Monad: A Monoid in the Class of Endofunctors

—

Ajax Peterson

The plan

- Briefly introduce Haskell for those who haven't seen it (the details aren't SUPER relevant)
- Develop an abstract idea for what a "monad" is (if we can even do that in a succinct manner)
- Get everyone in the room to understand a few monadic types, how to use them, and reasoning behind them
- We're all doing this together, seriously...

Haskell: Quickly

- Haskell is a purely functional programming language
  - Functional: primarily using functions to operate on data
  - Pure: functions are real functions
    - No side effects
    - Return value should be the same value for the same inputs
- Unlike many other programming languages
  - No for/while loops → favor recursion and higher-order functions
  - Function definitions are (usually) single expressions
  - Utilities are typically abstracted to be as broadly applicable as possible
    - Partially a result of Haskell's lack of overloading
  - We care A LOT about types
- I'm using Haskell as a vehicle to talk about monads, monads are not exclusive

Programming and Objects

- In most programming languages, we operate on objects:
  - `5`
  - `“ripley”`
  - `True`
  - `‘a’`

Programming and Objects

- In almost all languages, these objects have a type:
  - `5 :: Int`
  - `“ripley” :: String`
  - `True :: Bool`
  - `‘a’ :: Char`
- In Haskell, these are determined and checked at compile-time, which is useful for when we want to combine objects:
  - `5 + ‘a’ :: ? -- can the (+) function be applied to 5 and ‘a’?`
- As we continue forward, keep trying to figure out the types of the expression I’m showing

Combining Objects

- We can “combine”/transform objects in Haskell using functions:
  - ```
    f :: Int -> Int -> Int
    f x y = x + y
    ```
  - `f 5 3 :: Int → 8`
- Importantly, every operator in Haskell should be thought of as a function, so:
  - `(+) :: Int -> Int -> Int`
- This defines the plus operator as an infix function to be used like so:
  - `5 + 3 :: Int`
- This means I can still use them just like any other function:
  - `(+) 5 3 :: Int → 8`
- I can also define functions without a name (an anonymous function/lambda):
  - `(\x y -> x + y) :: Int -> Int -> Int`
    ```

Some type questions...

- Say I have this code:
  - ```
    f :: Int -> Int -> Int
    f x y = x * y + y
    ```
  - ```
    g :: Double -> Double
    g x = x + 3.0
    ```
- What are the types of these expressions?
  - `g 5.0 :: ?`
  - `g :: ?`
  - `f :: ?`
  - `f 7 5 :: ?`
  - `f 7 :: ?`

Some type questions...

- Say I have this code:
  - ```
    f :: Int -> Int -> Int
    f x y = x * y + y
    ```
  - ```
    g :: Double -> Double
    g x = x + 3.0
    ```
- What are the types of these expressions?
  - `g 5.0 :: Double`
  - `g :: Double -> Double`
  - `f :: Int -> Int -> Int`
  - `f 7 5 :: Int`
  - `f 7 :: Int -> Int`

Polymorphism

- Poly = “many”
- Morph = “form”
- Functions in Haskell exhibit polymorphism, i.e., they can take **many forms** depending on the input they are given:
  - `id :: forall a. a -> a`
    `id x = x`
  - `id “connor #1 java fan” :: String → “connor #1 java fan”`
    `id 5 :: Int → 5`
  - `(+) :: forall a. Num a => a -> a -> a`
    “type `a` MUST be a `Num` to be chill with this function”
  - `5 + 3 :: Int`
  - `“a” + “b” -- fails compile-time type checking b/c String is not a Num`

Containers

- Many languages provide constructs for 0 or more values within them
- This I will call a “container”
  - In Haskell, we typically denote a type like this as `c a`
    - `c` is a container type that contains objects of type `a`
  - Notably, lists are also container types, but they are denoted `[a]`
- What are some other container types we know?

Containers: Lists

- We can define a list in Haskell like so:
  - `[5, 3, 6] :: [Int]`
  - `[‘a’, ‘b’, ‘c’] :: [Char] -- this is equivalent to “abc”`
  - `[True, True, False, False] :: [Bool]`

Containers: Lists: Map

- Sometimes I want to apply a function to every member of a list:
  - `[5, 3, 9] → square → [25, 9, 81]`
- I have a function to do this in Haskell:
  - `map square [5, 3, 9] :: [Int] → [25, 9, 81]`
- This is our first instance of a “higher-order function” in Haskell
  - A function that takes another function as input

Let’s write `map`!

- First, what’s the type?

Let’s write `map`!

- First, what’s the type?
  - `map :: forall a b. (a -> b) -> [a] -> [b]`

Let’s write `map`!

- First, what’s the type?
  - `map :: forall a b. (a -> b) -> [a] -> [b]`
- Second, how do we define it?

Let's write `map`!

- First, what's the type?
  - `map :: forall a b. (a -> b) -> [a] -> [b]`
- Second, how do we define it?
  - ```
    map _ [] = []
    map f (x:xs) = f x : map f xs
    ```
    ```

The Haskell Process

1. Write code that works
2. Look for ways to abstract components out

What can we abstract out of this type?

