Refinement Framework Userguide

By Peter Lammich

March 12, 2013

Contents

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.

2 Guided Tour

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

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×nat) nres* **where**
  sum-max V $\equiv$ *do {*
    (-,s,m) $\leftarrow$ *WHILE* *(λ(V,s,m). V≠{})* *(λ(V,s,m). do {*
    x←SPEC (λx. x∈V);
    let V=V−{x};
    let s=s+x;
    let m=max m x;
    RETURN (V,s,m)
    }) (V,0,0);
    RETURN (s,m)
  }

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 5.1.

A while-loop has the form *WHILE b f σ_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 σ_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 σ_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 Φ* 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.

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., $\Psi \implies S \leq SPEC\ \Phi$.

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

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

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** $I = sum\text{-}max\text{-}invar\ V$] *refine-vcg*) — Invoke vcg

Note that we have explicitly 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 $WHILE^I\ b\ f\ \sigma_0$. 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,-,-)=σ *in finite* ($V_0 - V$))

definition *sum-max′* :: *nat set* $\Rightarrow$ (*nat*$\times$*nat*) *nres* **where**

```
sum-max′ V ≡ do {
  (-,s,m) ← WHILE_T^{sum-max′-invar V} (λ(V,s,m). V≠{}) (λ(V,s,m). do {
    x←SPEC (λx. x∈V);
    let V=V−{x};
    let s=s+x;
    let m=max m x;
    RETURN (V,s,m)
  }) (V,0,0);
  RETURN (s,m)
}
```

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

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 [1, 2].

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 ⇓ 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.

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

---

[1]Also called coupling invariant.

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×nat) nres* **where**
  sum-max'-impl $V \equiv$ *do* {
    $(\text{-},s,m) \leftarrow$ *WHILE$_T$* $(\lambda(V,s,m).\ \neg ls\text{-}isEmpty\ V)\ (\lambda(V,s,m).\ do$ {
      $x\leftarrow RETURN\ (the\ (ls\text{-}sel'\ V\ (\lambda x.\ True)))$;
      let $V=ls\text{-}delete\ x\ V$;
      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* $V \leq \Downarrow Id\ (sum\text{-}max'\ V')$
  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

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* ≤ *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*) ≫=
 (λ(*a*, *b*). *case b of* (*aa*, *ba*) ⇒ *dRETURN* (*aa*, *ba*)))

A simple folding gives us the desired refinement lemma

theorem *sum-max-code-refine*: *nres-of* (*sum-max-code V*) ≤ *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* ≠ {}
  shows *sum-max-code V = dRETURN* (*s,m*) ⟹ *s=*∑ (*ls-α V*) ∧ *m=Max* (*ls-α V*)

and *sum-max-code V ≠ 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 ≤ 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 ⇒ (nat×nat)* **where**
  sum-max'-code V ≡
 (let (a, b) =
      while (λ(V, s, m). ¬ ls-isEmpty V)
      (λ(a, b).
        case b of
        (aa, ba) ⇒
         let xa = the (ls-sel' a (λx. True)); xb = ls-delete xa a;
           xc = aa + xa; xd = max ba xa
        in (xb, xc, xd))
      (V, 0, 0)
 in case b of (aa, ba) ⇒ (aa, ba))

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

theorem *sum-max'-code-correct*:
  [[*ls-α V ≠ {}*]] ⟹ *sum-max'-code V = (∑ (ls-α V), Max (ls-α 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-max''-code ≤ sum-max'-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* $\Rightarrow$ *(nat×nat)* **where**
  sum-max''-code V $\equiv$
(the-res
  (dWHILET $(\lambda(V,\ s,\ m).\ \neg\ ls\text{-}isEmpty\ V)$
    $(\lambda(a,\ b).$
      case b of
      $(aa,\ ba) \Rightarrow$
        dRETURN (the (ls-sel' a $(\lambda x.\ True)))$ $\gg=$
        $(\lambda xa.$ *let xb = ls-delete xa a; xc = aa + xa; xd = max ba xa*
            in dRETURN (xb, xc, xd)))
    (V, 0, 0) $\gg=$
    $(\lambda(a,\ b).$ *case b of* $(aa,\ ba) \Rightarrow$ *dRETURN (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-max''-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** $-$

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* σ_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* $\Rightarrow$ *(nat×nat) nres* **where**
  sum-max-it V $\equiv$ *FOREACH V* $(\lambda x\ (s,m).\ RETURN\ (s+x,max\ m\ x))\ (0,0)$

theorem *sum-max-it-correct*:
  assumes *PRE*: $V \neq \{\}$ **and** *FIN*: *finite V*
  shows *sum-max-it V* $\leq$ *SPEC* $(\lambda(s,m).\ s=\sum V \wedge m=Max\ V)$
  using *PRE* **unfolding** *sum-max-it-def*
  apply *(intro FOREACH-rule[**where** I=$\lambda it\ \sigma$. 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*) (*λx* (*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*)))
  (*λx 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) $\Longrightarrow$ *s=$\sum$(ls-α V)* $\wedge$ *m=Max (ls-α V)*
  (**is** *?P1* $\Longrightarrow$ *?G1*)
  and *sum-max-it-code V $\neq$ dFAIL* (**is** *?G2*)
proof −
  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* $\Longrightarrow$ *?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   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 ($\lambda x.\ x \leq (2::nat)*(fst\ p + snd\ p)$) }*
  $\leq$ *do { let (x,y)=p; z$\leftarrow$SPEC ($\lambda z.\ z \leq x+y$);*
        a$\leftarrow$SPEC ($\lambda a.\ a \leq x+y$); 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
```

4 Arbitrary Recursion (TBD)

Explain REC and REC_T.

See examples/Recursion.

To Be Done

5 Reference

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←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 Φ; f}*.

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

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

REC$_T$ body Recursion for total correctness. Returns *FAIL* on nontermination.

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

WHILE$_T$ b f σ_0 Total correct while-loop. If there is a non-terminating path, the result is *FAIL*.

WHILE^I b f σ_0, *WHILE$_T$^{I} b f σ_0* While-loop with annotated invariant. It is asserted that the invariant holds.

FOREACH S f σ_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 σ_0 Foreach-loop with annotated invariant.

　　Alternative syntax: *FOREACH^I S f σ_0*.

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

FOREACH$_C$ S c f σ_0 Foreach-loop with explicit continuation condition.

　　Alternative syntax: *FOREACH$_C$ S c f σ_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.

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.

5.3 Proof Tools

Verification Condition Generator:

> **Method:** *intro refine-vcg*
>
> **Attributes:** *refine-vcg*
>
> Transforms a subgoal of the form $S \leq SPEC\ \Phi$ into verification conditions by decomposing the structure of S. Invariants for loops without annotation must be specified explicitly 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). ...) 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.

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

References

[1] 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.

[2] 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.