Lesson 10

Chapter 12

Functors and Applicatives

we define some type classes that require the presence of general functions

these general functions abstract common programming patterns

we have already seen examples of such generalizations : foldr, foldl, map, maptree

we go further

Functor -> Applicative -> Monad

class Functors

```
inc :: [Int] -> [Int]
inc []       = []
inc (n:ns) = n+1 : inc ns

map :: (a -> b) ->[a] –>[b]

map f []       = []
map f (x:xs) = f x : map f xs

inc = map (+1)
```

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

examples of Functor’s instances

instance Functor [] where
-- fmap :: (a -> b) -> [a] -> [b]
fmap = map

[a] is the same as [] a type list applied to the parameter type a

instances of Functor are always types T that have a type parameter another example:

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
--fmap :: (a -> b) -> Maybe a -> Maybe b
fmap   _ Nothing  = Nothing
fmap g  (Just x)     = Just (g  x)

fmap (+1) Nothing = Nothing

fmap (+1) Just 3 = Just 4

fmap not Just True = Just False
```

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
                     deriving Show

instance Functor Tree where
--fmap :: (a -> b) -> Tree a -> Tree b
fmap g (Leaf x)       = Leaf (g x)
fmap g (Node l  r)   = Node (fmap g l) (fmap g r)

fmap even Node (Leaf 1)  (Leaf 2)
Node (Leaf false) (Leaf True)
```

in general a Functor T a is a container that contains value of type a and fmap g applies g to each value in the container

```
instance Functor IO where
--fmap :: (a -> b) -> IO a -> IO b
fmap g mx = do
                x <- mx;
                return (g x)

fmap g mx = do { x <- mx; return (g x)}
```

example

the partially applied function type (a ->), represented as ((->) a), is a Functor:

```
fmap  :: (b -> c) -> (a -> b) -> (a -> c)
fmap = (.)
```

Functors => generalization

fmap can be used to process the values of any container (which is a Functor)

this generalization **propagates** to functions defined by mean of fmap

inc = fmap (+1)

inc (Just 1)   inc [1,2,3]   inc (Node (Leaf 1) (Leaf 2))

Functor laws

1) fmap id = id
2) fmap (g . h) = fmap g . fmap h

(1)states that fmap preserves the identity
(2) states that fmap preserves function composition

observe that fmap id :: f a -> f a, whereas id :: a -> a

together with the type of fmap, the laws guarantee that fmap is a mapping that does not reorder the values in the container that it is applied to a def. of fmap that does not obey the laws

```
instance Functor [] where
--fmap :: (a -> b) -> [a] -> [b ]
fmap g []       = []
fmap g (x: xs) = fmap g xs ++ [g x]
```

```
fmap id [1,2] = [2,1]
id [1,2] = [1,2]
```

In Chapter 16 we will see how one can show that a given Functor satisfies the Functor laws

interesting fact:
for any parameterized type in Haskell there is at most one function fmap that satisfies the laws

if we can make a parameterized type into a Functor (satisfying the laws), then fmap is unique

Applicatives (are Functors)

Functors map functions with one argument over containers,
Applicatives map functions with many parameters over containers

```
fmap0 :: a -> f a
fmap1 :: (a -> b) -> f a -> f b
fmap2 :: (a -> b -> c) -> f a -> f b -> f c
fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
..............
```

fmap2 (+) (Just 1) (Just 2) = Just 3

fmap :: (a -> b) -> f a -> f b
generalizes:
app :: (a -> b) -> a -> b
thanks to currying we don’t need app1, app2, app3,....
app1 g x = g x
app2 g x y =app1 (app1 g x) y
app3 g x y z =app1 (app1 (app1 g x) y) z

also here two basic functions are sufficient to define fmap0, 1, 2, 3,....

pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b

pure converts a value of type a into a structure f a
<*> is a generalized form of function application

g <*> x <*> y <*> x generalizes g x y z

typical use: pure g <*> x <*> y<*> z applicative style

```
fmap0 :: a -> f a
fmap0 = pure

fmap1 :: (a  ->b) -> f a -> f b
fmap1 g x = pure g <*> x

fmap 2 :: (a -> b -> c) -> f a -> f b -> f c
fmap2 g x y = pure g <*> x <*> y

fmap3 :: (a-> b -> c -> d) -> f a -> f b -> f c -> f d
fmap3 g x y z = pure g <*> x <*> y <*> z
```

```
g :: a -> b -> c -> d = (a -> (b -> (c -> d)))          observe types

pure g :: f (a -> (b -> (c -> d)))

(<*>) :: f (a -> B) -> f a -> f B    con B =(b ->(c -> d))

pure g <*> x  :: f B = f  (b ->(c -> d))

pure g <*> x <*> y :: f (c -> d)

pure g <*> x <*> y <*> z :: f d
```

definition

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

examples of Applicative:

```
instance Applicative Maybe where
--pure :: a -> Maybe a
pure = Just
--(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
Nothing <*> _ = Nothing
(Just g) <*> mx = fmap g mx
```

```
pure (+1) <*> Just 1
Just 2
pure (*) <*> Just 2 <*> Just 3
Just 6
pure (*) <*> Just 2 <*> Nothing
Nothing
```

lists are Applicatives

```
instance Applicative [] where
--pure :: a -> [a]
pure x = [x]
--(<*>) :: [a -> b] -> [a] -> [b]
gs <*> xs = [g x | g <- gs, x <- xs]
```

```
pure (+1) <*> [1,2,3]
[2,3,4]
[(*2),(+2)] <*> [1,2,3]
[2,4,6,3,4,5]
```

```
prods :: [Int] -> [Int] -> [Int]
prods [2,3] [3,4]
[6,8,9,12]
```

```
prods xs ys = pure (*) <*> xs <*> ys
```

xs and ys may be results of non-deterministic functions

```
failure = []
prods [2,3] [] = []
```

IO is Applicative

```
instance Applicative IO where
--pure :: a -> IO a
pure = return

--(<*>) :: IO (a -> b) -> IO a -> IO b
mg <*> mx = do {g <- mg; x <- mx; return (g x)}

getChars :: Int -> IO String
getChars 0 = []
getChars n = pure (:) <*> getChar <*> getChars  (n-1)
```