- `map :: forall a b. (a -> b) -> [a] -> [b]`

Very subtle...

- `fmap :: forall a b. (a -> b) -> m a -> m b`
- Now we can write mapping operations for ANY container type!

Another implementation for `map`

- If we think of a dictionary as a collection of 2-tuples, we could define `fmap` for dictionaries like so:
  - `{“a”: 3, “b”: 4}`
  - `→ {(“a”, 3), (“b”, 4)}`
  - `→ square`
  - `→ {(“a”, 9), (“b”, 16)}`
  - `→ {“a”: 9, “b”: 16}`
- For this one, our “map” over the dictionary is applying the function to every value

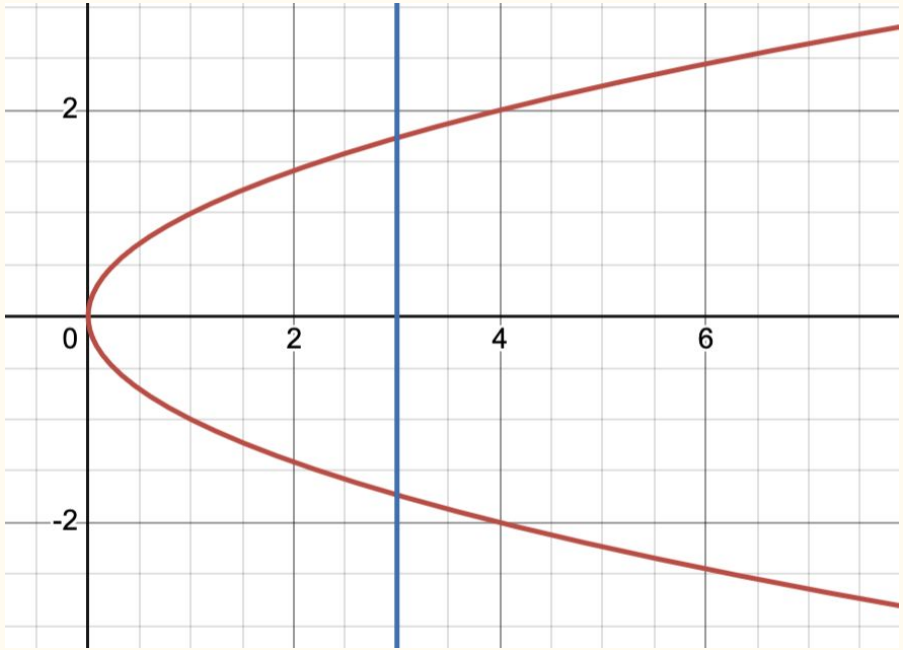
Containers but... more specific?

- What a cool function we've just generalized to find!
- Types that have a mapping operation `fmap` in Haskell are called `Functor`s:
- ```
  class Functor m where
      fmap :: (a -> b) -> m a -> m b
  ```
  ```

An issue in Haskell

- Functions are cool, right?
- Like I can do this:
  - `f 5 3`
- And then get a result!
- BUT...

🚨🚨🚨🚨🚨🚨🚨🚨🚨🚨BAD!!!!!



Coming back down to reality...

- Haskell has a function called `getLine` that reads input from the user
- What does that function evaluate to when invoked?

Coming back down to reality...

- Haskell has a function called `getLine` that reads input from the user
- What does that function evaluate to when invoked?
  - This is literally the vertical line test failing

Haskell functions are real functions

- When you evaluate the result of a function in Haskell, it must produce the same value every time
- In particular, this causes I/O to be a difficult problem to solve in Haskell
  - My `getLine` function should give me back a different string depending on what the user typed

Thinking back to what we were doing earlier...

- I can apply functions to a container using `fmap`:
  - `fmap square [5] → [25]`
- This function can be an anonymous function too if I wanted:
  - `fmap (\x -> x ^ 2) [5] → [25]`
- Now let's take this approach to thinking about what `fmap` is doing:
  - Look at every value in the container (don't get too caught up in the "every" part here)
  - Bind each value in the container to the name `x` in the lambda
  - Put the value calculated in the lambda back in the container
- Important takeaway: we are preserving the *structure* of the container throughout the operation
  - It was a list going in, it's a list coming out
  - I'm not directly applying my square function to the value in the list, the lambda is handling that transformation

Our little cheater

- Thinking back to I/O, my problem was that I can't apply transformations after the I/O operation because the function can't give back different values:
  - `putStrLn (getLine ++ " is cool") -- does not work`
- But we saw in the previous example that I can indirectly get the value in the list through the lambda!
- So let's do that:
  - `fmap (\line -> putStrLn line) getLine`

Well…

- So let's do that:
  - `fmap (\line -> putStrLn line) getLine`
- Well, I still haven't told you what type of `getLine` is, but let's make one up that works with how we're using it

Well...

- So let's do that:
  - `fmap (\line -> putStrLn line) getLine`
- Well, I still haven't told you what type of `getLine` is, but let's make one up that works with how we're using it:
  - `getLine :: [String]`
- Our `fmap` takes care of getting the value in the list without touching it directly!
- But it's kind of excessive to use a list type here. We're only getting back one `String` anyway... so let's make a more descriptive type!
  - `getLine :: IO String`
- And we ignore the actual definition of `fmap` for this type because it's evil... 😀

More

- So now our code's pretty good to do what we want it to do:
  - `fmap (\line -> putStrLn line) getLine`
- ...except what is the type of the lambda?
  - `String -> ?`

More

- So now our code's pretty good to do what we want it to do:
  - `fmap (\line -> putStrLn line) getLine`
- ...except what is the type of the lambda?
  - `String -> ?`
- Well, it should probably return something in the `IO` container type we just made because `putStrLn` is an I/O operation
  - Let's say it's `String -> IO ()` and that `putStrLn :: IO ()`
- But...

More problems jeez

- Recall our type definition for `fmap`:
  - `fmap :: Functor m => (a -> b) -> m a -> m b`
- The return type of the function we give to `fmap` isn't wrapped in the container type `m`!
- Soooooo... let's change that:
  - `? :: ? m => (a -> m b) -> m a -> m b`
- Let's check our example to make sure this works how we want now:
  - `? (\line -> putStrLn line) getLine`
- Side note: if we kept the original type signature for `fmap`, then we would just get a double wrapped value out of the other side, or `IO (IO ())`

The Monad: Invented

- The mystery function we made is called `bind`, or often invoked using its operator equivalent `>>=`, which is one of two defining functions for the monad:
  - ```
    class Monad m where
        (>>=) :: m a -> (a -> m b) -> m b
    ```
  - Note that the operands are flipped! The function comes second for clearer semantics with the operator variant
- The second function is called `return`, which allows us to wrap values computed in the passed-in function of `bind`
  - `return :: Monad m => a -> m a`
  - Sometimes the operation we're performing doesn't give us back a wrapped value, so we gotta wrap it ourselves
    ```

