The Hoare State Monad
Proof Pearl

Wouter Swierstra

Chalmers University of Technology
wouter@chalmers.se

Abstract. This pearl introduces the Hoare state monad, a variant of the state monad that facilitates the verification of stateful programs.

1 Introduction

Monads help structure functional programs. Yet proofs about monadic programs often start by expanding the definition of return and bind. This seems rather wasteful. If we exploit this structure when writing *programs*, why should we discard it when we writing *proofs*? This pearl examines how to verify functional programs inhabiting the state monad. It is our express aim to take advantage of the monadic structure of our programs to guide the verification process.

This pearl is a literate Coq script [13]. Although most proofs have been elided from the typeset version, the complete development is available from my homepage.

2 The state monad

Let me begin by motivating the state monad.

Consider the following inductive data type for binary trees:

```
Inductive Tree (a : Set) : Set :=
  | Leaf : a → Tree a
  | Node : Tree a → Tree a → Tree a.
```

Now suppose we want to define a function that replaces every value stored in a leaf of such a tree with a unique integer, i.e., no two leaves in the resulting tree should share the same label. The obvious solution is to define the following function:

```
Fixpoint relabel (a : Set) (t : Tree a) (s : nat) : Tree nat * nat
  := match t with
     | Leaf _ ⇒ (Leaf s, 1 + s)
     | Node l r ⇒ let (l', s') := relabel a l s
                  in let (r', s'') := relabel a r s'
                  in (Node l' r', s'') end.
```

This relabel function carries a natural number as it traverses the tree. It uses the argument number as the new label for the leaves. To make sure that no two leaves get assigned the same number, the number returned at a leaf is incremented. In the Node case, the number is threaded through the recursive calls appropriately.

While this solution is correct, it can be improved. It is all too easy to pass the wrong number to a recursive call, thereby forgetting to update the state. To preclude such errors, we show how the *state monad* may be used to carry the number implicitly as the tree is traversed.

For some fixed type of state $s : \mathsf{Set}$, the state monad is:

```
Definition State (a : Set) : Type := s → a * s.
```

A computation in the state monad State s a takes an initial state as its argument. Using this initial state, it performs some computation yielding a pair consisting of a value of type a and a final state.

The two monadic operations, return and bind, are defined as follows:

```
Definition return (a : Set) : a → State a := fun x s ⇒ (x, s).
Definition bind (a b : Set) : State a → (a → State b) → State b
  := fun c1 c2 s1 ⇒ let (x, s2) := c1 s1 in c2 x s2.
```

The return function lifts any pure value into the state monad, leaving the state untouched. The bind composes two computations. It passes both the state and the result arising from the first computation to the second computation.

In line with the notation used in Haskell [10], we introduce a pair of infix operators to write monadic computations. Firstly, we will write the bind as $\gg\!=$, a right-associative infix operator. Secondly, we will write $c_1 \gg c_2$ for bind c_1 (fun _ ⇒ c_2). This operator binds two computations, discarding the intermediate result.

Besides return and bind, there are two other operations in the state monad that manipulate the state:

```
Definition get : State s := fun s ⇒ (s, s).
Definition put : s → State unit := fun s _ ⇒ (tt, s).
```

The get function returns the current state, whereas put overwrites the current state with its argument.

We can now redefine our relabelling function to use the state monad:

```
Fixpoint relabel (a : Set) (t : Tree a) : State nat (Tree nat)
  := match t with
    | Leaf _ ⇒ get  >>= fun n ⇒
                put (S n) >>
                return (Leaf n)
    | Node l r ⇒ relabel  l >>= fun l' ⇒
                 relabel  r >>= fun r' ⇒
                 return (Node l' r') end.
```

Note that we have chosen to instantiate the type variable s to nat – the state carried around by the relabelling function is a natural number. By using the state monad, we no longer need to pass around this number by hand. This definition is much less error prone: all the 'plumbing' is handled by the monadic combinators.

3 The challenge

How can we prove that this relabelling function is correct?

