Monads and Comonads

Joe Isaacs

Churchill CompSci talks

November 16, 2016

Type classes

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Integer where
  (==) a b = intEq a b

triEq :: (Eq a) => a -> a -> a -> Bool
triEq a b c = (a==b) && (a==c) && (b==c)
```

Category theory background

A category contains:

- Objects $(A, B, C, \dots)$
- Morphisms. $(f : B \to C)$
- Each object must have an identity morphism. (id_A for an object A)
- Morphisms compose. ($f \circ g : A \to C$, if $g : A \to B$)

Categories must also follow three laws:

- $f \circ (g \circ h) = (f \circ g) \circ h$
- There must be a morphism $h : A \to C$ such that $h = f \circ g$

Each object A must contain an identity morphism $id_A : A \to A$.

Hask

- We take objects A, B to be types `A` and `B`
- We take morphisms $f : A \to B$ to be functions of type `f :: a -> b`
- Composition of morphisms $\circ$ will be function composition (written `(.)` in Haskell).
- $f \circ g$ goes to `(f . g) x` which in haskell in the same as `f (g (x))`.

Functor

- Functors F will map from a category C to a category D written as

$$F : C \to D$$

- An object A in C will be mapped to $F(A)$ in D
- (Covariant) functors map morphisms $f : A \to B$ to $F(f) : F(A) \to F(B)$
- $F(id_A) = id_{F(A)}$ where $id_A : A \to A$ and $id_{F(A)} : F(A) \to F(A)$
- $F(f \circ g) = F(f) \circ F(g)$ (functors distribute over morphism composition).

Functor in Haskell

- A functor F will map $f : A \to B$ to $F(f) : F(A) \to F(B)$
- `class Functor f where`
  `fmap :: (a -> b) -> f a -> f b`
- $id_A =$ `id :: a -> a`
- $id_{F(A)} =$ `fmap id :: f a -> f a`
- $F(h \circ g) =$ `fmap (h.g) :: f a -> f c`
- $F(h) \circ F(g) =$ `fmap h . fmap g :: f a -> f c`

where $\circ =$ `(.)`, $h =$ `h :: b -> c`, $g =$ `g :: a -> b` and $F =$ `Functor f`

Functor example `Maybe`

```
data Maybe a = Nothing | Just a
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap _ Nothing  = Nothing
```

- Must check $F(id_A) = id_{F(A)}$

```
  fmap id x
case (x == Just a)
  fmap id (Just a) a ==> Just (id a) ==> Just a
case (x == Nothing)
  fmap id Nothing ==> Nothing

==> fmap id = id
```

Functor example Maybe

Check $F(f \circ g) = F(f) \circ F(g)$

```
case (Just a)
      fmap (f.g) (Just a)
  ==> Just ((f.g) a)
  ==> Just (f (g a))

      (fmap f . fmap g) (Just a)
  ==> fmap f (fmap g (Just a))
  ==> fmap f (Just (g a))
  ==> Just (f(g a))
case (Nothing)
  fmap (f.g) Nothing == Nothing
      fmap f . fmap g Nothing
  ==> fmap f Nothing
  ==> Nothing
```

Monad

- Defined by $(M, join, unit)$
- M is also a functor
- $join$ is the transformation $M(Ma) \to Ma$ which satisfies

$$join \circ M(join) = join \circ join$$

  and

$$join \circ M(M(f)) = M(f) \circ join$$

- $unit$ is the transformation $a \to Ma$ which satisfies

$$join \circ M unit = join \circ unit = id_M$$

  from $M \to M$ and

$$unit \circ f = M(f) \circ unit$$

Monad in Haskell

We have seen the categorical definition of monads above, this could be a Haskell implementation.

