CMPT 383
Applicatives

Anders Miltner
Applicatives

- Applicatives for filling a gap left by Functors
- Provide better capabilities for multiple arguments
Where does fmap fall short

- fmap only works on unary functions
- Is this an issue? — remember, there’s partial application!
Extended example: Maybe and ++

- I have two elements of type Maybe String
  - \( h = \text{Just "Hello "} \)
  - \( s = \text{Just "World!"} \)
- I want to concatenate the values to get \( hs = \text{Just "Hello World"} \)
  - If either \( h \) or \( s \) is Nothing, just return Nothing
Remember Partial Application

- \( ++ :: [a] \rightarrow [a] \rightarrow [a] \)
- \( pa = \text{fmap} \ (++) \ (\text{Just} \ \text{"Hello"}) \)
- \( pa = \text{Just} \ (\lambda x \rightarrow \text{"Hello"} \ ++ x) \)
Now what?

• pa :: Maybe (String -> String)
  pa = Just (\x -> “Hello” ++ x)

• w = Maybe String
  w = Just “World”

• ???
We need something more

- It should have at least the capabilities of a Functor
- But it should have more — we want to be able to sequence computations
  - Intermediate step before we get to Monads
- This is what’s known as an “Applicative Functor”
Applicative Functor

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

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

- Identity: `pure id <*> v = v`
- Composition: `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- Homomorphism: `pure f <*> pure x = pure (f x)`
- Interchange: `u <*> pure y = pure ($ y) <*> u`
Why should all Applicatives be Functors

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

fmap :: (a -> b) -> (f a -> f b)
fmap f x = pure f <*> x
Example Instantiations: Maybe

```haskell
instance Applicative Maybe where
    pure f = Just f

    (<>*) Nothing _       = Nothing
    (<>*) _    Nothing    = Nothing
    (<>*) (Just f) (Just x) = Just (f x)
```

```
Maybe (String -> String)                       Maybe (String -> String)
Maybe (String -> (String -> String))         Maybe (String -> String)

pure (++) (<>*) (Just “Hello ”) (<>*) (Just “World!”) = Just “Hello World!”
```
Example Instantiations: [a]

```haskell
instance Applicative [] where
  pure f = [f]

  <*> fl xl = concat (fmap (\f -> fmap f xl) fl)
```

```haskell
pure (++) <*> ["Hello ","GoodBye "] <*> ["World!","Class!","it’s me!"]
["Hello World!","Hello Class!", "Hello it’s me!",
,"GoodBye World!","GoodBye Class!", "GoodBye it’s me!"]
```
**Example Instantiations: ZipList**

```haskell
newtype ZipList a = ZipList [a]

instance Applicative ZipList where
    pure f = ZipList [f..]

    (ZipList lf) (ZipList lx) = ZipList (helper lf lx)
    where helper [] _ = []
    helper _ [] = []
    helper (hf:tf) (hx:tx) = (hf hx):(helper tf tx)
```

```haskell
pure (++) <*> ["Hello ",{"GoodBye "] <*> ["World!",{"Class!",{"it’s me!"}]
["Hello World!",{"Goodbye Class!"]
```
Are all Functors Applicatives

- No!

- data Either a b = Left a | Right b