule FOREACHci-refine*[**where** $\Phi''=\Phi''$, *OF INJ REFS REF0 SV REF-PHI0*])
apply (*rule refl*)
apply (*erule* (*5*) *REFPHI*)
apply (*erule* (*9*) *REFSTEP*)
done

lemma *FOREACHi-refine-rcg*[*refine*]:
fixes $\alpha :: 'S \Rightarrow 'Sa$
fixes $S :: 'S\ set$
fixes $S' :: 'Sa\ set$
assumes *INJ*: $inj\text{-}on\ \alpha\ S$
assumes *REFS*: $S' = \alpha`S$
assumes *REF0*: $(\sigma 0,\sigma 0')\in R$
assumes *SV*: $single\text{-}valued\ R$
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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi\ it\ \sigma;\ \Phi'\ it'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \leq \Downarrow R\ (f'\ x'\ \sigma')$

shows *FOREACHi* Φ *S f* $\sigma 0 \leq \Downarrow R$ (*FOREACHi* Φ' S' f' $\sigma 0'$)
unfolding *FOREACHi-def*
apply (*rule FOREACHci-refine-rcg*)
apply (*rule assms*)+
using *assms* **apply** *auto*
done

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*: Φ'' S $\sigma 0$ (α‘S) $\sigma 0'$
assumes *REFSTEP*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'$. $[\![$
$x'=\alpha\ x$; $x\in it$; $x'\in it'$; $it'=\alpha$‘it; $it\subseteq S$; $it'\subseteq S'$;
$\Phi''\ it\ \sigma\ it'\ \sigma'$; $(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow(\{(\sigma, \sigma').\ (\sigma, \sigma') \in R \wedge \Phi''\ (it - \{x\})\ \sigma\ (it' - \{x'\})\ \sigma'\})\ (f'\ x'\ \sigma')$
shows *FOREACH S f* $\sigma 0 \leq \Downarrow R$ (*FOREACH* S' f' $\sigma 0'$)
unfolding *FOREACH-def*
apply (*rule FOREACHc-refine*[**where** $\Phi''=\Phi''$, *OF INJ REFS REF0 SV REF-PHI0*])
apply (*rule refl*)
apply (*erule* (*7*) *REFSTEP*)
done

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; $it\subseteq S$; $it'\subseteq S'$;
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \leq \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACH S f* $\sigma 0 \leq \Downarrow R$ (*FOREACH* S' f' $\sigma 0'$)
unfolding *FOREACH-def*
apply (*rule FOREACHc-refine-rcg*)
apply (*rule assms*)+
using *assms* **by** *auto*

lemma *FOREACHci-refine-rcg'*[*refine*]:
fixes 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 *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 $FOREACHc\ S\ c\ f\ \sigma 0 \le \Downarrow R\ (FOREACHci\ \Phi'\ S'\ c'\ f'\ \sigma 0')$
unfolding *FOREACHc-def*
apply (*rule FOREACHci-refine-rcg*)
apply (*rule assms*)
apply (*rule assms*)
apply (*rule assms*)
apply (*rule assms*)
apply (*erule* (*4*) *REFC*)
apply (*rule TrueI*)
apply (*rule REFSTEP*, *assumption+*)
done

lemma *FOREACHi-refine-rcg'*[*refine*]:
fixes 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*: $\bigwedge x\ it\ \sigma\ x'\ it'\ \sigma'.\ [\![$
$x'=\alpha\ x;\ x\in it;\ x'\in it';\ it'=\alpha`it;\ it\subseteq S;\ it'\subseteq S';$
$\Phi'\ it'\ \sigma';$
$(\sigma,\sigma')\in R$
$]\!] \Longrightarrow f\ x\ \sigma \le \Downarrow R\ (f'\ x'\ \sigma')$
shows $FOREACH\ S\ f\ \sigma 0 \le \Downarrow R\ (FOREACHi\ \Phi'\ S'\ f'\ \sigma 0')$
unfolding *FOREACH-def FOREACHi-def*
apply (*rule FOREACHci-refine-rcg'*)
apply (*rule assms*)+
apply *simp*
apply (*rule REFSTEP*, *assumption+*)
done

2.9.4 FOREACH with empty sets

lemma *FOREACHoci-emp* [*simp*] :
  $FOREACHoci\ \Phi\ \{\}\ R\ c\ f\ \sigma = do\ \{ASSERT\ (\Phi\ \{\}\ \sigma);\ RETURN\ \sigma\}$
apply (*simp add*: *FOREACHoci-def bind-RES image-def*)
apply (*simp add*: *WHILEIT-unfold FOREACH-cond-def*)
done

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

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

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

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

lemma *FOREACHi-emp* [*simp*] :
  $FOREACHi\ \Phi\ \{\}\ f\ \sigma = do\ \{ASSERT\ (\Phi\ \{\}\ \sigma);\ RETURN\ \sigma\}$
by (*simp add*: *FOREACHi-def*)

locale *transfer-FOREACH* = *transfer* +
  constrains $\alpha :: 'mc \Rightarrow 's\ nres$
  fixes $creturn :: 's \Rightarrow 'mc$
  and $cbind :: 'mc \Rightarrow ('s \Rightarrow 'mc) \Rightarrow 'mc$
  and $liftc :: ('s \Rightarrow bool) \Rightarrow 'mc \Rightarrow bool$

  assumes *transfer-bind*: $[\![\alpha\ m \le M;\ \bigwedge x.\ \alpha\ (f\ x) \le F\ x\ ]\!]$
    $\Longrightarrow \alpha\ (cbind\ m\ f) \le bind\ M\ F$
  assumes *transfer-return*: $\alpha\ (creturn\ x) \le RETURN\ x$

  assumes *liftc*: $[\![RETURN\ x \le \alpha\ m;\ \alpha\ m \ne FAIL]\!] \Longrightarrow liftc\ c\ m \longleftrightarrow c\ x$
begin

abbreviation *lift-it* :: $('x,'mc)\ set\text{-}iterator \Rightarrow$
  $('s \Rightarrow bool) \Rightarrow ('x \Rightarrow 's \Rightarrow 'mc) \Rightarrow 's \Rightarrow 'mc$
  where
  $lift\text{-}it\ it\ c\ f\ s0 \equiv it$
    $(liftc\ c)$
    $(\lambda x\ s.\ cbind\ s\ (f\ x))$
    $(creturn\ s0)$

```
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 –
  from IT obtain xs where xs-props:
    distinct xs   X = set xs   sorted-by-rel R xs   iterate = foldli xs
    unfolding set-iterator-genord-def by blast

  def RL≡λ(it,s). WHILE (FOREACH-cond c) (FOREACH-body F) (it, s) ≫=
      (λ(-, σ). RETURN σ)
  hence RL-def':
    ⋀it s. RL (it,s) = WHILE (FOREACH-cond c) (FOREACH-body F) (it, s)
≫=
    (λ(-, σ). RETURN σ) by auto

  {
    fix it s
    have RL (it, s) = (
      if (FOREACH-cond c (it,s)) then
        (FOREACH-body F (it,s) ≫= RL)
      else
        RETURN s)
      unfolding RL-def
      apply (subst WHILE-unfold)
      apply (auto simp: pw-eq-iff refine-pw-simps)
      done
  } note RL-unfold = this

  {
    fix it::'x list and x and s :: 's
    assume C: c s
    have do{ s'←F x s; RL (it, s')} ≤ RL (x # it,s)
      apply (subst (2) RL-unfold)
      unfolding FOREACH-cond-def FOREACH-body-def
      apply (rule pw-leI)
      using C apply (simp add: refine-pw-simps)
      done
  } note RL-unfold-step=this

  have RL-le: RL (xs,s0) ≤ FOREACHoci I X R c F s0
    unfolding FOREACHoci-def
    apply (rule le-ASSERTI)
    apply (rule bind2let-refine [of xs Id - λ-. RL(xs,s0) Id, simplified])
    apply (simp add: xs-props)
   apply (rule order-trans[OF - monoD[OF bind-mono1 WHILEI-le-WHILEIT]])
     apply (rule order-trans[OF - monoD[OF bind-mono1 WHILEI-weaken[OF
TrueI]]])
```

```
  apply (fold WHILE-def)
  apply (fold RL-def')
  by simp

{ fix S xs'
  have do { s←α S; RL (xs',s) } ≤ RL (xs,s0) ⟹
      α (foldli xs' (liftc c) (λx s. cbind s (f x)) S) ≤
      RL (xs, s0)
  proof (induct xs' arbitrary: S)
    case (Nil S) thus ?case
      by (simp add: RL-unfold FOREACH-cond-def)
    next
      case (Cons x xs' S)
      note ind-hyp = Cons(1)
      note pre = Cons(2)

      show ?case
      proof (cases α S)
        case FAIL with pre have RL (xs, s0) = FAIL by simp
        thus ?thesis by simp
      next
        case (RES S') note α-S-eq[simp] = this

        from transfer-bind [of S RES S'   f x   F x, OF - RS]
        have α-bind: α (cbind S (f x)) ≤ RES S' ≫= F x by simp

        { fix s
          assume s ∈ S'
          with liftc [of s S c]
          have c s = liftc c S by simp
        } note liftc-intro = this

        show ?thesis
        proof (cases liftc c S)
          case False note not-liftc = this
          with liftc-intro have ⋀s. c s ⟹ s ∉ S' by auto

          hence RES S' ≫= (λs. RL (x # xs', s)) = RES S'
            apply (simp add: bind-def Sup-nres-def image-iff RL-unfold
                         FOREACH-cond-def Ball-def Bex-def)
            apply (auto)
          done
          with pre not-liftc show ?thesis by simp
        next
          case True note liftc = this

          with liftc-intro have S'-simps: S' ∩ {s. c s} = S'   S' ∩ {s. ¬ c s} =
{}
            by auto
```

have $((\alpha\ (cbind\ S\ (f\ x))) \gg= (\lambda s.\ RL\ (xs', s))) \leq$
$((RES\ S' \gg= F\ x) \gg= (\lambda s.\ RL\ (xs', s)))$
by (*rule bind-mono*) (*simp-all add*: α*-bind*)
also have ... = $RES\ S' \gg= (\lambda s.\ F\ x\ s \gg= (\lambda s.\ RL\ (xs', s)))$ **by** *simp*
also have $\ldots \leq \alpha\ S \gg= (\lambda s.\ RL\ (x\ \#\ xs', s))$
apply (*simp add*: *RL-unfold*[*of* $x\ \#\ xs'$] *FOREACH-cond-def bind-RES S'-simps*)
apply (*simp add*: *FOREACH-body-def*)
done
also note *pre*
finally have *pre'*: $\alpha\ (cbind\ S\ (f\ x)) \gg= (\lambda s.\ RL\ (xs', s)) \leq RL\ (xs, s0)$ **by** *simp*

from *ind-hyp*[*OF pre'*] *liftc*
show *?thesis* **by** *simp*
qed
qed
qed
}
moreover
have $\alpha\ (creturn\ s0) \gg= (\lambda s.\ RL\ (xs, s)) \leq RL\ (xs, s0)$
proof –
have $\alpha\ (creturn\ s0) \gg= (\lambda s.\ RL\ (xs, s)) \leq$
$RETURN\ s0 \gg= (\lambda s.\ RL\ (xs, s))$
apply (*rule bind-mono*)
apply (*simp-all add*: *transfer-return*)
done
thus *?thesis* **by** *simp*
qed
ultimately have *lift-le*: $\alpha\ (lift\text{-}it\ (foldli\ xs)\ c\ f\ s0) \leq RL\ (xs, s0)$ **by** *simp*

from *order-trans*[*OF lift-le RL-le*, *folded xs-props*(*4*)]
show *?thesis* **.**
qed

lemma *transfer-FOREACHoi*[*refine-transfer*]:
fixes $X :: 'x\ set$ **and** $s0 :: 's$
assumes *IT*: *set-iterator-genord iterate X R*
assumes *RS*: $\bigwedge x\ s.\ \alpha\ (f\ x\ s) \leq F\ x\ s$
shows $\alpha\ (lift\text{-}it\ iterate\ (\lambda\text{-}.\ True)\ f\ s0) \leq FOREACHoi\ I\ X\ R\ F\ s0$
unfolding *FOREACHoi-def*
by (*rule transfer-FOREACHoci* [*OF IT RS*])

