Functors

• Just like category theory, Haskell has functors
  • All Haskell functors are category theoretic functors, but not vice-versa
  • Functors are very useful, and some of you have already used them!
Functor Class

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

  fmap id = id
  fmap (f . g) = fmap f . fmap g

High-level: turns functions on generics onto functions in the functor
(over the same generics)
Highest-Level View

- Functors contain data
- You can update the data, while maintaining the same container “shape”
Maybe Functor Instance

instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)

x = Just 2

case x of
    Nothing -> Nothing
    Just v -> Just (v+1)

fmap (\v. v+1) x
instance Functor List where
  fmap f [] = []
  fmap f (h:t) = (f h):(fmap f t)
instance Functor Tree where
    fmap f Leaf = Leaf
    fmap f Node(t1,a,t2) =
        Node(fmap f t1, f a, fmap f t2)
Pair Functor Instance

data Pair f g x = Pair (f x, g x)

instance (Functor g, Functor h) => Functor (Pair g h) where
  fmap :: (Functor g, Functor h) => (a -> b) -> Pair g h a -> Pair g h b
  fmap f (Pair (x, y)) = Pair (fmap f x, fmap f y)

y = Pair ([2], Node (Leaf, 3, Leaf))
:t y = Pair List Tree Int
instance Functor ((->) v) where
  fmap :: (a -> b) -> (v -> a) -> (v -> b)
  fmap f g = (f . g)
  fmap f g x = f(g(x))

fmap (\x. x+1) (\l. length l)

fmap (\x. show x) (\l. length l)

fmap show length
Is everything a functor?

• You can make products of things functors
• You can make sums of things functors
• You can make arrows functors
• Is everything a functor?
No

- T1 -> T2 is only a functor on the second value
- There is not a functor for the first value
Let’s try anyway

```haskell
newtype FlippedArrow a b = FlippedArrow (b -> a)

instance Functor (FlippedArrow v) where
  fmap :: (a -> b) -> (a -> v) -> (b -> v)
  fmap f g = ???
```
There is something fundamentally different about inputs to functions

• It’s generally pretty easy to be a functor

• Just apply the provided function to everywhere an “a” value shows up

• But we can’t do this to inputs!

• This is because being on the left of an arrow is contravariant
Is there a Functor-like thing?

class ContraFunctor f where
    fmapcontra :: (a -> b) -> f b -> f a

instance ContraFunctor (FlippedArrow v) where
    fmapcontra :: (a -> b) -> (b -> v) -> (a -> v)
    fmapcontra f (FlippedArrow g) = FlippedArrow (g . f)
Contravariance

- Shows up elsewhere in programming languages
- Contravariance appears in subtyping as well
  - In vs Out Params in C#
- Contravariance of inputs is why if T is a subtype of U, then it is not the case that List<T> is not a subtype of List<U>
- Just like negative of negative is positive, inputs to higher-order functions become contra-of-contravariant = covariant again
Takeaways

• Functors show up everywhere

• Almost everything can be turned into a functor

• The only time things can’t be turned into functors is when the relevant type variable shows up in a contravariant position