You could have invented monads

Roeland Matthijssens

26 september 2017

Functional Programming

Why?

See Jeffrey's chalk talk

Concepts

- No state change
- No side effects
- Function composition is crucial
- What about IO?

Demystification

- A monad is not a specific feature.
- They are simple data structures that happen to implement the monadic interface
- Being a monad is simply adhering to a certain(monadic) interface (bind, unit)

Monads

Disclaimer

No guarantees.

Slightly contrived example

Something you may want to do

- Add debugging statements to function calls
- Remember: no destructive changes (adding to a global log list)

Intuitive implementation

```
f x -> (x + 1, 'f was called')
g x -> (x + 1, 'g was called')
```

f and *g* both take a *float* and return a tuple (*float*, *string*)

How to use this in composition $f \cdot g$

```
y, logString_g = g(x)
z, logString_f = f(y)
log = logString_g + logString_f
return (z, log)
```

It can be done, but it is ugly and long winded if *f* and *g* simply took a *float* and returned a *float* we could write $f \cdot g$

```
f x -> x+1
g x -> x+2
f . g 3 -> 6
```

But they take a *float* and output a (*float*, *string*). Simply composing them would not work

```
f x -> (x+1, "f called.")
g x -> (x+2, "g called.")
f . g 3 ->
 Error: f expects a float but received a tuple
```

Lets redefine *f* and *g*

```
f' x log -> (x+1, log + "f called.")
g' x log -> (x+2, log + "g called.")
f' . g' 3 "" -> (6, "g called.f called.")
```

So now this works. Jeej

Unfortunatly we don't have an *f'* but an *f* which is inconvenient

Small Tangent

Simple type definition

f is a function that takes a *float* and returns a (*float*, *string*)

```
f :: float -> (float, string)
```

More complex type definition

f is a function that takes a function *g* and a *float* and returns a *string*
the function *g* that is an argument of *f* will take a *float* and return a (*float*, *string*)

```
f :: g -> float -> string
f :: (float->(float,string))->float->string
```

AKA currying

Bind

The main problem is that f does not take $(float, string)$ but just *float*. Lets create *bind*, a higher order function that can help us

Typedef of *bind*

- Inputs: bind takes a function that takes a *float* and returns a $(float, string)$ and a $(float, string)$
- Output: a new function that will take a $(float, string)$ and returns a $(float, string)$

```
bind ::
    (float -> (float, string)) ->
    (float, string) ->
    ((float, string) -> (float, string))
```

Define *bind*

```
bind f (float, string) =
    let (new_float, new_string) = f(float)
    in (new_float, string + new_string)
```

Cool, but what good does it do?

- Given a pair of debuggable functions f and g of type $float \to (float, string)$
- We can easily compose them using $(bind\ f) \cdot g$
- Or simply $bind\ f \cdot g$
- Lets call this composition function $>>=$
- $f >>= g >>= h\ 3$

- We now have half of our formal definition.
- Bear with me a bit longer

- Lets ask ourselves if we can create an identity (aka *unit*) function for this *bind* method we just defined
- So that $(unit >>= f) = (f >>= unit) = f$
- Of course there is
- We can just define unit as $unit\ x \rightarrow (x, \text{“”})$

- The fun thing about this *unit* function is that we can use it to *lift* a normal function to a debuggable function
- $lift\ f = unit \cdot f$
- This means that I can now compose a normal function with a debuggable functions.
- $debugable >>= lift\ normalfunction$
- Of cource the debug output of our lifted normal function is just the empty string (thats how we defined *unit*

Summary of example

- We started with simple functions
- We wanted to make them debuggable by having them output an additional string
- We need to be able to compose small functions into larger ones
- We did not want to write a lot of boiler plate
- We accomplished this by implementing *bind*, *unit*, *lift* and $>>=$

Second small example

Before we give the formal definition lets do another quick example

Multiple return values

```
double :: Int -> [Int]
double x -> [x, x]
primeFactors :: Int -> [Int]
primeFactors 30 -> [2, 3, 5]

double . primeFactors = ???
```

How to do $double \cdot primeFactors$

- First we get the $primeFactors$
- For each result in the returned list we apply $double$
- This returns a new list where each element is a list of length 2
- To form a single list of all results we simply flatten the list

Sounds familiar

```
bind: f x = concat(map f x)
unit: x = [x]
```

- We saw debuggable functions $Debuggable: a \rightarrow (a, string)$
- We saw multivalued functions $Multivalued: a \rightarrow [a]$
- Make *Debuggable* or *Multivalued* abstract and call it m
- In both cases you get a function $a \rightarrow m\ a$
- For that specific m we define *bind* and *unit* (the others are in function of these two)

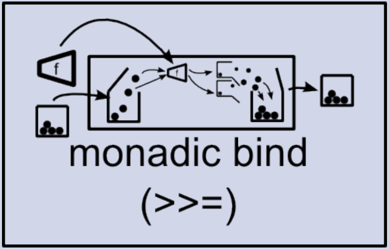
Formal definition

A monad is a triplet $(m, bind, unit)$. *bind* and *unit* must satisfy 3 mathematical laws

- left-unity
- right-unity
- associativity

Informal definition

- A monad is a box
- You define how you put a new item in this box (*unit*)
- You define how you can open a box to apply a function to the element inside of it (*bind*)
- If those two methods follow some basic rules (commutativity, ...) you can chain them nicely



What are they used for

The monad ('box') abstracts away a lot of computations that would be boiler plate. In our example passing the logging messages to each function is refactored in the monad's bind method

Other useful monads

- Maybe Monad: Abstracts away null checks
- State Monad: Abstracts away creating new changed states and passing it arround your functions (e.g. SeededRandom)
- Continuation Monad: Abstracts away call stack manipulations
- IO monad: Abstract away screen state manipulation
- ...

Caveat of box analogy

All 'boxes' can be seen as monads. But not all monads can be seen as boxes.

Questions?

- Need additional examples?
- Something not clear?
- Want a different explanation?
- You know where to find me !!

And when you understand monads fully, you are ready to learn about Arrows :)