

## Reader Monad Confusion

I've been using haskell for about a year now, and thought I was really beginning to get a handle on this monad thing. Other than a strange name, and confusing C programmers with the keyword return that doesn't return, they looked like they were just a way of connecting things together, kinda like function composition. In fact, I found myself frequently writing code like:

### Haskell Code

---

```
1 goldbach =
2   let primes = sieve [2..]
3       where sieve (p:xs) = p : sieve [x | x<-xs, x `mod` p /= 0]
4       p100 = takeWhile (<100) primes
5       pairs = [x+y | x<-p100, y<-p100]
6       evens = [4,6..100] :: [Int]
7       isPair x = x `elem` pairs
8       checkEvens = map isPair evens
9       conjecture = all id checkEvens
10  in conjecture
```

---

Here function composition, (not) cleverly disguised as a bunch of *lets* that accumulate, acts as the bind operator, providing sequencing. What I mean by this is that conjecture depends on checkEvens, which depends on isPair and evens, which depends on pairs, which depends on p100, which depends on primes, so even though the order of evaluation isn't explicit in pure functions, it is easy to write code so that in fact it is, by building up to the final answer in little pieces that depend on each other. With this in mind, even the Maybe monad seemed pretty straightforward, after it is defined as:

### Haskell Code

---

```
11 data Maybe a = Nothing | Just a
12
13 instance Monad Maybe where
14   (Just x) >>= k = k x
15   Nothing >>= _ = Nothing
16   return x = Just x
```

---

You just have to remember that k is a function that takes an x to a Maybe y like:

### Haskell Code

---

```
17 k :: Int -> Maybe String
18 k x = if x == 0 then Nothing else Just "Something"
```

---

So what happens if we **bind** Nothing with k? Well the Monad rule above which says that: **Nothing** >>= \_ = **Nothing** means Nothing bound to anything still results in Nothing, so Nothing >>= k equals Nothing. What about Just 1 bound with k, well the Monad rule above, **(Just x)** >>= **k** = **k x** means apply the function k to 1, which since 1 /= 0 returns *Just "Something"*. We can continue along, binding the output of k with another function, say k1, as long as the type of k1 is String -> Maybe whatever. So the analogy of binding with function composition is not really that far off.

In fact, with just a little more complicated binding, even the list Monad

### Haskell Code

---

```
19 instance Monad [] where
20   m >>= k = concat (map k m)
21   return x = [x]
```

---

is just a variation of this theme. The binding now specified by `m >>= k = concat (map k m)` So the next monad I took a look at was the Reader Monad. Looking through the source for Reader.hs, I came across:

Reader Defined

### Haskell Code

---

```
22 newtype Reader r a = Reader { runReader :: r -> a }
23 instance Monad (Reader r) where
24   return a = Reader $ \_ -> a
25   m >>= k = Reader $ \r -> runReader (k (runReader m r)) r
```

---

As Raymond's father used to say, **Holy Crap!** If this code is perfectly clear to you, then you can skip the rest of this article, for I doubt you will learn anything new. If not, read on, and I hope I can make it a little clearer for you what is going on here.

The first thing that confused me was the newtype constructor. We have a Reader on the left hand side with two arguments, and a Reader on the right hand side with what looks like a record that has a variable in it called runReader. Huh? Also, I see the following code somewhere:

runReader Example

### Haskell Code

---

```
26 r1 :: Reader Int String
27 r1 = do
28   a <- ask
29   return $ show (a+1)
30 test1 = putStrLn $ runReader r1 1
```

---

which dutifully prints 2. But!, again Reader has two arguments, one is an Int and one is a String, and now runReader isn't a field in a Record type but it looks like a function of two arguments as well. I'm very confused.

Maybe I'm just not smarter than the average bear, or maybe there are others out there who find this state of affairs similarly confusing, so let's see if we can dissect this stuff and make some sense of it.

## Areas of Confusion

1. What is the relationship between the Reader on the left hand side of the equals sign in the newtype constructor and the Reader on the right hand side?

