Functional Programming

Monads

H. Turgut Uyar

2015-2016

License

Topics

1. Introduction
   - Functors
   - Applicative Functors
   - Monads

Function Composition

example

```
-- show :: a -> String
--         a -> [Char]
-- length :: [a] -> Int

-- length . show $ 42   ~> 2
```

Composition with IO

example

```
-- getLine :: IO String
--            IO [Char]
-- length :: [a] -> Int

-- length . getLine     ~> type error
```

Composition with IO

```
-- fmapIO length getLine

fmapIO f p = do x <- p
                return (f x)
```

- what is the type of `fmapIO`?
  `fmapIO :: ([a] -> Int) -> IO [Char] -> IO Int`
- more general:
  `fmapIO :: (a -> b) -> IO a -> IO b`

Type Examples

```
data Box a = Box a
             deriving Show


data Maybe a = Nothing | Just a
               deriving Show
```

Functor Class

- extract value, apply function, wrap result
- functor

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

- infix `fmap`: <$>

Functor Example

```
instance Functor Box where
  fmap f (Box x) = Box (f x)

-- (+3) (Box 2)          ~> type error
-- fmap (+3) (Box 2)     ~> Box 5
-- (+3) <$> Box 2        ~> Box 5
```

Functor Example

```
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)

-- (+3) <$> Just 2       ~> Just 5
-- (+3) <$> Nothing      ~> Nothing
```

Functor Laws

- `fmap id == id`
- `fmap (f . g) == fmap f . fmap g`

Functor Law Examples

-
  ```
  fmap id (Box x)
  == Box (id x)
  == Box x
  == id (Box x)
  ```

-
  ```
  fmap (f . g) (Box x)
  == Box ((f . g) x)
  == Box (f (g x))
  ```

-
  ```
  (fmap f . fmap g) (Box x)
  == (fmap f) (fmap g (Box x)
  == (fmap f) (Box (g x))
  == Box (f (g x))
  ```

Lists as Functors

- `fmap :: (a -> b) -> f a -> f b`
- replace f with `[]`:
  `fmap :: (a -> b) -> [a] -> [b]`

```haskell
instance Functor [] where
  fmap = map
```

Functions as Functors

- `fmap :: (a -> b) -> f a -> f b`
- replace f with `(->) r`:
  `fmap :: (a -> b) -> ((->) r) a -> ((->) r) b`
- same as:
  `fmap :: (a -> b) -> (r -> a) -> (r -> b)`

```haskell
instance Functor ((->) r) where
  fmap = (.)
```

Applicative Functors

- extract function, extract value, apply, wrap result
- applicative functor

```haskell
class Functor f => Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b
  pure :: a -> f a
```

Applicative Functor Example

```haskell
instance Applicative Box where
  (Box f) <*> (Box x) = Box (f x)

  pure x = Box x

-- Box (+3) <$> Box 2   ~> type error
-- Box (+3) <*> Box 2   ~> Box 5
```

Applicative Functor Example

```
instance Applicative Maybe where
  Nothing <*> _ = Nothing
  Just f  <*> v = fmap f v

  pure x = Just x

-- Just (+3) <*> Just 2 ~> Just 5
-- Nothing   <*> Just 2 ~> Nothing
```

Applicative Functor Example

- how to add two Maybe values?

```
addMaybe :: Num a => Maybe a -> Maybe a -> Maybe a
addMaybe (Just x1) (Just x2) = Just (x1 + x2)
addMaybe _         _         = Nothing

-- OR:
addMaybe v1 v2 = (+) <$> v1 <*> v2
```

Applicative Functor Laws

- identity:
  `pure id <*> v == v`
- composition:
  `pure (.) <*> u <*> v <*> w == u <*> (v <*> w)`
- homomorphism:
  `pure f <*> pure x == pure (f x)`
- interchange:
  `u <*> pure y == pure ($ y) <*> u`
- as a consequence:
  `f <$> x == pure f <*> x`

Composing with IO

example

```
-- getLine :: IO String
-- readFile :: String -> IO String

-- readFile . getLine   ~> type error
```

IO Sequencing

- sequence I/O operations

```
-- bindIO getLine readFile

bindIO p q = do x <- p
                q x
```

- what is the type of bindIO?
  ```
  bindIO :: IO String -> (String -> IO String)
                      -> IO String
  ```
- more general:
  ```
  bindIO :: IO a -> (a -> IO b) -> IO b
  ```

Monads

- pattern: extract value, apply function
- monad

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

IO Monad

example

```
-- getLine :: IO String
-- readFile :: String -> IO String

readFileInteractive :: IO String
readFileInteractive = getLine >>= readFile


-- putStrLn :: String -> IO ()

catFileInteractive :: IO ()
catFileInteractive = getLine >>= readFile >>= putStrLn
```

Monads

- syntactic sugar for monads: `do`

example

```
catUserFile' = do filename <- getLine
                  content <- readFile filename
                  putStrLn content
```

Monad Example

```
userAge = getLine >>=
    (\l -> putStrLn $ show $ 2016 - (read l :: Int))


-- OR:
userAge = do line <- getLine
             let age = 2016 - (read line :: Int)
             putStrLn $ show age
```

Monad Example

```
instance Monad Box where
  Box x >>= f = f x

  return x = Box x

-- Box 18 >>= (\x -> Box (x ‘div‘ 2))      ~> Box 9
-- Box 18 >>= (\x -> return (x ‘div‘ 2))   ~> Box 9
```

Monad Example

```
instance Monad Maybe where
  Nothing >>= f = Nothing
  Just x  >>= f = f x

  return x = Just x

-- Just 18 >>= (\x -> return (x ‘div‘ 2))   ~> Just 9
```

Monad Example

```
half :: Monad m => Integer -> m Integer
half x = return (x ‘div‘ 2)

-- Box 18 >>= half              ~> Box 9
-- Just 18 >>= half             ~> Just 9

-- Just 18 >>= half >>= half    ~> Just 4
-- half <=< half $ Just 18      ~> Just 4
```

References

Required Reading: Thompson

- Chapter 18: Programming with monads

Required Reading: Lipovaa

- `http://learnyouahaskell.com/`
- Chapter 11: Functors, Applicative Functors and Monoids
- Chapter 12: A Fistful of Monads