Monads and Side Effects in Haskell

Alan Davidson

October 2, 2013

In this document, I will attempt to explain in simple terms what monads are, how they work, and how they are used to perform side effects in the Haskell programming language. I assume you already have some familiarity with Haskell syntax.

I will start from relatively simple, familiar ideas, and work my way up to our actual goals. In particular, I will start by discussing functors, then applicative functors, then monads, and finally how to have side effects in Haskell. I will finish up by discussing the functor and monad laws, and resources to learn more details.

1 Functors

A functor is basically a data structure that can be mapped over:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

This is to say, `fmap` takes a function that takes an `a` and returns a `b`, and a data structure full of `a`'s, and it returns a data structure with the same format but with every piece of data replaced with the result of running it through that function.

Example 1. *Lists are functors.*

```
instance Functor [] where
  fmap _ [] = []
  fmap f (x : xs) = (f x) : (fmap f xs)
```

We could also have simply made `fmap` equal to `map`, but that would hide the implementation, and this example is supposed to illustrate how it all works.

Example 2. *Binary trees are functors.*

```
data BinaryTree t = Empty
                  | Node t (BinaryTree t) (BinaryTree t)

instance Functor BinaryTree where
  fmap _ Empty = Empty
  fmap f (Node x left right) = Node (f x) (fmap f left) (fmap f right)
```

Note that these are not necessarily binary *search* trees; the elements in a tree returned from `fmap` are not guaranteed to be in sorted order (indeed, they're not even guaranteed to hold sortable data). The tree returned from `fmap` will have shape identical to the one passed into it, but the values in the returned tree are whatever we get back from the function we're mapping.

More complicated (i.e., non-binary) trees are also functors, but they make for a more complicated example.

2 Applicative Functors

An applicative functor is a bit like a regular, garden-variety functor, except that the function being mapped onto the data structure needs to be contained in the data structure, too. They're found in the `Control.Applicative` module.

```
class Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The `pure` function takes a thing and puts it in an applicative functor with the minimal context (i.e., the simplest version of the applicative functor that can hold the thing). The `<*>` operator is just like `fmap`, except that the functions are held in an applicative functor as well. Note, though, that this means there may be anywhere from zero to an infinite number of functions in our data structure!

It's worth noting that the actual definition of `Applicative` contains some extra functions I haven't mentioned, but they get automatically implemented if we just deal with these two.

Example 3. *`ZipList` is a type defined in the `Control.Applicative` module. It's implemented as a list, except that it is defined as an applicative functor in the following way:*

```
data ZipList t = ZipList [t]

instance Applicative ZipList where
  pure a = Ziplist (repeat a)
  ZipList [] <*> _ = ZipList []
  _ <*> ZipList [] = ZipList []
  ZipList (f : fs) <*> ZipList (x : xs) =
    let
      ZipList fxs = ZipList fs <*> ZipList xs
    in
      ZipList (f x : fs)
```

In other words, when using `<*>` on two `ZipList`s, the first function is applied to the first value, the second function is applied to the second value, etc.

The actual definition of `ZipList` is more concise because it uses `zipWith` to do the heavy lifting in `<*>`. My implementation is longer and more explicit so that it makes a more illustrative example than the real version does.

Example 4. *Recall that* `Either` *is a type constructor that takes two types: the first is the type of an error that it might give, and the second is the type that it contains if there are no errors. Using partial application, we can make the type constructor* `Either e` *that is either an error of type* `e` *or some value of a type not yet specified.*

`Either e` is an applicative functor.

```
data Either e t = Left e
                | Right t

instance Applicative Either e where
  pure = Right
  Left e <*> _ = Left e
  _ <*> Left e = Left e
  Right f <*> Right x = Right (f x)
```

Theorem 1. *All applicative functors are also functors.*

```
instance (Applicative a) => Functor a where
  fmap f = pure f <*>
```

From this theorem, we can see that applicative functors are at least as powerful as functors, since the latter can be implemented using the former.

3 Monads

Finally, we get to monads, which can be found in the `Control.Monad` module. Here's what they look like:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

Like applicative functors, there are several other functions defined in the `Monad` class that get implemented automatically, which I have omitted.

The function `return` is extremely misleadingly named, because it does not work like return statements in any other language I know of. Instead, it is a function that takes a thing and puts it in the minimum context to be a monad. You can think of `return` as being exactly the same as `pure` from applicative functors.

The `>>=` operator (pronounced "bind") is a bit odd in that its second argument is a function that knows about the monad it's used in. Bind takes something encapsulated in

a monad, and a function that can work on the unencapsulated version of that something and return its answer wrapped in the same monad. Bind gives back the answer from the function, after first merging in any extra context from the monad that the first argument came in.

Example 5. *`Maybe` is a monad.*

```
instance Monad Maybe where
  pure = Just
  Nothing >>= _ = Nothing
  Just x >>= f = f x
```

Example 6. *We could implement a way of accumulating debugging information as a data structure that keeps a list of strings associated with the data (the list contains the debugging information about how this data was generated). Such a debugging data structure is a monad.*

```
data Debugger t = Debugger t [String]

instance Monad Debugger where
  return a = Debugger a []
  Debugger x xDebug >>= f =
    let
      Debugger y yDebug = f x
    in
      Debugger y (xDebug ++ yDebug)
```

This example is inspired by the `Writer` datatype in the `Control.Monad.Trans.Writer` module, which is also a monad.

Theorem 2. *All monads are applicative functors.*

