21

Monads and Effects

Now that we know what the monad is for — it lets us compose embellished functions — the really interesting question is why embellished functions are so important in functional programming. We've already seen one example, the `Writer` monad, where embellishment let us create and accumulate a log across multiple function calls. A problem that would otherwise be solved using impure functions (e.g., by accessing and modifying some global state) was solved with pure functions.

21.1 The Problem

Here is a short list of similar problems, copied from Eugenio Moggi's seminal paper[1], all of which are traditionally solved by abandoning the purity of functions.

- Partiality: Computations that may not terminate

[1] https://core.ac.uk/download/pdf/21173011.pdf

- Nondeterminism: Computations that may return many results
- Side effects: Computations that access/modify state
  - Read-only state, or the environment
  - Write-only state, or a log
  - Read/write state
- Exceptions: Partial functions that may fail
- Continuations: Ability to save state of the program and then restore it on demand
- Interactive Input
- Interactive Output

What really is mind blowing is that all these problems may be solved using the same clever trick: turning to embellished functions. Of course, the embellishment will be totally different in each case.

You have to realize that, at this stage, there is no requirement that the embellishment be monadic. It's only when we insist on composition — being able to decompose a single embellished function into smaller embellished functions — that we need a monad. Again, since each of the embellishments is different, monadic composition will be implemented differently, but the overall pattern is the same. It's a very simple pattern: composition that is associative and equipped with identity.

The next section is heavy on Haskell examples. Feel free to skim or even skip it if you're eager to get back to category theory or if you're already familiar with Haskell's implementation of monads.

21.2 The Solution

First, let's analyze the way we used the `Writer` monad. We started with a pure function that performed a certain task — given arguments, it pro-

duced a certain output. We replaced this function with another function that embellished the original output by pairing it with a string. That was our solution to the logging problem.

We couldn't stop there because, in general, we don't want to deal with monolithic solutions. We needed to be able to decompose one log-producing function into smaller log-producing functions. It's the composition of those smaller functions that led us to the concept of a monad.

What's really amazing is that the same pattern of embellishing the function return types works for a large variety of problems that normally would require abandoning purity. Let's go through our list and identify the embellishment that applies to each problem in turn.

21.2.1 Partiality

We modify the return type of every function that may not terminate by turning it into a "lifted" type — a type that contains all values of the original type plus the special "bottom" value ⊥. For instance, the `Bool` type, as a set, would contain two elements: `True` and `False`. The lifted `Bool` contains three elements. Functions that return the lifted `Bool` may produce `True` or `False`, or execute forever.

The funny thing is that, in a lazy language like Haskell, a never-ending function may actually return a value, and this value may be passed to the next function. We call this special value the bottom. As long as this value is not explicitly needed (for instance, to be pattern matched, or produced as output), it may be passed around without stalling the execution of the program. Because every Haskell function may be potentially non-terminating, all types in Haskell are assumed to be lifted. This is why we often talk about the category **Hask** of Haskell

(lifted) types and functions rather than the simpler **Set**. It is not clear, though, that **Hask** is a real category (see this Andrej Bauer post[2]).

21.2.2 Nondeterminism

If a function can return many different results, it may as well return them all at once. Semantically, a non-deterministic function is equivalent to a function that returns a list of results. This makes a lot of sense in a lazy garbage-collected language. For instance, if all you need is one value, you can just take the head of the list, and the tail will never be evaluated. If you need a random value, use a random number generator to pick the n-th element of the list. Laziness even allows you to return an infinite list of results.

In the list monad — Haskell's implementation of nondeterministic computations — **`join`** is implemented as **`concat`**. Remember that **`join`** is supposed to flatten a container of containers — **`concat`** concatenates a list of lists into a single list. **`return`** creates a singleton list:

```
instance Monad [] where
    join = concat
    return x = [x]
```

The bind operator for the list monad is given by the general formula: **`fmap`** followed by **`join`** which, in this case gives:

```
as >>= k = concat (fmap k as)
```

Here, the function **`k`**, which itself produces a list, is applied to every element of the list **`as`**. The result is a list of lists, which is flattened using **`concat`**.

[2] http://math.andrej.com/2016/08/06/hask-is-not-a-category/

From the programmer's point of view, working with a list is easier than, for instance, calling a non-deterministic function in a loop, or implementing a function that returns an iterator (although, in modern C++[3], returning a lazy range would be almost equivalent to returning a list in Haskell).

A good example of using non-determinism creatively is in game programming. For instance, when a computer plays chess against a human, it can't predict the opponent's next move. It can, however, generate a list of all possible moves and analyze them one by one. Similarly, a non-deterministic parser may generate a list of all possible parses for a given expression.