Before we can talk about correctness, we need to establish the specification that we expect the relabel function to satisfy. One way of formulating the desired specification is by defining the following auxiliary function that flattens a tree to a list of labels:

```
Fixpoint flatten (a : Set) (t : Tree a) : list a
  := match t with
       | Leaf x ⇒ x :: nil
       | Node l r ⇒ flatten  l ++ flatten  r end.
```

We will prove that for any tree t and number x, the list flatten (fst (relabel t x) does not have any duplicates. This property does not completely characterise relabelling – we should also check that the argument tree has the same shape as the resulting tree. This is relatively easy to verify as the relabelling function clearly maps leaves to leaves and nodes to nodes. Proving that the resulting tree satisfies the proposed proposition, however, is not so easy.

4 Decorating the state monad

The relabel function in the previous section is simply typed. We can certainly use proof assistants such as Coq to formalise equational proofs about such functions. In this paper, however, we will take a slightly different approach.

We propose to use *strong specifications*, i.e., the type of our relabel function should capture information about its behaviour. We simultaneously complete the function definition and the proof that this definition satisfies its specification. This *correct-by-construction* approach to verification can be traced back to Martin-Löf [6].

To give a strong specification of our relabelling function, we will decorate computations in the state monad with additional propositional information. Recall that we defined the state monad as follows:

```
Definition State (a : Set) : Type := s → a * s.
```

We can refine this definition slightly: instead of accepting *any* initial state of type s, it requires an initial state that satisfies a given precondition. Furthermore, instead of returning *any* pair, it guarantees that the resulting pair satisfies a postcondition relating the initial state, resulting value, and final state. Bearing

these two points in mind, we arrive at the following definition of the *Hoare state monad*:

```
Definition Pre : Type := s → Prop.
Definition Post (a : Set) : Type := s → a → s → Prop.
Program Definition HoareState (pre : Pre) (a : Set) (post : Post a) : Set
  := forall i : { t : s | pre t }, { (x, f) : a * s | post i x f }.
```

We refer to this as the Hoare state monad as it enables Floyd-Hoare style pre- and postcondition reasoning about computations in the state monad [2, 3]. Throughout this paper, we will use Coq's Program tactic to program with strong specifications [12, 11].

We still need to define the return and bind functions for the Hoare state monad. The return function does not place any restriction on the input state; it simply returns its second argument, leaving the state intact:

```
Definition top : Pre := fun s ⇒ True.
Program Definition return (a : Set)
  : forall x, HoareState top a (fun i y f ⇒ i = f ∧ y = x)
  := fun x s ⇒ (x, s).
```

The definition of the return of the Hoare state monad is identical to the original definition of the state monad: we have only made its behaviour evident from its type. The Program tactic automatically discharges the trivial proofs necessary to complete the definition.

The bind of the Hoare state monad is a bit more subtle. Recall that the bind of the state monad has the following type:

```
State a → (a → State b) → State b
```

You might expect the definition of the bind of the Hoare state monad to have a type of the form:

```
HoareState P1 a Q1 → (a → HoareState P2 b Q2) → HoareState ... b ...
```

Before we consider the precondition and postcondition of the resulting computation, note that we can generalise this slightly. In the above type signature, the second argument of bind is not dependent. We can parametrise P_2 and Q_2 by the result of the first computation:

```
HoareState P1 a Q1
  → (forall (x : a), HoareState (P2 x) b (Q2 x))
  → HoareState ... b ...
```

This allows the pre- and postconditions of the second computation to refer to the results of the first computation.

Now we need to choose a suitable precondition and postcondition for the composite computation returned by the bind function. To motivate our choice

of pre- and postcondition, recall that the bind of the state monad is defined as follows:

```
Definition bind (a b : Set) : State a → (a → State b) → State b
  := fun c1 c2 s1 ⇒ let (x, s2) := c s1 in f x s2.
```

The bind function starts by running the first computation, and subsequently feeds its result to the second computation. So clearly the precondition of the composite computation should be imply the precondition of the first computation c_1—otherwise we could not justify running c_1 with the initial state s_1. Furthermore the postcondition of the first computation should imply the precondition of the second computation—if this wasn't the case, we could not give grounds for the call to c_2. These considerations lead to the following choice of precondition for the composite computation:

```
fun s1 ⇒ P1 s1 ∧ forall x s2, Q1 s1 x s2 → P2 x s2
```

What about the postcondition? Recall that a postcondition is a relation between the initial state, resulting value, and the final state. We would expect the postcondition of both argument computations to hold after executing the composite computation resulting from a call to bind. This composite computation, however, cannot refer to the initial state passed to the second computation or the results of the first computation: it can only refer to its own initial state and results. To solve this we existentially quantify over the results of the first computation, yielding the following postcondition for the bind operation:

```
fun s1 y s3 ⇒ exists x, exists s2, Q1 s1 x s2 ∧ Q2 x s2 y s3
```

In words, the postcondition of the composite computation states that there is an intermediate state s_2 and a value x resulting from the first computation, such that these satisfy the postcondition of the first computation Q_1. Furthermore, the postcondition of the second computation Q_2 relates these intermediate results to the final state s_3 and the final value y.