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

unfolding *FOREACHci-def*
by (*rule transfer-FOREACHoci* [*OF IT*[*unfolded set-iterator-def*] *RS*])

lemma *transfer-FOREACHc*[*refine-transfer*]:
  fixes X :: $'x$ *set* **and** *s0* :: $'s$
  assumes *IT*: *set-iterator iterate X*
  assumes *RS*: $\bigwedge x\ s.\ \alpha\ (f\ x\ s) \leq F\ x\ s$
  shows α (*lift-it iterate c f s0*) $\leq$ *FOREACHc X c F s0*
  unfolding *FOREACHc-def* **using** *assms* **by** (*rule transfer-FOREACHci*)

lemma *transfer-FOREACHI*[*refine-transfer*]:
  fixes X :: $'x$ *set* **and** *s0* :: $'s$
  assumes *IT*: *set-iterator iterate X*
  assumes *RS*: $\bigwedge x\ s.\ \alpha\ (f\ x\ s) \leq F\ x\ s$
  shows α (*lift-it iterate* (λ-. *True*) *f s0*) $\leq$ *FOREACHi I X F s0*
  unfolding *FOREACHi-def*
  using *assms* **by** (*rule transfer-FOREACHci*)

lemma *det-FOREACH*[*refine-transfer*]:
  fixes X :: $'x$ *set* **and** *s0* :: $'s$
  assumes *IT*: *set-iterator iterate X*
  assumes *RS*: $\bigwedge x\ s.\ \alpha\ (f\ x\ s) \leq F\ x\ s$
  shows α (*lift-it iterate* (λ-. *True*) *f s0*) $\leq$ *FOREACH X F s0*
  unfolding *FOREACH-def*
  using *assms* **by** (*rule transfer-FOREACHc*)

end

lemma *FOREACHoci-mono*[*refine-mono*]:
  assumes $\bigwedge x.\ f\ x \leq f'\ x$
  shows *FOREACHoci I S R c f s0* $\leq$ *FOREACHoci I S R c* f' *s0*
  using *assms* **apply** –
  unfolding *FOREACHoci-def FOREACH-body-def*
  apply (*refine-mono*)
  done
lemma *FOREACHoi-mono*[*refine-mono*]:
  assumes $\bigwedge x.\ f\ x \leq f'\ x$
  shows *FOREACHoi I S R f s0* $\leq$ *FOREACHoi I S R* f' *s0*
  using *assms* **apply** –
  unfolding *FOREACHoi-def FOREACH-body-def*
  apply (*refine-mono*)
  done
lemma *FOREACHci-mono*[*refine-mono*]:
  assumes $\bigwedge x.\ f\ x \leq f'\ x$
  shows *FOREACHci I S c f s0* $\leq$ *FOREACHci I S c* f' *s0*
  using *assms* **apply** –
  unfolding *FOREACHci-def FOREACH-body-def*
  apply (*refine-mono*)
  done

lemma *FOREACHc-mono*[*refine-mono*]:
  assumes $\bigwedge x.\ f\ x \leq f'\ x$
  shows *FOREACHc S c f s0* $\leq$ *FOREACHc S c f' s0*
  using *assms* **apply** −
  unfolding *FOREACHc-def*
  apply (*refine-mono*)
  done
lemma *FOREACHi-mono*[*refine-mono*]:
  assumes $\bigwedge x.\ f\ x \leq f'\ x$
  shows *FOREACHi I S f s0* $\leq$ *FOREACHi I S f' s0*
  using *assms* **apply** −
  unfolding *FOREACHi-def*
  apply (*refine-mono*)
  done
lemma *FOREACH-mono*[*refine-mono*]:
  assumes $\bigwedge x.\ f\ x \leq f'\ x$
  shows *FOREACH S f s0* $\leq$ *FOREACH S f' s0*
  using *assms* **apply** −
  unfolding *FOREACH-def*
  apply (*refine-mono*)
  done

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
apply (*case-tac* [!] *x*)
apply *simp-all*
done

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

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

definition *Sup-dres S* $\equiv$
if $S \subseteq \{dSUCCEEDi\}$ *then dSUCCEEDi*
else if $dFAILi \in S$ *then dFAILi*
else if $\exists\, a\ b.\ a \neq b \wedge dRETURN\ a \in S \wedge dRETURN\ b \in S$ *then dFAILi*
else dRETURN (*THE x.* $dRETURN\ x \in S$)

definition *Inf-dres S* $\equiv$
if $S \subseteq \{dFAILi\}$ *then dFAILi*
else if $dSUCCEEDi \in S$ *then dSUCCEEDi*
else if $\exists\, a\ b.\ a \neq b \wedge dRETURN\ a \in S \wedge dRETURN\ b \in S$ *then dSUCCEEDi*
else dRETURN (*THE x.* $dRETURN\ x \in S$)

fun *less-eq-dres* **where**
less-eq-dres dSUCCEEDi - $\longleftrightarrow$ *True* |
less-eq-dres - dFAILi $\longleftrightarrow$ *True* |
less-eq-dres (*dRETURN* (a::$'a$)) (*dRETURN b*) $\longleftrightarrow$ $a=b$ |
less-eq-dres - - $\longleftrightarrow$ *False*

definition *less-dres* **where** *less-dres* (a::$'a$ *dres*) $b \longleftrightarrow a \leq b \wedge \neg\ b \leq a$

instance

apply *intro-classes*
apply (*simp add*: *less-dres-def*)
apply (*case-tac x*, *simp-all*) []

apply (*case-tac x*, *simp-all*, *case-tac* [!] *y*,
simp-all, *case-tac* [!] *z*, *simp-all*) []

apply (*case-tac x*, *simp-all*, *case-tac* [!] *y*, *simp-all*) []

apply (*case-tac x*, *simp-all*, *case-tac* [!] *y*, *simp-all*) []

apply (*case-tac x*, *simp-all*, *case-tac* [!] *y*, *simp-all*) []

apply (*case-tac x*, *simp-all*, *case-tac* [!] *y*,
simp-all, *case-tac* [!] *z*, *simp-all*) []

apply (*case-tac x*, *simp-all*, *case-tac* [!] *y*, *simp-all*) []

apply (*case-tac x*, *simp-all*, *case-tac* [!] *y*, *simp-all*) []

apply (*case-tac x*, *simp-all*, *case-tac* [!] *y*,
simp-all, *case-tac* [!] *z*, *simp-all*) []

apply (*case-tac a*, *simp-all add*: *bot-dres-def*) []
apply (*case-tac a*, *simp-all add*: *top-dres-def*) []

apply (*case-tac x*)
apply (*auto simp add*: *Inf-dres-def*) [*3*]

apply (*case-tac z*, *simp-all add*: *Inf-dres-def*) []
apply (*auto*) []
apply (*case-tac x*, *fastforce+*) []
apply (*case-tac x*, *fastforce+*) []

apply *auto* []
apply (*case-tac x*)
apply *force*
apply *force*
apply (*subgoal-tac dRETURN a* $\leq$ *dRETURN aa* $\wedge$ *dRETURN a* $\leq$ *dRETURN b*)
apply *auto* []
apply *blast*
apply *force*

apply (*case-tac x*) []
apply *force*
apply *force*
apply *auto* []
apply (*subgoal-tac* (*THE x*. *dRETURN x* $\in$ *A*) = *aa*)

apply *force*
apply *force*

apply (*case-tac x*)
apply (*auto simp add*: *Sup-dres-def*) [*3*]
apply (*case-tac xa*, *simp-all*) []
apply (*subgoal-tac* (*THE x*. *dRETURN x* $\in$ *A*) = *aa*)
apply *force*
apply *force*

apply (*case-tac z*, *auto simp add*: *Sup-dres-def*) []
apply *force*
apply (*case-tac x*, *force+*) []

apply (*case-tac x*, *force*, *force*) []
apply (*subgoal-tac dRETURN aa* $\leq$ *dRETURN a* $\wedge$ *dRETURN b* $\leq$ *dRETURN a*)
apply *auto* []
apply *blast*

apply *force*

apply (*case-tac x*, *force+*)[]
done

end

abbreviation *dSUCCEED* $\equiv$ (*bot*::'*a dres*)
abbreviation *dFAIL* $\equiv$ (*top*::'*a dres*)

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

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

lemma *dres-order-simps*[*simp*]:
x$\leq$*dSUCCEED* $\longleftrightarrow$ *x*=*dSUCCEED*
dFAIL$\leq$*x* $\longleftrightarrow$ *x*=*dFAIL*
dRETURN r $\neq$ *dFAIL*
dRETURN r $\neq$ *dSUCCEED*
dFAIL $\neq$ *dRETURN r*
dSUCCEED $\neq$ *dRETURN r*
dFAIL$\neq$*dSUCCEED*
dSUCCEED$\neq$*dFAIL*
x=*y* $\Longrightarrow$ *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*

apply (*simp-all add*: *bot-unique top-unique*)
apply (*simp-all add*: *bot-dres-def top-dres-def*)
done

lemma *dres-Sup-cases*:
obtains $S\subseteq\{dSUCCEED\}$ **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** $S \subseteq \{dSUCCEED,\ dRETURN\ a\}$ $dRETURN\ a\in S$ $Sup\ S = dRETURN\ a$
proof −
show *?thesis*
apply (*cases* $S\subseteq\{dSUCCEED\}$)
apply (*rule that*(*1*), *assumption*)
apply (*simp add*: *Sup-dres-def bot-dres-def*)

apply (*cases* $dFAIL\in S$)
apply (*rule that*(*2*), *assumption*)
apply (*simp add*: *Sup-dres-def bot-dres-def top-dres-def*)

apply (*cases* $\exists\, a\ b.\ a\neq b \wedge dRETURN\ a\in S \wedge dRETURN\ b\in S$)
apply (*elim exE conjE*)
apply (*rule that*(*3*), *assumption+*)
apply (*auto simp add*: *Sup-dres-def bot-dres-def top-dres-def*) []

apply *simp*
apply (*cases* $\exists\, a.\ dRETURN\ a \in S$)
apply (*elim exE*)
apply (*rule-tac* $a{=}a$ **in** *that*(*4*)) []
apply *auto* [] **apply** (*case-tac xa*, *auto*) []
apply *auto* []
apply (*auto simp add*: *Sup-dres-def bot-dres-def top-dres-def*) []

apply *auto* **apply** (*case-tac x*, *auto*) []
done
qed

lemma *dres-Inf-cases*:
obtains $S\subseteq\{dFAIL\}$ **and** $Inf\ S = dFAIL$
| $dSUCCEED\in S$ **and** $Inf\ S = dSUCCEED$
| *a b* **where** $a\neq b$ $dRETURN\ a\in S$ $dRETURN\ b\in S$ $dSUCCEED\notin S$ $Inf\ S = dSUCCEED$
| *a* **where** $S \subseteq \{dFAIL,\ dRETURN\ a\}$ $dRETURN\ a\in S$ $Inf\ S = dRETURN\ a$
proof −
show *?thesis*
apply (*cases* $S\subseteq\{dFAIL\}$)
apply (*rule that*(*1*), *assumption*)

apply (*simp add*: *Inf-dres-def top-dres-def*)

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

apply (*cases* $\exists$ *a b*. *a*$\neq$*b* $\wedge$ *dRETURN a*$\in$*S* $\wedge$ *dRETURN b*$\in$*S*)
apply (*elim exE conjE*)
apply (*rule that*(*3*), *assumption*+)
apply (*auto simp add*: *Inf-dres-def bot-dres-def top-dres-def*) []

apply *simp*
apply (*cases* $\exists$ *a*. *dRETURN a* $\in$ *S*)
apply (*elim exE*)
apply (*rule-tac a*=*a* **in** *that*(*4*)) []
apply *auto* [] **apply** (*case-tac xa*, *auto*) []
apply *auto* []
apply (*auto simp add*: *Inf-dres-def bot-dres-def top-dres-def*) []

apply *auto* **apply** (*case-tac x*, *auto*) []
done
qed

lemma *dres-chain-eq-res*:
is-chain M $\Longrightarrow$
dRETURN r $\in$ *M* $\Longrightarrow$ *dRETURN s* $\in$ *M* $\Longrightarrow$ *r*=*s*
by (*metis chainD less-eq-dres.simps*(*4*))

lemma *dres-Sup-chain-cases*:
assumes *CHAIN*: *is-chain M*
obtains *M* $\subseteq$ {*dSUCCEED*} *Sup M* = *dSUCCEED*
| *r* **where** *M* $\subseteq$ {*dSUCCEED*,*dRETURN r*} *dRETURN r*$\in$*M* *Sup M* = *dRETURN r*
| *dFAIL*$\in$*M* *Sup M* = *dFAIL*
apply (*rule dres-Sup-cases*[*of M*])
using *dres-chain-eq-res*[*OF CHAIN*]
by *auto*

lemma *dres-Inf-chain-cases*:
assumes *CHAIN*: *is-chain M*
obtains *M* $\subseteq$ {*dFAIL*} *Inf M* = *dFAIL*
| *r* **where** *M* $\subseteq$ {*dFAIL*,*dRETURN r*} *dRETURN r*$\in$*M* *Inf M* = *dRETURN r*
| *dSUCCEED*$\in$*M* *Inf M* = *dSUCCEED*
apply (*rule dres-Inf-cases*[*of M*])
using *dres-chain-eq-res*[*OF CHAIN*]
by *auto*

lemma *dres-internal-simps*[*simp*]:

dSUCCEEDi = *dSUCCEED*
dFAILi = *dFAIL*
unfolding *top-dres-def bot-dres-def* **by** *auto*

Monad Operations

function *dbind* **where**
dbind dFAIL - = dFAIL
| *dbind dSUCCEED - = dSUCCEED*
| *dbind* (*dRETURN x*) *f* = *f x*
unfolding *bot-dres-def top-dres-def*
by *pat-completeness auto*
termination by *lexicographic-order*

setup ⟪
Adhoc-Overloading.add-variant
@{*const-name Monad-Syntax.bind*} @{*const-name dbind*}
⟫

lemma [*code*]:
dbind (*dRETURN x*) *f* = *f x*
dbind (*dSUCCEEDi*) *f* = *dSUCCEEDi*
dbind (*dFAILi*) *f* = *dFAILi*
by *simp-all*

lemma *dres-monad1*[*simp*]: *dbind* (*dRETURN x*) *f* = *f x*
by (*simp*)
lemma *dres-monad2*[*simp*]: *dbind M dRETURN* = *M*
apply (*cases M*)
apply (*auto*)
done