```
instance (Monad m) => Applicative m where
  pure = return
  fm <*> xm = fm >>= (\f ->
              xm >>= (\x ->
              return (f x)))
```

To put the implementation of `<*>` into English, we open up `fm` and take out the function(s) inside it, we open up `xm` and take out the value(s) inside it, apply the function(s) to the value(s), and then wrap the whole thing up again in a monad/applicative functor, merging in any extra data from the monads that contained the values and functions.

We can conclude from this that monads are at least as powerful as applicative functors, because the latter can be implemented using the former. We can also see that all monads are functors, because all monads are applicative functors and all applicative functors are functors.

Historical note: `Applicative` was added to Haskell long after `Monad` was. There may be legacy code that is an instance of `Monad` but which is not explicitly considered an instance of `Applicative` because it was written before `Applicative` was created.

4 Side Effects in Haskell

So, how can we use a monad to perform side effects in Haskell? The answer is that we can't! However, there is one particular monad, called `IO`, that has some secret sauce added on top of it which we can use to get side effects.

Suppose there is a type called `World`, which contains all the state of the external universe in it: it's got a copy of the contents of the hard drive, and a list of keypresses from the keyboard, and the current contents of the screen buffer, and the Internet connection, and everything else. Here's how to think about what `IO` does:

```
type IO t = World -> (t, World)
```

In other words, `IO t` is a function which takes a `World` and returns the `t` it's supposed to contain, along with a new, updated `World` formed by modifying the original one in the process of getting the `t`. This isn't actually how it's implemented, of course (where would you store the extra copies of the contents of your hard drive that each of the `World`s contains?), but it's a way of thinking about how this all works.

Here's the monad implementation for our `IO`:

```
instance Monad IO where
  return x world = (x, world)
  (ioX >>= f) world0 =
    let
      (x, world1) = ioX world0
    in
      f x world1  -- Has type (t, World)
```

Because Haskell functions are curried, the `return` function takes an argument called `x` and gives back a function that takes a `World` and returns `x` along with the "new, updated" `World` formed by not modifying the `World` it was given.

The implementation of bind is a bit trickier: the expression `(ioX >>= f)` has type `World -> (t, World)`. This is to say, it's a function that takes a `World`, called `world0`, which it uses to extract `x` from its `IO` monad. This gets passed to `f`, resulting in another `IO` monad, which again is a function that takes a `World` and returns a `t` and a new, updated `World`. We give it the `World` we got back from getting `x` out of its monad, and the thing it gives back to us is the `t` with a final version of the `World`.

So, okay, this `IO` appears to act like a monad, and we can see how it can modify the `World` it is given and give back an updated one. How does it know which `World` it's given in the first place?

If your program is compiled using GHC, you'll know that you need to have a `main` defined somewhere with type `IO ()`, and this is where program execution starts. This `main` is given the initial `World` to start everything off, and it passes the updated ones from each `IO` to the next. If you have an `IO` that is not reachable from `main`, it will never be executed, and it doesn't get a `World` passed to it.

If, instead, you're using GHCI to run your Haskell commands, everything is wrapped in an implicit `IO`, since the results get printed out to the screen. Every time you give it a new command, it passes in the current `World`, gets the result of your command back, calls `print` on it (which updates the `World` by modifying the contents of the screen or the list of defined variables or the list of loaded modules or whatever), and then saves the new `World` to give to the next command.

The neat thing about this is that there's only 1 `World` in existence at any given moment. Each `IO` takes that one and only `World`, consumes it, and gives back a single new `World`. Consequently, there's no way to accidentally run out of `World`s, or have multiple ones running around.

5 Extra Details

5.1 Applicative Functor Laws

Applicative functors need to satisfy a few extra laws that Haskell's type system cannot enforce on its own:

This:	must be equivalent to this:
`pure f <*> x`	`fmap f x`
`pure id <*> x`	`x`
`pure f <*> pure x`	`pure (f x)`
`pure ($ x) <*> f`	`f <*> pure x`
`pure (.) <*> f <*> g <*> x`	`f <*> (g <*> x)`

These should not be onerous things; they just ensure that applicative functors behave the way you intuitively think they should. Note that the first two laws of applicative functors imply this law of functors:

This:	must be equivalent to this:
`fmap id x`	`x`

5.2 Monad Laws

Like applicative functors, monads have a few laws that Haskell's type system cannot enforce but which should hold true anyway:

This:	must be equivalent to this:
`m >>= return`	`m`
`return x >>= f`	`f x`
`(m >>= f) >>= g`	`m >>= (\x -> f x >>= g)`

That last one is a little hard to parse. It's just saying that functions on monads should be associative: you can bind `m` to `f` and bind the result of that to `g`, and what you get back will be equivalent to binding `m` to a function that binds the result of `f` to `g`.

Again, these are not supposed to be a burden on the programmer; they're just there to make sure that things that claim to be monads act in somewhat intuitive ways.

6 Works Consulted

This was intended to be just a cursory introduction to monads and side effects in Haskell. This information has been distilled from other, more thorough sources, and if you're interested in learning more, I encourage you to take a look at them.

The vast majority of this information is taken from Learn You A Haskell For Great Good, which can be found at `http://LearnYouAHaskell.com`. It's a book, and you can read it online for free or purchase a physical copy. The information here has been taken from chapters 8, 11, 12, and 13.

The discussion of side effects was taken from the Haskell Wiki, at `http://haskell.org/haskellwiki/IO_inside`.