```
class Functor m => Monad m where
  unit :: a -> m a
  join  :: m (m a) -> m a
```

- $unit = \texttt{unit}$
- $join = \texttt{join}$
- The monad laws can now be written in Haskell as:

```
join . fmap join      = join . join
join . fmap unit      = join . unit     = id
unit . f              = fmap f . unit
join . fmap (fmap f) = fmap f . join
```

What do the laws mean?

We can explain all the laws with commutative diagrams, here is the first:

$$join \circ M(join) = join \circ join$$

$$\begin{tikzcd}
M^3 \arrow[r, "join"] \arrow[d, "M(join)"'] & M^2 \arrow[d, "join"] \\
M^2 \arrow[r, "join"] & M
\end{tikzcd}$$

When written in Haskell it looks like this.

```
join . fmap join = join . join
```

How Monads are really implemented in Haskell

- Haskell implements monad in different but equivalent way:

```
class Functor m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  fail   :: String -> m a
```

- `(>>=)` (pronounced bind)
- `return` = `unit`

Joining bind

- `join ::  m m a -> m a`
- `>>= ::  m a -> (a -> m b) -> m b`
- `(>>=)` can be derived from `unit` and `fmap`:

```
join x = x >>= id  (id :: m a -> m a)

f :: a -> b
x :: m a

x >>= f = join (fmap f x)  (fmap f :: m m b)
```

Monad laws using `(>>=)`

- ```
  m >>= return = m
  return m >>= f = f m
  (m >>= f) >>= g = m >>= (\x -> f x >>= g)
  ```
  These are left identity, right identity and associativity
- It can also be show that these three monad laws are equivalent to the four monad laws stated previously.
  ```

Example: Maybe

```
instance Monad Maybe where
  return x        = Just x
  fail _          = Nothing
  Just x  >>= f = f x
  Nothing >>= _ = Nothing
```

The Maybe instance of Monad must also follow the monad laws stated previously.

```
m >>= return = m

Just x >>= return ==> return x ==> Just x

Nothing >>= return ==> Nothing
```

Computations with possible errors

```
computeB :: a -> Maybe b
computeC :: b -> Maybe c

compute  :: a -> Maybe c
compute a = case computeB of
              Just b  -> computeC b
              Nothing -> Nothing
```

Computations with possible errors Cont.

```
computeB :: a -> Maybe b
computeC :: b -> Maybe c

compute  :: a -> Mayne c
compute a = computeB a >>= computeC
```

do-notation

```
do x                    ==> x

do let y = z            ==> let y = z in do x
   x

do y <- z               ==> z >>= \y -> do x
   x

do y                    ==> y >>= \_ -> do x
   x
```

Using do-notation

```
compute :: a -> Maybe c
compute a =
  do b <- computeB a
     c <- computeC b
     return c
```

Looks a lot like imperative code

IO with monads

- IO monad, with two operations
  - `putStr :: String -> IO ()` builds a monad that when run will print the string passed into `putStr` and return
  - `getLine :: IO String`, build a monad that when run will read a stream of characters for the keyboard up to a newline.
- This means the IO monad represents a sequence of computations, which are order in the same order as the binds.
- The IO monad will then be evaluated once it is returned to the Haskell runtime or `unsafePerformIO :: IO a -> a` is performed.

Example IO

```
main :: IO ()
main = do x <- getLine
          putStr ("return: " ++ x)
```

>>= for IO

```
instance Monad IO a where
  return a       = IO (\s -> (s,a))
  (IO m) (>>=) f = IO
      (\s -> case MUTABLE(m s) of (m',_) -> f m')
```

Comonad

- Dual of a monad
- Remember Monad has two morphisms
  - $unit : a \to Ma$
  - $join : M(Ma) \to Ma$
- A comonad C defines two transformations:
  - $extract : Ca \to a$
  - $duplicate : Ca \to C(Ca)$
- Satisfying three laws:
  - $extend \circ duplicate = I_C$
  - $C(extract) \circ duplicate = I_C$
  - $C(duplicate) \circ duplicate = duplicate \circ duplicate$
- In the triple $(C, extract, duplicate)$, C is a functor.

