A Hoare Logic for the State Monad
Proof Pearl

Wouter Swierstra

Chalmers University of Technology
wouter@chalmers.se

Abstract. This pearl examines how to verify functional programs written using the state monad. It uses Coq's Program framework to provide strong specifications for the standard operations that the state monad supports, such as return and bind. By exploiting the monadic structure of such programs during the verification process, it becomes easier to prove that they satisfy their specification.

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 writing *proofs*? This pearl examines how to verify functional programs written using the state monad. It is my express aim to take advantage of the monadic structure of these programs to guide the verification process.

This pearl is a literate Coq script [15]. Most proofs have been elided from the typeset version, but a complete development is available from my homepage. Throughout this paper, I will assume that you are familiar with Coq's syntax and have some previous exposure to functional programming using monads [16].

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, given by the relabel function below, keeps track of a natural number as it traverses the tree.

```
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.
```

The relabel function uses its 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, there is some room for improvement. It is all too easy to pass the wrong number to a recursive call, thereby forgetting to update the state. To preclude such errors, the *state monad* may be used to carry the number implicitly as the tree is traversed.

For some fixed type of state s : Set, the state monad is:

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

A computation in the state monad State 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. Two computations may be composed using the bind function. It passes both the state and the result arising from the first computation as arguments to the second computation.

In line with the notation used in Haskell [11], I will use a pair of infix operators to write monadic computations. Instead of bind, I will sometimes write $\gg=$, a right-associative infix operator. Secondly, I will write $c_1 \gg c_2$ instead of bind c_1 (**fun** _ ⇒ c_2). This operator binds two computations, discarding the intermediate result.

Besides return and bind, there are two other operations that may be used to construct computations in the state monad:

```
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 the relabelling function to use the state monad as follows:

```
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 the type variable s has been instantiated 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 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 straightforward to verify as the relabelling function clearly maps leaves to leaves and nodes to nodes. Proving that the resulting tree satisfies the proposed specification, 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, I will take a slightly different approach.

In this paper I will use *strong specifications*, i.e., the type of the relabel function will capture information about its behaviour. Simultaneously completing the function definition and the proof that this definition satisfies its specification yields programs that are *correct by construction*. This approach to verification can be traced back to Martin-Löf [6].

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

Definition State $(a : \mathsf{Set}) : \mathsf{Type} := s \to a * s$.

We can refine this definition slightly: instead of accepting *any* initial state of type s, the initial state should satisfy a given precondition. Furthermore, instead of returning *any* pair, the resulting pair should 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 a state monad enriched with Hoare logic [2, 3].

Definition Pre : Type := $s \to$ Prop.

Definition Post $(a : \mathsf{Set}) : \mathsf{Type} := s \to a \to s \to$ Prop.

Program Definition HoareState $(pre : \mathsf{Pre})$ $(a : \mathsf{Set})$ $(post : \mathsf{Post}\ a) : \mathsf{Set}$
  := **forall** $i : \{t : s \mid pre\ t\}, \{(x,f) : a * s \mid post\ i\ x\ f\}$.

Coq uses the notation $\{x : a \mid p\ x\}$ for strong specifications. Such a specification is inhabited by a pair consisting of a value x of type a, together with a proof that x satisfies the property p.

The code presented here uses Coq's Program framework [12, 13]. Defining a function that manipulates strong specifications using the Program framework is a two stage process: once we have defined the computational fragment, we are presented with a series of proof obligations that must be fulfilled before the Program framework can generate a complete Coq term. When defining the computational fragment, we do not have to manipulate proofs, but rather focus on programming with the first components of the strong specifications. In fact, the definition of the HoareState type above already uses one aspect of the Program framework: a projection is silently inserted to extract a value of type s from the variable i in the postcondition.

Although we have defined the HoareState type, we still need to define the return and bind functions. 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** $s \Rightarrow$ True.

Program Definition return $(a : \mathsf{Set})$
  : **forall** x, HoareState top a (**fun** $i\ y\ f \Rightarrow i = f \wedge y = x$)
  := **fun** $x\ s \Rightarrow (x, s)$.

