universität freiburg

Refinement To Code

Mathias Fleury

January 16, 2024

```
theory Refinement
  imports Refine_Imperative_HOL.IICF
begin
```

Now you need to install the AFP `isa-afp.org`.
Then you need to pick the IICF as base session.

What do we want?

- Express abstract algorithm
- Refine them to go from abstract to less abstract with algo changes and type changes
- Generate code

Refinement and Specification

WARNING

- Proofs are absolutely horrible
- And you will give up Isar
- Understanding goals is even worse than normal goals

Refinement and Specification

A new Language

Use the non-determinism exception monad. The result is `FAIL` or `RES S` where S is the set of all outcomes.

Beware: `RES {}` is `bot`. It is the non-refinable term.

The type is `'a nres`.

There is a nice do notation:

```
term ‹do {
  x ← f S;
  y ← SPEC (λy. y = x+1);
  if x = 1 then RETURN (x + y)
  else RETURN (2*x)
}›
```

WARNING: type errors are absolutely horrible, because the do-monad is overloaded.

And we have loops:

definition *my_sum_list* :: ‹*nat list* ⇒ *nat nres*› where

```
‹my_sum_list xs = (do {
  (x, _) ← WHILE_T λ(x, i). True (λ(x, i). i < length xs)
   (λ(x, i). do {
     ASSERT (i < length xs);
     let a = xs ! i;
     RETURN (x+a, Suc i)
   })
   (0, 0);
  RETURN x
  })›
```

1. The invariant part is optional
2. *ASSERT* are optional currently but necessary to generate code
3. What happens if the return type becomes *nat list nres*?

Refinement and Specification

Refining toward a Specification

```
lemma ‹my_sum_list xs ≤ SPEC(λa. a = sum_list xs)›
proof -
  have wf: ‹wf (measure (λ(_, i). length xs - i))›
    by auto
  show ?thesis
    unfolding my_sum_list_def
    apply (refine_vcg)
    — First: what does the goal even mean
    sorry
qed
```

The relation has to terminate

Solution:

```
definition my_sum_list2 :: ‹nat list ⇒ nat nres› where
 ‹my_sum_list2 xs = (do {
  (x, _) ← WHILE_T λ(x, i). i ≤length xs ∧ x = (sum_list (take i xs)) (λ(x, i). i <
length xs)
   (λ(x, i). do {
     ASSERT (i < length xs);
     let a = xs ! i;
     RETURN (x+a, Suc i)
   })
   (0, 0);
  RETURN x
})›
lemma ‹my_sum_list2 xs ≤ SPEC(λa. a = sum_list xs)›
proof -
  have wf: ‹wf (measure (λ(_, i). length xs - i))›
    by auto
  show ?thesis
    unfolding my_sum_list2_def
    apply (refine_vcg)
    apply (rule wf)
```

Refinement and Specification

Refining

Let's do the most stupid refinement possible, *int* instead of *nat*:

definition *my_sum_list3 :: ‹int list ⇒ int nres›* where

```
‹my_sum_list3 xs = (do {
  (x, _) ← WHILE_T^λ(x, i). i ≤length xs ∧ x = (sum_list (take i xs)) (λ(x, i). i <
length xs)
    (λ(x, i). do {
      ASSERT (i < length xs);
      let a = xs ! i;
      RETURN (x+a, Suc i)
    })
    (0, 0);
  RETURN x
})›
```

```
lemma
  assumes ‹(xs, ys) ∈ ⟨{(a,b). a = int b}⟩ list_rel›
  shows
    ‹my_sum_list3 xs ≤ ⇓{(a,b). a = int b} (my_sum_list2 ys)›
proof -
  show ?thesis
    unfolding my_sum_list3_def my_sum_list2_def
    apply refine_vcg
    oops
```

Remark that

- names are not lost by refinement
- everything is eagerly split

The hard-learned lessons:

- always put the invariants as a definition, never inline them
- put as many invariants as possible
- keep all properties through the invariants, do not drop them.
- refine as locally as possible

Refinement and Specification

Exercise

Refine the LRAT where you express clauses as lists.

Code Generation

Let's have a look at an example:

```
sepref_definition my_sum_list3_impl
  is ‹my_sum_list3›
  :: ‹(array_assn int_assn)^k →_a int_assn›
  unfolding my_sum_list3_def
  by sepref

export_code my_sum_list3_impl in SML_imp module_name Code
```

What happens without the assertion?

Code Generation

A New Monad

Does this seem familiar?

```
term ‹do {
   a ← return (1::int);
  if a = 0 then return 0
  else return (a + 1)
}›
```

Automatic translation with Sepref:

- translates all instruction
- must be deterministic (no *SPEC*/*RES*)
- no translation of success
- drops assertion

Code Generation

Imperative Representation

Relies on separation logic

term `<h ⊨ P * (Q :: assn) * true>`

where:

- `h` is the heap mapping addresses to values
- `P * Q * true` is an assertion over the heap

Let's have a look at:

term `list_assn`

Then we can get to Hoare triples:

term *<<P> f <Q>*$_t$*>*

Separation logic is a pain when for owning structures on the heap (like arrays of arrays).

Code Generation

Sepref

Sepref automatically translates constants according to the rules declared in

thm *sepref_fr_rules*

The hard-learned lessons:

- Sepref is slow (Sepref/SML faster than Sepref/LLVM IR) for large states
- The hard part: refining different components at the same time where the abstract version does not work
- The usual performance bugs remain around (allocating inside a loop instead of outside), but are harder to see
- For Sepref/SML: default is GMP integer

end