Functors, Applicative Functors and Monoids

Evan Misshula

2017-02-14

Outline

Polymorphism on a higher level

- Types are not part of a hierarchy
- We can think about how they should act
    - then connect them with typeclasses

Functors defined

Analogy with other typeclasses

Typeclasses define functions

- Eq *defines* concrete types that are equatable
  - functions ('=') and ('/')
- Ord *defines* concrete types that 'orderabe'
  - implements the 'compare' function
- Enum *defines* concrete types that enumerable
  - defines '..' a range

List Functor examples

Example (List Functor Examples)

- map:: (a -> b) -> [a] -> [b]

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

Functor code in the repl

List Functor in the repl

```
  :t map
  fmap (*2) [1..3]
  map (*2) [1..3]
i==i

map :: (a -> b) -> [a] -> [b]
[2,4,6]
[2,4,6]
```

Maybe Functor examples

Example (Maybe Functor Examples)

- type Maybe a = Nothing | Just a

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

Maybe Functor code in the repl

Maybe Functor in the repl

```
:t fmap
fmap (++ " HEY GUYS IM INSIDE THE JUST") (Just "Something seri
fmap (++ " HEY GUYS IM INSIDE THE JUST") Nothing
fmap (*2) (Just 200)
fmap (*2) Nothing
i==i

fmap :: Functor f => (a -> b) -> f a -> f b
Just "Something serious. HEY GUYS IM INSIDE THE JUST"
Nothing
Just 400
Nothing
```

Either Functor examples

Example (Either Functor Examples)

- data Either e a = Left e | Right a

```
instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left x) = Left x
```

IO is a Functor

IO is a Functor

```
instance Functor IO where
    fmap f action = do
result <- action
return (f result)
```

Play with the IO

Reversing a string

```
main = do line <- getLine
  let line' = reverse line
      putStrLn $ "You said " ++ line' ++ " backwards!"
      putStrLn $ "Yes, you really said" ++ line' ++ " backwards
```

Now we can do the same with fmap

```
main = do line <- fmap reverse getLine
  putStrLn $ "You said " ++ line ++ " backwards!"
  putStrLn $ "Yes, you really said" ++ line ++ " backwards!"
```

A general Functor example

A general Functor example

- Let's say we want to reverse a string and upcase it
- And we want to interleave '-'

```
import Data.Char
import Data.List

main = do line <- fmap (intersperse '-'
. reverse
. map toUpper) getLine
  putStrLn line
```

Functor in the repl

Let's look at the type of

- a partially applied fmap

  fmap (replicate 3) (Functor f) => f a -> f [a]

```
fmap (replicate 3) [1,2,3,4]
fmap (replicate 3) (Just 4)
fmap (replicate 3) (Right "blah")
fmap (replicate 3) (Left "foo")
1==1
```

```
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]
Just [4,4,4]
Right ["blah","blah","blah"]
Left "foo"
```

Functor Law intuition

If functors mean that something can be mapped over...

- then calling 'fmap' on a functor should
  - map a function over the functor

Functor Law intuition

If functors mean that something can be mapped over...

- then calling 'fmap' on a functor should
  - map a function over the functor
- Nothng else

The First Functor Laws

Definition (The First Functor Law)

states that if we map the identity (id) function over a functor, we get the functor

- fmap id = id

Identity in the Repl

Identity functions in the repl

```
fmap id (Just 3)
id (Just 3)
fmap id [1..5]
id [1..5]
fmap id []
fmap id Nothing
1==1

Just 3
Just 3
[1,2,3,4,5]
[1,2,3,4,5]
[]
Nothing
```

The Second Functor Law

Definition (The Second Functor Law says)

The Second Functor Law says that composing two functions and then mapping the composed function over a functor is the same as first mapping one function over the functor and then mapping the other one.

- fmap (f.g) = fmap f . fmap g
- fmap (f.g) F = fmap f (fmap g F)

Composition in the Repl

Composition functions in the repl

```
fmap ((+1).(*2)) (Just 3)
fmap (+1) (fmap (*2) (Just 3))
fmap  ((+1).(*2)) [1..5]
fmap (+1) (fmap (*2) [1..5])
1==1

Just 7
Just 7
[3,5,7,9,11]
[3,5,7,9,11]
```

What if we map a multi-parameter function over a functor?