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

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

$$\mathsf{State}\ a \rightarrow (a \rightarrow \mathsf{State}\ b) \rightarrow \mathsf{State}\ b$$

You might expect the definition of the revised bind function to have a type of the form:

$$\mathsf{HoareState}\ P_1\ a\ Q_1 \rightarrow (a \rightarrow \mathsf{HoareState}\ P_2\ b\ Q_2) \rightarrow \mathsf{HoareState} \ldots\ b \ldots$$

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 the bind function is not dependent. We can parametrise P_2 and Q_2 by the result of the first computation:

```
HoareState $P_1\ a\ Q_1$
  → (forall $(x : a)$, HoareState $(P_2\ x)\ b\ (Q_2\ x)$)
  → HoareState ... $b$ ...
```

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

```
Definition bind $(a\ b : \mathsf{Set}) : \mathsf{State}\ a \rightarrow (a \rightarrow \mathsf{State}\ b) \rightarrow \mathsf{State}\ b$
  := fun $c_1\ c_2\ s_1$ ⇒ let $(x, s_2) := c_1\ s_1$
                   in $c_2\ x\ s_2$.
```

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 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:

$$\mathbf{fun}\ s_1 \Rightarrow P_1\ s_1 \wedge \mathbf{forall}\ x\ s_2, Q_1\ s_1\ x\ s_2 \rightarrow P_2\ x\ s_2$$

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 below postcondition for the bind operation.

$$\mathbf{fun}\ s_1\ y\ s_3 \Rightarrow \mathbf{exists}\ x, \mathbf{exists}\ s_2, Q_1\ s_1\ x\ s_2 \wedge Q_2\ x\ s_2\ y\ s_3$$

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 redefine the two auxiliary functions get and put to use the HoareState type:

```
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. 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 the relabel function satisfies its specification?

Using the HoareState type, we now arrive at the definition of the relabelling function presented in Figure 1. The function definition of relabel is identical to the version using the state monad in Section 3. The only novel aspect is the 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

```
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.
```

Fig. 1. The revised definition of the relabel 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 $i, i + 1, ...i + \mathsf{size}\ t$.

This definition gives rise to two proof obligations, one for each branch of the pattern match in the relabel function. In the Leaf case, the proof obligation is trivial. It is discharged automatically by the Program framework. To solve the remaining obligation, we need to apply several tactics to trigger β-reduction and introduce the assumptions. After giving the variables in the context more meaningful names, we arrive at the proof state in Figure 2.

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 the induction hypotheses, *flattenL* and *flattenR*, the remaining goal becomes:

```
==================================
  seq i (size l) ++ seq lState (size r) = seq i (size l + size r)
```

```
1 subgoal
  i : nat
  t : Tree nat
  n : nat
  l : Tree nat
  lState : nat
  sizeL : lState = i + size l
  flattenL : flatten l = seq i (size l)
  r : Tree nat
  rState : nat
  sizeR : rState = lState + size r
  flattenR : flatten r = seq lState (size r)
  finalState : rState = n
  finalRes : t = Node l r
=============================
  n = i + size t ∧ flatten t = seq i (size t)
```

Fig. 2. Proving the obligation of the relabelling function

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.

It is interesting to note that extracting a Haskell program from this revised relabelling function yields the same extracted code as the original definition of relabel in Section 2. As Coq's extraction mechanism discards propositional information, using the HoareState type does introduce any computational overhead.

6 Wrapping It Up

Now suppose we need to show that relabel satisfies a weaker postcondition. For instance, consider the NoDup predicate on lists from the Coq standard libraries. A list satisfies the NoDup predicate if it does not contain duplicates. The predicate's definition is given below.

Inductive NoDup : list a → Prop :=
  | NoDup_nil : NoDup nil
  | NoDup_cons : **forall** $x\ xs, x \notin xs$ → NoDup xs → NoDup $(x :: xs)$.

How can we prove that the tree resulting from a call to the 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 the postcondition and strengthen the precondition explicitly. In line with Hoare Type Theory [10, 9, 8], 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 _ _ c ⇒ c.
```

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

```
Program Fixpoint finalRelabel (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 first argument to the do function is trivial. To complete this definition, however, we need to prove that the postcondition can be weakened appropriately. This proof 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 [10, 9, 8]. 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, the HoareState type generalises 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 relabel′ that carries around an (infinite) list of fresh labels.

