Dependent types, predicate transformers and refinement

Wouter Swierstra
with credit to Peter Hancock
Refinement calculus

- A single language for specifications & code;
- A logic describing valid \textit{refinement} steps that can be used to turn a specification into executable code.
Dependently typed languages

- A single language for specifications & code;
- A general purpose higher-order constructive logic...
- ...that is capable of describing other programming logics.
How can we embed a refinement calculus in a proof assistant?
How can we *program* with effects in a *dependently typed language*?
Related work

Ynot

- Axiomatic extension;
- Not executable;
- Rich logic;
- Easy to add new operations.

My thesis

- No axioms;
- Executable;
- More limited logic;
- More operations is more work.
Aims

• Show how existing languages are expressive enough to embed program logics...

• ...and use these logics to reason about effectful programs.
Predicates

• Predicates:

\[ \text{Pred } a = a \rightarrow \text{Set} \]

• We be working (mostly) with predicates on some fixed type of states.

• I’ll use the usual definition of inclusion:

\[ P \subset Q : \text{Pred } s \rightarrow \text{Pred } s \rightarrow \text{Set} \]

\[ P \subset Q = (s : S) \rightarrow P s \rightarrow Q s \]
Representing predicate transformers

record PT : Set
   pre : Pred S
   post : (s : S) -> pre s -> Pred S

• A precondition and postcondition, relating the final state to an input satisfying the precondition.

• I’ll write \([q,p]\) for such a record.
Example: skip

record PT : Set
  pre : Pred S
  post : (s : S) -> pre s -> Pred S

• Skip, the lowest possible hurdle:

  skip : PT
  skip = [pre,post]
  where
  pre = \s -> True
  post = \s pres s' -> s ≡ s'
Semantics

\[ wp : PT \rightarrow \text{Pred } S \rightarrow \text{Pred } S \]

\[ wp \ [\text{pre,post}] \ U \ s = \exists \ p : \text{pre } s, \ \text{post } s \ p \subset U \]
Weakest preconditions

Definition

Given $S$ a statement, the weakest-precondition of $S$ is a function mapping a precondition on the initial state ensuring that execution of $S$ terminates in $Q$.

More formally, let us use variable $x$ to denote abusively the tuple of variables correctness if and only if the first-order predicate below holds:

$$\forall x, P \Rightarrow wp(S, Q)$$

Formally, weakest-preconditions are defined recursively over the abstract state transformers where the predicate in parameter is a continuation.

Skip

$$wp(\texttt{skip}, R) = R$$

Wikipedia
• Remember:

skip : PT
wp : PT -> Pred S -> Pred S

• But now we can prove:

skipLemma :

(p : Pred s) -> (wp skip p ⊆ p)
Refinement

• We need to define a refinement relation between predicate transformers...

• and then use this to prove laws like:

  \text{skipLaw} : ([\text{pre}, \text{post}] : \text{PT}) \rightarrow
  (\text{pre} \subset \text{post}) \rightarrow [\text{pre}, \text{post}] \sqsubseteq \text{skip}
Refinement

record Refines ([pre1,post1]:PT) (pre2,post2]:PT) where
d : pre1 ⊂ pre2
r : (s : S) -> (p : pre1 s) ->
  post2 s (d s p) ⊂ post1 s p
Refinement laws

- The usual list of laws become provable theorems, rather than ‘arbitrary’ axioms

\[
\text{skipLaw} : ([\text{pre,post}] : PT) \rightarrow \\
(\text{pre} \subseteq \text{post}) \rightarrow \text{pt} \subseteq \text{skip}
\]

\[
\text{skipLaw} = \\
\quad \text{let } \text{sd} = \_ \_ \_ \rightarrow \text{true} \text{ in} \\
\quad \text{let } \text{sr} = \_s \text{ pres } s’ \text{ skipPost } \rightarrow \ldots \text{ in} \\
\quad \text{record } \{ \text{d} = \text{sd}; \text{r} = \text{sr} \}
\]
The whole story

- You can play this game for a small WHILE language, defining for every statement:
  - a predicate transformer;
  - a proof that this transformer satisfies the ‘usual’ wp semantics;
  - and a proof that the corresponding refinement law holds.
Assignments

assign : S -> PT

assign s = [pre, post]

where

pre s = True

post _ _ s' = (s' ≡ s)

Note: s replaces the entire state.
While

while : (S -> Bool) -> Pred S
    -> PT -> PT
while cond inv [bPre,bPost]
    = [pre,post]
where
pre = inv
post s pres s' = inv s'
    & not(cond s')
While

\[
while : (S \to \text{Bool}) \to \text{Pred } S
\to PT \to PT
\]

while \text{ cond inv } [bPre,bPost] = [pre,post]

where

pre = inv
post s pres s' = inv s'
& not(\text{cond } s')

Note: this is partial correctness
Sequencing

seq : PT -> PT -> PT

seq [pre1,post1] [pre2,post2] = [pre,post]

where

pre s = \exists (p : pre s), (t : S) -> post1 s p t -> pre2 t

post s pres s' = \exists (t : S),
                \exists (q : post1 s (fst pres) t, post2 t (snd pres t q)) s'
Shallow or deep?

• Now the statements are all identified with their representation as predicate transformers.

• Alternatively define:

```haskell
data Prog : Set where
  Skip : Prog
  Seq : Prog -> Prog -> Prog
  Spec : Pred S -> Prog...
```
Remaining work

• I have ‘prototype’ implementations of various language constructs in Agda and Coq – but it’s still very hard to use.

• I have avoided allocation of fresh variables and reasoning about ‘frame rules’

• Examples!
More related work

• Idea first appeared in Peter Hancock’s thesis;

• Structure closely resembles Altenkirch & Morris’s indexed containers (LICS ’09).