Once we have chosen the desired precondition and postcondition of bind, its definition is straightforward:

```
Program Definition bind : forall a b P1 P2 Q1 Q2,
  (HoareState P1 a Q1) →
  (forall (x : a), HoareState (P2 x) b (Q2 x)) →
  HoareState (fun s1 ⇒ P1 s1 ∧ forall x s2, Q1 s1 x s2 → P2 x s2)
             b
             (fun s1 y s3 ⇒ exists x, exists s2, Q1 s1 x s2 ∧ Q2 x s2 y s3)
  := fun a b P1 P2 Q1 Q2 c1 c2 s1 ⇒
       match c1 s1 with (x, s2) ⇒ c2 x s2 end.
```

This definition does give rise to two proof obligations: the intermediate state s_2 must satisfy the precondition of the second computation c_2; the application

$c_2\ x\ s_2$ must satisfy the postcondition of bind. Both these obligations are fairly straightforward to prove.

Before we have another look at the relabel function, we define the two auxiliary functions get and put.

```
Program Definition get : HoareState top s (fun i x f ⇒ i = f ∧ x = i)
  := fun s ⇒ (s, s).
Program Definition put (x : s) : HoareState top unit (fun _ _ f ⇒ f = x)
  := fun _ ⇒ (tt, x).
```

Both functions have the trivial precondition top. While the postcondition of the get function guarantees that it will return the current state without modifying it, the postcondition of the put function declares that the final state is equal to put's argument.

5 Relabelling revisited

Finally, we return to the original question: how can we prove that our relabel function satisfies its specification?

Using the Hoare State monad, we now arrive at the following definition of our relabelling function:

```
Fixpoint size (a : Set) (t : Tree a) : nat :=
  match t with
    | Leaf x ⇒ 1
    | Node l r ⇒ size  l + size  r end.
Fixpoint seq (x n : nat) : list nat :=
  match n with
    | 0 ⇒ nil
    | S k ⇒ x :: seq (S x) k end.
Program Fixpoint relabel (a : Set) (t : Tree a) :
  HoareState nat top
                (Tree nat)
                (fun i t f ⇒ f = i + size t ∧ flatten  t = seq i (size t))
 := match t with
    | Leaf x ⇒ get  >>= fun n ⇒
               put (n + 1) >>
               return (Leaf n)
    | Node l r ⇒ relabel  l >>= fun l' ⇒
                 relabel  r >>= fun r' ⇒
                 return (Node l' r') end.
```

The function definition of relabel is identical to the version using the state monad in Section 3. The only novel aspect is our choice of pre- and postcondition.

As we do not need any assumptions about the initial state, we choose the trivial precondition top. The postcondition uses two auxiliary functions, size and

