Class 8: Functor, Foldable

March 19

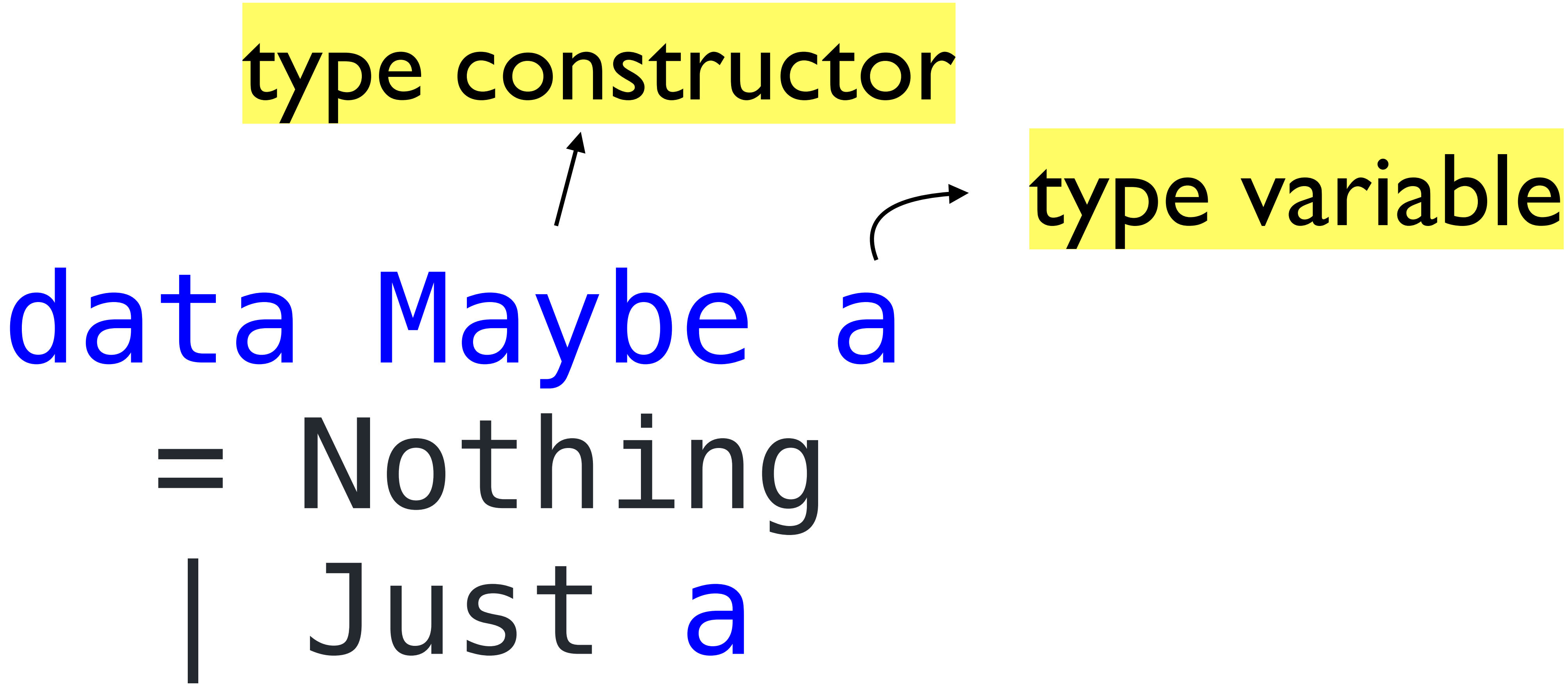
Q: how can I get better at finding and using library functions?

questions from homework

Q: how does “deriving” work? are there limitations?

questions from homework

generalizing map



type constructor

type variable

```
data Maybe a
  = Nothing
  | Just a
```

some polymorphic data types

```
data List a
  = Nil
  | Cons a (List a)
```

some polymorphic data types

```
data Tree a
  = Leaf
  | Branch (Tree a) a (Tree a)
```

some polymorphic data types

```
map       :: (a -> b) -> [a] -> [b]

treeMap   :: (a -> b) -> Tree a -> Tree b

maybeMap  :: (a -> b) -> Maybe a -> Maybe b
```

suspiciously similar type signatures

```
thingMap :: (a -> b) -> f a -> f b
```

generalizing…

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

…using a type class

a digression into kinds

```
ghci> :type 3
Int
```

(you can follow along in GHCi)

types have types too!
they're called *kinds*.

```
ghci> :kind Int
Int :: *

ghci> :kind Bool
Bool :: *

ghci> :kind Char
Char :: *
```

(you can follow along in GHCi)

```
data Maybe a
```

```
ghci> :kind Maybe Int
Maybe Int :: *

ghci> :kind Maybe
Maybe :: * -> *
```

(you can follow along in GHCi)

```
data List a
```

```
ghci> :kind [Int]
[Int] :: *

ghci> :kind []
[] :: * -> *
```

(you can follow along in GHCi)

some more conceptual examples…

```
Prelude> :k (->)
(->) :: * -> * -> *

Prelude> :k (->) Int Char
(->) :: *

Prelude> :k Int -> Char
(->) :: *
```

```
data Funny f a = Foo a (f a)
```

```
Prelude> :k Funny
Funny :: (* -> *) -> * -> *

Prelude> :k Funny Maybe
Funny Maybe :: * -> *
```

generalizing map

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

```
instance Functor Int where
  fmap = …
```

```
error:
Expected kind ‘* -> *’,
but ‘Int’ has kind ‘*’
```

non-example

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

```
instance Functor Maybe where
  fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap _ Nothing = Nothing
  fmap f (Just a) = Just (f a)
```

examples

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

```
instance Functor [] where
  fmap :: (a -> b) -> [a] -> [b]
  fmap _ [] = []
  fmap f (x : xs) = f x : fmap f xs
```

examples

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

```
instance Functor [] where
  fmap :: (a -> b) -> [a] -> [b]
  fmap = map
```

examples

(exercise: Functor instance for Tree + add3Tree)

generalizing fold

```
listFold :: (a -> b -> b) -> b -> [a] -> b

treeFold :: (a -> b -> b) -> b -> Tree a -> b
```

suspiciously similar type signatures

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
```

yet another type class

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
```

```
instance Foldable [] where
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr _ z [] = z
  foldr f z (x : xs) = f x (foldr f z xs)
```

examples

(exercise: Foldable instance for Tree + toListTree)

```haskell
any :: (a -> Bool) -> [a] -> Bool
any f = foldr ((||) . f) False
any :: (a -> Bool) -> [a] -> Bool
```

generalizes to

```
any :: Foldable t => (a -> Bool) -> t a -> Bool
```

```
elem :: Eq a => a -> [a] -> Bool
elem e = any (== e)
elem :: Eq a => a -> [a] -> Bool
```

generalizes to

```
elem :: (Foldable t, Eq a) => a -> t a -> Bool
```

```
sum :: [Int] -> Int
sum = foldr (+)
```

```haskell
sum :: [Int] -> Int
```

generalizes to

```haskell
sum :: (Foldable t, Num a) => t a -> a
```

(toList example)

today’s type classes

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

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
```