Monads

Table of Contents

Notes: Computerphile Monads

Consider a data constructor for an expression which captures integer division:

```
Data Expr = Val Int | Div Expr Expr
```

Let's write a function that can evaluate these expressions:

```
eval :: Expr -> Int
eval (Val n) = n
eval (Div x y) = (eval x) `div` (eval y)
```

But this is unsafe: if you attempt division by 0 you'll get an error. So let's define a safe division operation

```
safediv :: Int -> Int -> Maybe Int
safediv n m = if m == 0
                then Nothing
                else Just (n `div` m)
```

Now we can rewrite `eval` to be safe:

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = case eval x of
                    Nothing -> Nothing
                    Just n -> case eval y of
                                Nothing -> Nothing
                                Just m -> safediv n m
```

Now we have a program that will work safely. But it's pretty ugly and verbose. How can we make it better, and look more like the original code, while still being safe?

First observe that there's a common pattern here: 2 case analyses, doing the same thing. Let's abstract this out, introducing `m, f`:

```
case m of
    Nothing -> Nothing
    Just x -> f x
```

And let's give a name to this `m >== f`:

```
m >== f = case m of
            Nothing = Nothing
```

```
3               Just m -> f m
```

With this definition, let's rewrite `eval`:

```
1 eval :: Expr -> Maybe Int
2 eval (Val n) = return n
3 eval (Div x y) = eval x >>= (\n ->
4                  eval y >>= (\m ->
5                  safediv n m))
```

This is equivalent to the last definition of `eval`, but we've abstracted away all the case analyses. But we can still do better, with the syntactic sugar of the **`do`** notation, which gives a helpful shorthand for programs of this sort:

```
1 eval :: Expr -> Maybe Int
2 eval (Val n) = return n
3 eval (Div x y) = do n <- eval x
4                     m <- eval y
5                     safediv n m
```

This is much nicer. All the failure management is handled automatically.

Where do the monads come in?

So what does all this have to do with monads? Effectively we have rediscovered the `Maybe` monad, which comprises 3 things: the `Maybe` type constructor, and 2 functions:

- **`return`** `:: a -> Maybe`: a bridge between the pure and the impure
- `>>= :: Maybe a -> (a -> Maybe b)-> Maybe b`: sequencing

That's all a monad is:

1. Type constructor
2. **`return`** definition
3. >>= definition

What's the point?

1. The same idea works for other effects: I/O, mutable state, non-determinism, … Monads give a uniform framework for thinking about programming with effects.
2. Supports pure programming with effects: i.e. gives you a way to do impure things in a pure language

3. Use of effects is explicit in types: evaluator function here takes an `Expr` and returns a `Maybe Int`. You explicitly state what effects may be produced.
4. Provides ability to write functions that work for any effect, **effect polymorphism**. Haskell has libraries of generic effect functions.