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:

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

Both functions are defined in the same manner!
inc :: [Int] -> [Int]
inc [] = []
inc (n:ns) = n + 1 : inc ns

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

Using map

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (n:ns) = f n : map f ns

inc = map (+1)
sqr = map (^2)
Class of types that support mapping of function. For example, lists and trees.

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

$fmap$ takes a function of type $(a \rightarrow b)$ 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 $[\ ]$

```haskell
instance Functor [] where
    -- fmap :: (a -> b) -> [a] -> [b]
    fmap = map
```
Functor instance example 2: the Maybe type

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

> 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)
Functor laws

1. \( \text{fmap id} = \text{id} \)
2. \( \text{fmap} (g \cdot h) = \text{fmap} g \cdot \text{fmap} h \)

1. \text{fmap} preserves the identity function

2. \text{fmap} also preserves the function composition, where \( g \) has type \( b \rightarrow c \) and \( h \) has type \( a \rightarrow b \)

3. The functor laws ensure that \text{fmap} does perform a mapping operation, without altering the natural property of the data structure.
Benefits of Functors

1. \texttt{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 (inc) function can be used with any functor with \texttt{Int} type elements

```haskell
inc :: Functor f => f Int -> f Int
inc = \texttt{fmap} (+1)

> inc \texttt{(Just } 1\texttt{)}
\texttt{Just } 2

> inc \texttt{[1,2,3]}
\texttt{[2,3,4]}

> inc \texttt{(Node \texttt{(Leaf } 1\texttt{) (Leaf } 2\texttt{))}}
\texttt{Node \texttt{(Leaf } 2\texttt{) (Leaf } 3\texttt{)}}
```
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
```
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: `((pure g <*> x) <*> y) <*> z`

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

\[
data \text{Maybe } a = \text{Nothing} \mid \text{Just } a
\]

\[
\text{instance Applicative Maybe where}
\]

\[
\quad \text{pure :: } a \rightarrow \text{Maybe } a
\]

\[
\quad \text{pure} = \text{Just}
\]

\[
\quad (\langle*\rangle) :: \text{Maybe } (a\rightarrow b) \rightarrow \text{Maybe } a \rightarrow \text{Maybe } b
\]

\[
\quad \text{Nothing } \langle*\rangle _ = \text{Nothing}
\]

\[
\quad (\text{Just } g) \langle*\rangle mx = \text{fmap } g \text{ mx}
\]

> pure (+1) \langle*\rangle \text{Nothing}
Nothing
>
> pure (+) \langle*\rangle \text{Just 2} \langle*\rangle \text{Just 3}
Just 5
>
> \text{mult3 } x y z = x*y*z
>
> pure \text{mult3} \langle*\rangle \text{Just 1} \langle*\rangle \text{Just 2} \langle*\rangle \text{Just 4}
Just 8
Applicative functor instance example: Maybe (Cont.)

![Diagram of Maybe applicative functor example]

1. Function wrapped in a context
2. Value in a context
3. Unwrap both and apply the function to the value
4. New value in a context

Picture source:
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html
Applicative functor instance example 2: list type []

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]

Picture source:
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html
Applicative laws

pure \text{id} <*> x = x
pure (g x) = pure g <*> pure x
x <*> pure y = pure (\g \rightarrow g y) <*> x
x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z

1. pure preserves the identity function
2. pure also preserves function application
3. When an effectful function is applied to a pure argument, the order in which the two components are evaluated does not matter.
4. The operator <*> is associative (modulo types that are involved).
Monads

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

- 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.
Monad instance example 1: Maybe

```haskell
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
```

```haskell
> (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)]
```
Monad laws

return \( x \) >>= f = f x \quad -- \text{left identity}

mx >>= \text{return} = mx \quad -- \text{right identity}

(mx >>= f) >>= g = mx >>= (\lambda x \rightarrow (f x >>= g))

1. If we return a value and then feed it into a monadic function, this should give the same result as simply applying the function to the value.

2. If we feed the result of a monadic computation into the function return, this should give the same result as simply performing the computation.

3. >>= is associative