

CS 252:

Advanced Programming Language Principles



Applicative Functors

Prof. Tom Austin

San José State University

Review: what is a functor?

A functor is something
that can be mapped over.

Functor lab review

Examples of functors?

- Lists
- Maybe values
- Either values
- IO (a little oddly)

Limits of functors

With functors:

```
fmap (+1) [1, 2, 3]
```

But what does this code do?

```
fmap (+) [1, 2, 3]
```

And how can we make use of the result?

Motive: adding Maybe values

```
addMaybe m1 m2 = case (m1, m2) of
  ((Just x), (Just y)) -> Just (x+y)
  (_, Nothing) -> Nothing
  (Nothing, _) -> Nothing
```

```
main = do
  print $ addMaybe (Just 3) (Just 4)
```

Motive: adding Maybe values

```
addMaybe m1 m2 = case (m1, m2) of  
  ((Just x), my) -> fmap (+) x my
```

```
  (Nothing, _) -> Nothing
```

```
main = do
```

```
  print $ addMaybe (Just 3) (Just 4)
```

Motive: adding Maybe values

```
addMaybe m1 m2 = fmap (+) m1 <*> m2
```

???

```
main = do
```

```
  print $ addMaybe (Just 3) (Just 4)
```

```
> let jf = Just (\x -> x + 1)
```

```
> let jf = fmap (+) (Just 1)
```

```
> jf (Just 3)
```

Equivalent
definition

[Some long error about types]

```
> import Control.Applicative
```

```
> jf <*> (Just 3)
```

```
Just 4
```

Control.Applicative

- `pure` boxes up an item.
- `<*>`
 - infix operator similar to `fmap`.
 - the function itself is in a functor.

What is pure?

pure wraps an item in the base functor.

```
> pure 7
```

```
7
```

```
> pure 7 :: [Int]
```

```
[7]
```

```
> pure 7 :: Maybe Int
```

```
Just 7
```

Applicative functors

```
class (Functor f) => Applicative f
```

where

```
pure :: a -> f a
```

```
(<*>) :: f (a -> b) -> f a  
-> f b
```

The function is
inside a functor,
unlike fmap

Applicative Maybe

```
instance Applicative Maybe
```

```
  where
```

```
    pure = Just
```

```
    Nothing <*> _ = Nothing
```

```
    (Just f) <*> x = fmap f x
```

```
> Just (+3) <*> Just 4
```

```
Just 7
```

```
> pure (+3) <*> Nothing
```

```
Nothing
```

```
> pure (+) <*> Just 3 <*> Just 4
```

```
Just 7
```

```
> (+) <$> Just 3 <*> Just 4
```

```
Just 7
```



Infix
operator for
fmap

Applicative []

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

```
> (*) <$> [1,2,3] <*> [1,0,0,1]
```

```
[1,0,0,1,2,0,0,2,3,0,0,3]
```

```
> pure 7 :: [Int]
```

```
[7]
```

Applicative IO

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

Applicative IO in action

```
import Control.Applicative
```

```
main = do
```

```
  a <- (++) <$> getLine <*> getLine
```

```
  putStrLn a
```

liftA2

The `liftA2` function lets us apply a normal function to two functors more easily.

```
liftA2 :: (Applicative f) =>
  (a -> b -> c)
  -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

liftA2 example

```
> (:) <$> Just 3 <*> Just [4]  
Just [3,4]
```

```
> liftA2 (:) (Just 3) (Just [4])  
Just [3,4]
```

```
> (+) <$> Just 3 <*> Just 4  
Just 7
```

```
> liftA2 (+) (Just 3) (Just 4)
```

Applicative functor laws

$$1. \text{pure } f \langle * \rangle x = \text{fmap } f \ x$$

$$2. \text{pure } \text{id} \langle * \rangle v = v$$

$$3. \text{pure } (.) \langle * \rangle u \langle * \rangle v \langle * \rangle w = \\ u \langle * \rangle (v \langle * \rangle w)$$

$$4. \text{pure } f \langle * \rangle \text{pure } x = \text{pure } (f \ x)$$

$$5. u \langle * \rangle \text{pure } y = \text{pure } (\$ \ y) \langle * \rangle u$$

Monoids

An associative binary function & a value that acts as an identity with respect to that function.

`1 * x`

`x + 0`

`lst ++ []`

Monoids

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
  mconcat :: [m] -> m
  mconcat = foldr mappend mempty
```

Monoid Rules

$$1. \text{empty} \text{ `mappend` } x = x$$

$$2. x \text{ `mappend` } \text{empty} = x$$

$$3. (x \text{ `mappend` } y) \text{ `mappend` } z = x \text{ `mappend` } (y \text{ `mappend` } z)$$

Lab: Applicative Functors

Faceted values hide secrets by showing different “views” of data based on the observer’s authority.

This lab uses applicative functors to track these differing views. The starter code describes the problem in more detail.

Faceted Values Capture Multiple Views

```
var secret = truebank;  
var y = true;  
if (secret)  
    y = false;
```

```
y = <bank?false:true>
```

A faceted value captures that

$y = \text{false}$ for bank

$y = \text{true}$ for $\neg\text{bank}$

```
var x = 0;  
if (<bank?true:false>  
    x = 40 + 2;  
else x = 37 - 21;
```

→ x = <bank?42:16>

```
var f = <bank?add1:sub1>;  
var r = f(4);
```

→ r = <bank?5:3>