FIT2102 PASS - Week 10

Nicholas Cheng

October 19, 2020

What is a Monad?

Monads are applicative functors.

Recall Functors and Applicatives

We have seen that **Applicatives** share a relationship with **Functors**.
The most notable thing is that you can define *fmap* in terms of *apply*.

Still not getting it

If we say that Applicative is a subclass of Functor, then a Monad is a subclass of Applicative.
In other words, *Functor* $\rightarrow$ *Applicative* $\rightarrow$ *Monad*.

Just show me the typeclass!

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

Monads have three core operations. Of these three, only the first operation is *mandatory*. It's called **bind**.
Notice that *return* looks familiar. What does it resemble?

Does bind look familiar to you?

```
fmap  :: Functor f => (a -> b) -> f a -> f b
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
(>>=) :: Monad f => f a -> (a -> f b) -> f b
```

Here's an example

```
Prelude> let andOne x = [x, 1]
Prelude> andOne 10 [10,1]
Prelude> :t fmap andOne [4, 5, 6]
fmap andOne [4, 5, 6] :: Num t => [[t]]
Prelude> fmap andOne [4, 5, 6]
[[4,1],[5,1],[6,1]]
```

Notice that another level of nested context is added. Suppose we wanted to remove that level of nesting, what do we do?

concat

```haskell
concat :: Foldable t => t [a] -> [a]
```

In English, *concat* removes a level of nesting.

Monad is a generalization of concat

```haskell
concat :: Foldable t => t [a] -> [a]
join   :: Monad m => m (m a) -> m a
```

Exercise: Define bind

```
-- keep in mind this is (>>=) flipped
bind :: Monad m => (a -> m b) -> m a -> m b
bind = undefined
```

Define it in terms of *join* and *fmap*.

A Monad is NOT

Impure Monadic functions are pure functions.

A way to do imperative programming in Haskell There are monads where order doesn't matter.

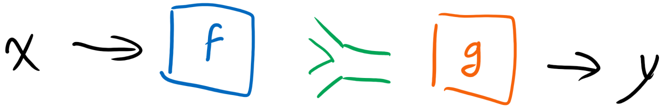
A value It's a typeclass. It's more about the operations and relationships between elements in a domain.

About strictness *bind* and *return* are nonstrict.

Extra knowledge. Doubt this is needed for your unit.

Do syntax

It’s syntactic sugar for sequencing operations. Think of it as an operator that allows you to take inputs from one function and feed it to another function.
That’s a mouthful, here’s crudely drawn illustration.



Here's something in do syntax

```haskell
bindingAndSequencing :: IO ()
bindingAndSequencing = do
  putStrLn "name pls:"
  name <- getLine
  putStrLn ("y helo thar: " ++ name)
```

Same thing, without the sugar

```haskell
bindingAndSequencing' :: IO ()
bindingAndSequencing' =
  putStrLn "name␣pls:" >>
  getLine >>=
  \name -> putStrLn ("y␣helo␣thar:␣" ++ name)
```

References

Chapter 18, Haskell Programming From First Principles.
I trust you'll know how to get your hands on a copy.