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

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

lemma *dbind-mono*[*refine-mono*]:
$[\![M \le M'; \bigwedge x.\ dRETURN\ x \le M \Longrightarrow f\ x \le f'\ x]\!] \Longrightarrow$
$dbind\ M\ f \le dbind\ M'\ f'$
apply (*cases M*, *simp-all*)
apply (*cases M'*, *simp-all*)
done

lemma *dbind-mono1*[*simp*, *intro*!]: *mono dbind*
apply (*rule monoI*)
apply (*rule le-funI*)

apply (*rule dbind-mono*)
by *auto*

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

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

lemma *dr-mono-bind'*: *mono* ($\lambda F\ s.$ *dbind* (*f s*) *F*)
apply *rule*
apply (*rule le-funI*)
apply (*rule dbind-mono*)
apply (*auto dest*: *le-funD*)
done

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

lemma [*refine-mono*]:
dbind dSUCCEED f = *dSUCCEED*
dbind dFAIL f = *dFAIL*
by (*simp-all*)

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

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

interpretation *dres-while*!: *generic-WHILE dbind dRETURN*
dWHILEIT dWHILEI dWHILET dWHILE
apply *unfold-locales*
apply (*auto simp*: *dWHILEIT-def dWHILEI-def dWHILET-def dWHILE-def*)
done

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

Syntactic criteria to prove $s \neq dSUCCEED$

lemma *dres-ne-bot-basic*[*refine-transfer*]:
  $dFAIL \neq dSUCCEED$
  $dRETURN\ x \neq dSUCCEED$
  $[\![m{\neq}dSUCCEED;\ \bigwedge x.\ f\ x \neq dSUCCEED]\!] \Longrightarrow dbind\ m\ f \neq dSUCCEED$
  $dASSERT\ \Phi \neq dSUCCEED$
  $[\![m1{\neq}dSUCCEED;\ m2{\neq}dSUCCEED]\!] \Longrightarrow If\ b\ m1\ m2 \neq dSUCCEED$
  $[\![\bigwedge x.\ f\ x{\neq}dSUCCEED]\!] \Longrightarrow Let\ x\ f \neq dSUCCEED$
  $[\![\bigwedge x1\ x2.\ g\ x1\ x2 \neq dSUCCEED]\!] \Longrightarrow prod\text{-}case\ g\ p \neq dSUCCEED$
  apply (*auto split*: *prod.split*)
  apply (*cases m*, *auto*) []
  apply (*cases* Φ, *auto*) []
  done

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 $\forall x.\ RECT\ B\ x \neq dSUCCEED$
  unfolding *RECT-def*
  apply (*split split-if*)
  apply (*subst all-conj-distrib*)
  apply (*intro impI conjI*)
  apply *simp*
  apply (*intro impI*)
  apply (*erule gfp-cadm-induct*[*rotated*])
  apply (*intro allI*)
  apply (*rule A*)
  apply *simp*

  apply *rule*
  apply *simp*
  apply (*intro allI*)
  apply (*drule-tac x=x* **in** *point-chainI*)
  apply (*erule dres-Inf-chain-cases*)
  apply (*auto simp*: *Inf-fun-def INF-def dest*!: *subset-singletonD*) []
  apply (*auto simp*: *Inf-fun-def INF-def*) []
  apply (*auto simp*: *Inf-fun-def INF-def*) []
  apply *metis*
  apply *simp*
  done

lemma *dres-ne-bot-dWHILEIT*[*refine-transfer*]:
  assumes $\bigwedge x.\ f\ x \neq dSUCCEED$

shows *dWHILEIT I b f s* $\neq$ *dSUCCEED* **using** *assms*
unfolding *dWHILEIT-def iWHILEIT-def WHILEI-body-def*
apply *refine-transfer*
done

lemma *dres-ne-bot-dWHILET*[*refine-transfer*]:
assumes $\bigwedge x.\ f\ x \neq dSUCCEED$
shows *dWHILET b f s* $\neq$ *dSUCCEED* **using** *assms*
unfolding *dWHILET-def iWHILET-def iWHILEIT-def WHILEI-body-def*
apply *refine-transfer*
done

end

2.11 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-of r* $\equiv$ *case r of*
dSUCCEEDi $\Rightarrow$ *SUCCEED*
| *dFAILi* $\Rightarrow$ *FAIL*
| *dRETURN x* $\Rightarrow$ *RETURN x*

lemma *nres-of-simps*[*simp*]:
nres-of dSUCCEED = *SUCCEED*
nres-of dFAIL = *FAIL*
nres-of (*dRETURN x*) = *RETURN x*
apply −
unfolding *nres-of-def bot-dres-def top-dres-def*
by (*auto simp del*: *dres-internal-simps*)

lemma *nres-of-mono*: *mono nres-of*
  apply (*rule*)
  apply (*case-tac x*, *simp-all*, *case-tac y*, *simp-all*)
  done

lemma *nres-transfer*:
  $nres\text{-}of\ dSUCCEED = SUCCEED$
  $nres\text{-}of\ dFAIL = FAIL$
  $nres\text{-}of\ a \leq nres\text{-}of\ b \longleftrightarrow a \leq b$
  $nres\text{-}of\ a < nres\text{-}of\ b \longleftrightarrow a < b$
  $is\text{-}chain\ A \Longrightarrow nres\text{-}of\ (Sup\ A) = Sup\ (nres\text{-}of`A)$
  $is\text{-}chain\ A \Longrightarrow nres\text{-}of\ (Inf\ A) = Inf\ (nres\text{-}of`A)$
  apply *simp-all*
  apply (*case-tac a*, *simp-all*, *case-tac* [!] *b*, *simp-all*) [*1*]

  apply (*simp add*: *less-le*)
  apply (*case-tac a*, *simp-all*, *case-tac* [!] *b*, *simp-all*) [*1*]

  apply (*erule dres-Sup-chain-cases*)
    apply (*cases* $A=\{\}$)
    apply *auto* []
    apply (*subgoal-tac* $A=\{dSUCCEED\}$, *auto*) []

    apply (*case-tac* $A=\{dRETURN\ r\}$)
    apply *auto* []
    apply (*subgoal-tac* $A=\{dSUCCEED,dRETURN\ r\}$, *auto*) []

    apply (*drule imageI*[**where** $f=nres\text{-}of$])
    apply *auto* []

  apply (*erule dres-Inf-chain-cases*)
    apply (*cases* $A=\{\}$)
    apply *auto* []
    apply (*subgoal-tac* $A=\{dFAIL\}$, *auto*) []

    apply (*case-tac* $A=\{dRETURN\ r\}$)
    apply *auto* []
    apply (*subgoal-tac* $A=\{dFAIL,dRETURN\ r\}$, *auto*) []

    apply (*drule imageI*[**where** $f=nres\text{-}of$])
    apply (*auto intro*: *bot-Inf*[*symmetric*]) []
  done

lemma *nres-correctD*:
  assumes $nres\text{-}of\ S \leq SPEC\ \Phi$
  shows
  $S=dRETURN\ x \Longrightarrow \Phi\ x$
  $S \neq dFAIL$

using *assms* **apply** −
apply (*cases S*, *simp-all*)+
done

Transfer Theorems Setup

interpretation *dres*!: *dist-transfer nres-of*
apply *unfold-locales*
apply (*simp add*: *nres-transfer*)
done

lemma *det-FAIL*[*refine-transfer*]: *nres-of* (*dFAIL*) $\leq$ *FAIL* **by** *auto*
lemma *det-SUCCEED*[*refine-transfer*]: *nres-of* (*dSUCCEED*) $\leq$ *SUCCEED* **by** *auto*
lemma *det-SPEC*: Φ *x* $\Longrightarrow$ *nres-of* (*dRETURN x*) $\leq$ *SPEC* Φ **by** *simp*
lemma *det-RETURN*[*refine-transfer*]:
nres-of (*dRETURN x*) $\leq$ *RETURN x* **by** *simp*
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*
using *assms*
apply (*cases m*)
apply (*auto simp*: *pw-le-iff refine-pw-simps*)
done

interpretation *det-assert*!: *transfer-generic-Assert-remove*
bind RETURN ASSERT ASSUME
nres-of
by *unfold-locales*

interpretation *det-while*!: *transfer-WHILE*
dbind dRETURN dWHILEIT dWHILEI dWHILET dWHILE
bind RETURN WHILEIT WHILEI WHILET WHILE nres-of
apply *unfold-locales*
apply (*auto intro*: *det-bind*)
done

interpretation *det-foreach*!:
transfer-FOREACH nres-of dRETURN dbind dres-case True True
apply *unfold-locales*
apply (*blast intro*: *det-bind*)
apply *simp*
apply (*case-tac m*)
apply *simp-all*
done

2.11.2 Transfer to Plain Function

interpretation *plain*!: *transfer RETURN* **.**

lemma *plain-RETURN*[*refine-transfer*]: *RETURN* $a \leq$ *RETURN* a **by** *simp*
lemma *plain-bind*[*refine-transfer*]:
$[\![RETURN\ x \leq M;\ \bigwedge x.\ RETURN\ (f\ x) \leq F\ x]\!] \Longrightarrow RETURN\ (Let\ x\ f) \leq bind\ M\ F$
  apply (*erule order-trans*[*rotated*,*OF bind-mono*])
  apply *assumption*
  apply *simp*
  done

interpretation *plain-assert*!: *transfer-generic-Assert-remove*
  bind RETURN ASSERT ASSUME
  RETURN
  by *unfold-locales*

interpretation *plain*!: *transfer-FOREACH RETURN* $(\lambda x.\ x)$ *Let* $(\lambda x.\ x)$
  apply (*unfold-locales*)
  apply (*erule plain-bind*, *assumption*)
  apply *simp*
  apply *simp*
  done

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) $\leq S'$
  using *assms*
  by (*cases* S, *simp-all*)

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*
  by *unfold-locales* (*auto simp add*: *Sup-upper Sup-least*)

lemma *nrec-admissible*: *nrec.admissible* (λ(*f*::'*a* $\Rightarrow$ '*b nres*).
  ($\forall$ *x0*. *f x0* $\leq$ *SPEC* (*P x0*)))
  apply (*rule ccpo.admissibleI*[*OF nrec.ccpo*])
  apply (*unfold fun-lub-def*)
  apply (*intro allI impI*)
  apply (*rule Sup-least*)
  apply *auto*
  done

declaration ⟪ *Partial-Function.init nrec* @{*term nrec.fixp-fun*}
  @{*term nrec.mono-body*} @{*thm nrec.fixp-rule-uc*}
  (**SOME* @{*thm fixp-induct-nrec*'}*) *NONE* ⟫

lemma *bind-mono-pfun*[*partial-function-mono*]:
  ⟦ *nrec.mono-body B*; $\bigwedge$*y*. *nrec.mono-body* (λ*f*. *C y f*) ⟧ $\Longrightarrow$
    nrec.mono-body (λ*f*. *bind* (*B f*) (λ*y*. *C y f*))
  apply *rule*
  apply (*rule Refine-Basic.bind-mono*)
  apply (*blast dest*: *monotoneD*)+
  done

2.12.2 Deterministic Result Monad

interpretation *drec*!:
  partial-function-definitions op $\leq$ *Sup*::'*a dres set* $\Rightarrow$ '*a dres*
  by *unfold-locales* (*auto simp add*: *Sup-upper Sup-least*)

lemma *drec-admissible*: *drec.admissible* (λ(*f*::'*a* $\Rightarrow$ '*b dres*).
  ($\forall$ *x*. *P x* $\longrightarrow$
    (*f x* $\neq$ *dFAIL* $\wedge$

```
    (∀ r. f x = dRETURN r ⟶ Q x r))))
proof -
  have [simp]: fun-ord (op ≤::'b dres ⇒ - ⇒ -) = op ≤
    apply (intro ext)
    unfolding fun-ord-def le-fun-def
    by (rule refl)

  have [simp]: ⋀A x. {y. ∃f∈A. y = f x} = (λf. f x)`A by auto

  show ?thesis
    apply (rule ccpo.admissibleI[OF drec.ccpo])
    apply (unfold fun-lub-def)
    apply clarsimp
    apply (drule-tac x=x in point-chainI)
    apply (erule dres-Sup-chain-cases)
    apply (simp only:)
    apply simp
    apply (simp only:)
    apply auto []
    apply (simp only:)
    apply force
    done
qed

declaration ⟪ Partial-Function.init drec @{term drec.fixp-fun}
  @{term drec.mono-body} @{thm drec.fixp-rule-uc} NONE ⟫

lemma drec-bind-mono-pfun[partial-function-mono]:
  ⟦ drec.mono-body B; ⋀y. drec.mono-body (λf. C y f) ⟧ ⟹
     drec.mono-body (λf. dbind (B f) (λy. C y f))
  apply rule
  apply (rule dbind-mono)
  apply (blast dest: monotoneD)+
  done

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*: $(f,f)\in Id$ **by** *simp*
lemma *idtrans-arg*: $(f,f')\in Id \Longrightarrow (a,a')\in Id \Longrightarrow (f\ a,f'\ a')\in Id$ **by** *simp*

2.13.2 Setup

