How to really get rid of side effects completely? Introduction to Free Monad.

Ray Shih

Hi

- My name is **Ray Shih**
- B.S., M.S. of NTU CSIE
- Worked in WOOMOO
- Work in Mobiusbobs
- A **Fullstack Software Engineer**, including
  - iOS, Android, Web backend/Frontend
- Learn Haskell as a hobby :D

What is a Monad?

- `(>>=) :: m a -> (a -> m b) -> m b`
- `Monad Laws`
- `Do syntax!`

```haskell
main = do
  a <- readLn :: IO Int
  b <- readLn :: IO Int
  print $ a + b
```

What is Free Monad?

What is Free Monad

- A free monad generated by a functor is a special case of the more general free (algebraic) structure over some underlying structure. For an explanation of the general case, culminating with an explanation of free monads, see the article on free structures.

- from: https://wiki.haskell.org/Free_monad

- WAT ???????

What can
Free Monad do?

Entirely
Side Effect Free
Program!

Ordinary Program with Side Effect

```
main :: IO ()
main = do
  a <- readLn :: IO Int
  b <- readLn :: IO Int
  print $ a + b
```

Side Effect ->
Inverse of Control!!

```haskell
program :: IS ()
program = do
  a <- readInt
  b <- readInt
  output $ show $ a + b
```

```haskell
program = do
  a <- readInt
  b <- readInt
  output $ show $ a + b
```

Run it like normal IO Monad

```haskell
main = runProgram program
```

```haskell
program = do
  a <- readInt
  b <- readInt
  output $ show $ a + b
```

Or test it like pure function

```haskell
testProgram program ["1", "2"] == ["3"]
```

How to write
an Interpreter
with
self-defined instruction set?

```haskell
data IS a = Input (String -> IS a)
          | Output String (IS a)
          | Return a

p :: IS ()
p =
  (Input (\s ->
    Output ("Hello " ++ s)
         (Return ())))


run :: IS () -> IO ()
run (Input f) = do
  s <- getLine
  run $ f s

run (Output s next) = do
  putStrLn s
  run next

run (Return a) = return a
```

Try to make it a Monad!

```haskell
instance Monad IS where
  (Input genNext) >>= f =
    Input $ (\s -> genNext s >>= f)

  (Output s next) >>= f =
    Output s (next >>= f)

  (Return a) >>= f = f a

input = Input (\s -> Return s)
output s = Output s (Return ())

p :: IS ()
p =
  (Input (\s ->
    Output ("Hello " ++ s)
         (Return ())))
```

```haskell
instance Monad IS where
  (Input genNext) >>= f =
    Input $ (\s -> genNext s >>= f)

  (Output s next) >>= f =
    Output s (next >>= f)

  (Return a) >>= f = f a

input = Input (\s -> Return s)
output s = Output s (Return ())

p :: IS ()
p = do
  s <- input
  output $ "Hello " ++ s
```

Cool!

But there is a problem

Don’t Repeat Yourself!

- We need to define/refine monad every time we create new DSL or add new instruction to existed DSL.

How?
By extracting the recursive structure

Fix Point !!

Review: y combinator

(oversimplified version)

Recursive function can be generated by finding the fix point of recursive function generator

```
yCom g = f where f = g f

genFib f 0 = 0
genFib f 1 = 1
genFib f n = f (n - 1) + f (n - 2)

fib = yCom genFib
```

```
     a
   Fx a
Fx (Fx a)
```

The type of Recursive Construct can be obtained by finding the fix point of ???

Lets place some holes for now

```
data A = Fx A
```

How to extract?

```
data A = Fx A

data IS a
  = Input (String -> IS a)
  | Output String (IS a)
  | Result a
```

How to extract?

```
data A = Fx (f A)

data IS a
  = Input (String -> a)
  | Output String a
  | Result a
```

How to extract?

```
data A = Fx (f A)
       | Result a

data IS a
  = Input (String -> a)
  | Output String a
```

How to extract?

```
data Free f a = Fx (f A)
              | Result a

data IS a
  = Input (String -> a)
  | Output String a
```

How to extract?

```
data Free f a = Fx (f (Free f a))
              | Result a

data IS a
  = Input (String -> a)
  | Output String a

-- ex: Free IS ()
```

