Calculating correct programs

Wouter Swierstra

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Program calculation - The dream of the 70s

Instead of *writing programs*, we should *derive* a executable program from its specification.

Program calculation – The dream of the 70s

Instead of *writing programs*, we should *derive* a executable program from its specification.

The *refinement calculus* provides a precise logic, defining when such a derivation is valid.

In other words, it describes how to compute an *implementation* from a *specification*.

Why care about program calculation?

Nobody **proves** their programs correct…

… let alone **calculates** a program from a specification.

Why care about program calculation?

Nobody **proves** their programs correct…

… let alone **calculates** a program from a specification.

But understanding program calculation answers questions:

- What constitutes a specification?
- What programs satisfy a specification?
- What steps are valid when deriving a program from its specification?

Challenge

- The refinement calculus mixes specifications and programs.
- Interactive proof assistants based on type theory provide a single framework for proving and programming.
- How can we perform such refinement proofs in a proof assistant such as Coq?

Refinement calculus

Refinement calculus: specifications

Specifications are typically given in the form of a precondition and postcondition.

The specification $[p, q]$ is satisfied by a program that, provided the precondition p holds initially, terminates in a state where the postcondition q holds.

Refinement

The central notion of the refinement calculus is that of *program refinement*,

$$p_1 \sqsubseteq p_2$$

This refinement holds precisely when

$$\forall P,\ \mathsf{wp}(p_1, P) \Rightarrow \mathsf{wp}(p_2, P)$$

This notion of refinement can be applied *both* to programs and specifications.

Intuitively, when p_2 refines p_1 we may think of p_2 as ‘more specific’ than p_1.

Refinement calculations

Starting from a specification S, we can iteratively refine it:

$$S \sqsubseteq P_1 \sqsubseteq ... \sqsubseteq P_n \sqsubseteq C$$

Here S is a specification of the form $[p, q]$ and C is a piece of executable code. The intermediate programs P_i are a mix of code and specifications.

Refinement laws

Rather than prove every step of such a calculation correct in terms of weakest precondition semantics, there are numerous derived laws.

Lemma (skip)

If $pre \Rightarrow post$, then $[pre, post] \sqsubseteq$ skip

Lemma (Following assignment)

For any term E, we have $[pre, post] \ \sqsubseteq \ [pre, post[w \backslash E]]; w ::= E$

Refinement laws

Rather than prove every step of such a calculation correct in terms of weakest precondition semantics, there are numerous derived laws.

Lemma (skip)

If $pre \Rightarrow post$, then $[pre, post] \sqsubseteq$ skip

Lemma (Following assignment)

For any term E, we have $[pre, post] \sqsubseteq [pre, post[w\backslash E]]; w ::= E$

Note: Deciding how to apply these laws requires creativity!

Refinement calculations: example

$$[x = X \wedge y = Y, x = Y \wedge y = X]$$

Refinement calculations: example

$$[x = X \wedge y = Y, x = Y \wedge y = X]$$

$\sqsubseteq$ { by the following assignment law }

$$[x = X \wedge y = Y, t = Y \wedge y = X]; \text{x ::= t}$$

Refinement calculations: example

$$[x = X \wedge y = Y, x = Y \wedge y = X]$$

$\sqsubseteq$ { by the following assignment law }

$[x = X \wedge y = Y, t = Y \wedge y = X]$; x ::= t

$\sqsubseteq$ { by the following assignment law }

$[x = X \wedge y = Y, t = Y \wedge x = X]$; y ::= x; x ::= t

Refinement calculations: example

$$[x = X \wedge y = Y, x = Y \wedge y = X]$$

$\sqsubseteq$ { by the following assignment law }

$[x = X \wedge y = Y, t = Y \wedge y = X]$; x ::= t

$\sqsubseteq$ { by the following assignment law }

$[x = X \wedge y = Y, t = Y \wedge x = X]$; y ::= x; x ::= t

$\sqsubseteq$ { by the following assignment law }

$[x = X \wedge y = Y, y = Y \wedge x = X]$; t ::= y; y ::= x; x ::= t

Refinement calculations: example

$[x = X \wedge y = Y, x = Y \wedge y = X]$

$\sqsubseteq$ { by the following assignment law }

$[x = X \wedge y = Y, t = Y \wedge y = X]$; x ::= t

$\sqsubseteq$ { by the following assignment law }

$[x = X \wedge y = Y, t = Y \wedge x = X]$; y ::= x; x ::= t

$\sqsubseteq$ { by the following assignment law }

$[x = X \wedge y = Y, y = Y \wedge x = X]$; t ::= y; y ::= x; x ::= t

$\sqsubseteq$ { by the law for skip }

skip ; t ::= y; y ::= x; x ::= t

Refinement on paper

Calculating programs from their specification on paper has its drawbacks:

- Complex derivations require a great deal of bookkeeping – and it's easy to make mistakes.
- Upon completion, you still need to transcribe the derived program to a programming language.

Can we do better?