```
ML ⟪
structure Refine-Autoref = struct
  open Refine-Misc;
  (*****************************)
  (* Theorem Collections *)
  (*****************************)

  structure prg-thms = Named-Thms
    ( val name = @{binding autoref-prg}
      val description = Automatic Refinement: Program translation rules. );

  structure exp-thms = Named-Thms
    ( val name = @{binding autoref-ex}
      val description = Automatic Refinement: Expression translation rules.);

  structure spec-thms = Named-Thms
    ( val name = @{binding autoref-spec}
      val description = Automatic Refinement: Specification rules
        ^decomposition rules. );

  structure other-thms = Named-Thms
    ( val name = @{binding autoref-other}
      val description = Automatic Refinement: Rules for subgoals of
      ^other types );

  structure elim-thms = Named-Thms
    ( val name = @{binding autoref-elim}
      val description = Automatic Refinement: Preprocessing elim rules );

  structure simp-thms = Named-Thms
    ( val name = @{binding autoref-simp}
      val description = Automatic Refinement: Preprocessing simp rules );

  type config = {
```

```
  prg-thms : thm list,
  exp-thms : thm list,
  spec-thms : thm list,
  other-thms : thm list,
  elim-thms : thm list,
  simp-ss : simpset,
  trace : bool
};


(****************************)
(* Tactics *)
(****************************)

(*
  The following function classifies the current subgoal into one of
  the following categories. It is used by the main tactic to decide how
  to proceed.
*)
datatype ref-goal =                  (* Argument order is always (C,R,A) *)
  Rg-prg-ref of term * term * term  | (* C ≤⇓R A *)
  Rg-var-ref of term * term * term  | (* (C,A)∈R where A is var/free *)
  Rg-spec-ref of term * term * term | (* (C,A)∈?R, where type C has var *)
  Rg-exp-ref of term * term * term  | (* (C,A)∈R [where hd R is constant] *)
  Rg-other   of term                | (* Any other conlcusion *)
  Rg-invalid of term;                 (* Invalid *)

fun dest-refinement-goal ctxt i st = let
  (* Test to detect schematic variables on wrong side *)
  fun test-var t res = if exists-subterm is-Var t then
    Rg-invalid t else res;
in
  if i <= nprems-of st then case
    Logic.concl-of-goal (prop-of st) i |> HOLogic.dest-Trueprop
    |> Envir.beta-eta-contract
  of
    (Const (@{const-name Orderings.ord-class.less-eq}, -) $
      c $ (Const (@{const-name Refine-Basic.conc-fun},-) $ r $ a)) =>
      test-var a (Rg-prg-ref (c,r,a))
  | (Const (@{const-name Set.member},-) $
      (Const(@{const-name Product-Type.Pair},-)$c$a)$r) =>
    if is-Free a orelse is-Var a then test-var a (Rg-var-ref (c,r,a))
    else if exists-subtype is-TVar (fastype-of c) then
      test-var a (Rg-spec-ref (c,r,a))
    else test-var a (Rg-exp-ref (c,r,a))
  | t => Rg-other t
  else (Rg-invalid (prop-of st))
end;
```

```
(* Check whether term is a goal suited for idtrans. In positive case,
  return head function and number of arguments.
  Check whether goal has the form: (-,t)∈- where t=f a1 ...an,
  and return f and n.
*)
fun dest-idtrans-aexp (-,r,a) = (
  case strip-comb a of
    (c as (Const -),args) => SOME (c,length args)
  | (f as (Free -),args) => SOME (f,length args)
  | - => NONE);

fun idtrans-tac ctxt a i st = case dest-idtrans-aexp a of
  SOME (f,argn) => let
    val certify = cterm-of (Proof-Context.theory-of ctxt);
    val idtrans0-thm = @{thm idtrans0};
    val f-var = idtrans0-thm |> concl-of |> HOLogic.dest-Trueprop
      |> strip-comb |> snd |> hd |> strip-comb |> snd |> hd;

    val inst = [(certify f-var,certify f)];

    fun do-inst 0 = Drule.cterm-instantiate inst idtrans0-thm
    | do-inst n = do-inst (n−1) RS @{thm idtrans-arg};

    val thm = do-inst argn;
  in rtac thm i st end
| - => no-tac st;

fun preprocess-tac ctxt (cfg:config) =
  ((Method.assm-tac ctxt ORELSE' full-simp-tac (#simp-ss cfg))
     THEN-ALL-NEW (TRY o REPEAT-ALL-NEW
       (Tactic.eresolve-tac (#elim-thms cfg)))
     );

fun trace-rg ctxt rg = let
  fun ts-term t = Pretty.string-of (Syntax.pretty-term ctxt t);
  fun ts-triple (c,r,a) = (^ts-term c^ | ^ts-term r^ | ^ts-term a^)
in tracing (
  case rg of
    Rg-prg-ref t => prg-ref: ^ts-triple t
  | Rg-exp-ref t => exp-ref: ^ts-triple t
  | Rg-var-ref t => var-ref: ^ts-triple t
  | Rg-spec-ref t => spec-ref: ^ts-triple t
  | Rg-other t => other: ^ts-term t
  | Rg-invalid t => invalid: ^ts-term t
) end

fun trace-sg ctxt i st = ();
  (*if i <= nprems-of st then tracing (@{make-string} (cprem-of st i))
  else ();*)
```

```
fun autoref-tac (cfg:config) sstep ctxt i st = let
  val recurse-tac = if sstep then K all-tac else autoref-tac cfg sstep ctxt;
  val rg = dest-refinement-goal ctxt i st;
  val - = if #trace cfg then
    (trace-rg ctxt rg; trace-sg ctxt i st)
    else ();

in
  case rg of
    Rg-prg-ref - => (
      preprocess-tac ctxt cfg THEN-ALL-NEW
        ( resolve-tac (#prg-thms cfg)
              ORELSE' ( resolve-tac (#exp-thms cfg) THEN-ALL-NEW-FWD
recurse-tac)
          ORELSE' rprems-tac ctxt (* Match with induction premises from REC
*)
        )
    ) i st
  | Rg-var-ref (c,r,a) => (
    preprocess-tac ctxt cfg THEN-ALL-NEW atac)  i st
  | Rg-spec-ref (c,r,a) =>
    (preprocess-tac ctxt cfg THEN-ALL-NEW
      (resolve-tac (#spec-thms cfg) THEN-ALL-NEW-FWD recurse-tac)
    ) i st
  | Rg-exp-ref (c,r,a) => (preprocess-tac ctxt cfg THEN-ALL-NEW
    FIRST' [resolve-tac (#exp-thms cfg), idtrans-tac ctxt (c,r,a)]
      THEN-ALL-NEW-FWD recurse-tac
    ) i st
  | Rg-other - => ((
      triggered-mono-tac ctxt ORELSE'
      REPEAT-ALL-NEW (CHANGED-PROP o ares-tac (#other-thms cfg))) i
st)
  | Rg-invalid - => (no-tac st)
end;

fun dflt-config ctxt add-spec-thms trace = {
  prg-thms = prg-thms.get ctxt,
  exp-thms = exp-thms.get ctxt,
  spec-thms = add-spec-thms @ spec-thms.get ctxt,
  other-thms = other-thms.get ctxt,
  elim-thms = elim-thms.get ctxt,
  simp-ss = HOL-basic-ss addsimps (simp-thms.get ctxt),
  trace=trace
};

fun autoref-method trace ss as-thms ctxt = let
  val cfg = dflt-config ctxt as-thms trace
in
```

```
    if ss then
      SIMPLE-METHOD' (
        CHANGED-PROP o (autoref-tac cfg true ctxt))
    else
      SIMPLE-METHOD' (
        CHANGED-PROP o REPEAT-DETERM' (CHANGED-PROP o
          autoref-tac cfg false ctxt))
  end