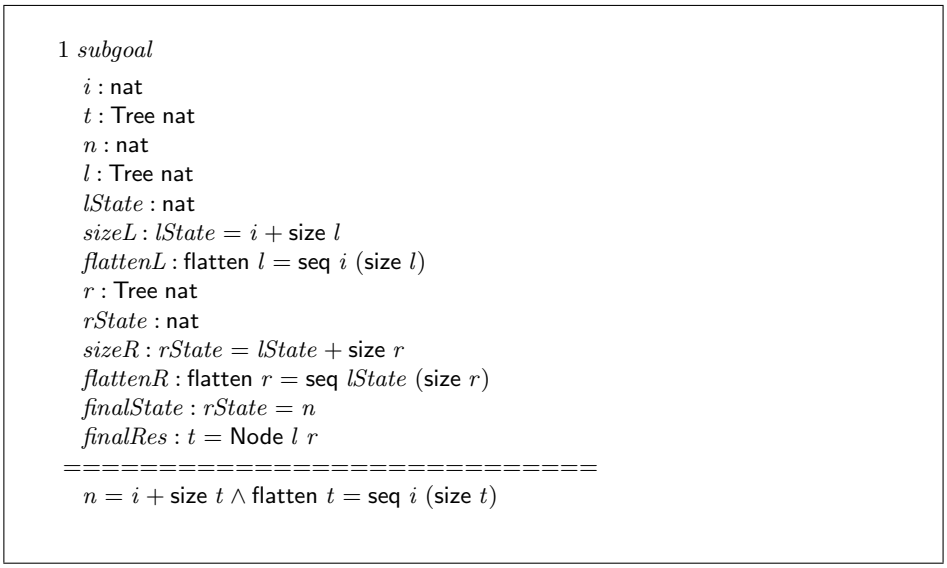

Fig. 1: Proving the obligation of the relabelling function.

seq, and consists of two parts. First of all, the final state should be exactly size t larger than the initial state, where t refers to the resulting tree. Furthermore, when the relabelling function is given an initial state i, flattening t should yield the sequence $i, i+1, ...i+$ size t.

This gives us one obligation that cannot be solved automatically. We need to apply several tactics to trigger β-reduction and introduce our assumptions. After giving the variables in the context more meaningful names, we arrive at the proof state in Figure 1.

To complete the proof, we must prove that the postcondition holds for the tree Node l r under the assumption that it holds for recursive calls to l and r. The first part of the conjunction follows immediately from the assumptions *finalRes*, *sizeR*, and *sizeL* and the associativity of addition. The second part of the conjunction is a bit more interesting. After applying our induction hypotheses, *flattenL* and *flattenR*, the remaining goal becomes:

$$\text{seq } i \text{ (size } l) +\!\!+ \text{ seq } \mathit{lState} \text{ (size } r) = \text{seq } i \text{ (size } l + \text{size } r)$$

To complete the proof we need to use the assumption *sizeL*. If we had chosen the obvious postcondition flatten t = seq i (size t) we would not have been able to complete this proof. Once we apply *sizeL* we can use one last lemma to complete the proof:

Lemma SeqSplit : **forall** y x z, seq x $(y+z)$ = seq x y ++ seq $(x+y)$ z.

This lemma is easy to prove by induction on y.

6 Wrapping it up

Now suppose we need to show that relabel satisfies a weaker postcondition. For example, the NoDup predicate is defined in the Coq libraries as follows:

```
Inductive NoDup : list a → Prop :=
  | NoDup_nil : NoDup nil
  | NoDup_cons : forall x xs, x ∉ xs → NoDup xs → NoDup (x :: xs).
```

How can we prove that the tree resulting from a call to our relabelling function satisfies NoDup (flatten t)?

We cannot define a relabelling function that has this postcondition—the induction hypotheses are insufficient to complete the required proofs in the Node case. We can, however, weaken our postcondition and strengthen our precondition explicitly. In line with Hoare Type Theory [9, 8, 7], we call this operation do:

```
Program Definition do (s a : Set) (P1 P2 : Pre s) (Q1 Q2 : Post s a) :
  (forall i, P2 i → P1 i) → (forall i x f, Q1 i x f → Q2 i x f) →
  HoareState s P1 a Q1 → HoareState s P2 a Q2
    := fun str wkn c ⇒ c.
```

This function has no computational content. It merely changes the precondition and postcondition associated with a computation in the Hoare state monad. We can now define the final version of our relabelling function as follows:

```
Program Fixpoint final (a : Set) (t : Tree a) :
  HoareState  (top nat) (Tree nat) (fun i t f ⇒ NoDup (flatten t))
    := do _ _ (relabel a t).
```

The precondition is unchanged. As a result, the *str* argument is filled in automatically. To complete this definition, however, we need to prove that the postcondition can be weakened appropriately. This boils down to showing that the list seq i (size t) does not have any duplicates. Using one last lemma, forall $n\ x\ y, x < y \rightarrow \neg$In x (seq $y\ n$), we complete the proof.

7 Discussion

Related work

This pearl draws inspiration from many different sources. Most notably, it is inspired by recent work on Hoare Type Theory [9, 8, 7]. Ynot, the implementation of Hoare Type Theory in Coq, postulates the existence of return, bind, and do to use Hoare logic to reason about functions that use mutable references. This paper shows how these functions may be *defined* in Coq, rather than postulated. Furthermore, we have generalised their presentation somewhat: where

Hoare Type Theory has specifically been designed to reason about mutable references, this pearl shows that the HoareState type can be used to reason about *any* computation in the state monad.