Even though we may interpret functions returning lists as non-deterministic, the applications of the list monad are much wider. That's because stitching together computations that produce lists is a perfect functional substitute for iterative constructs — loops — that are used in imperative programming. A single loop can be often rewritten using `fmap` that applies the body of the loop to each element of the list. The `do` notation in the list monad can be used to replace complex nested loops.

My favorite example is the program that generates Pythagorean triples — triples of positive integers that can form sides of right triangles.

```
triples = do
    z <- [1..]
    x <- [1..z]
    y <- [x..z]
    guard (x^2 + y^2 == z^2)
    return (x, y, z)
```

[3] http://ericniebler.com/2014/04/27/range-comprehensions/

The first line tells us that `z` gets an element from an infinite list of positive numbers `[1..]`. Then `x` gets an element from the (finite) list `[1..z]` of numbers between 1 and `z`. Finally `y` gets an element from the list of numbers between `x` and `z`. We have three numbers $1 \leqslant x \leqslant y \leqslant z$ at our disposal. The function `guard` takes a `Bool` expression and returns a list of units:

```
guard :: Bool -> [()]
guard True = [()]
guard False = []
```

This function (which is a member of a larger class called `MonadPlus`) is used here to filter out non-Pythagorean triples. Indeed, if you look at the implementation of bind (or the related operator `>>`), you'll notice that, when given an empty list, it produces an empty list. On the other hand, when given a non-empty list (here, the singleton list containing unit `[()]`), bind will call the continuation, here `return (x, y, z)`, which produces a singleton list with a verified Pythagorean triple. All those singleton lists will be concatenated by the enclosing binds to produce the final (infinite) result. Of course, the caller of `triples` will never be able to consume the whole list, but that doesn't matter, because Haskell is lazy.

The problem that normally would require a set of three nested loops has been dramatically simplified with the help of the list monad and the `do` notation. As if that weren't enough, Haskell let's you simplify this code even further using list comprehension:

```
triples = [(x, y, z) | z <- [1..]
                     , x <- [1..z]
```

```
    , y <- [x..z]
    , x^2 + y^2 == z^2]
```

This is just further syntactic sugar for the list monad (strictly speaking, `MonadPlus`).

You might see similar constructs in other functional or imperative languages under the guise of generators and coroutines.

21.2.3 Read-Only State

A function that has read-only access to some external state, or environment, can be always replaced by a function that takes that environment as an additional argument. A pure function `(a, e) -> b` (where `e` is the type of the environment) doesn't look, at first sight, like a Kleisli arrow. But as soon as we curry it to `a -> (e -> b)` we recognize the embellishment as our old friend the reader functor:

```
newtype Reader e a = Reader (e -> a)
```

You may interpret a function returning a `Reader` as producing a mini-executable: an action that given an environment produces the desired result. There is a helper function `runReader` to execute such an action:

```
runReader :: Reader e a -> e -> a
runReader (Reader f) e = f e
```

It may produce different results for different values of the environment.

Notice that both the function returning a `Reader`, and the `Reader` action itself are pure.

To implement bind for the `Reader` monad, first notice that you have to produce a function that takes the environment `e` and produces a `b`:

```
ra >>= k = Reader (\e -> ...)
```

Inside the lambda, we can execute the action `ra` to produce an `a`:

```
ra >>= k = Reader (\e -> let a = runReader ra e
                         in ...)
```

We can then pass the `a` to the continuation `k` to get a new action `rb`:

```
ra >>= k = Reader (\e -> let a  = runReader ra e
                             rb = k a
                         in ...)
```

Finally, we can run the action `rb` with the environment `e`:

```
ra >>= k = Reader (\e -> let a = runReader ra e
                             rb = k a
                         in runReader rb e)
```

To implement `return` we create an action that ignores the environment and returns the unchanged value.

Putting it all together, after a few simplifications, we get the following definition:

```
instance Monad (Reader e) where
    ra >>= k = Reader (\e -> runReader (k (runReader ra e)) e)
    return x = Reader (\e -> x)
```

21.2.4 Write-Only State

This is just our initial logging example. The embellishment is given by the `Writer` functor:

```
newtype Writer w a = Writer (a, w)
```

For completeness, there's also a trivial helper `runWriter` that unpacks the data constructor:

```
runWriter :: Writer w a -> (a, w)
runWriter (Writer (a, w)) = (a, w)
```

As we've seen before, in order to make `Writer` composable, `w` has to be a monoid. Here's the monad instance for `Writer` written in terms of the bind operator:

```
instance (Monoid w) => Monad (Writer w) where
    (Writer (a, w)) >>= k = let (a', w') = runWriter (k a)
                            in Writer (a', w `mappend` w')
    return a = Writer (a, mempty)
```