```
relabel′ : forall a b, Tree a → State (list b) (Tree b)
```

To prove that relabel meets the required specification, Hutton and Fulger prove various lemmas relating relabel′ and relabel. 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. Using the HoareState type, it may be possible to rule out errors by strengthening the precondition, thereby eliminating the need for this additional pattern match. Furthermore, the HoareState type presented here is slightly more general as its postcondition may also refer to the result of the computation.

Similar monadic structures to the one presented here have appeared in the verification of the seL4 microkernel [1] and security protocol verification [14]. There are a few differences between these approaches and the development presented here. Firstly, the postconditions presented here are ternary relations between the initial state, result, and final state. As a result, we do not need to introduce auxiliary variables to relate intermediate results. Sprenger and Basin [14] construct a Hoare logic on top of a weakest-precondition calculus. They present a shallow embedding of a series of logical rules that describe how the return and bind behave. On the other hand, Cock et al. [1] present their rules are presented as predicate transformers, using Isabelle/HOL's verification condition generator to infer the weakest precondition of a computation. The approach taken here focuses on programming with strong specifications in type theory, where the type of a computation fixes the desired pre- and postcondition.

Further Work

I have not 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 somehow canonical, for instance, by showing that the HoareState type forms a monad in some category of strong specifications. McKinna's thesis [7] on the categorical structure of strong specifications may form the starting point for such research.

Using the HoareState type to write larger programs will lead to larger proof obligations. For this approach to scale, it is important to provide a suitable set of custom tactics to alleviate the burden of proof. Some tactics that are already provided by the Program framework proved useful in the development presented here, but further automation might still be necessary.

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

References

[1] Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In: Munoz, C., Ait, O. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 167–182. Springer, Heidelberg (2008)

[2] Floyd, R.W.: Assigning meanings to programs. Mathematical Aspects of Computer Science 19 (1967)

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

[4] Hutton, G., Fulger, D.: Reasoning about effects: seeing the wood through the trees. In: Proceedings of the Ninth Symposium on Trends in Functional Programming (2008)

[5] Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: POPL 2006: 33rd Symposium on Principles of Programming Languages, pp. 42–54. ACM Press, New York (2006)

[6] Martin-Löf, P.: Constructive mathematics and computer programming. In: Proceedings of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages, pp. 167–184. Prentice-Hall, Inc., Englewood Cliffs (1985)

[7] McKinna, J.: Deliverables: a categorical approach to program development in type theory. Ph.D thesis, School of Informatics at the University of Edinburgh (1992)

[8] Nanevski, A., Morrisett, G.: Dependent type theory of stateful higher-order functions. Technical Report TR-24-05, Harvard University (2005)

[9] Nanevski, A., Morrisett, G., Birkedal, L.: Polymorphism and separation in Hoare Type Theory. In: ICFP 2006: Proceedings of the Eleventh ACM SIGPLAN Internation Conference on Functional Programming (2006)

[10] Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: Reasoning with the awkward squad. In: ICFP 2008: Proceedings of the Twelfth ACM SIGPLAN International Conference on Functional Programming (2008)

[11] Peyton Jones, S. (ed.): Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge (2003)

[12] Sozeau, M.: Subset coercions in Coq. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 237–252. Springer, Heidelberg (2007)

[13] Sozeau, M.: Un environnement pour la programmation avec types dépendants. Ph.D thesis, Université de Paris XI (2008)

[14] Sprenger, C., Basin, D.: A monad-based modeling and verification toolbox with application to security protocols. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 302–318. Springer, Heidelberg (2007)

[15] The Coq development team. The Coq proof assistant reference manual. LogiCal Project, Version 8.2 (2008)

[16] Wadler, P.: The essence of functional programming. In: POPL 1992: Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 1–14 (1992)