Why teaching functional programming to undergraduates at CUNY is important

Evan Misshula

2018-03-28

What if we map a multi-parameter function over a functor?

- Look at the type signature

```
a = fmap (*) [1..4]
:t a
fmap (\f -> f 9) a
1==1
```

```
a :: (Num a, Enum a) => [a -> a]
[9,18,27,36]
```

What if we want to take a function out of a Just

Let's take a Just (3 *) and map

and map it over Just 5

```
:set +m
:{
class (Functor f) => Applicative f where
    pure :: a -> f a;
    (<*>) :: f (a -> b) -> f a -> f b
:}
```

Maybe Applicative

Let's look at the Applicative for Maybe

```
:set +m
:{
instance Applicative MyMaybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
:}
```

Maybe Applicative inside the repl

Using the Maybe Applicative

```
-- :add Control.Applicative
Just (+3) <*> Just 9
pure (*2) <*> Just 10
pure (+3) <*> Just 9
Just (++"!!") <*> Just "Go now"
Nothing <*> Just "woot"
1==1



<interactive>:587:1: error:
    • Could not deduce (Applicative Maybe) arising from a use o
      from the context: Num b
        bound by the inferred type of it :: Num b => Maybe b
        at <interactive>:587:1-20
    • In the expression: Just (+ 3) <*> Just 9
      In an equation for ‘it’: it = Just (+ 3) <*> Just 9
```

Fmap as an infix operator

Control.Applicative exports a function called <$>

which is fmap as an infix operator

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

Compare Applicatives in the repl

Infix fmap in the repl

```
(++) <$> Just "John " <*> Just "Travolta"
(++) "John " "Travolta"
1==1

<interactive>:597:1: error:
    • No instance for (Applicative Maybe) arising from a use of
    • In the expression: (++) <$> Just "John " <*> Just "Travol
      In an equation for ‘it’:
          it = (++) <$> Just "John " <*> Just "Travolta"
John Travolta
```

Lists are Applicative Functors

Definition (Definition of the Applicative for a list)

- Literally a Cartesian product of functions and list values

```
:set +m
:{
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x<- xs]
:}
```

Applicative Functors of lists in the repl

Applicative Functors of lists in the repl

```
[(*0),(+100),(^2)] <*> [1..4]
[(+),(*)] <*>[1,2]<*>[3,4]
(++) <$> ["ha","heh","hmm"] <*> ["?","!","."]
1==1

[0,0,0,0,101,102,103,104,1,4,9,16]
[4,5,5,6,3,4,6,8]
["ha?","ha!","ha.","heh?","heh!","heh.","hmm?","hmm!","hmm."]
```

IO is an Applicative

Let's see how the IO Applicative is implemented:

```
:set +m
:{
instance Applicative IO where
    pure = return
    a <*> b = do
f <- a
x <- b
return (f x)
:}
```

Concatenating IO strings

Two ways to concatenate two lines of user input string

- Imperative code

```
:set +m
:{
myAction :: IO String
myAction = do
    a <- getLine
    b <- getLine
    return $ a ++ b
:}
```

Applicative way to concatenate two lines of user input string

- Applicative code

```
:set +m
:{
myAction :: IO String
myAction = (++)
    <$> getLine
    <*> getLine
:}
```

The first Applicative Functor Law

Theorem (The first Applicative Functor Law)

- *pure f <*> x = fmap f x*