end ⟫
```

setup ⟪ *Refine-Autoref.prg-thms.setup* ⟫
setup ⟪ *Refine-Autoref.exp-thms.setup* ⟫
setup ⟪ *Refine-Autoref.spec-thms.setup* ⟫
setup ⟪ *Refine-Autoref.other-thms.setup* ⟫
setup ⟪ *Refine-Autoref.elim-thms.setup* ⟫
setup ⟪ *Refine-Autoref.simp-thms.setup* ⟫

method-setup *refine-autoref* =
⟪ (*Args.mode trace* −− *Args.mode ss* −− *Attrib.thms*)
  >> (*fn* ((*trace*,*ss*),*as-thms*) => *fn ctxt*
      => *Refine-Autoref.autoref-method trace ss as-thms ctxt*) ⟫
Refinement Framework: *Automatic data refinement*

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 $Let\ x\ f \le\Downarrow R\ (Let\ x'\ f')$
  using *assms*
  by (*auto simp*: *pw-le-iff refine-pw-simps*)

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*: $\bigwedge f\ f'\ x\ x'.\ \llbracket\ \bigwedge x\ x'.\ (x,x')\in R \Longrightarrow f\ x \le\Downarrow S\ (f'\ x');\ (x,x')\in R\ \rrbracket$

$\Longrightarrow$ *body f x* $\leq\Downarrow S$ (*body' f' x'*)
assumes *M*: *mono body*
shows *REC body x* $\leq\Downarrow S$ (*REC body' x'*)
by (*rule REC-refine*[*OF M R0 RS*])

lemma *RECT-autoref* [*autoref-prg*]:
assumes *R0*: $(x,x')\in R$
assumes *RS*: $\bigwedge f\, f'\, x\, x'.\ [\![\ \bigwedge x\, x'.\ (x,x')\in R \Longrightarrow f\ x \leq\Downarrow S\ (f'\ x');\ (x,x')\in R\ ]\!]$
$\Longrightarrow$ *body f x* $\leq\Downarrow S$ (*body' f' x'*)
assumes *M*: *mono body*
shows *RECT body x* $\leq\Downarrow S$ (*RECT body' x'*)
by (*rule RECT-refine*[*OF M R0 RS*])

lemma *RETURN-autoref* [*autoref-prg*]:
$[\![(x,\ x') \in R;$ *single-valued R*$]\!] \Longrightarrow$ *RETURN x* $\leq \Downarrow R$ (*RETURN x'*)
using *RETURN-refine-sv* **.**

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*: $\bigwedge\sigma\ \sigma'\ x\ x'.\ [\![\ (\sigma,\sigma')\in R;\ (x,x')\in Id\ ]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACH S f* $\sigma 0 \leq \Downarrow R$ (*FOREACH S' f'* $\sigma 0'$)
apply (*rule FOREACH-refine*[*OF inj-on-id*, **where** Φ''=λ- - - -. *True*])
using *assms* **by** *auto*

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*: $\bigwedge\sigma\ \sigma'\ x\ x'.\ [\![\ (\sigma,\sigma')\in R;\ (x,x')\in Id\ ]\!] \Longrightarrow f\ x\ \sigma$
$\leq \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACH S f* $\sigma 0 \leq \Downarrow R$ (*FOREACHi I S' f'* $\sigma 0'$)
unfolding *FOREACH-def FOREACHc-def FOREACHi-def*
apply (*rule FOREACHci-refine*[*OF inj-on-id*, **where** Φ''=λ- - - -. *True*])
using *assms* **by** *auto*

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

apply (*rule FOREACHc-refine*[*OF inj-on-id*, **where** Φ''=λ- - - -. *True*])
using *assms* **by** *auto*

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*: $\bigwedge\sigma\ \sigma'\ x\ x'.\ [\![\ (\sigma,\sigma')\in R;\ (x,x')\in Id\ ]\!] \Longrightarrow f\ x\ \sigma$
$\le \Downarrow R\ (f'\ x'\ \sigma')$
shows *FOREACHc* $S\ c\ f\ \sigma 0 \le \Downarrow R$ (*FOREACHci* $I\ S'\ c'\ f'\ \sigma 0'$)
unfolding *FOREACH-def FOREACHc-def FOREACHi-def*
apply (*rule FOREACHci-refine*[*OF inj-on-id*, **where** Φ''=λ- - - -. *True*])
using *assms* **by** *auto*

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* $R1\ R2 \longleftrightarrow (a,a')\in R1 \wedge (b,b')\in R2$ **by** *auto*

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

lemma *prod-id-split*[*autoref-simp*]: *Id* = *rprod Id Id* **by** *auto*

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* $f\ x,f'\ x')\in R$ **by** *auto*

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

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

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 $(f,\ \textit{prod-case}\ f'\ (a',b'))\in R$
  using *assms* **by** *auto*

lemma *prod-case-autoref-prg*[*autoref-prg*]:
  assumes $f \le\Downarrow R\ (f'\ a'\ b')$
  shows $f \le\Downarrow R\ (\textit{prod-case}\ f'\ (a',b'))$
  using *assms* **by** *auto*

lemma *pair-autoref*[*autoref-ex*]:
  $[\![\ (a,a')\in Ra;\ (b,b')\in Rb\ ]\!] \Longrightarrow ((a,b),(a',b'))\in \textit{rprod}\ Ra\ Rb$
  by *auto*

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

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

lemma *TAG-remove*[*simp*]: $\textit{TAG}\ s\ t = t$ **by** *simp*

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

lemma *TAG-dest*:
  assumes $(c::'c,a::'a)\in R$
  shows $(c,\textit{TAG}\ \textit{name}\ a)\in R$
  using *assms* **by** *auto*

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 $(c,a)\in R$

by *fact*

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 $(c,a){\in}Id$
by *fact*

2.13.4 Convenience Lemmas

Introduce autoref, determinize, optimize

lemma *autoref-det-optI*:
assumes $m \le \Downarrow R\ a$
assumes $\alpha\ c' \le m$
assumes $c = c'$
shows $\alpha\ c \le \Downarrow R\ a$
using *assms* **by** *auto*

Introduce autoref, determinize

lemma *autoref-detI*:
assumes $m \le \Downarrow R\ a$
assumes $\alpha\ c \le m$
shows $\alpha\ c \le \Downarrow R\ a$
using *assms* **by** *auto*

To be used in optimize phase:

lemma *prod-case-rename*:
prod-case $(\lambda a\ \text{-}.\ f\ a) = f\ o\ fst$
prod-case $(\lambda\text{-}\ b.\ f\ b) = f\ o\ snd$
by (*auto*)

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*) **by** (*simp add*: *RELATES-def*)
lemma (**in** *map*) *drh*[*refine-dref-RELATES*]:
RELATES (*build-rel* α *invar*) **by** (*simp add*: *RELATES-def*)

lemma (**in** *uprio*) *drh*[*refine-dref-RELATES*]: *RELATES* (*build-rel* α *invar*)
by (*simp add*: *RELATES-def*)
lemma (**in** *prio*) *drh*[*refine-dref-RELATES*]: *RELATES* (*build-rel* α *invar*)
by (*simp add*: *RELATES-def*)

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

lemma (**in** *set-sel'*) *pick-ref*[*refine-hsimp*]:
$[\![$ *invar s*; α $s \neq \{\}]\!] \Longrightarrow$ *the* (*sel'* *s* (λ-. *True*)) $\in \alpha$ *s*
by (*auto elim*!: *sel'E*)

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*)
unfolding *IT-tag-def* **by** (*rule iteratei-rule*)

lemma (**in** *map-iteratei*) *it-is-iterator*[*refine-transfer*]:
invar m $\Longrightarrow$ *set-iterator* (*IT-tag iteratei m*) (*map-to-set* (α *m*))
unfolding *IT-tag-def* **by** (*rule iteratei-rule*)

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

definition *prio-pop-min q* $\equiv$ *do* {
ASSERT (*dom q* $\neq \{\}$);

$SPEC\ (\lambda(e,w,q').$
$q'{=}q(e{:=}None) \wedge$
$q\ e = Some\ w \wedge$
$(\forall\ e'\ w'.\ q\ e' = Some\ w' \longrightarrow w{\le}w')$
$)$
$\}$

lemma (**in** *uprio-pop*) *prio-pop-min-refine*[*refine*]:
$(q,q'){\in}build\text{-}rel\ \alpha\ invar \Longrightarrow RETURN\ (pop\ q)$
$\le \Downarrow (rprod\ Id\ (rprod\ Id\ (build\text{-}rel\ \alpha\ invar)))$
$(prio\text{-}pop\text{-}min\ q')$
unfolding *prio-pop-min-def*
apply *refine-rcg*
apply *clarsimp*
apply (*erule* (*1*) *popE*)
apply (*rule pw-leI*)
apply (*auto simp*: *refine-pw-simps intro*: *ranI*)
done

lemma *dres-ne-bot-iterate*[*refine-transfer*]:
assumes *B*: *set-iterator* (*IT-tag it r*) *S*
assumes *A*: $\bigwedge x\ s.\ f\ x\ s \neq dSUCCEED$
shows $IT\text{-}tag\ it\ r\ c\ (\lambda x\ s.\ dbind\ s\ (f\ x))\ (dRETURN\ s) \neq dSUCCEED$
apply (*rule-tac* $I{=}\lambda\text{-}\ s.\ s{\neq}dSUCCEED$ **in** *set-iterator-rule-P*[*OF B*])
apply (*rule dres-ne-bot-basic A* | *assumption*)+
done

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 $foldli\ l\ c\ f\ \sigma \le foldli\ l\ c\ f'\ \sigma'$
proof −
{ fix *l*
have $foldli\ l\ c\ f\ bot = bot$ **by** (*induct l*) (*auto simp*: *STRICT*)
} note [*simp*] = *this*
{ fix *l*
have $foldli\ l\ c\ f'\ top = top$ **by** (*induct l*) (*auto simp*: *STRICT*)
} note [*simp*] = *this*

show *?thesis*
using *B*
apply (*induct l arbitrary*: $\sigma\ \sigma'$)
apply (*auto simp*: *A STRICT dest*: *COND*)
done

qed

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-case True True c*) *f* σ
    $\le$ *foldli l* (*dres-case True True c*) *f′* σ
  apply (*rule it-mono-aux*)
  apply (*simp-all split*: *dres.split-asm add*: *STRICT A*)
  done

lemma *it-mono-aux-dres*:
  assumes *A*: $\bigwedge a\ x.\ f\ a\ x \le f'\ a\ x$
  shows *foldli l* (*dres-case True True c*) ($\lambda x\ s.$ *dbind s* (*f x*)) σ
    $\le$ *foldli l* (*dres-case True True c*) ($\lambda x\ s.$ *dbind s* (*f′ x*)) σ
  apply (*rule it-mono-aux-dres′*)
  apply (*simp-all*)
  apply (*rule dbind-mono*)
  apply (*simp-all add*: *A*)
  done

lemma *iteratei-mono′*:
  assumes *L*: *set-iteratei* α *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-tag it s* (*dres-case True True c*) *f* σ
    $\le$ *IT-tag it s* (*dres-case True True c*) *f′* σ
  proof −
    from *set-iteratei.iteratei-rule*[*OF L*, *OF I*, *unfolded set-iterator-foldli-conv*]
    obtain *l0* **where** *l0-props*: *distinct l0* α *s* = *set l0* *it s* = *foldli l0* **by** *blast*

    from *it-mono-aux-dres′* [*of f f′ l0 c* σ]
    show *?thesis*
      unfolding *IT-tag-def l0-props*(*3*)
      by (*simp add*: *STRICT A*)
  qed

lemma *iteratei-mono*:
  assumes *L*: *set-iteratei* α *invar it*
  assumes *A*: $\bigwedge a\ x.\ f\ a\ x \le f'\ a\ x$
  assumes *I*: *invar s*
  shows *IT-tag it s* (*dres-case True True c*) ($\lambda x\ s.$ *dbind s* (*f x*)) σ
    $\le$ *IT-tag it s* (*dres-case True True c*) ($\lambda x\ s.$ *dbind s* (*f′ x*)) σ
 proof −
    from *set-iteratei.iteratei-rule*[*OF L*, *OF I*, *unfolded set-iterator-foldli-conv*]
    obtain *l0* **where** *l0-props*: *distinct l0* α *s* = *set l0* *it s* = *foldli l0* **by** *blast*

from *it-mono-aux-dres* [*of f f′ l0 c σ*]
show *?thesis*
unfolding *IT-tag-def l0-props*(*3*)
by (*simp add*: *A*)
qed

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* ≡ (*c*,*a*)∈*r*

2.16.1 Set

lemma (**in** *set-empty*) *set-empty-et*[*autoref-ex*]:
(*empty* (),{})∈(*build-rel* (*α*::′*s* ⇒ ′*x set*) *invar*)
by (*auto simp*: *empty-correct*)

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*′∈*s*′)
using *assms* **by** (*auto simp*: *memb-correct*)

lemma (**in** *set-ball*) *set-ball-et*[*autoref-ex*]:
DETREFe s (*build-rel α invar*) *s′* ⟹ *DETREFe P Id P′*
⟹ *DETREFe* (*ball s P*) *Id* (∀ *x*∈*s′*. *P′ x*)

by (*auto simp*: *ball-correct*)

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* ($\exists x \in s'.\ P'\ x$)
by (*auto simp*: *bexists-correct*)

lemma (**in** *set-size*) *set-size-et*[*autoref-ex*]:
DETREFe s (*build-rel* α *invar*) s'
$\Longrightarrow$ *DETREFe* (*size s*) *Id* (*card* s')
by (*auto simp*: *size-correct*)

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'))
by (*auto simp*: *size-abort-correct*)

lemma (**in** *set-union*) *set-union-et*[*autoref-ex*]:
assumes *DETREFe s1* (*build-rel* α*1 invar1*) *s1'*
assumes *DETREFe s2* (*build-rel* α*2 invar2*) *s2'*
shows *DETREFe* (*union s1 s2*) (*build-rel* α*3 invar3*) ($s1' \cup s2'$)
using *assms* **by** (*simp add*: *union-correct*)

lemma (**in** *set-union-dj*) *set-union-dj-et*[*autoref-ex*]:
assumes $s1' \cap s2' = \{\}$
assumes *DETREFe s1* (*build-rel* α*1 invar1*) *s1'*
assumes *DETREFe s2* (*build-rel* α*2 invar2*) *s2'*
shows *DETREFe* (*union-dj s1 s2*) (*build-rel* α*3 invar3*) ($s1' \cup s2'$)
using *assms* **by** (*simp add*: *union-dj-correct*)

lemma (**in** *set-diff*) *set-diff-et*[*autoref-ex*]:
assumes *DETREFe s1* (*build-rel* α*1 invar1*) *s1'*
assumes *DETREFe s2* (*build-rel* α*2 invar2*) *s2'*
shows *DETREFe* (*diff s1 s2*) (*build-rel* α*1 invar1*) ($s1' - s2'$)
using *assms* **by** (*simp add*: *diff-correct*)

lemma (**in** *set-filter*) *set-filter-et*[*autoref-ex*]:
assumes *DETREFe P Id* P'
assumes *DETREFe s* (*build-rel* α*1 invar1*) s'
shows *DETREFe* (*filter P s*) (*build-rel* α*2 invar2*) ($\{e \in s'.\ P'\ e\}$)
using *assms* **by** (*simp add*: *filter-correct*)

lemma (**in** *set-inter*) *set-inter-et*[*autoref-ex*]:
assumes *DETREFe s1* (*build-rel* α*1 invar1*) *s1'*
assumes *DETREFe s2* (*build-rel* α*2 invar2*) *s2'*
shows *DETREFe* (*inter s1 s2*) (*build-rel* α*3 invar3*) ($s1' \cap s2'$)
using *assms* **by** (*simp add*: *inter-correct*)

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

$DETREFe\ x\ Id\ x' \Longrightarrow DETREFe\ s\ (build\text{-}rel\ \alpha\ invar)\ s'$
$\Longrightarrow DETREFe\ (ins\ x\ s)\ (build\text{-}rel\ \alpha\ invar)\ (insert\ x'\ s')$
by (*auto simp*: *ins-correct*)

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

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

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

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 $DETREFe\ (subset\ s1\ s2)\ Id\ (s1' \subseteq s2')$
using *assms* **by** (*simp add*: *subset-correct*)

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 $DETREFe\ (equal\ s1\ s2)\ Id\ (s1' = s2')$
using *assms* **by** (*simp add*: *equal-correct*)

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

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

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 $DETREFe\ (disjoint\ s1\ s2)\ Id\ (s1' \cap s2' = \{\})$
using *assms* **by** (*simp add*: *disjoint-correct*)

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-rel α2 invar2) s2'
  shows RETURN (disjoint-witness s1 s2) ≤ ⇓Id (SPEC
    (λr. if (s1'∩s2'={}) then r=None else (∃ e∈s1'∩s2'. r=Some e)))
  using assms
  apply –
  apply (simp, intro impI conjI)
  apply (simp add: disjoint-witness-None)
  apply (elim conjE)
  apply (cases disjoint-witness s1 s2)
  apply (drule (2) disjoint-witness-correct) apply blast
  apply (drule (2) disjoint-witness-correct) apply blast
  done

lemma (in set-sel') set-sel'-et[autoref-ex]:
  assumes DETREFe P Id P'
  assumes DETREFe s (build-rel α invar) s'
  shows RETURN (sel' s P) ≤⇓Id
    (SPEC (λr. if (∀x∈s'. ¬ P' x) then r=None else (∃x∈s'. P' x ∧ r=Some
x)))
  using assms apply –
  apply (cases sel' s P)
  apply (auto dest: sel'-someD sel'-noneD)
  done

lemma (in set-to-list) set-to-list-et[autoref-ex]:
  assumes DETREFe s (build-rel α invar) s'
  shows RETURN (to-list s) ≤⇓Id (SPEC (λl. set l = s' ∧ distinct l))
  using assms to-list-correct by auto

lemma (in list-to-set) list-to-set-et[autoref-ex]:
  assumes DETREFe l Id l'
  shows DETREFe (to-set l) (build-rel α invar) (set l')
  using assms by (auto simp: to-set-correct)

lemma (in set-sel') pick-et[autoref-ex]:
  assumes s'≠{}
  assumes DETREFe s (build-rel α invar) s'
  shows RETURN (the (sel' s (λ-. True))) ≤⇓Id (SPEC (λx. x∈s'))
  using assms by (auto elim!: sel'E)
```

