Functors and
Type Constructor Classes

Outline

- **Kinds, types, and type constructors**
- Functor and Foldable
- Type classes for abstraction
- Applicative functors

Kinds

Like a super-simple type system for types *(only keeps track of arity)*

A regular type has kind *

```
String      Int      [Bool]      Maybe Int

Tree a      Map k v      Int -> Int      a -> a -> Bool
```

A *type constructor* takes one or more *types* as arguments and produces a type

```
Maybe :: * -> *      Tree :: * -> *      [] :: * -> *

Map :: * -> * -> *      (->) :: * -> * -> *
```

Haskell functions and values always have regular types!

Outline

- Kinds, types, and type constructors
- **Functor and Foldable**
- Type classes for abstraction
- Applicative functors

Functor *class of data structures that can be mapped over*

type constructor!

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

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs

tmap :: (a -> b) -> Tree a -> Tree b
tmap _ Leaf          = Leaf
tmap f (Node x l r) = Node (f x) (tmap f l) (tmap f r)

mmap :: (a -> b) -> Maybe a -> Maybe b
mmap _ Nothing  = Nothing
mmap f (Just x) = Just (f x)
```

Functor *class of data structures that can be mapped over*

type constructor!

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

```
instance Functor [] where
  fmap _ []     = []
  fmap f (x:xs) = f x : fmap f xs

instance Functor Tree where
  fmap _ Leaf          = Leaf
  fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)

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

Functor laws

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

Equations that every Functor instance should satisfy:

```
fmap id  <==>  id
```

```
fmap (f . g)  <==>  fmap f . fmap g
```

Means that `fmap` preserves the *structure* of values

… code written against the Functor interface can assume this

Foldable *class of data structures that can be accumulated over*

type constructor!

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

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

tfold :: (a -> b -> b -> b) -> b -> Tree a -> b
tfold _ b Leaf    = b
tfold f b (Node x l r) = f x (tfold f b l) (tfold f b r)
```

Accumulator functions have different types

… can we refactor **`tfold`** *to make them the same?*

Foldable *class of data structures that can be accumulated over*

type constructor!

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

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

tfold :: (a -> b -> b -> b) -> b -> Tree a -> b
tfold _ b Leaf   = b
tfold f b (Node x l r) = f x (tfold f b l) (tfold f b r)

tfoldr :: (a -> b -> b) -> b -> Tree a -> b
tfoldr _ b Leaf   = b
tfoldr f b (Node x l r) = tfoldr f (f x (tfoldr f b r)) l
```

Foldable *class of data structures that can be accumulated over*

type constructor!

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

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

tfold :: (a -> b -> b -> b) -> b -> Tree a -> b
tfold _ b Leaf   = b
tfold f b (Node x l r) = f x (tfold f b l) (tfold f b r)

instance Foldable Tree where
  foldr _ b Leaf   = b
  foldr f b (Node x l r) = foldr f (f x (foldr f b r)) l
```

Outline

- Kinds, types, and type constructors
- Functor and Foldable
- **Type classes for abstraction**
- Applicative functors

Type classes as an abstraction mechanism

abstraction: *to separate a concept from its specific instances and make it reusable*

- **higher-order functions**
  names and makes reusable the *implementation* of high-level programming patterns for working with a *single data structure*

- **type classes**
  names and makes reusable the *interface* of high-level programming patterns for working with a *variety of data structures*

Why abstract these shared interfaces?

- ***reuse* functions over *classes* of data types**
  by instantiating Foldable or (especially) Monad
  you have access to tons of *library functions*

- **write code that is *extensible* with *new* data types**
  describe the *interface* of a data type you expect
  then program against that interface

- ***describe precisely* the properties of your data type**
  type classes induce a *classification scheme* for data types

Outline

- Kinds, types, and type constructors
- Functor and Foldable
- Type classes for abstraction
- **Applicative functors**

Applicative functors *functors that support application*

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

inject → `pure`

apply → `(<*>)`

```
instance Applicative Maybe where
  pure = Just
  Just f <*> Just x = Just (f x)
  _      <*> _      = Nothing

instance Applicative [] where
  pure x = [x]
  (f:fs) <*> xs = map f xs ++ (fs <*> xs)
  []     <*> _  = []
```

Applicative functor laws

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

Equations that every Applicative instance should satisfy:

identity

```
pure id <*> v  <==>  v
```

homomorphism

```
pure f <*> pure x  <==>  pure (f x)
```

Applicative functor laws (cont.)

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

Equations that every Applicative instance should satisfy:

composition

```
pure (.) <*> u <*> v <*> w
          <==>
     u <*> (v <*> w)
```

interchange

```
u <*> pure y  <==>  pure ($ y) <*> u
```

Relationship to Functor

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

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

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