Embedding in Coq

Embedding in Coq

Together with Joao Alpuim, we showed how to *embed* the refinement calculus in Coq, enabling us to:

- state and prove refinement laws;
- use such laws to interactively derive a program from its specification;
- use the full power of Coq to automate proofs and guide the development;
- generate an executable program from a completed derivation.

Basic definitions

We can represent specifications as a pair of a pre- and postcondition:

```
Definition Pred (A : Type) : Type := A -> Type.

Record PT (A : Type) : Type :=
  MkPT { pre : Pred S;
         post : ∀ s : S, pre s -> Pred (A × S)}
```

Note: the postcondition is a relation between an input state `s` that satisfies the precondition, the final result returned and the output state.

Refinement

We can assign a weakest precondition semantics to pre- and postcondition pairs `PT` as predicate transformers.

Next we can define a `Refinement` relation on `PT`, written $pt_1 \sqsubseteq pt_2$:

- the precondition of pt_1 implies that of pt_2
- the postcondition of pt_2 implies that of pt_1

And we can show that it is sound and complete with respect to the weakest precondition semantics.

Derived laws

We can already prove general properties of refinements, such as:

```
Lemma strengthenPost :
  (∀ s x s', Q1 s (x,s') -> Q2 s (x,s')) ->
  [ P , Q2 ] ⊑ [ P , Q1 ].
```

Derived laws

We can already prove general properties of refinements, such as:

```
Lemma strengthenPost :
  (∀ s x s', Q1 s (x,s') -> Q2 s (x,s')) ->
  [ P , Q2 ] ⊑ [ P , Q1 ].
```

But we haven't said anything about our *programs* yet.

Syntax

We can describe the syntax of the various effects using a Coq data type.

```
Inductive Term (a : Type) : Type :=
  | New    : v -> (Ptr -> Term a) -> Term a
  | Read   : Ptr -> (v -> Term a) -> Term a
  | Write  : Ptr -> v -> Term a  -> Term a
  | Spec   : PT b -> (b -> Term a) -> Term a
  | Return : a -> Term a.
```

For now, we assume a fixed type for representing addresses (`Ptr`) and values stored on the heap (`v`).

Semantics?

An inductive data type represents the *abstract syntax* of our language, but what about the semantics?

And how can we relate this to the notion of refinement?

Semantics

To define the semantics of terms, we associate a suitable pre- and postcondition with each syntactic construct.

```
Fixpoint semantics (t: Term a) : PT a :=
  match t with
    | Spec s => s
    ...
```

Most constructs follow the familiar rules for the semantics of state, even if they are 'bottom-up'.

$$\frac{}{\{\,\mathsf{True}\,\}\ \mathsf{Return}\ y\ \{\,s = s' \wedge x = y\,\}}\ \textsc{Return}$$

$$\frac{p \overset{s}{\mapsto} v \qquad \{\,P\,\}\ k\ v\ \{\,Q\,\}}{\{\,P\,\}\ \mathsf{Read}\ p\ k\ \{\,Q\,\}}\ \textsc{Read}$$

$$\frac{p \overset{s}{\mapsto} \_ \qquad \{\,P\,\}\ k\ \{\,Q\,\}}{\{\,P\ (s\ [p\ \mapsto\ v])\,\}\ \mathsf{Write}\ p\ v\ k\ \{\,Q\ (s\ [p\ \mapsto\ v])\ x\ s'\,\}}\ \textsc{Write}$$

$$\frac{p \notin dom\ (s) \qquad \{\,P\,\}\ k\ p\ \{\,Q\,\}}{\{\,P\ (s\ [p\ \mapsto\ v])\,\}\ \mathsf{New}\ v\ k\ \{\,Q\ (s\ [p\ \mapsto\ v])\ x\ s'\,\}}\ \textsc{New}$$

$$\frac{\{\,P_1\,\}\ b\ \{\,Q_1\,\} \qquad \textbf{forall}\ s, \neg\, c(s) \wedge I\ s \rightarrow P_2\ s \qquad \{\,P_2\,\}\ k\ \{\,Q_2\,\}}{\left\{\begin{array}{c} I\ s \wedge (\forall\, t, c(t) \wedge I\ t \rightarrow P_1\ t) \wedge \\ \forall\, t\ t', c(t) \wedge I\ t \wedge Q_1\ t\ t' \rightarrow I\ t' \end{array}\right\}\ \mathsf{While}\ c\ \textbf{do}\ b\ \textbf{od}\ k\ \left\{Q_2\right\}}\ \textsc{While}$$

Figure 2: Semantics of WHILE

(Read our paper at your leisure)

Refinement of programs

1. We have defined a refinement relation on pre- and postcondition pairs PT
2. We have defined a semantics for terms, mapping each term to a value of type PT.

These two pieces together give a refinement relation on terms.

Proof engineering

Refinement proofs

- We can prove various properties of our refinement relation (e.g., transitivity)
- We can prove typical refinement calculus laws (e.g., the following assignment rule)
- Using these lemmas, we can transcribe refinement calculations from paper to our theorem prover.