Revisiting Our Example

- Back to our example, now using the bind operator:
  - `getLine >>= (\line -> putStrLn line)`
- Let's make some semantic sense of this line:
  - `getLine` → "get a line of input from the user"
  - `>>=` → "and then"
  - `\line` → "bind the line of input we got to the name `line`"
  - `putStrLn line` → "print the line we got back out to the screen"

More abstractly now...

- That's all cool, but still... what's a monad?
- A monad is any type that:
  - Contains 0 or more values (i.e., is a container type)
  - Can have computations applied to those values **and you can retrieve the computation's results**
    - This is related to the function passed to bind returning a wrapped value!
  - Preserves the structure of the type across computations
    - The final return value of bind is a wrapped value
- Or more casually for I/O...
  - Monads allow you to abuse a mapping operation to access an impure return value

Context-Dependent Definitions

- The following are monadic types and, as such, can have bind applied to them. Included is the relevant name for the bind operator for that type
- `IO a` – “and then” (sequencing I/O operations)
  - `getLine >>= putStrLn`
  - “read a line **and then** print it”
- `[a]` – “flat map” (unconventional to use the `>>=` operator here)
  - `[1, 3, 5] >>= (\x -> [x, x + 1]) → [1, 2, 3, 4, 5, 6]`
  - “for each value `x`, calculate `x` and its successor and put those values where `x` used to be”
- `Maybe a` – “and then” (propagading error/null values)
  - `divMaybe 10 5 >>= (\x -> return (div x 5)) → Just 0`
  - “try dividing 10 by 5 and then, if the division succeeded, divide that value by 5”

Related: Applicative Functors

- What if I have some function like:
  - ```
    f :: Int -> Int -> Int -> Int
    f x y z = x + y + z
    ```
- How could I apply this function to a bunch of wrapped values that I have?
  - `let a = Just 5`
  - `let b = Just 3`
  - `let c = Just 6`
- If I try to use `fmap`, I can apply it to the first value:
  - `fmap f a :: Maybe (Int -> Int -> Int)`
- But that type kinda sucks (I end up wrapping the resulting function 😭 😭)
    ```

Related: Applicative Functors

- Luckily, the `Maybe` type is also an `Applicative`, which has the following function defined on it:
  - `(<*>) :: Applicative m => m (a -> b) -> m a -> m b`
  - ^ omg look it's a wrapped function!
- So now I can write my code like this:
  - `f <$> a <*> b <*> c :: Maybe Int`
  - "unwrap a and apply f to a, unwrap the resulting function and b and apply that function to b, unwrap the next resulting function and c and apply that function to c"

Related: Kinds

- Kinds are a way of categorizing type constructors in Haskell
  - Except there's really only one which is then derived...
- `*` – the kind of non-parameterized (nullary) type constructors
  - `Int, Double, String, Char, ()`
- `* -> *` – the kind of unary-parameterized type constructors
  - `Maybe a, IO a, [a]`
  - I've been calling these "container types" because most store values of the parameterized type
- `* -> ... -> *` – the kind of n-ary-parameterized type constructors
  - `Either a b` – binary-parameterized type constructor (`* -> * -> *`)
- `#` – "unlifted" types
  - Meant for types don't allow the "bottom" value (expressions always computable)

Please ask me questions...