Software System Design and Implementation

Functors, Applicative and Monads

Gabriele Keller

The University of New South Wales
School of Computer Science and Engineering
Sydney, Australia
Data constructors revisited

data Point = Point Float Float

Point :: Float -> Float -> Point
Point 0.0 1.25 :: Point
Data constructors revisited

```
data Shape  
  = Circle  Point Float  
  | Path  [Point]  

Circle :: Point -> Float -> Shape  
Path  :: [Point] -> Shape  
```
Data constructors revisited

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

```
Leaf ::
Node :: a -> (Tree a) -> (Tree a) -> (Tree a)
```
Data constructors revisited

data Either a b
  = Left  a
  | Right b

Left  :: a -> Either a b
Right :: b -> Either a b
Data constructors revisited

data Maybe a
  = Nothing
  | Just a

Nothing :: Maybe a
Just    :: a -> Maybe a
Type Constructors

- **Data constructors** map values to values:

```
Just :: a -> Maybe a
```

```
True
False
```

```
Just True
Just False
```

```
Bool
```

```
Maybe Bool
```

types a **sets of values**
Type Constructors

- **Type constructor** map types to type:

  \[ \text{Maybe} :: \ast \rightarrow \ast \]

  - `Char`
  - `Maybe Int`
  - `Float`
  - `Maybe Bool`
  - `Maybe Char`
  - `Maybe (Maybe Int)`
  - `Maybe Float`
  - `Maybe Bool`

  Kinds are **sets of types**
Kinds

- Char
- [Int]
- Bool
- Float
- ... Maybe Bool

- Either
- (,)

- Maybe
- Tree
- [ ]
- ... *

- * -> *

- * -> *

- * -> *
Generalising map

• map on lists:

\[
\begin{align*}
\text{map} & : (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{map } f \ [\ ] & = [\ ] \\
\text{map } f \ (x : xs) & = f \ x : \text{map } f \ xs
\end{align*}
\]

• map for other unary type constructors:

\[
\begin{align*}
\text{treeMap} & : (a \rightarrow b) \rightarrow \text{Tree } a \rightarrow \text{Tree } b \\
\text{treeMap } f \ \text{Leaf} & = \text{Leaf} \\
\text{treeMap } f \ (\text{Node } x \ \text{leftSubtree} \ \text{rightSubtree}) & = \text{Node } f \ x \ (\text{treeMap } f \ \text{leftSubtree}) \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (\text{treeMap } f \ \text{rightSubtree})
\end{align*}
\]
Generalising map

- map on the Maybe type:

```
maybeMap :: (a -> b) -> Maybe a -> Maybe b
maybeMap f Nothing  = Nothing
maybeMap f (Just x) = Just (f x)
```

```
fmap :: (a -> b) -> f a -> f b
```
Functors

• We have seen how type classes can be used to group types according to the operations supported on their values:

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

instance Eq Bool where
  (==) True  True  = True
  (==) False False = True
  (==) _     _     = False
  (/=) b1    b2    = not (b1 == b2)
```
Functors

• We can also use type classes to group type constructors:

\[ f:: * \rightarrow * \]

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

instance Functor Tree where
    fmap f Leaf  =  Leaf
    fmap f (Node a t1 t2)
        =  Node (f a) (fmap f t1) (fmap f t2)
```
What properties should map have?

• Should leave the structure intact:

\[
\begin{align*}
\text{fmap id } xs &= xs \\
\text{fmap (f . g) } xs &= ((\text{fmap f}) . (\text{fmap g})) \; xs
\end{align*}
\]

• These properties are not enforced by the compiler
  - it’s the programmers responsibility to ensure
  - these are quickcheckable properties, but proofs are often straight forward
  - these abstractions are very useful to understand code
Applicative

• Applicative are functors with two additional operations:

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

• Properties

\[
\begin{align*}
\text{pure } \text{id} & \llhd v = v \\
\text{pure } (\cdot) & \llhd u \llhd v \llhd w = u \llhd (v \llhd w) \\
\text{pure } f & \llhd \text{pure } x = \text{pure } (f \ x) \\
\text{pure } y & \llhd \text{pure } y = \text{pure } (\_ \ y) \llhd u
\end{align*}
\]
Monads

• Monads

```haskell
class Applicative m => Monad m where

    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
```
Monads

• Properties

\[
\begin{align*}
\text{return } a \gg= k &= k a \\
 m \gg= \text{return } &= m \\
 m \gg= (\lambda x \to k \ x \gg= h) &= (m \gg= k) \gg= h
\end{align*}
\]
Monads

- **Do-notation:**

```haskell
incMaybe :: Num a => Maybe a -> Maybe a
incMaybe (Just x) = Just (x + 1)
incMaybe _        = Nothing

incM mx          = mx >>= \x ->
                   return (x + 1)

addM mx my       = mx >>= \x ->
                   my >>= \y ->
                   return (x + y)

addM mx my = do
             x <- mx
             y <- my
             return (x + y)
```