The Hoare State Monad

Wouter Swierstra

The State Monad

Wouter Swierstra

Relabelling a tree

```
data Tree a = Leaf a
    | Node (Tree a) (Tree a)

relabel :: Tree a -> Tree Int
```

Relabelling by hand

```
relabel :: Tree a -> Tree Int
relabel t = fst (worker 0 t)
  where
  worker :: Int -> (Tree Int, Int)
  worker i (Leaf _)   = (Leaf i, i + 1)
  worker i (Node l r) = ...
```

Recursive step

```
worker i (Node l r) =
  let (l’, i’)  = relabel l i
      (r’, i’’) = relabel r i’
  in (Node l’ r’, i’’)
```

Recursive step

```
worker i (Node l r) =
  let (l’, i’)  = relabel l i
      (r’, i’’) = relabel r i’
  in (Node l’ r’, i’’)
```

Easy to make a mistake!

The State Monad

```
type State a = Int -> (a , Int)


return :: a -> State a

(>>=) :: State a

  -> (a -> State b)

  -> State b
```

Return

```
type State a = Int -> (a , Int)


return :: a -> State a

return x = \i -> (x, i)
```

Bind

```
type State a = Int -> (a , Int)


(>>=) :: State a -> (a -> State b)-> State b
c >>= f = \i -> let (x, i’) = c i
                in f x i’
```

Relabelling, mark II

```
relabel :: Tree a -> State (Tree Int)
relabel (Leaf _) = \i -> (Leaf i, i+1)
relabel (Node l r) =
  relabel l >>= \l' ->
  relabel r >>= \r' ->
  return (Node l' r')
```

Relabelling with **do**

```
relabel :: Tree a -> State (Tree Int)
relabel (Node l r) =
  do l’ <- relabel l
     r’ <- relabel r
     return (Node l’ r’)
```

Reasoning about monads

- How can we prove that the relabelling function is correct?
- Usual approach: expand definitions of return and bind, perform equational reasoning.
- Why not exploit monadic structure during the proof?

Challenge:
verify the relabelling function, without expanding the definitions of return and bind.

Coq

- An interactive proof assistant based on type theory.
- Consists of two distinct parts:
  - a total functional language;
  - a tactic language
- I'm assuming some knowledge of dependent types...

Strong specifications

- Consider the following type for division:

```
(n : nat) ->
{d : nat | d > 0} ->
{(q,r) : nat × nat | d * q + r = n}
```

- The type explains how the function behaves.
- The Program tactic enables the separation of concerns.

Idea:
Decorate the state monad with pre- and postconditions.

Pre- and postconditions

- Define the following types:

```
Pre = Nat -> Prop

Post (a : Set) = Nat -> a -> Nat -> Prop
```

The Hoare State Monad

Define the Hoare type:

```
HoareState P A Q =
  {i : Nat | P i} ->
    {(x,f) : A × Nat | 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?

Return

```
return : (x : A) ->
  HoareState
    (\i -> True)
    A
    (\i y f -> i = f /\ x = y)
return x = \i -> (x,i)
```

Return

```
return : (x : A) ->
  HoareState
    (\i -> True)
    A
    (\i y f -> i = f /\ x = y)
return x = \i -> (x,i)
```

Need to complete one trivial proof.

Bind - I

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

Bind - II

```
bind : HoareState P1 A Q1 ->
  ((x:A) -> HoareState (P2 x) B (Q2 x)) ->
  HoareState ... B ...
```

What should the pre- and postconditions be?

Bind's precondition

```
\s1 -> P1 s1
    /\ forall x s2, Q1 s1 x s2 -> P2 x s2
```

The initial state must satisfy the first computations precondition

Bind’s precondition

```
\s1 -> P1 s1
   /\ forall x s2, Q1 s1 x s2 -> P2 x s2
```

The initial state must satisfy the first computations precondition

Bind’s precondition

```
\s1 -> P1 s1
    /\ forall x s2, Q1 s1 x s2 -> P2 x s2
```

The intermediate state satisfies the second computation’s precondition.

Bind's precondition

```
\s1 -> P1 s1
    /\ forall x s2, Q1 s1 x s2 -> P2 x s2
```

The intermediate state satisfies the second computation's precondition.

Bind’s postcondition

```
\s1 y s3 -> exists x, exists s2,
    Q1 s1 x s2 /\ Q2 x s2 y s3
```

There is an intermediate results 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.

```
c >>= f = \i -> let (x, i’) = c i
                in f x i’
```

Using the Hoare State Monad

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

The program remains unchanged.

Relabelling, revisited

- For our relabelling function:
  - the precondition is trivial;
  - for the postcondition we choose:

```
\i t f -> flatten t = [i .. i + size t]
```

Relabelling, revisited

- For our relabelling function:
  - the precondition is trivial;
  - for the postcondition we choose:

```
\i t f -> flatten t = [i .. i + size t]
```

Postcondition not strong enough!

Relabelling, revisited

- For our relabelling function:
  - the precondition is trivial;
  - for the postcondition we choose:

```
\i t f -> flatten t = [i .. i + size t]
  /\ f = i + size t
```

Demo

Acknowledgements

- Graham Hutton and Diana Fulger, for suggesting the problem.
- Greg Morrisett, Aleks Nanevski, Thorsten Altenkirch, and many others for suggesting the solution.
- Matthieu Sozeau for Program support.

- Draft paper and working code now ready if you're interested.