Paradigmas de Programação

Week 10
Applicative Functors

Alexandra Mendes

Applicative Functors

- Functors: useful concept for values that can be mapped over. For mapping functions over functors, we used functions that take only one parameter.
- But what happens when we map a function that takes two parameters over a functor?

```
>:t fmap (++) (Just 'a')
>:t fmap (++) ["a","b"]
>:t fmap (*) [1,2,3]
>let f = fmap (*) [1,2,3]
>fmap (\x -> x 10) f
```

Applicative Functors

- Functors: useful concept for values that can be mapped over. For mapping functions over functors, we used functions that take only one parameter.
- But what happens when we map a function that takes two parameters over a functor?

```
>:t fmap (++) (Just 'a')
>:t fmap (++) ["a","b"]
>:t fmap (*) [1,2,3]
>let f = fmap (*) [1,2,3]
>fmap (\x -> x 10) f
```

What if we want to apply a function that is inside a functor to a value that is also inside a functor? For example, [1*] to [2]?

Applicative Functors

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

- pure takes a value of any type and returns an applicative functor with that value inside it (puts it in the default/minimal context).
- <*> takes a functor that has a function in it and another functor and (sort of) extracts that function from the first functor and then maps it over the second one.
- If a type constructor is part of the Applicative typeclass, it's also in Functor, so we can use fmap on it.

Example of Applicative instance

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

Example of Applicative instance

Examples:

```
> Just (*3) <*> Just 9
Just 27
> [(*1),(*2)] <*> [9]
[9,18]
> [(*1),(*2)] <*> pure 9
[9,18]
> pure (+3) <*> Just 10
Just 13
> Just (++"I love") <*> Nothing
Nothing
> Nothing <*> Just "Programming"
Nothing
>Just (++"Programming") <*> pure "I love "
Just "I love Programming"
```

Applicative Functor

What's happening here?

```
> pure (+) <*> Just 6 <*> Just 4
Just 10
> pure (+) <*> Just 6 <*> Nothing
Nothing
> pure (+) <*> Nothing <*> Just 4
Nothing
```

Applicative Functor

What's happening here?

```
> pure (+) <*> Just 6 <*> Just 4
Just 10
> pure (+) <*> Just 6 <*> Nothing
Nothing
> pure (+) <*> Nothing <*> Just 4
Nothing
```

- Allows us to apply functions that expect parameters that are not wrapped in functors and use that function to operate on several values that are in functor contexts.
- The function can take as many parameters as we want, because it's always partially applied step by step between occurences of `<*>`.
- `pure f <*> x` equals `fmap f x`

Applicative Functor

`pure f <*> x` equals `fmap f x`

Putting a function in a default context and then extract and apply it to a value inside another applicative functor is the same as we did the same as just mapping that function over that applicative functor.

Thus, writing

`pure f <*> x <*> y <*> ...`

has the same effect as writing

`fmap f x <*> y <*> ...`

Applicative Functor

Control.Applicative exports the function

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

`<$>` is just `fmap` as an infix operator.

```
> (++) <$> Just "Simply" <*> Just " the best"
Just "Simply the best"
```

Applicative Functors: List

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

```
> [(+1),(+2),(+3)] <*> [2]
[3,4,4]
> [(+1),(+2),(+3)] <*> [1,2,3]
[2,3,4,3,4,5,4,5,6]
> [(^10),(3-),(*12)] <*> [2]
[1024,1,24]
> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
```

Applicative Functors: List

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

Has the same effect as:

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

Applicative Functors: List

Using the applicative style on lists is often a good replacement for list comprehensions:

```
> [ x*y | x <- [1,2,3], y <- [10,20,30]]
[10,20,30,20,40,60,30,60,90]
```

In applicative style:

```
> (*) <$> [1,2,3] <*> [10,20,30]
[10,20,30,20,40,60,30,60,90]
```

Applicative Functor: IO

```
instance Applicative IO where
    pure = return
    a <*> b = do
        f <- a
        x <- b
        return (f x)
```

Examples (all equivalent):

```
> (++) <$> getLine <*> getLine
> return (++) <*> getLine <*> geLine
> pure (++) <*> getLine <*> geLine
```

Exercise

Given the following type of expressions

```
data Expr a = Var a | Val Int | Add (Expr a) (Expr a)
                deriving Show
```

that contain variables of some type a, show how to make this type into an instance of the Applicative classes.

Solution

```
data Expr a = Var a | Val Int | Add (Expr a) (Expr a)
              deriving Show

instance Applicative Expr where
-- pure :: a -> Expr a
pure = Var
-- <*> :: Expr (a -> b) -> Expr a -> Expr b
Var g <*> e = fmap g e
Val n <*> e = Val n
(Add l r) <*> e = Add (l <*> e) (r <*> e)
```

The Functor: Expr

```
instance Functor Expr where
-- fmap :: (a -> b) -> Expr a -> Expr b
  fmap f (Var a) = Var (f a)
  fmap f (Val n) = Val n -- f takes values of type a,
                         -- not Int.
  fmap f (Add l r) = Add (fmap f l) (fmap f r)
```

The Monad: Expr

```haskell
instance Monad Expr where
  -- >>-= :: Expr a -> (a -> Expr b) -> Expr b
  (Var x) >>= g = g x
  (Val n) >>= g = Val n
  (Add l r) >>= g = Add (l >>= g) (r >>= g)
```

Exercise

The default Applicative for lists combines every function with every value, e.g.:

```
[f,g] <*> [x1,x2,x3]
   = [f x1, f x2, f x3, g x1, g x2, g x3]
```

Another way to make lists an applicative functor is to combine functions and values pointwise, e.g.:

```
[f,g,h] <*> [x1,x2,x3] = [f x1, g x2, h x3]
```

The library `Control.Applicative` provides a newtype `ZipList` for which the function pure makes an infinite list of copies of its argument, and the operator `<*>` applies each argument function to the corresponding argument value at the same position.

Exercise

Consider the type:

```
newtype ZipList a = Z [a] deriving Show
```

Declare ZipList as an instance of Functor and Applicative.
Hint: use the `zip` function.

Solution

```
instance Functor ZipList where
-- fmap :: (a -> b) -> ZipList a -> ZipList b
fmap g (Z xs) = Z (fmap g xs)

instance Applicative ZipList where
-- pure :: a -> ZipList a
pure x  = Z (repeat x)
-- <*> :: ZipList (a -> b) -> ZipList a -> ZipList b
(Z gs)  <*> (Z  xs) = Z [gs | (x,y) <- zip gs xs]
```

The ZipList wrapper around the list type is required because each type can only have at most one instance declaration for a given class.