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?

```
> let jf = Just (\x -> x + 1)
> let jf = fmap (+) (Just 1)
> jf (Just 3)
  [Some long error about types]
> import Control.Applicative
> jf <*> (Just 3)
Just 4
```

Equivalent definition

`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

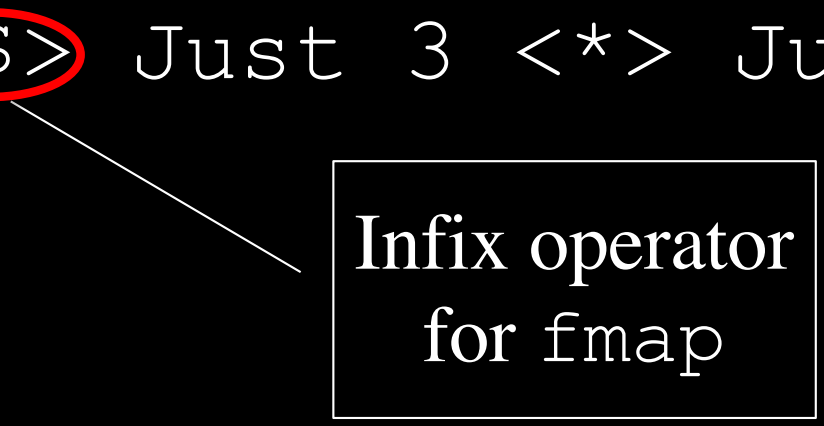
```
class (Functor f) => Applicative f
  where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a
                        -> f b
```

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
```



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.pure f <*> x = fmap f x
2.pure id <*> v = v
3.pure (.) <*> u <*> v <*> w =
                    u <*> (v <*> w)
4.pure f <*> pure x = pure (f x)
5.u <*> pure y = pure ($ y) <*> 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. `mempty `mappend` x = x`
2. `x `mappend` mempty = x`
3. `(x `mappend` y) `mappend` z =`
   `x `mappend` (y `mappend` z)`

Lab: Applicative Functors

Details in Cavas.