Monads

A4M36TPJ, 2013/2014

Introduction

- In pure-functional languages **no side-effects** are allowed.
- Functions in pure-functional languages **depend only on input arguments**.
- Monads can be used to **simulate (not only) side-effects**.

Debuggable Functions

- We have functions **f** and **g** that both map floats to floats, but we'd like to modify these functions to also output strings for **debugging purposes**.

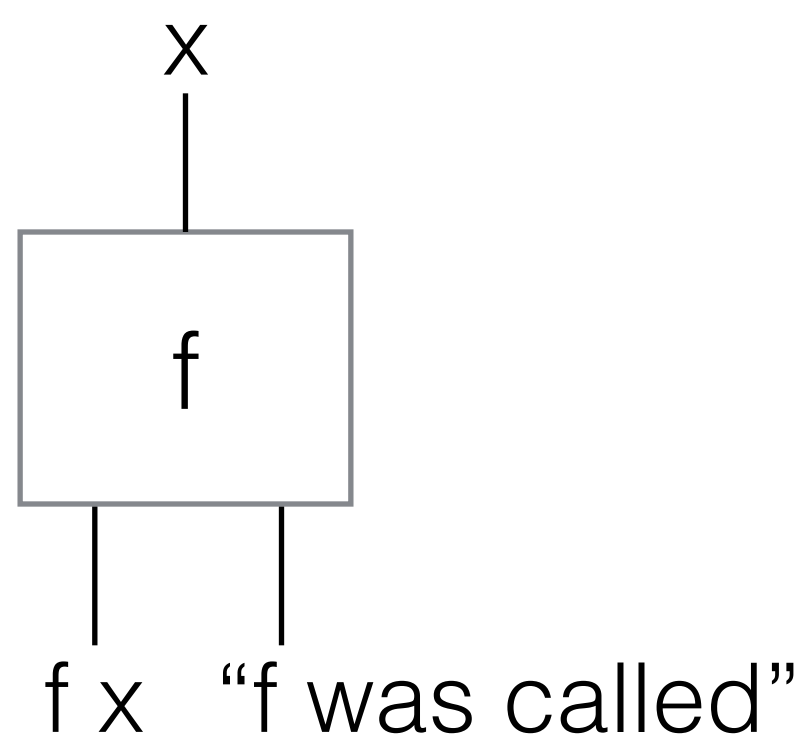
f,g : Float -> Float

Debuggable Functions

- How can we modify the types of **f** and **g** to admit side effects?
- The only possible way is for these strings to be returned alongside the floating point numbers.

f',g' :: Float -> (Float,String)

Debuggable Functions



Debuggable Functions

- What about function composition?