Non-interactive refinement

Example: formalizing the derivation of swap:

```
Definition swap : Term :=
  skip; t := x; x := y; y := t;

Definition swapSpec : PT := ...

Lemma swapDerivation :
  swapSpec ⊑ swap.
  Proof.
    ...
```

Non-interactive refinement

Example: formalizing the derivation of swap:

```
Definition swap : Term :=
  skip; t := x; x := y; y := t;

Definition swapSpec : PT := ...

Lemma swapDerivation :
  swapSpec ⊑ swap.
  Proof.
    ...
```

But this is not yet playing to Coq's strengths as an **interactive** theorem prover...

Interactive refinement

Instead of assuming we know the program we want to end up with *a priori*, we formulate our derivations as follows:

```
Lemma swapDerivation :
  { c : Term | swapSpec ⊑ c
             /\ isExecutable c}.
```

Now we need to rephrase the usual refinement lemmas to work on goals of this form.

For example, the 'following assignment rule' fills in part of the program `c`, but leaves a goal to complete the remainder of the derivation (hopefully with an easier refinement problem left).

Guiding principles

- All laws have the same general form of conclusion:

```
{c : Term | spec ⊑ c /\ isExecutable c}
```

- There is at least one lemma implementing the refinement rule associated with the different language constructs. For compound statements (if, while, sequential composition) there are usual several variants.
- The order of hypotheses is chosen to maximize the chance of early failure.
- Never assume anything about the shape of the pre- or postcondition of the specifications involved.

Example: `writeLemma`

```
Lemma writeLemma
  (ptr : Ptr) (y : v) (spec : PT a) (t : Term a)
  (H : ...)
  (Step :  Spec [ ... , ...] ⊑ t)
  : Spec spec ⊑ Write b ptr y t.
```

- `H` states the requirement that the precondition of `spec` implies that `ptr` is a valid address;
- The `Step` proof is the ‘continuation’ of the refinement development, where the state has been updated accordingly.

Adding automation

We have defined a collection of *tactics* that let you apply such lemmas (and automate some of the associated book keeping);

```
Ltac WRITE ptr v :=
  eapply (writeLemma ptr v );
  simpl_goal.
```

Here `simpl_goal` is a custom tactic that unfolds the definition of refinement, splits any conjunction assumptions, substitutes equalities in our context, triggers beta reduction, etc.

Example: swap

```
Definition swapRefinement (P Q : Ptr) :
  {c : Term unit & SWAP P Q ⊑ c}.
Proof.
  READ Q x.
  NEW x T.
  READ P y.
  WRITE Q y.
  READ T z.
  WRITE P z.
  RETURN tt.
  (* Two simple proofs *)
  * ... (* lookup P s = lookup Q s' *)
  * ... (* lookup Q s = lookup P s' *)
Qed.
```

Proof debugging

There are many more advanced libraries for reasoning about stateful computations in Coq that provide:

- better proof automation;
- richer (separation) logics;
- smarter heap models;
- ...

Proof debugging

There are many more advanced libraries for reasoning about stateful computations in Coq that provide:

- better proof automation;
- richer (separation) logics;
- smarter heap models;
- ...

But if you have written a program, and you get stuck during its verification with incomprehensible open subgoals, there's very little support for debugging the verification effort.

Here we can inspect the remaining specification at any intermediate point, stepping through the commands one by one.

Further support

This encourages a 'forward' development – but we can equally well use the following assignment rule to refine the 'end' of the program.

We can check the remaining specification at any point – and apply weakening/strengthening rules to keep things tidy.

We can split a complex specification into separate subgoals and combine the resulting developments – this is where a proof assistant really helps.

Extraction

Given any refinement development proving

```
{c : Term | spec ⊑ c /\ isExecutable c}
```

we can project out the `Term` and generate OCaml/Haskell code for it.

We can write a small interpreter in OCaml/Haskell that maps `Write` statements to assignments, etc.

Validation

- We have shown that the semantics induced by the refinement relation coincide with their usual axiomatic weakest precondition semantics.

  It works in *theory*.[1]

Validation

- We have shown that the semantics induced by the refinement relation coincide with their usual axiomatic weakest precondition semantics.

  It works in *theory*.[1]

- Several case studies, deriving a program that does a binary search for the integer square root and (the heart of) a union-find data structure.

  It works in *practice*.[2]

Validation

- We have shown that the semantics induced by the refinement relation coincide with their usual axiomatic weakest precondition semantics.

  It works in *theory*.[1]

- Several case studies, deriving a program that does a binary search for the integer square root and (the heart of) a union-find data structure.

  It works in *practice*.[2]

[1] For a suitably definition of theory.

[2] For a suitably definition of practice.

Further work

- Piggyback on existing Coq developments with better heap models, such as Ynot;
- There is development focuses on a fixed collection of effects – but can be adapted easily enough to describe others – exceptions, non-determinism, or general recursion – each yielding their own theory of refinement.

Questions?