Wouter Swierstra
Nijmegen, 13/09/10
Me and my research

Wouter Swierstra
Nijmegen, 13/09/10
How can we write better software?
How can we write better software?

- Model checking
- Static typing
- Theorem proving
- Automatic testing
- Static analysis
- Best software engineering practices
Type Theory
Per Martin-Löf

• A foundation of constructive mathematics;
• a functional programming language.
Type Theory
Per Martin-Löf

• A foundation of constructive mathematics;
  • a functional programming language.

Really?
What about...

- mutable references?
- arrays?
- exceptions?
- concurrency?
- a GUI?
- a foreign function interface?
- network communication?
- a compiler?
- general recursion?
- file manipulation?
- random numbers?
- ...

...
PhD Thesis

- **Goal:** Reason about effectful programs.
- **Solution:** Implement a pure and total specification of effects in type theory. Replace specs with “real effects” on compilation.
- **Result:** Write and reason about effectful programs.
What’s missing?

• I’ve studied this approach for individual effects – but what is the common theme?

• Proofs in type theory can be hard – what reasoning principles can we use to make them easier.
This year:
common theme
This year:
common theme

• Study how Hancock-Setzer *interaction structures* can be used for effectful, dependently-typed programming.
This year: common theme

• Study how Hancock-Setzer interaction structures can be used for effectful, dependently-typed programming.

• ... and find grant money to continue this line of research.
Today:
reasoning principle
Relabelling a tree

**Inductive** Tree (a : Set) : Set :=

| Leaf : a -> Tree a |
| Node : Tree a -> Tree a -> Tree a |

relabel : forall a, Tree a -> Tree nat
relabel
Relabelling 1.0

Fixpoint relabel (a : Set)
  (t : Tree a) (s : nat) : Tree nat * nat
Relabelling 1.0

Fixpoint relabel (a : Set) (t : Tree a) (s : nat) : Tree nat * nat

:= match t with
    | Leaf _ => (Leaf s, s + 1)
    | Node l r =>
        let (l', s') := relabel l s
        in let (r', s'') := relabel r s'
        in (Node l' r', s'')
end
Recursive step

| Node l r =>
| ---
| let (l', s') := relabel l s
| in let (r', s'') := relabel r s'
| in (Node l' r', s'')
Recursive step

\[
\text{Node } l \ r \Rightarrow \\
\text{let } (l', s') := \text{relabel } l \ s \\
\text{in let } (r', s'') := \text{relabel } r \ s' \\
\text{in } (\text{Node } l' \ r', s'')
\]

Easy to make a mistake!
The state monad

(* For some fixed type s *)

**Definition** State (a : Set) : Type
  := s -> a * s

return : a -> State a

bind : State a
  -> (a -> State b)
  -> State b
Return

Definition State (a : Set) : Type
  := s -> a * s

Definition return (a : Set)
  : a -> State a :=
  fun x => fun s => (x, s)
Bind

**Definition** State (a : Set) : Type

:= s -> a * s

**Definition** bind (a b : Set)

(c1 : State a) (c2 : a -> State b) : State b

:= fun s => let (x, s’) := c1 s

in c2 x s’
Definition State (a : Set) : Type := s -> a * s

Definition bind (a b : Set) (c1 : State a) (c2 : a -> State b) : State b := fun s => let (x, s') := c1 s in c2 x s'

I’ll use an infix operator, >>= , instead of bind
Relabelling 2.0

Node l r =>
  relabel l >>= fun l' =>
  relabel r >>= fun r' =>
  return (Node l' r')
Relabelling 2.0

Node l r =>
  relabel l >>= fun l' =>
  relabel r >>= fun r' =>
  return (Node l' r')

No more passing around the state explicitly!
Challenge:
verify the relabelling function,
without expanding the
definitions of return and bind.
Strong specifications

• Strong specifications:
  • define a value;
  • together with a proof that that value satisfies the spec.