21.2.5 State

Functions that have read/write access to state combine the embellishments of the `Reader` and the `Writer`. You may think of them as pure functions that take the state as an extra argument and produce a pair value/state as a result: `(a, s) -> (b, s)`. After currying, we get them into the form of Kleisli arrows `a -> (s -> (b, s))`, with the embellishment abstracted in the `State` functor:

```
newtype State s a = State (s -> (a, s))
```

Again, we can look at a Kleisli arrow as returning an action, which can be executed using the helper function:

```
runState :: State s a -> s -> (a, s)
runState (State f) s = f s
```

Different initial states may not only produce different results, but also different final states.

The implementation of bind for the `State` monad is very similar to that of the `Reader` monad, except that care has to be taken to pass the correct state at each step:

```
sa >>= k = State (\s -> let (a, s') = runState sa s
                            sb = k a
                        in runState sb s')
```

Here's the full instance:

```
instance Monad (State s) where
    sa >>= k = State (\s -> let (a, s') = runState sa s
                            in runState (k a) s')
    return a = State (\s -> (a, s))
```

There are also two helper Kleisli arrows that may be used to manipulate the state. One of them retrieves the state for inspection:

```
get :: State s s
get = State (\s -> (s, s))
```

and the other replaces it with a completely new state:

```
put :: s -> State s ()
put s' = State (\s -> ((), s'))
```

21.2.6 Exceptions

An imperative function that throws an exception is really a partial function — it's a function that's not defined for some values of its arguments. The simplest implementation of exceptions in terms of pure total functions uses the `Maybe` functor. A partial function is extended to a total function that returns `Just a` whenever it makes sense, and `Nothing` when it doesn't. If we want to also return some information about the cause of the failure, we can use the `Either` functor instead (with the first type fixed, for instance, to `String`).

Here's the `Monad` instance for `Maybe`:

```
instance Monad Maybe where
    Nothing >>= k = Nothing
    Just a  >>= k = k a
    return a = Just a
```

Notice that monadic composition for `Maybe` correctly short-circuits the computation (the continuation `k` is never called) when an error is detected. That's the behavior we expect from exceptions.

21.2.7 Continuations

It's the "Don't call us, we'll call you!" situation you may experience after a job interview. Instead of getting a direct answer, you are supposed to provide a handler, a function to be called with the result. This style of programming is especially useful when the result is not known at the time of the call because, for instance, it's being evaluated by another thread or delivered from a remote web site. A Kleisli arrow in this case returns a function that accepts a handler, which represents "the rest of the computation":

```
data Cont r a = Cont ((a -> r) -> r)
```

The handler `a -> r`, when it's eventually called, produces the result of type `r`, and this result is returned at the end. A continuation is parameterized by the result type. (In practice, this is often some kind of status indicator.)

There is also a helper function for executing the action returned by the Kleisli arrow. It takes the handler and passes it to the continuation:

```
runCont :: Cont r a -> (a -> r) -> r
runCont (Cont k) h = k h
```

The composition of continuations is notoriously difficult, so its handling through a monad and, in particular, the `do` notation, is of extreme advantage.

Let's figure out the implementation of bind. First let's look at the stripped down signature:

```
(>>=) :: ((a -> r) -> r) ->
  (a -> (b -> r) -> r) ->
  ((b -> r) -> r)
```

Our goal is to create a function that takes the handler `(b -> r)` and produces the result `r`. So that's our starting point:

```
ka >>= kab = Cont (\hb -> ...)
```

Inside the lambda, we want to call the function `ka` with the appropriate handler that represents the rest of the computation. We'll implement this handler as a lambda:

```
runCont ka (\a -> ...)
```

In this case, the rest of the computation involves first calling **kab** with **a**, and then passing **hb** to the resulting action **kb**:

```
runCont ka (\a -> let kb = kab a
                  in runCont kb hb)
```

As you can see, continuations are composed inside out. The final handler **hb** is called from the innermost layer of the computation. Here's the full instance:

```
instance Monad (Cont r) where
  ka >>= kab = Cont (\hb -> runCont ka (\a -> runCont (kab a) hb))
  return a = Cont (\ha -> ha a)
```

21.2.8 Interactive Input

This is the trickiest problem and a source of a lot of confusion. Clearly, a function like **getChar**, if it were to return a character typed at the keyboard, couldn't be pure. But what if it returned the character inside a container? As long as there was no way of extracting the character from this container, we could claim that the function is pure. Every time you call **getChar** it would return exactly the same container. Conceptually, this container would contain the superposition of all possible characters.