The relabelling problem is taken from Hutton and Fulger [4], who give an equational proof. Their proof, however, revolves around defining an intermediate function:

$$label' : \textbf{forall}\ a\ b, \mathsf{Tree}\ a \to \mathsf{State}\ (\mathsf{list}\ b)\ (\mathsf{Tree}\ b)$$

The $label'$ function carries around an (infinite) list of fresh labels that are used to relabel the leaves of the argument tree. To prove that *label* meets the required specification, Hutton and Fulger prove various lemmas relating *label* and $label'$. It is not clear how their proof techniques can be adapted to other functions in the state monad.

Similar techniques have been used by Leroy [5] in the Compcert project. His solution, however, revolves around defining an auxiliary data type:

$$\begin{array}{l}\textbf{Inductive}\ \mathsf{Res}\ (a : \mathsf{Set})\ (t : s) : \mathsf{Set} := \\ \quad \mid \mathsf{Error} : \mathsf{Res}\ a\ t \\ \quad \mid \mathsf{OK} : a \to \textbf{forall}\ (t' : s), \mathsf{R}\ t\ t' \to \mathsf{Res}\ a\ t.\end{array}$$

Where R is some relation between states. Unfortunately, the bind of this monad yields less efficient extracted code, as it requires an additional pattern match on the Res resulting from the first computation. Furthermore, the Hoare state monad presented here is slightly more general as its postcondition may also refer to the result of the computation.

Cock et al. have used a similar monad in the verification of the seL4 microkernel [1]. There are a few differences between their monad and the one presented here. Firstly, we have chosen the postconditions to be ternary relations between the initial state, result, and final state. As a result, we do not need to introduce 'ghost variables' to relate intermediate results. Secondly, their rules are presented as predicate transformers, using Isabelle/HOL's verification condition generator to infer the weakest precondition of a computation. This paper, on the other hand, focuses on programming with strong specifications in type theory, where the type of a computation fixes the desired pre- and postcondition. Finally, their monad also handles non-determinism, a topic I have steered clear of in this paper.

Further work

The Hoare state monad as presented here seems to work quite well in practice. I have not, however, provided justification for the choice of pre- and postcondition of bind and return. Other choices are certainly possible. For instance, we could choose the following type for return:

$$\textbf{forall}\ x, \mathsf{HoareState}\ \mathsf{top}\ a\ (\textbf{fun}\ i\ y\ f \Rightarrow \mathsf{True})$$

Clearly this is a bad choice—applying the return function will no longer yield any information about the computation. It would be interesting to investigate if the choices presented here are indeed the weakest precondition and the strongest postcondition.

Acknowledgements I would like to thank Matthieu Sozeau for developing Coq's Program support and helping me to use it. Both Matthieu and Jean-Philippe Bernardy provided invaluable feedback on a draft version of this paper. Finally, I would like to thank Thorsten Altenkirch, Peter Hancock, Graham Hutton, and James McKinna for their useful suggestions.

References

[1] David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Cesar Munoz and Otmane Ait, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

[2] Robert W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19, 1967.

[3] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[4] Graham Hutton and Diana Fulger. Reasoning about effects: seeing the wood through the trees. In *Proceedings of the Ninth Symposium on Trends in Functional Programming*, 2008.

[5] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL '06: 33rd Symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[6] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 167–184. Prentice-Hall, Inc., 1985.

[7] Aleksandar Nanevski and Greg Morrisett. Dependent type theory of stateful higher-order functions. Technical Report TR-24-05, Harvard University, 2005.

[8] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare Type Theory. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN Internation Conference on Functional Programming*, 2006.

[9] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP '08: Proceedings of the Twelfth ACM SIGPLAN Internation Conference on Functional Programming*, 2008.

[10] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[11] Matthieu Sozeau. Subset coercions in Coq. In *Types for Proofs and Programs*, volume 4502 of *LNCS*. Springer-Verlag, 2007.

[12] Matthieu Sozeau. *Un environnement pour la programmation avec types dépendants*. PhD thesis, Université de Paris XI, 2008.

[13] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2008. Version 8.2.