Monads using pseudocode

Satyadev Nandakumar

September 13, 2022

Contents

1 Overview

We try to present Monads using pseudocode. For the pseudocode, we will assume that we have an anonymous function with syntax $(\lambda x.y)$. This is different from the syntax in Haskell, and we hope that it makes monads more readable.

The strategy in these notes is to descibe examples in the following way : first describe what the problem we want to solve is, then describe the Monadic solution in pseudocode, followed by the code.

2 About Monads

Monads are a class of objects which support essentially two operations. From experience, it has been found that, in order to support modularity in a pure functional language, it is essential to support these two operations.

Modularity implies that there should be a layer of abstraction, a "black box", such that the outside world can interact with our code only through specific functions. Inside our code, we are free to implement the functionality in any way we wish - we may have side-effects, we may change the implementation inside our code without affecting the interface etc.

"What happens inside a monad stays inside the monad."

We know that this modularity is taken for granted by programmers using imperative languages, object-oriented languages etc. Pure functional languages are usually at a disadvantage - for example, when we change a function from an ordinary recursive version to a tail recursive version, we often have to pass extra arguments. This will change not only the definition of the function, but also every call to the function. This breaks modularity. An implementation change in the function will also imply that every user of the function has to change their code. This is what we intend to avoid. And how exactly does a monad mitigate this problem?

A monad guarantees that it has two operations.

1. A "boxing" operation which takes in a value, and encapsulates it inside a monad (think of this as a box with the value inside).

2. We do not provide an unboxing operation. Like a weary parent scolding a naughty child, we say "No, you cannot have the value. Just tell me what you want to do with the value. I will do that, and give you back the result, all nicely wrapped up inside a box". This operation is called bind. That analogy might clarify the type of bind in Haskell.

```
>>= :: Monad m => m a -> (a->m b) -> m b
```

Think of a parent having "a boxed value", a function that takes a plain value as argument, and returns a "boxed b". The parent performs the function, returning the "boxed b". Like the strict parent, we do not give the user "unboxed a" and allow the user to directly apply the function themselves. We do it for them.

The philosophy behind the bind operation is not strange. Object-oriented code often has private variables which can be manipulated only through get and set functions. Moreover, any operation involving these variables can be performed by a user only through member methods of the object. There is no way to directly access the private variables of an object. It may help to think of the values inside a monad as private variables of an object. For bind, the above analogy of a parent doing an operation for a child is the one that I have found to be helpful.

A further point about bind ...

Suppose the functions we need to apply, have type

```
Monad m => m a -> (a -> m a) -> m a
```

That is to say, `b` is the same as `a`. If this is the case, then the result of the bind application is the same type as the starting monad. This means that we can have long chains of applications of bind of the following form.

```
monadic_value >>= function1 >>= function2 >>= function3
```

since the result of `monadic_value >>= function1` is also a monad of the same type as the initial value, and so on. Hence, it is also common to say that Monads are "pipes with types", i.e. similar to Unix pipes in their method of action, but with a type system to check that only proper outputs are channelled to the next functions down the chain.

This means that the bind operation can be involved in long chains of operations starting from a monadic value. (Of course, all the functions must have type $(a \to ma)$.)

Now, let us look at a few standard monads, and understand them from the viewpoint that monads are tools for enchancing the modularity of the code. We first look at three built-in monads.

3 The `Maybe` Monad

3.1 Maybe monad - definition of return and bind

The data type `Maybe` is defined as

```
data Maybe a = Nothing | Just a
```

The purpose of the `Maybe` monad is to indicate whether a value is valid (`Just x`) or whether there was an error, in which case we just carry forward `Nothing`.

Now, we see how it helps to make `Maybe` into a monad. First of all, the "boxing" function should take a value `v`, and return `Just v`. What about the bind operation? Does it make sense to add a bind operation to `Maybe`? If we want to add a bind, what should it do? Remember the parent who has a monad value applies a function taking a plain value and returns the boxed result. That analogy suggests that bind in Maybe should do the following.

```
bind(Nothing, f )    = Nothing
bind(Just x, f )     = f x
```

3.2 How to use the Maybe monad

To demonstrate one way in which we can use the `Maybe` data type, we may define a safe division operation below.

```
safeDiv x y
  | (y==0)     =  Nothing
  | otherwise  =  return (x/y)
```

The phrase `return (x/y)` wraps the value `(x/y)` value inside a `Maybe` monad, specifically, storing it as `Just (x/y)`. Then `safeDiv 8 2` should evaluate to `Just 4`.

Assume that we have a function

```
add_one x = return (x+1)
```

Then we can chain the operations as follows:

```
safeDiv 3 4 >>= add_one
```

This will reduce to

```
(Just 0.75) >>= add_one
```

which yields `Just 1.75`. We know that we can chain these kind of operations indefinitely, as long as the functions like `add_one` take care to use return to finally result in a monadic value.

4 The List Monad

4.1 The List monad : definition of return and bind

The most common data type in Haskell, namely lists, are monads themselves. We have seen that lists are applicative functors. Now, in order to make lists into monads, all that remains is to implement the `return` and $\rangle\rangle =$ operations in a way that makes sense as monadic operations.

Of these, `return` is easy. It should just create a list out of a given value.

```
return x = [x]
```

What should bind do? It should take a list (this is the monadic first argument), and a function f that takes a value of type a and returns a list of elements of type b. The reasonable thing to do is to apply this function f to each of the elements in the first list. The result of application of f is itself a list. Now, we have two possible choices to implement bind.