2.16.2 Map

TODO: Still incomplete

```
lemma (in map-empty) map-empty-t[autoref-ex]:
  DETREFe (empty ()) (build-rel α invar) Map.empty
  by (auto simp: empty-correct)


lemma (in map-lookup) map-lookup-t:
```

assumes *DETREFe k Id k'*
assumes *DETREFe m (build-rel α invar) m'*
shows *DETREFe (lookup k m) Id (m' k')*
using *assms* **by** (*auto simp*: *lookup-correct*)

lemma (**in** *map-update*) *map-update-t*[*autoref-ex*]:
assumes *DETREFe k Id k'*
assumes *DETREFe v Id v'*
assumes *DETREFe m (build-rel α invar) m'*
shows *DETREFe (update k v m) (build-rel α invar) (m'(k'↦v'))*
using *assms* **by** (*auto simp*: *update-correct*)

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-rel α invar) [k'↦v']*
using *assms* **by** (*auto simp*: *sng-correct*)

2.16.3 Unique Priority Queue

TODO: Still incomplete

lemma (**in** *uprio-empty*) *uprio-empty-t*[*autoref-ex*]:
DETREFe (empty ()) (build-rel α invar) Map.empty
by (*auto simp*: *empty-correct*)

lemma (**in** *uprio-isEmpty*) *uprio-isEmpty-t*[*autoref-ex*]:
assumes *DETREFe s (build-rel α invar) s'*
shows
DETREFe (isEmpty s) Id (s'=Map.empty)
DETREFe (isEmpty s) Id (dom s'={})
using *assms* **by** (*auto simp*: *isEmpty-correct*)

lemma (**in** *uprio-insert*) *uprio-insert-t*[*autoref-ex*]:
assumes *DETREFe s (build-rel α invar) s'*
assumes *DETREFe e Id e'*
assumes *DETREFe a Id a'*
shows *DETREFe (insert s e a) (build-rel α invar) (s'(e' ↦ a'))*
using *assms* **by** (*simp add*: *insert-correct*)

lemma (**in** *uprio-pop*) *uprio-pop-t*[*autoref-ex*]:
assumes *dom q'≠{}*
assumes *DETREFe q (build-rel α invar) q'*
shows *RETURN (pop q) ≤ ⇓(rprod Id (rprod Id (build-rel α invar)))*
(SPEC (λ(e, w, rq').
rq' = q'(e := None) ∧
q' e = Some w ∧ (∀ e' w'. q' e' = Some w' ⟶ w ≤ w')))
apply (*rule SPEC-refine-sv*)
apply (*intro rprod-sv single-valued-Id br-single-valued*)
using *assms*

```
  apply auto
  apply (erule (1) popE)
  apply (auto simp: ran-def)
  done

  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* $\Rightarrow$ (*nat*$\times$*nat*) *nres* **where**
  sum-max $V \equiv$ *do* {
    (-,s,m) $\leftarrow$ *WHILE* ($\lambda(V,s,m).\ V\neq\{\}$) ($\lambda(V,s,m).$ *do* {
      $x \leftarrow$ *SPEC* ($\lambda x.\ x\in V$);

$\quad\quad$ *let* $V{=}V{-}\{x\}$;
$\quad\quad$ *let* $s{=}s{+}x$;
$\quad\quad$ *let* $m{=}max\ m\ x$;
$\quad\quad$ $RETURN\ (V,s,m)$
$\quad$ $\})\ (V,0,0)$;
$\quad$ $RETURN\ (s,m)$
$\}$

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 $WHILE\ b\ f\ \sigma_0$, 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 $WHILE\ b\ f\ \sigma_0$ 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 $WHILE_T\ b\ f\ \sigma_0$ 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 $RETURN\ x$ specifies precisely the result x.

The specification statement $SPEC\ \Phi$ 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., $\Psi \Longrightarrow S \leq SPEC\ \Phi$.

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

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 $x \in V$ $\quad$ *sum-max-invar* $V_0\ (V,s,m)$
shows *sum-max-invar* $V_0\ (V - \{x\}, s+x, \mathit{max}\ m\ x)$

In our case the proof is rather straightforward, it only requires the lemma *it-step-insert-iff*, that handles the $V_0 - (V - \{x\})$ terms that occur in the invariant.

using *assms* **unfolding** *sum-max-invar-def* **by** (*auto simp*: *it-step-insert-iff*)

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 *sum-max* $V \leq SPEC\ (\lambda(s,m).\ s = \sum V \wedge m = Max\ V)$

The precondition $V \neq \{\}$ is necessary, as the *Max*-operator from Isabelle/HOL's standard library is not defined for empty sets.

using *PRE* **unfolding** *sum-max-def*
apply (*intro WHILE-rule*[**where** I=*sum-max-invar* V] *refine-vcg*) — Invoke vcg

Note that we have explicitely instantiated the rule for the while-loop with the invariant. If this is not done, the verification condition generator will stop at the WHILE-loop.

apply (*auto intro*: *sum-max-invar-step*) — Discharge step
unfolding *sum-max-invar-def* — Unfold invariant definition
apply (*auto*) — Discharge remaining goals
done

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

definition *sum-max'* :: *nat set* $\Rightarrow$ (*nat*$\times$*nat*) *nres* **where**
  sum-max' $V \equiv$ *do* {
    $(\text{-},s,m) \leftarrow \mathit{WHILE}_T{}^{\mathit{sum\text{-}max'\text{-}invar}\ V}$ $(\lambda(V,s,m).\ V{\neq}\{\})$ $(\lambda(V,s,m).$ *do* {
      $x{\leftarrow}\mathit{SPEC}\ (\lambda x.\ x{\in}V)$;
      let $V{=}V{-}\{x\}$;
      let $s{=}s{+}x$;
      let $m{=}\mathit{max}\ m\ x$;
      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'* $V \leq \mathit{SPEC}\ (\lambda(s,m).\ s{=}\sum V \wedge m{=}\mathit{Max}\ V)$
  using *NE FIN* **unfolding** *sum-max'-def*
  apply (*intro refine-vcg*) — Invoke vcg

This time, the verification condition generator uses the annotated invariant. Moreover, it leaves us with a variant. We have to specify a well-founded relation, and show that the loop body respects this relation. In our case, the set V decreases in each step, and is initially finite. We use the relation *finite-psubset* and the *inv-image* combinator from the Isabelle/HOL standard library.

  apply (*subgoal-tac wf* (*inv-image finite-psubset fst*),
    assumption) — Instantiate variant
  apply *simp* — Show variant well-founded

  unfolding *sum-max'-invar-def* — Unfold definition of invariant
  apply (*auto intro*: *sum-max-invar-step*) — Discharge step

  unfolding *sum-max-invar-def* — Unfold definition of invariant completely
  apply (*auto intro*: *finite-subset*) — Discharge remaining goals
  done

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* (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*)
  }

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* (λ*x*. *True*)), i.e., selection of an element that satisfies the predicate λ*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′*)∈*build-rel ls-α ls-invar*
  shows *sum-max-impl V* ≤ ⇓*Id* (*sum-max V′*)

Let R be a *refinement relation*[1], that relates concrete and abstract values.

Then, the function $\Downarrow R$ maps a result-set over abstract values to the greatest result-set over concrete values that is compatible w.r.t. R. The value *FAIL* is mapped to itself.

[1] Also called coupling invariant.

Thus, the proposition $S \leq \Downarrow R\ S'$ means, that S refines S' w.r.t. R, i.e., every value in the result of S can be abstracted to a value in the result of S'.

Usually, the refinement relation consists of an invariant I and an abstraction function α. In this case, we may use the *br I* α-function to define the refinement relation.

In our example, we assume that the input is in the refinement relation specified by list-sets, and show that the output is in the identity relation. We use the identity here, as we do not change the datatypes of the output.

The proof is done automatically by the refinement verification condition generator. Note that the theory *Collection-Bindings* sets up all the necessary lemmas to discharge refinement conditions for the collection framework.

using *assms* **unfolding** *sum-max-impl-def sum-max-def*
apply (*refine-rcg*) — Decompose combinators, generate data refinement goals
apply (*refine-dref-type*) — Type-based heuristics to instantiate data refinement goals
apply (*auto simp add*: *ls-correct refine-hsimp*) — Discharge proof obligations
done

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* $V \leq$ *SPEC* $(\lambda(s,m).\ s=\sum V' \wedge m=Max\ V')$
proof –
note *sum-max-impl-refine*
also note *sum-max-correct*
finally show *?thesis* **using** *assms* **.**
qed

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

definition *sum-max'-impl* :: *nat ls* $\Rightarrow$ (*nat*$\times$*nat*) *nres* **where**
sum-max'-impl V $\equiv$ *do* {
(-,s,m) $\leftarrow$ $WHILE_T$ $(\lambda(V,s,m).\ \neg$*ls-isEmpty* $V)$ $(\lambda(V,s,m).$ *do* {
$x \leftarrow RETURN$ (*the* (*ls-sel'* V $(\lambda x.\ True)$));
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* $\Longrightarrow$ *sum-max'-impl* $V \leq \Downarrow Id$ (*sum-max'* V')
unfolding *sum-max'-impl-def sum-max'-def*
apply *refine-rcg*

apply *refine-dref-type*
apply (*auto simp*: *refine-hsimp ls-correct*)
done

theorem *sum-max'-impl-correct*:
assumes $(V, V')\in$ *build-rel ls-α ls-invar* **and** $V'\neq\{\}$
shows *sum-max'-impl V* $\leq$ *SPEC* $(\lambda(s,m).\ s=\sum V' \wedge m=Max\ V')$
using *ref-two-step*[*OF sum-max'-impl-refine sum-max'-correct*] *assms*

Note that we do not need the finiteness precondition, as list-sets are always finite. However, in order to exploit this, we have to unfold the *build-rel* construct, that relates the list-set on the concrete side to the set on the abstract side.

apply (*auto simp*: *build-rel-def*)
done

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*

This is done automatically by the transfer procedure of our framework.

unfolding *sum-max-impl-def*
apply (*refine-transfer*)+
done

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* (λ*x*. *True*))) ≫=
(λ*xa*. *let xb* = *ls-delete xa a*; *xc* = *aa* + *xa*; *xd* = *max ba xa*

in dRETURN (*xb*, *xc*, *xd*)))
(V, *0*, *0*) $\gg=$
(λ(*a*, *b*). *case b of* (*aa*, *ba*) $\Rightarrow$ *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*
using *sum-max-code-aux*[*folded sum-max-code-def*] **.**

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*

The proof is done by transitivity, and unfolding some definitions:

using *nres-correctD*[*OF order-trans*[*OF sum-max-code-refine sum-max-impl-correct*, *of V ls-*α *V*]] *assms*
by *auto*

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*
unfolding *sum-max'-impl-def*
apply (*refine-transfer*)
done

thm *sum-max'-code-aux*[*no-vars*]
definition *sum-max'-code* :: *nat ls* $\Rightarrow$ (*nat*$\times$*nat*) **where**
sum-max'-code V $\equiv$
(*let* (*a*, *b*) =
while (λ(V, *s*, *m*). $\neg$ *ls-isEmpty V*)
(λ(*a*, *b*).
case b of
(*aa*, *ba*) $\Rightarrow$
let xa = *the* (*ls-sel' a* (λ*x*. *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*) $\Rightarrow$ (*aa*, *ba*))

theorem *sum-max'-code-refine*: *RETURN* (*sum-max'-code V*) $\leq$ *sum-max'-impl V*
using *sum-max'-code-aux*[*folded sum-max'-code-def*] **.**

theorem *sum-max'-code-correct*:
$[\![ls\text{-}\alpha\ V \neq \{\}]\!] \Longrightarrow sum\text{-}max'\text{-}code\ V = (\sum (ls\text{-}\alpha\ V),\ Max\ (ls\text{-}\alpha\ V))$
using *order-trans*[*OF sum-max'-code-refine sum-max'-impl-correct*,
of V ls-α V]
by *auto*

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*:
$RETURN\ ?sum\text{-}max''\text{-}code \leq sum\text{-}max'\text{-}impl\ V$
unfolding *sum-max'-impl-def*
apply (*refine-transfer the-resI*) — Using *the-resI* for internal monad and result extraction
done

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* (λ*x*. *True*))) ≫=
(λ*xa*. *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*: $RETURN\ (sum\text{-}max''\text{-}code\ V) \leq sum\text{-}max'\text{-}impl\ V$
using *sum-max''-code-aux*[*folded sum-max''-code-def*] **.**

theorem *sum-max''-code-correct*:
$[\![ls\text{-}\alpha\ V \neq \{\}]\!] \Longrightarrow sum\text{-}max''\text{-}code\ V = (\sum (ls\text{-}\alpha\ V),\ Max\ (ls\text{-}\alpha\ V))$
using *order-trans*[*OF sum-max''-code-refine sum-max'-impl-correct*,
of V ls-α V]
by *auto*

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*)
using *PRE* **unfolding** *sum-max-it-def*
apply (*intro FOREACH-rule*[**where** *I*=λ*it* σ. *sum-max-invar V* (*it*,σ)] *refine-vcg*)
apply (*rule FIN*) — Discharge finiteness of iterated set
apply (*auto intro*: *sum-max-invar-step*) — Discharge step
unfolding *sum-max-invar-def* — Unfold invariant definition
apply (*auto*) — Discharge remaining goals
done

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'*)
unfolding *sum-max-it-impl-def sum-max-it-def*

Note that we specified *inj-on-id* as additional introduction rule. This is due to the very general iterator refinement rule, that may also change the set over that is iterated.

using *assms*
apply *refine-rcg* — This time, we don't need the *refine-dref-type* heuristics, as no schematic refinement relations are generated.
apply (*auto simp*: *refine-hsimp*)
done

schematic-lemma *sum-max-it-code-aux*:
nres-of ?sum-max-it-code ≤ *sum-max-it-impl V*
unfolding *sum-max-it-impl-def*
apply *refine-transfer*
done

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*) ≤ *sum-max-it-impl V*
using *sum-max-it-code-aux*[*folded sum-max-it-code-def*] **.**

theorem *sum-max-it-code-correct*:
assumes *ls-α V* ≠ {}
shows
sum-max-it-code V = *dRETURN* (*s*,*m*) ⟹ *s*=∑(*ls-α V*) ∧ *m*=*Max* (*ls-α V*)
(**is** *?P1* ⟹ *?G1*)
and *sum-max-it-code V* ≠ *dFAIL* (**is** *?G2*)
proof −
note *sum-max-it-code-refine*[*of V*]
also note *sum-max-it-impl-refine*[*of V ls-α V*]
also note *sum-max-it-correct*
finally show *?P1* ⟹ *?G1 ?G2* **using** *assms* **by** *auto*
qed

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*:

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

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

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 (λx. x≤(2::nat)*(fst p + snd p)) }
  ≤ do { let (x,y)=p; z←SPEC (λz. z≤x+y);
         a←SPEC (λa. a≤x+y); ASSERT (x>2); RETURN (a+z)}
  apply (rule pw-leI)
  apply (auto simp add: refine-pw-simps split: prod.split)

  apply (rename-tac a b x)
  apply (case-tac x≤a+b)
  apply (rule-tac x=0 in exI)
  apply simp
  apply (rule-tac x=a+b in exI)
  apply (simp)
  apply (rule-tac x=x−(a+b) in exI)
  apply simp
  done
```

