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 $i \ (\text{Node } l \ r) =$

let $(l', \ i') = \text{relabel } l \ i$
$(r', \ i'') = \text{relabel } r \ i'$

in $(\text{Node } l' \ r', \ i'')$

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
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

\[ c >>= f = \lambda i \rightarrow \text{let } (x, i') = c i \]
\[ \text{in } f x i' \]
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**

```haskell
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:

\[(n : \text{nat}) \rightarrow \{d : \text{nat} \mid d > 0\} \rightarrow \{(q,r) : \text{nat} \times \text{nat} \mid d \times q + r = n\}\]

• 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:

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

Define the Hoare type:

\[
\text{HoareState } P A Q = \{ i : \text{Nat} \mid P i \} \rightarrow \{(x,f) : A \times \text{Nat} \mid Q i x f\}
\]
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 - 1

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 \rightarrow P1 \ s1

/\ \forall \ x \ s2, \ Q1 \ s1 \ x \ s2 \rightarrow \ P2 \ x \ s2

The initial state must satisfy the first computations 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 initial state must satisfy the first computations 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

\( s_1 \rightarrow P_1 s_1 \)

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

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

\( s_1 y s_3 \rightarrow \exists x, \exists s_2,\)

\( Q_1 s_1 x s_2 \land Q_2 x s_2 y s_3 \)

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.

\[ c \ggg f = \lambda i \to \text{let } (x, i') = c i \text{ in } f x i' \]
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:

\[
i \ t \ f \to \text{flatten } t = [i .. i + \text{size } t]
\]
Relabelling, revisited

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

\[
i \, t \, f \rightarrow \text{flatten } t = [i \ldots i + \text{size } t]
\]

Postcondition not strong enough!
Relabelling, revisited

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

\[
\begin{align*}
\forall i \ t \ f & \rightarrow \text{flatten } t = [i .. i + \text{size } t] \\
\forall f & = i + \text{size } t
\end{align*}
\]
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.