Suppose the input list is `[a1, a2, a3]`, and `f a1 = [b1, b2]`, `f a2 = [b3]` and `f a3 = [b4, b5]`.

One possible result is to enumerate the results as a list: `[[b1, b2], [b3], [b4, b5]]`. But in this case, the return type of f is $[b]$ and the return type of bind is $[[b]]$. But these must be the same!

The second possibility is to *concatenate* the results: `[b1, b2, b3, b4, b5]`. This causes the result of bind and the result of the application of f both to be $[b]$. In fact, this is actually what is done.

```
bind ( list, f ) = concat ( map (f, list) )
```

4.2 How to use the list monad

Consider the following function which squares the number in a list:

```
squareM x = return (x*x)
```

Because of the `return`, the function can be used in a monadic bind. Let's write one more function, which takes a value x and returns a list of positive and negative versions of x.

```
posNeg x = [x,-x]
```

This function illustrates that the function used in the bind may not always return singleton lists.

Now, we can concatenate these functions in chains of bind:

```
[1,2,3] >>= posNeg >>= squareM
```

returns `[1,1,4,4,9,9]`.

5 The IO monad

We know that monads are meant to modularize code in such a way that we can have a "black box" inside which we can implement functions like `return` and $\rangle\rangle =$ (bind) in any manner we can as long as we are consistent with the type system. This opens up the possibility that we may engage in truly nefarious activities inside a monad, even go so far as to have side effects. Side effects will be accomplished by holding an internal "state of the world", as we will soon see.

Keep in mind that `IO` is one of the most complicated Monads in the Haskell system. The sections above should have convinced you that Monads by themselves are not that complicated, and in fact, in many situations, lead to elegant code from a user's perspective. Now, let us look at the IO Monad.

First of all, what is the `IO` type? The Haskell report 2010 unhelpfully says in Section 6.1.7 that "IO The IO type is abstract: no constructors are visible to the user." [Haskell 2010] However, GHC, which is the standard implementation of the Haskell language, does give some implementation details. [StackOverflow 1]

```
Prelude> :i IO
newtype IO a
= GHC.Types.IO (GHC.Prim.State# GHC.Prim.RealWorld
                -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
-- Defined in ‘GHC.Types’
instance Monad IO -- Defined in ‘GHC.Base’
instance Functor IO -- Defined in ‘GHC.Base’
```

This is helpful if we look carefully. First of all, it says that `IO` is a polymorphic type parametrized by type a. It also says that IO is a Functor and a Monad. There is also a type signature, which uses internal representations from the GHC implementation. We can use the approximate definition, as in the Haskell Wiki [Haskell Wiki], that

```
type IO a          =       RealWorld  -> (a,RealWorld)
```

That is, an `IO a` is a function taking the current state of the real world (for example, the state of the terminal, file descriptors, signals etc.), and returns a tuple containing the new state of the world, and `a` as components.

The new state of the world may be used for further IO actions. This is how chaining will take place. We will see this in a moment.

5.1 How to use the IO monad

1. Consider the particular main function defined below.

```
main = putStrLn "Hello"
```

The function main has the type `RealWorld -> ((), RealWorld)`. That is, it starts with an initial realworld, and returns the empty tuple type, and a realworld, possibly different from the realworld it received. This change may happen, for example, because it received a terminal as part of the realworld, and then printed out some message. The printing out (side-effect) results in a new realworld, which main then passes.

2. For a slightly different example, consider

```
main = getLine
```

which can also be written as

```
main = do name <- getLine; return name
```

The function main here has the type `RealWorld -> (String, RealWorld)`, which we call `IO String`.

3. Let us look at getchar which starts with a real world, and returns a character and a transformed world (where, for example, read pointer to the next token is advanced by one position). So according to our discussion,

```
getChar: : RealWorld -> (Char, RealWorld)
```

Using the type synonym for `IO`, we say that `getChar :: IOChar`.

The general drift of the above discussion is this. The monadic operations involving IO always start with an implicit RealWorld value, which is not mentioned in the Haskell type signature. The return type is a "tuple" containing a value and the transformed RealWorld. In our approximate story, the output type is `(a, RealWorld)`. In Haskell, the type synonym is is `IO a`.

5.2 return and bind

Since the IO monad is complicated, we give only an approximate idea of return and bind.

1. The monadic operator `return` just wraps a plain value inside an `IO`. Recalling that an `IO a` is a type synonym for a function that maps a RealWorld value to a tuple containing two parts - a value of type `a` and a possibly different RealWorld value - the return monadic operator can be defined as

```
return:: a -> IO a
return x = (\ w -> (x,w))
```

2. The monadic operator $\rangle\rangle =$ (bind) should do the following. It takes two arguments. The first argument is an `IO a` monad. The second argument is a function, which takes a plain value, and returns an `IO b` monad. The bind operator must then use the function given to apply it on the monadic `a` (i.e. the boxed version of a) that it already has. Keep in mind that the box also contains the current state of the world.

The definition of bind (following [Haskell Wiki]) is: (Note that this is not valid Haskell code, it is just a way to explain what goes on.)

```
(>>=) :: IO a -> (a -> IO b) -> IO b
x >>= f = ( \ initial_world ->
                                let
                                   (m,intermediate_world) = x initial_world
                                   (n,final_world) = f m intermediate_world
                                in
                                   (n,final_world))
```

Acknowledgments

The author wishes to thank Subin Pulari for comments on an earlier draft of the paper

6 Bibliography

To be filled.