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 $b\ f\ \sigma_0$ 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 σ_0 Foreach-loop with explicit continuation condition and annotated invariant.

Alternative syntax: *FOREACH*$_C{}^I$ S c f σ_0.

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

3.5.2 Refinement

op $\leq$::$'a$ *nres* $\Rightarrow$ $'a$ *nres* $\Rightarrow$ *bool* Refinement ordering. $S \leq S'$ means, that every result in S is also a result in S'. Moreover, S may only fail if S' fails. $\leq$ 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: $S \leq \Downarrow R\ S' \longleftrightarrow \Uparrow R\ S \leq S'$.

br α 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* Φ 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 $\alpha = RETURN$ (transfer to plain function for total correct code generation), and $\alpha = nres\text{-}of$ (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 $min\text{-}dist\ v\ v' = (LEAST\ d.\ dist\ v\ d\ v')$
definition $conn\ v\ v' = (\exists\, d.\ dist\ v\ d\ v')$

Properties

In this subsection, we prove some properties of paths.

lemma
shows $connI[intro]$: $dist\ v\ d\ v' \Longrightarrow conn\ v\ v'$
and $connI\text{-}id[intro]$: $conn\ v\ v$
and $connI\text{-}succ[intro]$: $conn\ v\ v' \Longrightarrow v'' \in succ\ v' \Longrightarrow conn\ v\ v''$
by $(auto\ simp$: $conn\text{-}def\ intro$: $dist\text{-}suc\ dist\text{-}z)$

lemma $min\text{-}distI2$:
$[\![\ conn\ v\ v'\ ; \bigwedge d.\ [\![\ dist\ v\ d\ v'; \bigwedge d'.\ dist\ v\ d'\ v' \Longrightarrow d \le d'\ ]\!] \Longrightarrow Q\ d\ ]\!]$
$\Longrightarrow Q\ (min\text{-}dist\ v\ v')$
unfolding $min\text{-}dist\text{-}def$
by $(rule\ LeastI2\text{-}wellorder[$**where** $Q{=}Q$ **and** $a{=}SOME\ d.\ dist\ v\ d\ v'])$
$(auto\ simp$: $conn\text{-}def\ intro$: $someI)$

lemma $min\text{-}distI\text{-}eq$:
$[\![\ dist\ v\ d\ v'; \bigwedge d'.\ dist\ v\ d'\ v' \Longrightarrow d \le d'\ ]\!] \Longrightarrow min\text{-}dist\ v\ v' = d$
by $(force\ intro$: $min\text{-}distI2\ simp$: $conn\text{-}def)$

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

lemma $dist\text{-}z\text{-}iff[simp]$: $dist\ v\ 0\ v' \longleftrightarrow v'{=}v$
by $(auto\ elim$: $dist.cases\ intro$: $dist.intros)$

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

lemma $min\text{-}dist\text{-}z[simp]$: $min\text{-}dist\ v\ v = 0$
by $(rule\ min\text{-}distI2)\ (auto\ intro$: $dist\text{-}z)$

lemma $min\text{-}dist\text{-}z\text{-}iff[simp]$: $conn\ v\ v' \Longrightarrow min\text{-}dist\ v\ v' = 0 \longleftrightarrow v'{=}v$
by $(rule\ min\text{-}distI2)\ (auto\ intro$: $dist\text{-}z)$

lemma $min\text{-}dist\text{-}is\text{-}dist$: $conn\ v\ v' \Longrightarrow dist\ v\ (min\text{-}dist\ v\ v')\ v'$
by $(auto\ intro$: $min\text{-}distI2)$
lemma $min\text{-}dist\text{-}minD$: $dist\ v\ d\ v' \Longrightarrow min\text{-}dist\ v\ v' \le d$
by $(auto\ intro$: $min\text{-}distI2)$

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

lemma $min\text{-}dist\text{-}succ$:
$[\![\ conn\ v\ v'; v'' \in succ\ v'\ ]\!] \Longrightarrow min\text{-}dist\ v\ v'' \le Suc\ (min\text{-}dist\ v\ v')$
by $(rule\ min\text{-}distI2[$**where** $v'{=}v'])$
$(auto\ elim$: $dist.cases\ intro!$: $min\text{-}dist\text{-}minD\ dist\text{-}suc)$

lemma $min\text{-}dist\text{-}suc$:

assumes c: *conn* v v' *min-dist* v $v' = Suc\ d$
shows $\exists\, v''.$ *conn* v $v'' \wedge v' \in$ *succ* $v'' \wedge$ *min-dist* v $v'' = d$
proof –
from *min-dist-is-dist*[*OF* $c(1)$]
have *min-dist* v $v' = Suc\ d \longrightarrow$ *?thesis*
proof *cases*
case (*dist-suc* d' v'') **then show** *?thesis*
using *min-dist-succ*[*of* v v'' v'] *min-dist-minD*[*of* v d v'']
by (*auto simp*: *connI*)
qed *simp*
with c **show** *?thesis* **by** *simp*
qed

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

lemma *min-dist-less*:
assumes *conn src* v *min-dist src* $v = d$ **and** $d' < d$
shows $\exists\, v'.$ *conn src* $v' \wedge$ *min-dist src* $v' = d'$
using *assms*
proof (*induct* d *arbitrary*: v)
case (*Suc* d) **with** *min-dist-suc*[*of src* v] **show** *?case*
by (*cases* $d' = d$) *auto*
qed *auto*

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

corollary *min-dist-le*:
assumes c: *conn src* v **and** d': $d' \le$ *min-dist src* v
shows $\exists\, v'.$ *conn src* $v' \wedge$ *min-dist src* $v' = d'$
using *min-dist-less*[*OF* c, *of min-dist src* v d'] d' c
by (*auto simp*: *le-less*)

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

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

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 (λv. v∈C); let C = C−{v};
      if v=dst then RETURN (True,{},{},{},d)
      else do {
        (V,N) ← FOREACHi (FE-invar V N v) (succ v) (λw (V,N).
          if (w∉V) then do {
            ASSERT (w∉N);
            RETURN (insert w V, insert w N)
          } else RETURN (V,N)
        ) (V,N);
        ASSERT (bfs-invar' src dst (f,V,C,N,d));
        if (C={}) then do {
          let C=N;
          let N={};
          let d=d+1;
          RETURN (f,V,C,N,d)
        } else RETURN (f,V,C,N,d)
      }
    })
    (False,{src},{src},{},0::nat);
  if f then RETURN (Some d) else RETURN None
  }
```

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 ∪ succ v, C − {v}, N ∪ (succ v − V), d)
  using assms
proof (simp (no-asm-use) add: bfs-invar′-def, intro conjI)
  case goal6 then show ?case
    by (metis Un-iff Diff-iff connI-succ le-SucE min-dist-succ)
next
  case goal7 then show ?case
    by (metis Un-iff connI-succ min-dist-succ)
qed blast+
```

The first part of the invariant is preserved if the *if* $C=\{\}$-statement is executed. This is the verification condition for the loop-invariant. Note that preservation of the second part of the invariant is proven easily inside the main proof.

```
lemma invar-empty-step:
  assumes bfs-invar′ src dst (False, V, {}, N, d)
  shows bfs-invar′ src dst (False, V, N, {}, Suc d)
  using assms
proof (simp (no-asm-use) add: bfs-invar′-def, intro conjI impI allI)
  case goal3 then show ?case
    by (metis le-SucI)
next
  case goal4 then show ?case
    by (metis le-SucE subsetD)
next
  case (goal5 v) then show ?case
    using min-dist-suc[of src v Suc d] by metis
next
  case goal6 then show ?case
    by (metis Suc-n-not-le-n)
qed blast+
```

The invariant holds initially.

```
lemma invar-init: bfs-invar src dst (False, {src}, {src}, {}, 0)
  by (auto simp: bfs-invar-def bfs-invar′-def empty-assm-def)
    (metis min-dist-suc min-dist-z-iff)
```

The invariant is preserved if we break the loop.

```
lemma invar-break:
  assumes bfs-invar src dst (False, V, C, N, d)
  assumes dst∈C
  shows bfs-invar src dst (True, {}, {}, {}, d)
  using assms unfolding bfs-invar-def bfs-invar′-def empty-assm-def
  by auto
```

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

```
lemma invar-final-succeed:
```

assumes *bfs-invar′ src dst* (*True*, *V*, *C*, *N*, *d*)
shows *min-dist src dst* = *d*
using *assms* **unfolding** *bfs-invar′-def*
by *auto*

If the loop terminated normally, there is no path between *src* and *dst*.

The lemma is formulated as deriving a contradiction from the fact that there is a path and the loop terminated normally.

Note the proof language *Isar* that was used here. It allows one to write human-readable proofs in a theorem prover.

lemma *invar-final-fail*:
assumes *C*: *conn src dst* — There is a path between *src* and *dst*.
assumes *INV*: *bfs-invar′ src dst* (*False*, *V*, {}, {}, *d*)
shows *False*
proof –

We make a case-distinction whether $d' \leq d$:

have *min-dist src dst* ≤ *d* ∨ *d* + *1* ≤ *min-dist src dst* **by** *auto*
moreover **{**

Due to the invariant, any node with a path of length at most d is contained in V

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

This applies in the first case, hence we obtain that *dst*∈*V*

assume *A*: *min-dist src dst* ≤ *d*
moreover from *C* **have** *dist src* (*min-dist src dst*) *dst*
by (*blast intro*: *min-dist-is-dist*)
ultimately have *dst*∈*V* **by** *blast*

However, as C and N are empty, the invariant also implies that *dst* is not in V:

moreover from *INV* **have** *dst*∉*V* **unfolding** *bfs-invar′-def* **by** *auto*

This yields a contradiction:

ultimately have *False* **by** *blast*
} **moreover** **{**

In the case $d{+}1 \leq d'$, we also obtain a node that has a shortest path of length $d{+}1$:

assume *d*+*1* ≤ *min-dist src dst*
with *min-dist-le*[*OF C*] **obtain** *v′* **where**
conn src v′   *min-dist src v′* = *d* + *1*
by *auto*

However, the invariant states that such nodes are either in N or are successors of C. As N and C are both empty, we again get a contradiction.

```
    with INV have False unfolding bfs-invar'-def by auto
  } ultimately show ?thesis by blast
qed
```

Finally, we prove our algorithm correct: The following theorem solves both verification tasks.

Note that a proposition of the form $S \sqsubseteq SPEC\ \Phi$ states partial correctness in our framework, i.e., S refines the specification Φ.

