20

Monads: Programmer's Definition

Programmers have developed a whole mythology around monads. It's supposed to be one of the most abstract and difficult concepts in programming. There are people who "get it" and those who don't. For many, the moment when they understand the concept of the monad is like a mystical experience. The monad abstracts the essence of so many diverse constructions that we simply don't have a good analogy for it in everyday life. We are reduced to groping in the dark, like those blind men touching different parts of the elephant end exclaiming triumphantly: "It's a rope," "It's a tree trunk," or "It's a burrito!"

Let me set the record straight: The whole mysticism around the monad is the result of a misunderstanding. The monad is a very simple concept. It's the diversity of applications of the monad that causes the confusion.

As part of research for this post I looked up duct tape (a.k.a., duck tape) and its applications. Here's a little sample of things that you can do with it:

- sealing ducts
- fixing CO_2 scrubbers on board Apollo 13
- wart treatment
- fixing Apple's iPhone 4 dropped call issue
- making a prom dress
- building a suspension bridge

Now imagine that you didn't know what duct tape was and you were trying to figure it out based on this list. Good luck!

So I'd like to add one more item to the collection of "the monad is like..." clichés: The monad is like duct tape. Its applications are widely diverse, but its principle is very simple: it glues things together. More precisely, it composes things.

This partially explains the difficulties a lot of programmers, especially those coming from the imperative background, have with understanding the monad. The problem is that we are not used to thinking of programming in terms of function composition. This is understandable. We often give names to intermediate values rather than pass them directly from function to function. We also inline short segments of glue code rather than abstract them into helper functions. Here's an imperative-style implementation of the vector-length function in C:

```
double vlen(double * v) {
    double d = 0.0;
    int n;
    for (n = 0; n < 3; ++n)
        d += v[n] * v[n];
    return sqrt(d);
}
```

Compare this with the (stylized) Haskell version that makes function composition explicit:

```
vlen = sqrt . sum . fmap (flip (^) 2)
```

(Here, to make things even more cryptic, I partially applied the exponentiation operator `(^)` by setting its second argument to `2`.)

I'm not arguing that Haskell's point-free style is always better, just that function composition is at the bottom of everything we do in programming. And even though we are effectively composing functions, Haskell does go to great lengths to provide imperative-style syntax called the `do` notation for monadic composition. We'll see its use later. But first, let me explain why we need monadic composition in the first place.

20.1 The Kleisli Category

We have previously arrived at the writer monad by embellishing regular functions. The particular embellishment was done by pairing their return values with strings or, more generally, with elements of a monoid. We can now recognize that such an embellishment is a functor:

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

instance Functor (Writer w) where
    fmap f (Writer (a, w)) = Writer (f a, w)
```

We have subsequently found a way of composing embellished functions, or Kleisli arrows, which are functions of the form:

```
a -> Writer w b
```

It was inside the composition that we implemented the accumulation of the log.

We are now ready for a more general definition of the Kleisli category. We start with a category **C** and an endofunctor m. The corresponding Kleisli category **K** has the same objects as **C**, but its morphisms are different. A morphism between two objects a and b in **K** is implemented as a morphism:

$$a \to m\ b$$

in the original category **C**. It's important to keep in mind that we treat a Kleisli arrow in **K** as a morphism between a and b, and not between a and $m\ b$.

In our example, m was specialized to `Writer w`, for some fixed monoid `w`.

Kleisli arrows form a category only if we can define proper composition for them. If there is a composition, which is associative and has an identity arrow for every object, then the functor m is called a *monad*, and the resulting category is called the Kleisli category.

In Haskell, Kleisli composition is defined using the fish operator `>=>`, and the identity arrow is a polymorphic function called `return`. Here's the definition of a monad using Kleisli composition:

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

Keep in mind that there are many equivalent ways of defining a monad, and that this is not the primary one in the Haskell ecosystem. I like it for its conceptual simplicity and the intuition it provides, but there are other definitions that are more convenient when programming. We'll talk about them momentarily.

In this formulation, monad laws are very easy to express. They cannot be enforced in Haskell, but they can be used for equational rea-

soning. They are simply the standard composition laws for the Kleisli category:

```
(f >=> g) >=> h = f >=> (g >=> h) -- associativity
return >=> f = f                 -- left unit
f >=> return = f                 -- right unit
```

This kind of a definition also expresses what a monad really is: it's a way of composing embellished functions. It's not about side effects or state. It's about composition. As we'll see later, embellished functions may be used to express a variety of effects or state, but that's not what the monad is for. The monad is the sticky duct tape that ties one end of an embellished function to the other end of an embellished function.

Going back to our `Writer` example: The logging functions (the Kleisli arrows for the `Writer` functor) form a category because `Writer` is a monad:

```
instance Monoid w => Monad (Writer w) where
    f >=> g = \a ->
        let Writer (b, s) = f a
            Writer (c, s') = g b
        in Writer (c, s `mappend` s')
    return a = Writer (a, mempty)
```

Monad laws for `Writer w` are satisfied as long as monoid laws for `w` are satisfied (they can't be enforced in Haskell either).

There's a useful Kleisli arrow defined for the `Writer` monad called `tell`. It's sole purpose is to add its argument to the log:

```
tell :: w -> Writer w ()
tell s = Writer ((), s)
```

We'll use it later as a building block for other monadic functions.

20.2 Fish Anatomy

When implementing the fish operator for different monads you quickly realize that a lot of code is repeated and can be easily factored out. To begin with, the Kleisli composition of two functions must return a function, so its implementation may as well start with a lambda taking an argument of type `a`:

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g = \a -> ...
```

The only thing we can do with this argument is to pass it to `f`:

```
f >=> g = \a -> let mb = f a
                in ...
```

At this point we have to produce the result of type `m c`, having at our disposal an object of type `m b` and a function `g :: b -> m c`. Let's define a function that does that for us. This function is called *bind* and is usually written in the form of an infix operator:

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

For every monad, instead of defining the fish operator, we may instead define bind. In fact the standard Haskell definition of a monad uses bind:

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

Here's the definition of bind for the `Writer` monad:

```
(Writer (a, w)) >>= f = let Writer (b, w') = f a
                        in Writer (b, w `mappend` w')
```

It is indeed shorter than the definition of the fish operator.

It's possible to further dissect bind, taking advantage of the fact that `m` is a functor. We can use `fmap` to apply the function `a -> m b` to the contents of `m a`. This will turn `a` into `m b`. The result of the application is therefore of type `m (m b)`. This is not exactly what we want — we need the result of type `m b` — but we're close. All we need is a function that collapses or flattens the double application of `m`. Such a function is called `join`:

```
join :: m (m a) -> m a
```

Using `join`, we can rewrite bind as:

```
ma >>= f = join (fmap f ma)
```

That leads us to the third option for defining a monad:

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

Here we have explicitly requested that `m` be a `Functor`. We didn't have to do that in the previous two definitions of the monad. That's because any type constructor `m` that either supports the fish or bind operator is automatically a functor. For instance, it's possible to define `fmap` in terms of bind and `return`:

```
fmap f ma = ma >>= \a -> return (f a)
```

For completeness, here's **join** for the **Writer** monad:

```
join :: Monoid w => Writer w (Writer w a) -> Writer w a
join (Writer ((Writer (a, w')), w)) = Writer (a, w `mappend` w')
```

20.3 The do Notation

One way of writing code using monads is to work with Kleisli arrows — composing them using the fish operator. This mode of programming is the generalization of the point-free style. Point-free code is compact and often quite elegant. In general, though, it can be hard to understand, bordering on cryptic. That's why most programmers prefer to give names to function arguments and intermediate values.

When dealing with monads it means favoring the bind operator over the fish operator. Bind takes a monadic value and returns a monadic value. The programmer may choose to give names to those values. But that's hardly an improvement. What we really want is to pretend that we are dealing with regular values, not the monadic containers that encapsulate them. That's how imperative code works — side effects, such as updating a global log, are mostly hidden from view. And that's what the **do** notation emulates in Haskell.

