CSCE 314
Programming Languages
Functors, Applicatives, and Monads

Dr. Hyunyoung Lee
Motivation – Generic Functions

A common programming pattern can be abstracted out as a definition.

For example:

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

sqr :: [Int] -> [Int]
sqr [] = []
sqr (n:ns) = n^2 : sqr ns
```

Both functions are defined in the same manner!
Using map

\[
\begin{align*}
\text{inc} & : [\text{Int}] \rightarrow [\text{Int}] \\
\text{inc} \ [n] & = \ [] \\
\text{inc} \ (n:ns) & = \ n + 1 : \text{inc} \ ns \\
\text{sqr} & : [\text{Int}] \rightarrow [\text{Int}] \\
\text{sqr} \ [n] & = \ [] \\
\text{sqr} \ (n:ns) & = \ n^2 : \text{sqr} \ ns
\end{align*}
\]

\[\text{inc} = \text{map} \ (+1)\]

\[\text{sqr} = \text{map} \ (^2)\]
Functors

Class of types that support mapping of function. For example, lists and trees.

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

fmap takes a function of type (a->b) and a structure of type (f a), applies the function to each element of the structure, and returns a structure of type (f b).

Functor instance example 1: the list structure []

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

(f a) is a data structure that contains elements of type a
Functor instance example 2: the Maybe type

```haskell
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)
```

Now, you can do

```haskell
> fmap (+1) Nothing
Nothing
> fmap not (Just True)
Just False
```
Functor instance example: the Maybe type (Cont.)

Picture source:
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html
Functor instance example 3: the Tree type

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)

Now, you can do

> fmap (+1) (Node (Leaf 1) (Leaf 2))
Node (Leaf 2) (Leaf 3)
> fmap (even) (Node (Leaf 1) (Leaf 2))
Node (Leaf False) (Leaf True)
Benefits of Functors

1. \( \text{fmap} \) can be used to process the elements of any structure that is functorial.

2. Allows us to define generic functions that can be used with any functor.

Example: increment (\( \text{inc} \)) function can be used with any functor with \( \text{Int} \) type elements

\[
\text{inc} :: \text{Functor } f \Rightarrow f \text{ Int} \rightarrow f \text{ Int} \\
\text{inc} = \text{fmap (+1)} \\
> \text{inc } \text{(Just 1)} \\
\text{Just 2} \\
> \text{inc } [1,2,3] \\
[2,3,4] \\
> \text{inc (Node (Leaf 1) (Leaf 2))} \\
\text{Node (Leaf 2) (Leaf 3)}
\]
Want to be more flexible?

Functors abstract the idea of mapping a function over each element of a structure.

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

The first argument of `fmap` is a function that takes one argument, but we want more flexibility! We want to be able to use functions that take any number of arguments.

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

Only one argument function!
Applicative

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

The function `pure` takes a value of any type as its argument and returns a structure of type `f a`, that is, an applicative functor that contains the value.

The operator `<*>` is a generalization of function application for which the argument function, the argument value, and the result value are all contained in `f` structure.

` <*> ` associates to the left: `((g <*> x) <*> y) <*> z`

`fmap g x = pure g <*> x = g <$> x `
Applicative functor instance example 1: Maybe

```haskell
data Maybe a = Nothing | Just a

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) <*> Nothing
Nothing
> pure (+) <*> Just 2 <*> Just 3
Just 5
> mult3 x y z = x*y*z
> pure mult3 <*> Just 1 <*> Just 2 <*> Just 4
Just 8
```
Applicative functor instance example: Maybe (Cont.)

Picture source:
Instance Applicative [] where

-- pure :: a -> [a]
pure x = [x]

-- (<>*) :: [a -> b] -> [a] -> [b]
gs <*> xs = [ g x | g <- gs, x <- xs ]

pure transforms a value into a singleton list. 
<>* takes a list of functions and a list of arguments, and applies each function to each argument in turn, returning all the results in a list.

> pure (+1) <*> [1,2,3]
[2,3,4]
> pure (+) <*> [1,3] <*> [2,5]
[3,6,5,8]
> pure (:) <*> "ab" <*> ["cd","ef"]
["acd","aef","bcd","bef"]
Applicative functor instance example: `[]` (Cont.)

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

Roughly, a monad is a strategy for combining computations into more complex computations.

Another pattern of effectful programming (applying pure functions to (side-)effectful arguments)

(>>=) is called “bind” operator.

Note: return may be removed from the Monad class in the future, and become a library function instead.

```haskell
class (Applicative m) => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    return = pure
```
Monad instance example 1: Maybe

data Maybe a = Nothing | Just a

instance Monad Maybe where
  -- (>>=): Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing >>= _ = Nothing
  (Just x) >>= f = f x

> div2 x = if even x then Just (x \div\ 2) else Nothing

> (Just 10) >>= div2
  Just 5
> (Just 10) >>= div2 >>= div2
  Nothing
> (Just 10) >>= div2 >>= div2 >>= div2
  Nothing
Monad instance example 2: list type []

instance Monad [] where

-- (>>>=): [a] -> (a -> [b]) -> [b]
xs >>= f = [y | x <- xs, y <- f x]

pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = do x <- xs
    y <- ys
    return (x,y)

> pairs [1,2] [3,4]
[[(1,3),(1,4),(2,3),(2,4)]]