The actual specification that we prove here precisely reflects the two verification tasks: *If the algorithm fails, there is no path between src and dst, otherwise it returns the length of the shortest path.*

The proof of this theorem first applies the verification condition generator (*apply* (*intro refine-vcg*)), and then uses the lemmas proved beforehand to discharge the verification conditions. During the *auto*-methods, some trivial verification conditions, e.g., those concerning the invariant of the inner loop, are discharged automatically. During the proof, we gradually unfold the definition of the loop invariant.

```
definition bfs-spec src dst ≡ SPEC (
  λr. case r of None ⇒ ¬ conn src dst
          | Some d ⇒ conn src dst ∧ min-dist src dst = d)
theorem bfs-correct: bfs src dst ≤ bfs-spec src dst
  unfolding bfs-def bfs-spec-def
  apply (intro refine-vcg)

  unfolding FE-invar-def
  apply (auto simp:
    intro: invar-init invar-break )

  unfolding bfs-invar-def
  apply (auto simp:
    intro: invar-succ-step invar-empty-step invar-final-succeed)

  unfolding empty-assm-def
  apply (auto intro: invar-final-fail)

  unfolding bfs-invar'-def
  apply auto
  done

end

end
```

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* $\leq$ *Graph.bfs-spec* (*ls-*$\alpha\circ$*succ*) *src dst*
proof −

As list-sets are always finite, our setting satisfies the finitely-branching assumption made about graphs

  interpret *Graph ls-*$\alpha\circ$*succ*
    by *unfold-locales simp*

The refinement framework can prove automatically that the implementation refines the abstract algorithm.

The notation $S \leq \Downarrow R\ S'$ means, that the program S refines the program S' w.r.t. the refinement relation (also called coupling invariant occasionally) R.

In our case, the refinement relation is the identity, as the result type was not refined.

  have *bfs-impl succ src dst* $\leq \Downarrow$*Id* (*Graph.bfs* (*ls-*$\alpha\circ$*succ*) *src dst*)
    unfolding *bfs-impl-def bfs-def*

    apply (*refine-rcg*)
    apply (*refine-dref-type*)

    apply (*rule inj-on-id* | *simp add*: *refine-hsimp*
      hs-correct hs.sng-correct ls-correct ls.sng-correct
      split: *prod.split prod.split-asm*) +
    done

The result then follows due to transitivity of refinement.

  also have ... $\leq$ *bfs-spec src dst*
    by (*simp add*: *bfs-correct*)
  finally show *?thesis* **.**
qed

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 ≤ bfs-impl succ src dst
  unfolding bfs-impl-def
  apply (refine-transfer)
  done

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) ≤ bfs-impl succ src dst
  using bfs-code-refine-aux[folded bfs-code-def] .
```

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-code succ src dst = *dRETURN None*
    $\Longrightarrow \neg$(*Graph.conn* (*ls-*α $\circ$ *succ*) *src dst*)
  bfs-code succ src dst = *dRETURN* (*Some d*)
    $\Longrightarrow$ *Graph.conn* (*ls-*α $\circ$ *succ*) *src dst*
       $\wedge$ *Graph.min-dist* (*ls-*α $\circ$ *succ*) *src dst* = *d*
  bfs-code succ src dst $\neq$ *dFAIL*
proof –
  interpret *Graph ls-*$\alpha$$\circ$*succ*
    by *unfold-locales simp*

  from *order-trans*[*OF bfs-code-refine bfs-impl-correct*, *of succ src dst*]
  show *bfs-code succ src dst* = *dRETURN None*
    $\Longrightarrow \neg$(*Graph.conn* (*ls-*α $\circ$ *succ*) *src dst*)
    bfs-code succ src dst = *dRETURN* (*Some d*)
    $\Longrightarrow$ *Graph.conn* (*ls-*α $\circ$ *succ*) *src dst*
       $\wedge$ *Graph.min-dist* (*ls-*α $\circ$ *succ*) *src dst* = *d*
    bfs-code succ src dst $\neq$ *dFAIL*
    apply (*unfold bfs-spec-def*)
    apply (*auto split*: *option.split-asm*)
    done
qed

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* $\Rightarrow$ *nat dlist*
where
  nat-list $\equiv$ *dlist-of-list*

definition *succ* :: *nat* $\Rightarrow$ *nat dlist*
where
  succ n = *nat-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-code-succ m n = *bfs-code succ* (*nat-of-integer m*) (*nat-of-integer n*)

ML-val ⟪
(∗ *Execute algorithm for some node pairs.* ∗)
@{*code bfs-code-succ*} *1 1*;
@{*code bfs-code-succ*} *1 2*;
@{*code bfs-code-succ*} *1 5*;
@{*code bfs-code-succ*} *1 7*;
⟫

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 **by** (*simp add*: *RELATES-def*)

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

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

lemma [*simp*]: *single-valued* (*converse word-nat-rel*)
unfolding *word-nat-rel-def*
apply (*auto simp del*: *build-rel-def*)
by (*metis injI order-antisym order-eq-refl word-le-nat-alt*)

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

definition test-impl :: word32 ⇒ word32 ⇒ word32 set nres where
  test-impl x y ≡ do {
    let S={};
    (S,-,-) ← WHILE (λ(S,x,y). x>0) (λ(S,x,y). do {
      let S=S∪{y};
      let x=x − 1;
      let y=y + 1;
      RETURN (S,x,y)
    }) (S,x,y);
    RETURN S
  }

lemma test-impl-refine:
  assumes x'+y'<2^len-of TYPE(32)
  assumes (x,x')∈word-nat-rel
  assumes (y,y')∈word-nat-rel
  shows test-impl x y ≤ ⇓(map-set-rel word-nat-rel) (test x' y')
proof −
  from assms show ?thesis
    unfolding test-impl-def test-def
    apply (refine-rcg)
```

apply (*refine-dref-type*)
apply (*auto simp*: *refine-hsimp*)
done
qed

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*}
apply (*subst rfoldl.simps*, *simp*)+
done

lemma *rfoldl-refines*[*refine*]:
assumes *SV*: *single-valued Rs*
assumes *REFF*: ⋀*x x′ s s′*. ⟦ (*s*,*s′*)∈*Rs*; (*x*,*x′*)∈*Rl* ⟧
⟹ *f s x* ≤ ⇓*Rs* (*f′ s′ x′*)
assumes *REF0*: (*s0*,*s0′*)∈*Rs*
assumes *REFL*: (*l*,*l′*)∈*map-list-rel Rl*
shows *rfoldl f s0 l* ≤ ⇓*Rs* (*rfoldl f′ s0′ l′*)
using *REFL*[*simplified*] *REF0*
apply (*induct arbitrary*: *s0 s0′ rule*: *list-all2-induct*)
apply (*simp add*: *REF0 RETURN-refine-sv*[*OF SV*])
apply (*simp only*: *rfoldl-simps*)
apply (*refine-rcg*)
apply (*rule REFF*)
apply *simp-all*
done

lemma *transfer-rfoldl*[*refine-transfer*]:
assumes ⋀*s x*. *RETURN* (*f s x*) ≤ *F s x*

```
  shows RETURN (foldl f s l) ≤ rfoldl F s l
  using assms
  apply (induct l arbitrary: s)
  apply simp
  apply (simp only: foldl-Nil foldl-append rfoldl-simps)
  apply simp
  apply (rule order-trans[rotated])
  apply (rule refine-transfer)
  apply assumption
  apply assumption
  apply simp
  done
```

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 (λs x. do {
    ASSERT (x≠{});
    y←SPEC (λy. y∈x);
    RETURN (s+y)
  }) s0 l


lemma [simp]:
  pick-sum s [] = RETURN s
  pick-sum s (x#l) = do {
    ASSERT (x≠{}); y←SPEC (λy. y∈x); pick-sum (s+y) l
  }
  unfolding pick-sum-def
  apply simp-all
  done

lemma foldl-mono:
  assumes ⋀x. mono (λs. f s x)
  shows mono (λs. foldl f s l)
  apply (rule monoI)
  using assms
  apply (induct l)
  apply (auto dest: monoD)
  done

lemma pick-sum-correct:
  assumes NE: {}∉set l
  assumes FIN: ∀x∈set l. finite x
  shows pick-sum s0 l ≤ SPEC (λs. s ≤ foldl (λs x. s+Max x) s0 l)
  using NE FIN
```

```
  apply (induction l arbitrary: s0)
  apply (simp)
  apply (simp)
  apply (intro refine-vcg)
  apply blast
  apply simp
  apply (rule order-trans)
  apply assumption
  apply (rule SPEC-rule)
  apply (erule order-trans)
  apply (rule monoD[OF foldl-mono])
  apply (auto intro: monoI)
  done

definition pick-sum-impl s0 l ≡
  rfoldl (λs 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 (λx x′. (x,x′)∈build-rel ls-α ls-invar) l l′
  shows pick-sum-impl s0 l ≤ ⇓Id (pick-sum s0 l′)
  unfolding pick-sum-def pick-sum-impl-def
  using A
  apply (refine-rcg)
  apply (refine-dref-type)
  apply (simp-all add: refine-hsimp)
  done

schematic-lemma pick-sum-code-aux: RETURN ?f ≤ pick-sum-impl s0 l
  unfolding pick-sum-impl-def
  apply refine-transfer
  done

thm pick-sum-code-aux[no-vars]
definition pick-sum-code s0 l ≡
  foldl (λs 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
  using pick-sum-code-aux[folded pick-sum-code-def] .

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)
  unfolding exploresl-def apply −
  apply (refine-autoref)
  done
```

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*)
  unfolding *exploresl-def* **apply** −
  apply (*refine-autoref*
    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*]
)
  done

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*)
  unfolding *exploresl-def* **apply** −
  apply (*refine-autoref*)

Push refinement as far as possible

  apply (*refine-autoref (ss)*)+
  back — There are many alternatives. However, already the first goal of the first alternative, i.e. (*?cb10*, *False*) ∈ *?Rb10*, indicates the problem — there is no specification to translate booleans.

```
oops
```

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)
  unfolding explores1-def
  apply refine-autoref
  done
```

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*)
  unfolding *explores1'-def*
  apply *refine-autoref*
  done

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 (λ*D* (*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 ≤ SPEC (λr. r ⟶ (v0,vd)∈E*)
  unfolding dfs-def
  apply (rule RECT-rule[where
    Φ=λ(V,v). (v0,v)∈E* ∧ V⊆E* ``{v0} and
    V=finite-psupset (E* ``{v0}) <*lex*> {}
    ])
    apply (refine-mono)

    apply blast

    apply (simp)

    apply (intro refine-vcg)
      apply simp

      apply simp

      apply (rule-tac I=λ- r. r ⟶ (v0, vd) ∈ E* in FOREACHc-rule)
        apply simp
        apply (rule finite-subset[of - (E* ``{v0})])
        apply (auto) []
        apply simp

        apply simp

        apply simp

        apply (tactic ⟪ Refine-Misc.rprems-tac @{context} 1 ⟫)
        apply simp
        apply (auto) []
        apply (auto simp: finite-psupset-def) []

      apply assumption

      apply assumption
  done
```

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 $RETURN\ (?f) \leq \Downarrow Id\ (dfs\ E\ vd\ v0)$
 apply (*rule autoref-det-optI*[**where** α=*RETURN*])

Data Refinement Phase

 using *IdI*[*of E*] *IdI*[*of vd*] *IdI*[*of v0*] — Declare parameters as identities
 unfolding *dfs-def*
 apply (*refine-autoref* — Automatic refinement
  spec-Id[**where** *'a='v*]
  spec-R[**where** *'c='v hs* **and** *'a='v set*]
)

Transfer Phase

 apply (*subgoal-tac* $\bigwedge b.\ set\text{-}iterator\ (\lambda c\ f.$
  $(IT\text{-}tag\ ls\text{-}iteratei\ (succi\ b))\ c\ f)$
  $(E\ ``\ \{b\}))$ — Prepare transfer of iterator
 apply (*refine-transfer the-resI*
) — Transfer using monad. Hence *the-resI*

 apply (*subgoal-tac* — Show iterator (Left over from subgoal-tac above)
  $(E\ ``\ \{b\}) = ls\text{-}\alpha\ (succi\ b))$
 apply (*simp only*:)
 apply (*rule ls.it-is-iterator*)
 apply (*auto simp*: *E-def*) [*2*]

Optimization Phase. We apply no optimizations here.

 apply (*rule refl*)
 done

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* $\equiv$
  $(\lambda f\ (a,\ b).$
   if $b = t$ *then dRETURN True*
   else if hs-memb b a then dRETURN False
    else let $xa = hs\text{-}ins\text{-}dj\ b\ a$
     in IT-tag ls-iteratei (*succi b*)

(*dres-case True True* (*op* = *False*))
(λx *s*. *s* $\gg=$ (λs. *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*
unfolding *dfs-rec-def*
apply (*rule RECT-unfold*)
unfolding *dfs-rec-body-def*
apply (*refine-mono*)
done

We fold the definitions, to get a nice refinement lemma

lemma *dfs-impl-refine*:
RETURN (*dfs-impl succi t s0*)
$\leq$ $\Downarrow$*Id* (*dfs* {(*s*,*s'*). *s'*$\in$*ls*-α (*succi s*)} *t s0*)
using *dfs-impl-refine-aux*[*folded dfs-rec-body-def dfs-rec-def*]
unfolding *dfs-impl-def* **.**

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 –
note *dfs-impl-refine*[*of succi vd v0*]
also note *dfs-correct*[*OF F*, *of vd*, *unfolded E-def*]
finally show *?thesis* **using** *R* **by** (*auto simp*: *E-def*)
qed

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.