- Look at the type signature

```
let a = fmap (*) [1..4]
:t a
fmap (\f -> f 9) a
1==1
```

```
a :: [Integer -> Integer]
[9,18,27,36]
```

What if we want to take a function out of a Just

Let's take a Just (3 *) and map

and map it over Just 5

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

Maybe Applicative

Let's look at the Applicative for Maybe

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

Maybe Applicative inside the repl

Using the Maybe Applicative

```
:m Control.Applicative
Just (+3) <*> Just 9
pure (*2) <*> Just 10
pure (+3) <*> Just 9
Just (++"!!") <*> Just "Go now"
Nothing <*> Just "woot"
1==1



Just 12
Just 20
Just 12
Just "Go now!!"
Nothing
```

Fmap as an infix operator

Control.Applicative exports a function called <$>

which is fmap as an infix operator

```
(<$>) :: (Functor f) => (a->b) -> f a -> f b
f <$> x = fmap f x
```

Compare Applicatives in the repl

Infix fmap in the repl

```
(++) <$> Just "John " <*> Just "Travolta"
(++) "John " "Travolta"
1==1

Just "John Travolta"
John Travolta
```

Lists are Applicative Functors

Definition (Definition of the Applicative for a list)

- Literally a Cartesian product of functions and list values

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x<- xs]
```

Applicative Functors of lists in the repl

Applicative Functors of lists in the repl

```
[(*0),(+100),(^2)] <*> [1..4]
[(+),(*)] <*>[1,2]<*>[3,4]
(++) <$> ["ha","heh","hmm"] <*> ["?","!","."]
1==1

[0,0,0,0,101,102,103,104,1,4,9,16]
[4,5,5,6,3,4,6,8]
["ha?","ha!","ha.","heh?","heh!","heh.","hmm?","hmm!","hmm."]
```

IO is an Applicative

Let's see how the IO Applicative is implemented:

```
instance Applicative IO where
    pure = return
    a <*> b = do
f <- a
x <- b
return (f x)
```

Concatenating IO strings

Two ways to concatenate two lines of user input string

- Imperative code

```
myAction :: IO String
myAction = do
    a <- getLine
    b <- getLine
    return $ a ++ b
```

Applicative way to concatenate two lines of user input string

- Applicative code

```
myAction :: IO String
myAction = (++)
    <$> getLine
    <*> getLine
```

The first Applicative Functor Law

Theorem (The first Applicative Functor Law)

$pure\ f <*> x = fmap\ f\ x$

Some lessons we've skipped

Defining types

- *data* will define a new algebraic type
- *type* creates a type synonym
- *newtype* creates new types from old types

Applicative Functor in two ways

function left, each argument right

```
:m Control.Applicative
[(+1),(*100),(*5)] <*> [1..3]
1==1


[2,3,4,100,200,300,5,10,15]
```

function left, every argument right

```
getZipList $ ZipList [(+1),(*10
-- getZipList $
-- ZipList [(+1),(*100),(*5)]
--  <*> ZipList [1,2,3]
1==1

[2,200,15]
```

The newtype keyword

'newtype' takes one type and wrap it

- to present it as another type

ZipList [a] }

- data can have multiple value contstructors

type vs. newtype vs. data examples

'data' to make new types

- Here are additive and multiplicative types with multiple constructors

```
data Profession = Fighter | Archer | Wizard
data Species = Human | Elf | Orc | Goblin
data PlayerCharacter = PlayerCharacter Species Profession
```

Using newtype to drive typeclass properties

newtype

```
newtype CharList = CharList {getCharList :: [Char]} deriving(Ec
CharList "this will be shown!"
CharList "benny" == CharList "benny"
CharList "benny" == CharList "oisters"
1==1


CharList {getCharList = "this will be shown!"}
True
False
```

Monoid Definition

Definition (Monoid definition)

A data type, category or set is a monoid if it has a binary operation $\bullet$ which is associative and has an identity.

- $\forall a, b, c \in S, (a \bullet b) \bullet c = a \bullet (b \bullet c)$
- $e \bullet a = a \bullet e = a$

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

Monoid functions defined

Defining the monoid functions

- 'mempty' is just the identity function
- mappend is the binary function
  - it doesn't just append
- mconcat reduces a list of monoid values and reduces them to one by applying mappend

Monoid Laws

Theorem (The Monoid Laws are just the definition in Haskell)

- $mappend\ mempty\ x = x$
- $mappend\ x\ mempty = x$
- $mappend\ (mappend\ x\ y)\ z = mappend\ x\ (mappend\ y\ z)$

Monoid examples

Example (List is a monoid)

- [] with (++) is a monoid
  - id = ""
- Natural numbers with (*) is a monoid
  - id = 1
- Natural numbers with (+) is a monoid
  - id = 0