Comonad in Haskell

```
class Functor w => Comonad w where
  extract :: w a -> a
  extend  :: (w a -> b) -> w a -> w b
```

since

```
duplicate :: m a   -> m m a
fmap f    :: m m a -> m b
```

We can define extend as `extend f = (fmap f) . duplicate`
A Comonad must also obey these laws:

```
extend extract       = id
extract . extend f   = f
extend f . extend g = extend (f. extend g)
```

CArray and associated functions

```
data CArray i a = CA (Array i a) i

indices :: Array i a -> [i]

bounds :: Array i a -> [i]

array :: [i] -> [a] -> Array i a

(!) :: Array i a -> a

(?) :: CArray i a -> a    (a safe ! with an offset)
```

Array filters are comonads

```
instance Comonad (CArray i) where
  extract (CA arr c) = arr!c
  extend f (CA x c)  =
    let  es' = map (\i -> (i,f(CA x i))) (indices x)
    in   CA (array (bounds x) es') c

laplace1D :: Num a => CArray Integer a -> a
laplace1D a = (a ? (-1)) + (a ? 1)  - 2 * (a ? 0)
```

`(?)` will try to get the value at `i+i'` of the value `a` where `(CA a i)`

Example from 'A Notation for Comonads' by Dominic Orchard and Alan Mycroft

Questions?

Mixing Monads

- What happens if we want to have both IO and logging in the same functions?
- Use monad transformers.
- Monad transformers require another function to be defined `lift :: m a -> t m a`.
- This must satisfy the laws:
  - `lift . return = return`
  - `lift (m >>= f) = lift m >>= (lift . f)`
- This is captured in Haskell by:

```
class MonadTrans t where
  lift :: m a -> t m a
```

Logging with IO

```
type LoggingIO l a = WriterT l IO a

log :: String -> LoggingIO [String] ()
log s = tell [s]

example :: LoggingIO [String] ()
example = do log "Print 1"
             lift (print 1)

main = do str <- execWriterT example
          (print str)
```

We cannot just compose any Monad

- Remember the type of `join :: M(M a) -> M a`
- If we extend this to monad transformers we get

```
join :: M(N(M(N a))) -> M(N a)
```

- We cannot infer the `join` method for transformers just from the `join` functions for the transformer and the monad being transformed.
- To allow composition we must also have a function

```
distrib :: t (m a) -> m (t a)
```

Monoid

A monoid is an object M with two morphisms $\oplus : M \times M \to M$ and $unit : M$.
Where both

$$unit \oplus M = M = M \oplus unit \text{ (left and right identity)}$$

and

$$M \oplus (M \oplus M) = (M \oplus M) \oplus M \text{ (associativity)}$$

```
class Monoid a where
  mzero   :: a
  mappend :: a -> a -> a
```

$unit = \texttt{mzero}$ and $\oplus = \texttt{mappend}$

Writer Monad Motivation: Logging

- Want to store order set of stages throughout a computation.

```
addOne :: Int -> Int -> [String]
addOne x = (x+1,["Added 1 to " ++ show x])

applyLogger :: Monoid m =>
               (v,m) -> (v -> (o, m)) -> (o,m)
applyLogger (input,log) f = (o, log `mappend` l)
  where (o,l) = f input

addOne 0 `applyLogger` addOne
`applyLogger` addone

> (3,["Added 1 to 0"
    ,"Added 1 to 1","Added 1 to 2"])
```

Write Monad

```
newtype Writer w a = Writer { runWriter :: (a,w) }

instance (Monoid w) => Monad (Writer w) where
  return x = writer (x,mempty)
  (Writer (x,v)) >>= f = let (Writer (y,v')) = f x
                         in Writer (y,v `mappend` v')
```

Logging with the Write Monad

```
logNumber :: Int -> Writer [String] Int
logNumber x = Writer (x, ["Recorded " ++ (show x))

runWriter
(do a <- logNumber 10
   b <- logNumber 11
   return (a*b))
> (110,["Recorded 10", "Recorded 11"])
```