

Functor and Applicative

Dr. Mattox Beckman

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
DEPARTMENT OF COMPUTER SCIENCE

Objectives

- ▶ Implement the Functor and Applicative type classes for a user-defined type.
- ▶ Use the Functor and Applicative type classes to generalize the `map` function.

Motivation

Example Types

```
1 data Tree a = Node a [Tree a]
2 data Maybe a = Just a | Nothing
```

- Suppose we want to write the `map` function for these types. What will they look like?

The Functor Typeclass

The Functor Typeclass

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

- ▶ You can use this to define a map for many different types.
- ▶ The `f` type you pass in must be a parameterized type.

Examples

```
1 instance Functor Maybe where
2   fmap f (Just x) = Just (f x)
3   fmap _ Nothing  = Nothing
4 instance Functor [] where
5   fmap f [] = []
6   fmap f (x:xs) = f x : fmap f xs
```

Why This is Useful

- ▶ If you define a type and declare it to be a `Functor`, then other people can use `fmap` on it.
- ▶ You can also write functions that use `fmap` that can accept any `Functor` type.

Using Functor

```
1 Main> let incAnything x = fmap (+1) x
2 Main> incAnything [10,20]
3 [11,21]
4 Main> incAnything (Just 30)
5 Just 31
6 Main> incAnything (Foo 30)
7 Foo 31
```

Applicative Functors

- ▶ We can take this up one level.

The Applicative Typeclass

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

- ▶ The <*> operator 'lifts' function applications.

Declaring Our Own Applicative

Complete Foo

```
1 import Control.Applicative
2
3 data Foo a = Foo a
4
5 instance Show a => Show (Foo a) where
6   show (Foo a) = "Foo " ++ show a
7
8 instance Functor Foo where
9   fmap f (Foo a) = Foo $ f a
10
11 instance Applicative Foo where
12   pure a = Foo a
13   (Foo f) <*> (Foo x) = Foo $ f x
```

Sample Run

```
1 Main> let inc = (+1)
2 Main> fmap inc (Foo 30) -- fmap works
3 Foo 31
4 Main> inc <$> (Foo 30) --- synonym for fmap
5 Foo 31
6 Main> Foo inc <*> Foo 20 -- (Foo f) <*> (Foo a) = (Foo (f a))
7 Foo 21
8 Main> let plus a b = a + b
9 Main> :t plus <$> (Foo 20)
10 plus <$> (Foo 20) :: Num a => Foo (a -> a)
```

- Do you remember the type of <*>?

Applicatives

```
1 Main> let plus a b = a + b
2 Main> :t plus <$> (Foo 20)
3 plus <$> (Foo 20) :: Num a => Foo (a -> a)
4 Main> plus <$> (Foo 20) <*> (Foo 30)
5 Foo 50
```

- ▶ Note that plus did not have to know about Foo.
- ▶ Note also that Foo did not have to know about Applicative.
- ▶ If we can define pure and <*> and fmap for it, we can use this trick.

Details

- There are some laws that applicatives are supposed to obey.

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`

- Haskell does not enforce these.

Credit

- ▶ Many of the examples were stolen off the Haskell Wikibooks page.