If you're familiar with quantum mechanics, you should have no problem understanding this analogy. It's just like the box with the Schrödinger's cat inside — except that there is no way to open or peek inside the box. The box is defined using the special built-in **IO** functor. In our example, **getChar** could be declared as a Kleisli arrow:

```
getChar :: () -> IO Char
```

(Actually, since a function from the unit type is equivalent to picking a value of the return type, the declaration of `getChar` is simplified to `getChar :: IO Char`.)

Being a functor, `IO` lets you manipulate its contents using `fmap`. And, as a functor, it can store the contents of any type, not just a character. The real utility of this approach comes to light when you consider that, in Haskell, `IO` is a monad. It means that you are able to compose Kleisli arrows that produce `IO` objects.

You might think that Kleisli composition would allow you to peek at the contents of the `IO` object (thus "collapsing the wave function," if we were to continue the quantum analogy). Indeed, you could compose `getChar` with another Kleisli arrow that takes a character and, say, converts it to an integer. The catch is that this second Kleisli arrow could only return this integer as an `(IO Int)`. Again, you'll end up with a superposition of all possible integers. And so on. The Schrödinger's cat is never out of the bag. Once you are inside the `IO` monad, there is no way out of it. There is no equivalent of `runState` or `runReader` for the `IO` monad. There is no `runIO`!

So what can you do with the result of a Kleisli arrow, the `IO` object, other than compose it with another Kleisli arrow? Well, you can return it from `main`. In Haskell, `main` has the signature:

```
main :: IO ()
```

and you are free to think of it as a Kleisli arrow:

```
main :: () -> IO ()
```

From that perspective, a Haskell program is just one big Kleisli arrow in the `IO` monad. You can compose it from smaller Kleisli arrows using monadic composition. It's up to the runtime system to do something with the resulting `IO` object (also called `IO` action).

Notice that the arrow itself is a pure function — it's pure functions all the way down. The dirty work is relegated to the system. When it finally executes the `IO` action returned from `main`, it does all kinds of nasty things like reading user input, modifying files, printing obnoxious messages, formatting a disk, and so on. The Haskell program never dirties its hands (well, except when it calls `unsafePerformIO`, but that's a different story).

Of course, because Haskell is lazy, `main` returns almost immediately, and the dirty work begins right away. It's during the execution of the `IO` action that the results of pure computations are requested and evaluated on demand. So, in reality, the execution of a program is an interleaving of pure (Haskell) and dirty (system) code.

There is an alternative interpretation of the `IO` monad that is even more bizarre but makes perfect sense as a mathematical model. It treats the whole Universe as an object in a program. Notice that, conceptually, the imperative model treats the Universe as an external global object, so procedures that perform I/O have side effects by virtue of interacting with that object. They can both read and modify the state of the Universe.

We already know how to deal with state in functional programming — we use the state monad. Unlike simple state, however, the state of the Universe cannot be easily described using standard data structures. But we don't have to, as long as we never directly interact with it. It's

enough that we assume that there exists a type `RealWorld` and, by some miracle of cosmic engineering, the runtime is able to provide an object of this type. An `IO` action is just a function:

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

Or, in terms of the `State` monad:

```
type IO = State RealWorld
```

However, `>=>` and `return` for the `IO` monad have to be built into the language.

21.2.9 Interactive Output

The same `IO` monad is used to encapsulate interactive output. `RealWorld` is supposed to contain all output devices. You might wonder why we can't just call output functions from Haskell and pretend that they do nothing. For instance, why do we have:

```
putStr :: String -> IO ()
```

rather than the simpler:

```
putStr :: String -> ()
```

Two reasons: Haskell is lazy, so it would never call a function whose output — here, the unit object — is not used for anything. And, even if it weren't lazy, it could still freely change the order of such calls and thus garble the output. The only way to force sequential execution of two

functions in Haskell is through data dependency. The input of one function must depend on the output of another. Having `RealWorld` passed between `IO` actions enforces sequencing.

Conceptually, in this program:

```
main :: IO ()
main = do
    putStr "Hello "
    putStr "World!"
```

the action that prints "World!" receives, as input, the Universe in which "Hello " is already on the screen. It outputs a new Universe, with "Hello World!" on the screen.

21.3 Conclusion

Of course I have just scratched the surface of monadic programming. Monads not only accomplish, with pure functions, what normally is done with side effects in imperative programming, but they also do it with a high degree of control and type safety. They are not without drawbacks, though. The major complaint about monads is that they don't easily compose with each other. Granted, you can combine most of the basic monads using the monad transformer library. It's relatively easy to create a monad stack that combines, say, state with exceptions, but there is no formula for stacking arbitrary monads together.