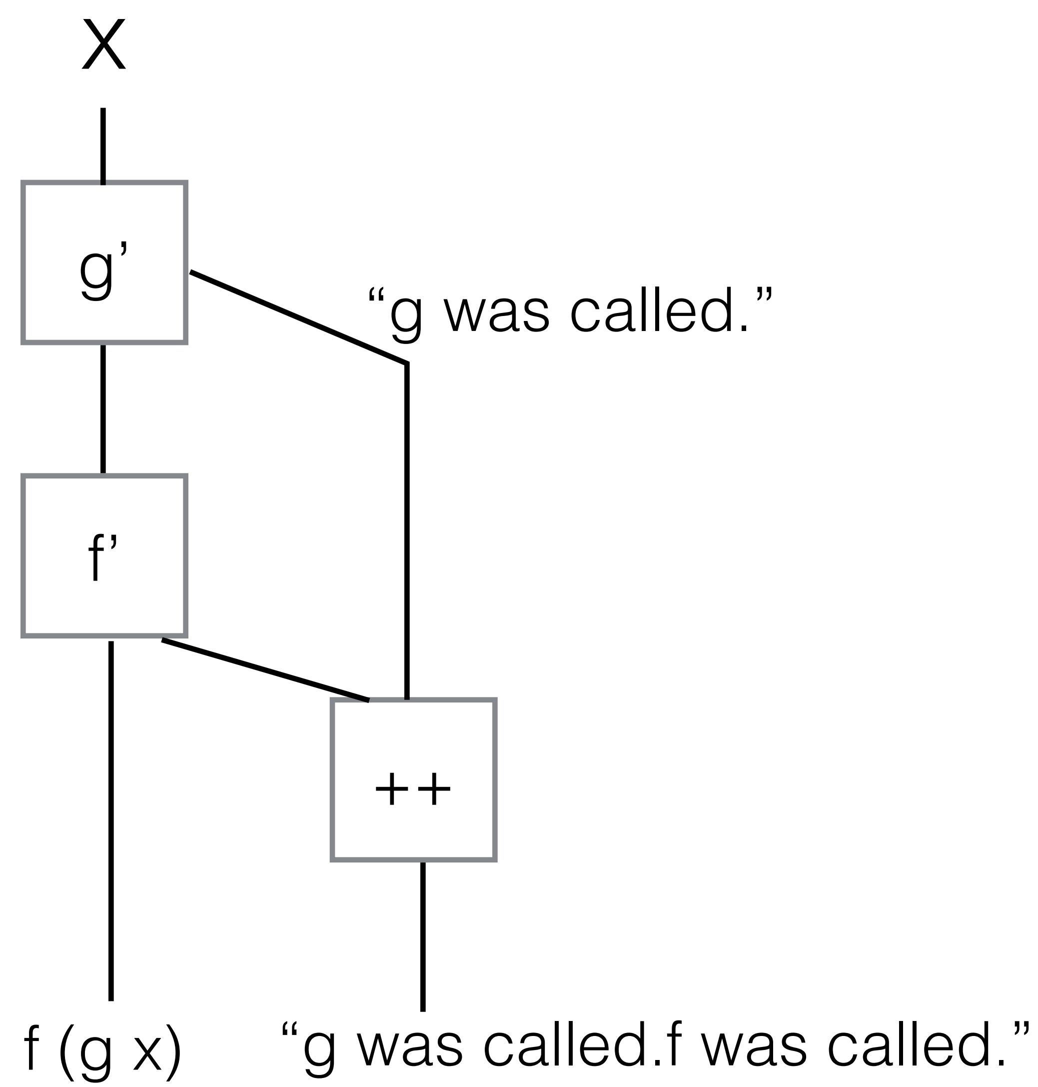
f’ . g’

  - These functions cannot be composed straightforward.

  - Return type of **g’** is not same as input type of **f’**.

Debuggable Functions

- We would like to compose functions **f’** and **g’** same way as **f** and **g**.



Debuggable Functions

- To implement previous diagram you can do:

```
let (y,s) = g' x
    (z,t) = f' y in (z,s++t)
```

- But you have to do it every time you want to compose functions **f’** and **g’**.

Debuggable Functions

- How can we do it easier programmatically?
- We need to find **higher-order** function which will do this **plumbing** for us.
- As the problem is that the output from **g'** can't simply be plugged into the input to **f'**, we need to 'upgrade' **f’**.

Debuggable Functions

- We introduce new function bind with the following type:

bind f' :: (Float,String) -> (Float,String)

bind :: (Float -> (Float,String)) ->
((Float,String) -> (Float,String))

Debuggable Functions

- bind must serve two purposes:
  - It must apply **f'** to the correct part of **g' x**.
  - Concatenate the string returned by **g'** with the string returned by **f'**.

bind f' (gx,gs) = let (fx,fs) = f' gx in (fx,gs++fs)

Debuggable Functions

- Given a pair of debuggable functions, **f'** and **g'**, we can now compose them together to make a new debuggable function **bind f' . g’**.
- We will write this composition as **f’ * g’**.

Debuggable Functions

- Even though the output of **g'** is incompatible with the input of **f'** we still have a nice easy way to concatenate their operations.
- And this suggests another question: Is there an 'identity' debuggable function?

Debuggable Functions

- Identity have the following properties:

  f . id = f and id . f = f

- According that we are looking for the function **unit**:

  unit * f = f * unit = f

- The function **unit** does not change the output of the function **f**.

Debuggable Functions

unit x = (x,"")

Debuggable Functions

- The unit allows us to 'lift' any function into a debuggable one.

lift f x = (f x,”")

lift f = unit . f

Debuggable Functions Summary

- The functions, **bind** and **unit**, allow us to compose debuggable functions in a straightforward way, and compose ordinary functions with debuggable functions in a natural way.

