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

```haskell
data Tree a = Node a [Tree a]
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

```
class Functor f where
    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

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap _ Nothing  = Nothing
instance Functor [] where
    fmap f [] = []
    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

```
Main> let incAnything x = fmap (+1) x
Main> incAnything [10,20]
[11,21]
Main> incAnything (Just 30)
Just 31
Main> incAnything (Foo 30)
Foo 31
```

Applicative Functors

- We can take this up one level.

The Applicative Typeclass

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

- The <*> operator ‘lifts’ function applications.

Declaring Our Own Applicative

Complete Foo

```
import Control.Applicative

data Foo a = Foo a

instance Show a => Show (Foo a) where
  show (Foo a) = "Foo " ++ show a

instance Functor Foo where
  fmap f (Foo a) = Foo $ f a

instance Applicative Foo where
  pure a = Foo a
  (Foo f) <*> (Foo x) = Foo $ f x
```

Sample Run

```
Main> let inc = (+1)
Main> fmap inc (Foo 30) -- fmap works
Foo 31
Main> inc <$> (Foo 30) --- synonym for fmap
Foo 31
Main> Foo inc <*> Foo 20  -- (Foo f) <*> (Foo a) = (Foo (f a))
Foo 21
Main> let plus a b = a + b
Main> :t plus <$> (Foo 20)
plus <$> (Foo 20) :: Num a => Foo (a -> a)
```

- Do you remember the type of <*>?

Applicatives

```
Main> let plus a b = a + b
Main> :t plus <$> (Foo 20)
plus <$> (Foo 20) :: Num a => Foo (a -> a)
Main> plus <$> (Foo 20) <*> (Foo 30)
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.