Before define the monad

```
data Free f r =
  Free (f (Free f r)) | Pure r
    deriving (Functor)

data IS next = Input (String -> next)
             | Output String next
             deriving (Functor)
```

Define the monad!!

```
data Free f r =
  Free (f (Free f r)) | Pure r
    deriving (Functor)

data IS next = Input (String -> next)
             | Output String next
             deriving (Functor)

instance (Functor f) => Monad (Free f) where
  (Free x) >>= f = Free (fmap (>>= f) x)
  (Pure r) >>= f = f r
```

```haskell
instance (Functor f) => Monad (Free f) where
  (Free x) >>= f = Free (fmap (>>= f) x)
  (Pure r) >>= f = f r
```

```haskell
p = (Free (Input (\s -> (Pure s))))
  >>= (\s -> (Free (Output s (Pure ()))))
```

```
instance (Functor f) => Monad (Free f) where
  (Free x) >>= f = Free (fmap (>>= f) x)
  (Pure r) >>= f = f r
```

```
p = (Free (Input (\s -> (Pure s)))) >>= f
  where f =
    (\s -> (Free (Output s (Pure ()))))
```

```
instance (Functor f) => Monad (Free f) where
  (Free x) >>= f = Free (fmap (>>= f) x)
  (Pure r) >>= f = f r
```

```
p = (Free
      (fmap (>>= f)
        (Input (\s -> (Pure s)))))
  where f =
    (\s -> (Free (Output s (Pure ()))))
```

```
instance (Functor f) => Monad (Free f) where
  (Free x) >>= f = Free (fmap (>>= f) x)
  (Pure r) >>= f = f r
```

```
p = (Free
      (Input (\s -> ((Pure s) >>= f)))))
  where f =
    (\s -> (Free (Output s (Pure ()))))
```

```haskell
instance (Functor f) => Monad (Free f) where
  (Free x) >>= f = Free (fmap (>>= f) x)
  (Pure r) >>= f = f r
```

```haskell
p = (Free
      (Input (\s -> (f s))))
  where f =
    (\s -> (Free (Output s (Pure ()))))
```

```
instance (Functor f) => Monad (Free f) where
  (Free x) >>= f = Free (fmap (>>= f) x)
  (Pure r) >>= f = f r
```

```
p = (Free
      (Input (\s ->
        (Free (Output s (Pure ()))))))
```

liftF

```
input = (Free (Input (\s -> Pure s)))
output s = (Free (Output s (Pure ())))
```

```
liftF i = Free (fmap Pure i)

input = liftF $ Input (\s -> s)
output s = liftF $ Output s ()
```

```
readInt :: Free IS Int
readInt = read <$> input
```

```
program = do
  a <- readInt
  b <- readInt
  output $ show $ a + b
```

```
runProgram program

testProgram program ["1", "2"] == [“3"]
```

Complete Interpreter

```haskell
runProgram :: Free IS () -> IO ()
runProgram (Free (Input genNext)) = do
  s <- getLine
  runProgram $ genNext s

runProgram (Free (Output s next)) = do
  putStrLn s
  runProgram next

runProgram (Pure r) = return r
```

So What is Free Monad? Given any Functor, we can get a monad for free!

```
instance (Functor f) => Monad (Free f) where
  (Free x) >>= f = Free (fmap (>>= f) x)
  (Pure r) >>= f = f r
```

Use 3rd Party Free Monad Lib

```haskell
import Control.Monad.Free
  ( Free(Free, Pure)
  , liftF)

data IS a = Input (String -> a)
          | Output String a
          deriving (Functor)

input = liftF $ Input (\s -> s)

output :: String -> Free IS ()
output s = liftF $ Output s ()

p :: Free IS ()
p = do
  s <- input
  output $ "Hello " ++ s

run :: Free IS () -> IO ()
run (Free (Input f)) = do
  s <- getLine
  run $ f s

run (Free (Output s next)) = do
  putStrLn s
  run next

run (Pure a) = return a
```

Free Monad + Interpreter Pattern

Cool Example: Software Thread!!

```
interleave (Atomic m1) (Atomic m2) = do
    next1 <- atomic m1
    next2 <- atomic m2
    interleave next1 next2

interleave t1 (Return _) = t1
interleave (Return _) t2 = t2

runThread (interleave thread1 thread2)
```

References

- http://www.haskellforall.com/2012/06/you-could-have-invented-free-monads.html
- https://github.com/ekmett/free/