Exercise: Show that **lift f * lift g = lift (f.g)**

Multivalued Functions

- Consider functions **sqrt** and **cbrt** that compute the square root and cube root, respectively, of a real number. These are straightforward functions of type
Double -> Double.

- Consider a version of these functions that works with complex numbers.

- Every complex number, besides zero, has **two** square roots. Similarly, every non-zero complex number has **three** cube roots.

sqrt',cbrt' :: Complex Double -> [Complex Double]

Multivalued Functions

- Suppose we want to find the sixth root of a real number. We can just concatenate the cube root and square root functions. In other words we can define **sixthroot x = sqrt (cbrt x)**.

- How do we define a function that finds all six sixth roots of a complex number using **sqrt'** and **cbrt'**?

Multivalued Functions

- We face the similar problem like in Debuggable Functions. The return type (list) is not compatible with the input type (complex).
- We declare higher-order function bind with the following type:

```
bind :: (Complex Double -> [Complex Double]) ->
            ([Complex Double] -> [Complex Double])
```

Multivalued Functions

```
bind :: (Complex Double -> [Complex Double])
->      ([Complex Double] -> [Complex Double])
```

```
bind f x = concat (map f x)
```

```
unit x = [x]
```

Multivalued Functions

f * g = bind f . g

lift f = unit . f

Random Numbers

random :: StdGen -> (a,StdGen)

- To generate a random number you need a seed, and after you've generated the number you need to update the seed to a new value.

- A function that is conceptually a randomised function **a -> b** can be written as a function **a -> StdGen -> (b,StdGen)** where StdGen is the type of the seed.

Random Numbers

```
bind :: (a -> StdGen -> (b,StdGen)) ->
    (StdGen ->(a,StdGen)) -> (StdGen -> (b,StdGen))

bind f x seed = let (x',seed') = x seed in f x' seed’

unit :: a -> (StdGen -> (a,StdGen))

unit x g = (x,g)
```

Random Numbers Complete Example in Haskell

```
import Random

bind :: (a -> StdGen -> (b,StdGen)) -> (StdGen ->
(a,StdGen)) -> (StdGen -> (b,StdGen))

bind f x seed = let (x',seed') = x seed in f x' seed'

unit x g = (x,g)

lift f = unit . f
```

Random Numbers Complete Example in Haskell

```
addDigit n g =

        let (a,g') = random g in (n + a `mod` 10,g')

shift = lift (*10)

test :: Integer -> StdGen ->  (Integer,StdGen)

test = bind addDigit . bind shift . addDigit

g = mkStdGen 123

main = print $ test 0 g
```

Summary

type Debuggable a = (a,String)

type Multivalued a = [a]

type Randomised a = StdGen -> (a,StdGen)

m ∈ {Debuggable, Multivalued, Randomised}

- We're given a function **a -> m b** but we need to somehow apply this function to an object of type **m a** instead of one of type **a**.
- In each case we do so by defining a function called bind of type **(a -> m b) -> (m a -> m b)** and introducing a kind of identity function **unit :: a -> m a**.

Summary

- The triple of objects (m,unit,bind) is the monad, and to be a monad they must satisfy the Monad laws such as **unit * f = f * unit = f**, …

Monads in Haskell

- Haskell is a lazy evaluated pure-functional language.

- Monads are there used for I/O operations, State and other standard side-effects.

- In Haskell we write **bind** as infix operator **>>=**. So **bind f x** is written as **x >>= f**.

- **unit** function is called **return**.

- From previous examples **Debuggable is the Writer monad**, **Multivalued is the List monad** and **Randomised is the State monad**.

Monads in Haskell

```
return 7 >>= (\x -> Writer (x+1,"inc."))
         >>= (\x -> Writer (2*x,"double."))
         >>= (\x -> Writer (x-1,"dec."))
```

Haskell Syntax

```
do x <- y
          more code


y >>= (\x -> do
               more code).
```

Haskell Syntax

```
do
    let x = 7
    y <- Writer (x+1,"inc\n")
    z <- Writer (2*y,"double\n")
    Writer (z-1,"dec\n")
```

References

- http://www.haskell.org/haskellwiki/Monad
- http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html?m=1