From proposition to program

Embedding the refinement calculus in Coq

Wouter Swierstra Joao Alpuim

Utrecht University

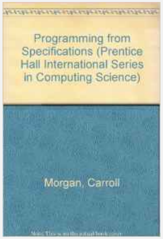
Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

The dream of the 70's

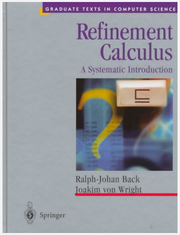
Instead of writing programs, we should derive a executable program from its specification.

The refinement calculus



Programming from Specifications (Prentice Hall International Series in Computing Science)

Morgan, Carroll



GRADUATE TEXTS IN COMPUTER SCIENCE

Refinement Calculus

A Systematic Introduction

Ralph-Johan Back
Joakim von Wright

Springer

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.

Research goals

- The refinement calculus mixes specifications and programs.
- Proof assistants based on type theory provide a single framework for proving and programming.
- Can we use such proof assistants calculate programs from their specification?

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

Given 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*,*

$$[pre, post] \ \sqsubseteq \ [pre, post[w \backslash E]]; w ::= E$$

Note: Deciding how to apply these laws requires creativity!

Refinement calculations: example

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

$\sqsubseteq$ { 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?

The Coq proof assistant

The interactive proof assistant Coq:

- based on a type theory with dependent types;
- a small functional language Gallina;
- many proof tactics that allow the user to construct complex proofs interactively.

This work

Our paper shows how to embed the refinement calculus in the proof assistant 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 : Type :=
  MkPT { pre : Pred S;
         post : forall s : S, pre s -> Pred S}.
```

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

Semantics

For each such a pair of precondition and postcondition, we can define the usual semantics as predicate transformer:

```
Definition semantics (pt : PT) : Pred S -> Pred S
   := fun P s =>
        { p : pre pt s
        & forall s’, post pt s p s’ -> P s’}.
```

Refinement

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

- 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 correct with respect to the weakest precondition semantics.

Derived laws

Even though we have not introduced the WHILE language yet, we can already prove refinement properties, such as:

```
Lemma strengthenPost :
  (forall (s s' : S), Q1 s s' -> Q2 s s') ->
  Refinement [ P , Q2 ] [ P , Q1 ].
```

Using the definitions so far, we can formulate and prove typical refinement rules.

The WHILE language

$$
\begin{aligned}
S ::=\ & \text{skip} \\
|\ & S1; S2 \\
|\ & x ::= a \\
|\ & \text{if } e \text{ then } S1 \text{ else } S2 \\
|\ & \text{while } e \text{ do } S \\
|\ & [p\ ,\ q]
\end{aligned}
$$

Expressions consist of variables, constants, and various numeric or boolean operators.

We can define a suitable abstract syntax tree for expressions and programs as an inductive data type in Coq.

Semantics of WHILE

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?

Overview

So far we have defined:

- Pre- and postconditions `PT`
- A refinement relation on `PT`
- Syntax of WHILE

Still missing:

- A semantics of WHILE, mapping to `PT`
- Extending our refinement relation to work between programs

Semantics of WHILE

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

For example, for the SKIP command we choose:

$$\frac{}{\{\ \text{True}\}\ \text{skip}\ \{\text{s = s'}\}}\ \textsc{Skip}$$

Semantics of While

The rule for sequential composition is slightly more complicated:

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \qquad \{P_2\}\ c_2\ \{Q_2\}}{\{P_1\ s \wedge \forall t, Q_1\ s\ t \rightarrow P_2\ t\}\ c_1; c_2\ \{\exists (t : S), Q_1\ s\ t \wedge Q_2\ t\ s'\}}$$

Do these definitions make sense?

Remember, we are free to choose the pre- and postconditions of every syntactic construct.

How can you be sure that your choices are correct?

We have proven that for every syntactic construct, our choice of `PT` coincides with the usual weakest precondition semantics for that construct.

This provides at least some evidence that our choices for pre- and postcondition are sound with respect to the usual semantics.

Overview

So far we have defined:

- Pre- and postconditions $\mathtt{PT}$
- A refinement relation on $\mathtt{PT}$
- Syntax of WHILE
- A semantics of WHILE, mapping to $\mathtt{PT}$

Still missing:

- Extending our refinement relation to work between programs

Refinement of programs

- We have defined a refinement relation on pre- and postcondition pairs `PT`
- We have defined semantics for the WHILE language as a value of type `PT`
- Together, this gives us a refinement relation on WHILE programs.

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: formalising the derivation of swap:

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

Definition swapSpec : PT := ...

Lemma swapDerivation :
  Refinement swapSpec swap.
  ...
```

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 : While | Refinement 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 : While | Refinement spec c /\ isExecutable c}
```

- There is at least one lemma implementing the refinement rule associated with the different language constructs. For compound statements 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.

Validation

- We have done non-trivial case study, deriving a program that does a binary search for the integer square root – ‘our system works’
- We have shown that the semantics induced by the refinement relation coincide with their usual axiomatic (weakest precondition) semantics – ‘our definitions are correct’

Further work

- Combine with existing work on separation logic in Coq;
- Exhance the (simplistic) model of the heap with richer types, such as arrays and objects;
- Extend the WHILE language with more constructs