You might be wondering then, why use monads at all? If we want to make side effects invisible, why not stick to an imperative language? The answer is that the monad gives us much better control over side effects. For instance, the log in the **Writer** monad is passed from function to function and is never exposed globally. There is no possibility of

garbling the log or creating a data race. Also, monadic code is clearly demarcated and cordoned off from the rest of the program.

The `do` notation is just syntactic sugar for monadic composition. On the surface, it looks a lot like imperative code, but it translates directly to a sequence of binds and lambda expressions.

For instance, take the example we used previously to illustrate the composition of Kleisli arrows in the `Writer` monad. Using our current definitions, it could be rewritten as:

```
process :: String -> Writer String [String]
process = upCase >=> toWords
```

This function turns all characters in the input string to upper case and splits it into words, all the while producing a log of its actions.

In the `do` notation it would look like this:

```
process s = do
    upStr <- upCase s
    toWords upStr
```

Here, `upStr` is just a `String`, even though `upCase` produces a `Writer`:

```
upCase :: String -> Writer String String
upCase s = Writer (map toUpper s, "upCase ")
```

This is because the `do` block is desugared by the compiler to:

```
process s =
    upCase s >>= \upStr ->
        toWords upStr
```

The monadic result of `upCase` is bound to a lambda that takes a `String`. It's the name of this string that shows up in the `do` block. When reading the line:

```
upStr <- upCase s
```

we say that **upStr** *gets* the result of **upCase s**.

The pseudo-imperative style is even more pronounced when we inline **toWords**. We replace it with the call to **tell**, which logs the string **"toWords "**, followed by the call to **return** with the result of splitting the string **upStr** using **words**. Notice that **words** is a regular function working on strings.

```
process s = do
    upStr <- upCase s
    tell "toWords "
    return (words upStr)
```

Here, each line in the do block introduces a new nested bind in the desugared code:

```
process s =
    upCase s >>= \upStr ->
      tell "toWords " >>= \() ->
        return (words upStr)
```

Notice that **tell** produces a unit value, so it doesn't have to be passed to the following lambda. Ignoring the contents of a monadic result (but not its effect — here, the contribution to the log) is quite common, so there is a special operator to replace bind in that case:

```
(>>) :: m a -> m b -> m b
m >> k = m >>= (\_ -> k)
```

The actual desugaring of our code looks like this:

```
process s =
  upCase s >>= \upStr ->
    tell "toWords " >>
      return (words upStr)
```

In general, **do** blocks consist of lines (or sub-blocks) that either use the left arrow to introduce new names that are then available in the rest of the code, or are executed purely for side-effects. Bind operators are implicit between the lines of code. Incidentally, it is possible, in Haskell, to replace the formatting in the **do** blocks with braces and semicolons. This provides the justification for describing the monad as a way of overloading the semicolon.

Notice that the nesting of lambdas and bind operators when desugaring the **do** notation has the effect of influencing the execution of the rest of the **do** block based on the result of each line. This property can be used to introduce complex control structures, for instance to simulate exceptions.

Interestingly, the equivalent of the **do** notation has found its application in imperative languages, C++ in particular. I'm talking about resumable functions or coroutines. It's not a secret that C++ futures form a monad[1]. It's an example of the continuation monad, which we'll discuss shortly. The problem with continuations is that they are very hard to compose. In Haskell, we use the **do** notation to turn the spaghetti of "my handler will call your handler" into something that looks very much like sequential code. Resumable functions make the same transformation possible in C++. And the same mechanism can be applied to turn the spaghetti of nested loops[2] into list comprehensions or "gener-

[1] https://bartoszmilewski.com/2014/02/26/c17-i-see-a-monad-in-your-future/

[2] https://bartoszmilewski.com/2014/04/21/getting-lazy-with-c/

ators," which are essentially the `do` notation for the list monad. Without the unifying abstraction of the monad, each of these problems is typically addressed by providing custom extensions to the language. In Haskell, this is all dealt with through libraries.