You could have invented monads

Roeland Matthijssens

7 september 2017

Functional Programming

Why?

See Jeffrey's chalk talk

Limitations

- No state change
- No side effects
- What about IO?

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

- $f\ x \rightarrow (x + 1, \text{“}f\ \textit{was called}\text{”})$
- $g\ x \rightarrow (x + 2, \text{“}g\ \textit{was called}\text{”})$

f and g both take a float and return a tuple $(\mathit{float}, \mathit{string})$

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 $f \cdot g$
- $f\,x \rightarrow x + 1$
- $g\,x \rightarrow x + 2$
- $f \cdot g\,3 \rightarrow 6$

- But they take a *float* and output a $(\mathit{float}, \mathit{string})$. Simply composing them would not work
- $f\,x \rightarrow (x + 1, \text{“}f\ \mathit{called.}\text{”})$
- $g\,x \rightarrow (x + 2, \text{“}g\ \mathit{called.}\text{”})$
- $f \cdot g\,3 \rightarrow$ Error: f takes only one argument but got 2

- Lets redefine f and g
- $f'\ x\ logSoFar \rightarrow (x + 1, logSoFar + \text{“}f\ called.\text{”})$
- $g'\ x\ logSoFar \rightarrow (x + 2, logSoFar + \text{“}g\ called.\text{”})$
- $f' \cdot g'\ 3\ \text{“”} \rightarrow (6, \text{“}g\ called.f\ called.\text{”})$
- So now this works. Jeej
- Unfortunatly we don’t have a f' but an f which is inconvenient

Small Tangent

Simple type definition

- f is a function that takes a float and returns a $(\mathit{float}, \mathit{string})$
- $f :: \mathit{float} \to (\mathit{float}, \mathit{string})$

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 $(\mathit{float}, \mathit{string})$
- $f :: g \to \mathit{float} \to \mathit{string}$
- $f :: (\mathit{float} \to (\mathit{float}, \mathit{string})) \to \mathit{float} \to \mathit{string}$

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 $bind$

- Iinputs: bind takes a function that takes a $float$ and returns a $string$ and a $(float, string)$
- Output: a new function that will take a $(float, string)$ and returns a $(float, string)$
- $bind :: (float \rightarrow (float, string)) \rightarrow (float, string) \rightarrow ((float, string) \rightarrow (float, string))$

Define *bind*

- $bind\,f'\ (g_f, g_s) =$
- $\quad let(f_f, f_s) = f'(g_f)$
- $\quad in(f_f, g_s + f_s)$

Cool, but what good does it do?

- Given a pair of debuggable functions f and g of type $\mathit{float} \rightarrow (\mathit{float}, \mathit{string})$
- We can easily compose them using $(\mathit{bind}\, f) \cdot g$
- Or simply $\mathit{bind}\, f \cdot g$
- Lets call this composition function $\otimes$

- 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 $unit \bigotimes f = f \bigotimes unit = f$
- 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.
- $debugable \bigotimes lift\ normalfunction$
- Of cource the debug output of our lifted normal function is just the empty string (thats how we defined *unit*

Summary of example

- We stared 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 smaller ones
- We did not want to write a lot of boiler plate
- We accomplished this by implementing $bind$, $unit$, $lift$ and $\otimes$

Second small example

Before we give the formal definition lets do another quick example

Multiple return values

Real numbers

- $sqrt :: real \rightarrow real$
- $cbrt :: real \rightarrow real$
- $sixthroot = sqrt \cdot cbrt$

Complex numbers

- $sqrt :: complex \rightarrow [complex]$
- $cbrt :: complex \rightarrow [complex]$
- $sixthroot = ???$

How to do $sqrt \cdot cbrt$

- First we apply the $cbrt$
- For each result in the returned list we apply $sqrt$
- This returns a new list of length 3 where each element is a list of length 2
- To form a single list of all 6 roots we simply flatten the list

Sounds familiar

- $bind\,f\;x = concat(map\,f\;x)$
- $unit\;x = [x]$
- $f \bigotimes g = bind\,f \cdot g$
- $lift\,f = unit \cdot f$

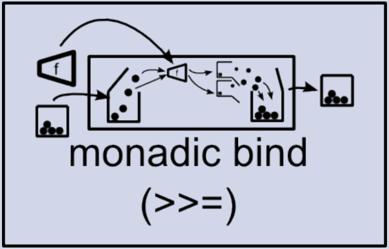
- We saw debuggable functions $Debuggable\ a = (a, string)$
- We saw multivalued functions $Multivalued\ a = [a]$
- Make $Debuggable$ or $Multivalued$ abstract and call it m
- In both cases you get a function $a \to m\ a$
- For that specific m we define $bind$ and $unit$ (the others are in function of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

Formal definition

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

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 usefull 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
- ...

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 for Arrows