2. Why is there a Record field on the right hand side?
3. What is that `r ->` a doing there?

## Answers

1. The Reader on the left hand side is a type constructor. It is what you use when you write type signatures. In that sense it is like `Int` or `String` or `Maybe`. The Reader on the right hand side is what you use to make some data of type Reader (the left hand side). It is called a data constructor. I think the best way to keep these straight is to remember `Maybe`. In the definition of `k` above, the type signature told us that `k` takes `Int` and returns a `Maybe String`. Those are all type constructors. To make a `Maybe String`, the `k` function returns either a `Nothing`, or a `Just x`, where `x` is a string. The `Nothing` and `Just` are data constructors (really just functions) whose results are `Maybes`. We have every right to rename, say `Just` to `Maybe`, and all we will do is make things more confusing, but still legal. Then `Maybe` would have two meanings, one as (still) a type constructor, and another (now) as a data constructor. Well, that is what we have done with `Reader`. The `Reader` on the right hand side is a data constructor. Perhaps we would have been better off if it had been called `ReaderData` instead of `Reader`, but it wasn't, so get used to it. Even though it is the same word, its meaning is totally different on each side of the newtype declaration.
2. Okay, so why is there a record on the right hand side of the newtype constructor? Normally we see record types like this:

### Haskell Code

---

```
31 data Birthday = BirthdayData {
32     month :: Int,
33     day    :: Int,
34     year   :: Int }
```

---

and we think of them as just variables ... parts of a `Birthday`. But in reality, they are functions (called selector functions). They are functions that take some `Birthday` data, and return an `Int`. So if you had:

### Haskell Code

---

```
35 henry = BirthdayData 6 15 1954
36 howOldIsHenry = 2009 - (year henry)
```

---

`howOldIsHenry` would return 55. (Am I really that old?) and `year` in the definition of `howOldIsHenry` is actually a function, that takes a `Birthday` data type, which you can construct by calling the `BirthdayData` data constructor with 3 integers, and returns the year portion of the `Birthday`. So please remove the imperative mind set that what is inside a `Record` type is a bunch of variables, it is actually a bunch of functions that operate on the data type.

3. Okay, so what about that  $(e \rightarrow a)$  on the right hand side. Remember that functions are *first class objects* in haskell. They are data, just like Ints, Strings, or anything else. As such, they need a type constructor, and that type constructor is spelled  $(\rightarrow)$ . We usually write it in its infix form, so if you want to describe the type constructor of a function from a to b, you would write  $(a \rightarrow b)$ , though you are perfectly free to write the more confusing version of  $((\rightarrow) a b)$  if you desire. Now returning to the newtype definition above, well, whole thing is really a shortcut way of writing:

### Haskell Code

---

```
37 newtype Reader e a = Reader (e->a)
38 runReader (Reader f) x = f x
```

---

I think this is much clearer. It tells us that a Reader type constructor takes two type parameters, an e and an a. You create some Reader data by suppling a function (f) which takes an e to an a. You then wrap that function (f) inside an data constructor (confusingly also named Reader) such that if you provide the (Reader f) thing to runReader, along with another argument (x) of type e, you will get back f x (f of x) with returns something of type a.

We can see this in action with the following code:

### Haskell Code

---

```
39 r2 :: Reader Int String
40 r2 =
41   let f x = show (x+1)
42       in Reader f
43
44 test2 = putStrLn $ runReader r2 1
```

---

which again prints out 2. Look ma, no monads.....yet. So while the type of r2 is Reader Int String, looking at it we see that it is really a function (f) from an Int  $\rightarrow$  String, wrapped inside a Reader data constructor. Okay, I hope this has cleared up the confusion surrounding the type constructor of a Reader. At least it did for me once I figured it all out. Now let's move on the the functions that make Reader a Monad. The line that needs serious decoding is:

### Haskell Code

---

```
45 m >>= k = Reader $ \r -> runReader (k (runReader m r)) r
```

---

But, before we get to that, let's understand why return is defined as:

### Haskell Code

---

```
46 return a = Reader $ \_ -> a
```

---

Now, the point of having a Reader is so that we can have some kind of environment hanging around in the background, without having to pass it around to all of our functions explicitly. Once we've set up this environment, which can be any arbitrarily complicated data structure, we can reference it with the *ask*

function while we are running inside the Reader Monad. In the example above, the environment was a simple `Int` value, but in general it could be contents of every file on your computer. (though that might be a wee bit large.) The great thing about this is that it gives you, in a sense, access to *global* variables in a controlled *local* (Reader) environment. Now remember, `runReader` takes a function, `f`, which has access to an environment, `e`, and returns a value, `a`. That function may or may not care what is in it's environment. It is perfectly free to ignore it's environment if it wants to. A function which ignores its environment, and always returns the same result is known as the constant function, written `const` in haskell. (`const :: a -> b -> a`) Now consider the meaning of `return` for Monads. It must take an arbitrary data type, `a`, and inject it into the Monad. In the Reader monad, if we want the result of running the reader to be `a`, then we must create a function that takes it's environment `e` as an input, ignores it, and returns `a` as its result. Haskell has already provided us with such a function, and if we rewrite the definition of `return` above as:

### Haskell Code

---

```
47 return a = Reader (const a)
```

---

we will get `1` back when we say: `runReader (return 1) "anything"` Just as we would if we said: `runReader (Reader (const 1)) "anything"` all `return` does is wrap the constant function inside a Reader data constructor. The constant functions ignores it environment, which in this case is the string "anything".

Okay, now lets move on to deciphering:

### Haskell Code

---

```
48 m >>= k = Reader $ \r -> runReader (k (runReader m r)) r
```

---

I think this would be a lot clearer if it were rewritten in an equivalent form like:

Reader Bind

### Haskell Code

---

```
49 (Reader f1) >>= f2 = Reader $ \e -> runReader (f2 (f1 e)) e
```

---

now there is one less `runReader` to deal with, and the idea that we are dealing with two different functions is a little more obvious. To understand this, remember that the purpose of `>>=` is to *bind* together a Monad with a function that returns a Monad. Also recall that `runReader` take two arguments, a function to run and an environment to carry around. Now working from the inside out, we see `f1` is a function. Thanks to pattern matching, we have extracted it from the Reader data constructor and named it `f1`. Now `f1` takes its environment `e`, and returns an `a`. But where does this environment magically come from? Well, even if we have a long sequence of operations bound together in our Reader Monad, say `f1 >>= f2 >>= f3 ... >>= fn`, at some point we are going to take all of them and say `runReader (f1 >>= f2 >>= f3 ... >>= fn) environment`

where **environment** will be the argument supplied to `f1`, which gets passed on to `f2` etc. That argument shows up as `e` in the definition above. So `f1` takes that `e` and returns an `a`. Let's rewrite the above with this in mind:

### Haskell Code

---

```
50 (Reader f1) >>= f2 = Reader $ \e -> runReader (f2 a) e
```

---

Now what is `f2`? `f2` is a function that takes an `a` and returns a *Reader b*. After all, that is what binding is all about. Let's make that substitution.

### Haskell Code

---

```
51 (Reader f1) >>= f2 = Reader $ \e -> runReader (Reader b) e
```

---

Now what does `runReader (Reader b) e` do? `Reader b` is a function that takes an `e` and returns a `c`, (`c` does not have to be the same type as either `a` or `b`). `runReader` takes that function, supplies the `e` and returns the `c`. Let's rewrite again:

### Haskell Code

---

```
52 (Reader f1) >>= f2 = Reader $ \e -> c
```

---

but this is the same as:

### Haskell Code

---

```
53 (Reader f1) >>= f2 = Reader (const c)
```

---

which if you supply to `runReader`, along with an environment, will return `c`, where I mean return in the sense of `C`, and not in the sense of `Monad`. So, the definition of `bind` is a way of chaining all of these functions together in such a way that the externally supplied environment `e` is always the first argument to the functions we create. Pretty tricky, I must say.

## Conclusion

So what have we learned? For me a couple of things. One, is you have to be very careful about keeping straight when something is a type constructor, like `Maybe`, and when it is an data constructor, like `Nothing` or `Just`, especially when it uses the same name, like `Reader`. Next, banish your `C` concepts that `Records` are lists of variables. They are functions that can do arbitrary things once they peel off their data constructor. Finally, when you find a complicated expression that is difficult to understand, work from the inside out, figure out what the inputs and outputs are, and substitute them in the expression to try to simplify it. Eventually it should become obvious.

## Comments

The raw lhs file is available for you to play with if you are so inclined. You can also read this page in pdf format Please address any complaints, comments, or misc. personal abuse to Henry Laxen Thanks.