• Notation in Coq:

\{n : \text{nat} \mid n > 7\}
Program

• Coq’s Program framework for working with strong specifications
• let’s you define functions manipulating strongly specified values,
• and collects assumptions and obligations.
• You need to prove any proof obligations (using tactics) before Program generates a complete Coq term.
Idea:
Decorate the state monad with pre- and postconditions.
Pre- and postconditions

Define the following types:

\[ \text{Pre} := s \to \text{Prop} \]
\[ \text{Post} (a : \text{Set}) := s \to a \to s \to \text{Prop} \]
The Hoare State Type

Define the following type:

\[
\text{HoareState } P \ a \ Q \ := \\
\{ i : s \ | \ P i \} \rightarrow \\
\{ (x,f) : a \times s \ | \ Q i x f \}
\]
Remaining questions

• How can we define return?
• How can we define bind?
• How can we use these functions to verify our relabelling function?
Definition return (x : a) : HoareState

   (fun i => True)
   a
   (fun i y f => i = f /\ x = y)

:= fun i => (x,i)
**Definition** return \((x : a) : \)

\[ \text{HoareState} \]

\[ \text{(fun } i \rightarrow \text{True)} \]

\[ a \]

\[ \text{(fun } i \ y \ f \rightarrow i = f \ \wedge \ x = y) \]

:= \text{fun } i \rightarrow (x,i)

Need to complete one trivial proof.
What should the pre- and postconditions be?

HoareState P1 A Q1 ->
(A -> HoareState P2 B Q2) ->
HoareState ... B ...

Bind - l
Bind - II

\[\text{HoareState } P_1 \ A \ Q_1 \rightarrow\]
\[((x:A) \rightarrow \text{HoareState } (P_2 \ x) \ B \ (Q_2 \ x)) \rightarrow\]
\[\text{HoareState} \ ... \ B \ ...\]

What should the pre- and postconditions be?
Bind’s precondition

\( s_1 \rightarrow P_1 s_1 \)

\( \forall x s_2, Q_1 s_1 x s_2 \rightarrow P_2 x s_2 \)

The initial state must satisfy the first computation’s precondition
Bind’s precondition

\( \forall s_1 \rightarrow P_1 s_1 \)

\( \forall \forall x s_2, Q_1 s_1 x s_2 \rightarrow P_2 x s_2 \)

The initial state must satisfy the first computation’s precondition
Bind’s precondition

\[ s_1 \rightarrow P_1 \ s_1 \]

\[ \forall x \ s_2, \ Q_1 \ s_1 \ x \ s_2 \rightarrow P_2 \ x \ s_2 \]

The intermediate state satisfies the second computation’s precondition.
Bind’s precondition

\( \forall s_1 \rightarrow P_1 s_1 \)

// \( \forall x s_2, Q_1 s_1 x s_2 \rightarrow P_2 x s_2 \)

The intermediate state satisfies the second computation’s precondition.
Bind’s postcondition

\( s_1 \ y \ s_3 \rightarrow \text{exists } x, \text{exists } s_2, \)
\[ Q_1 \ s_1 \ x \ s_2 \ \land \ Q_2 \ x \ s_2 \ y \ s_3 \]

There is an intermediate result and an intermediate state relating the two computations.
Implementing bind

• The definition of bind is **exactly the same** as for the state monad...

• ...but we need to fulfill one or two proof obligations.
Using the Hoare State Monad

To verify programs in the state monad, all we need to do is change the type signature, that is, choose the pre- and postconditions.

The program remains unchanged.
Relabelling revisited

HoareState

(fun i => True)

(Tree nat)

(fun i t f =>

  flatten t = [i .. i + size t])
Relabelling revisited

HoareState

(fun i => True)

(Tree nat)

(fun i t f =>

  flatten t = [i .. i + size t]

  \ f = i + size t)
The proof

• The definition gives rise to two proof obligations, one for every case branch.
• We’ve automated away all work involved in keeping track of the state;
• The proof for the recursive case is only about 5 lines long (but uses some fancy Program tactics).
Discussion

- Other choices for pre- and postconditions?
- Is the HoareState type a monad?
- Further automation using Ltac?