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

```haskell
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

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

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 \textit{class of data structures that can be accumulated over} type constructor!

\begin{center}
\begin{verbatim}
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b

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
rfold _ b Leaf = b
rfold f b (Node x l r) = f x (rfold f b l) (rfold f b r)
\end{verbatim}
\end{center}

\begin{itemize}
  \item Accumulator functions have different types
  \item … can we refactor \texttt{tfold} to make them the same?
\end{itemize}
Foldable class of data structures that can be accumulated over

class Foldable t where
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

\[
\text{class Foldable } t \text{ where } \\
\text{foldr : } (a \to b \to b) \to b \to t~a \to b
\]

instance Foldable [] where
\[
\text{foldr } _{}~b~[] &= b \\
\text{foldr}~f~b~(x:\!xs) &= f~x~(\text{foldr}~f~b~xs)
\]

\[
tfold : (a \to b \to b \to b) \to b \to \text{Tree}~a \to b \\
\text{tfold } _{}~b~\text{Leaf} &= b \\
\text{tfold}~f~b~(\text{Node}~x~l~r) &= f~x~(\text{tfold}~f~b~l)~(\text{tfold}~f~b~r)
\]

instance Foldable Tree where
\[
\text{foldr } _{}~b~\text{Leaf} &= b \\
\text{foldr}~f~b~(\text{Node}~x~l~r) &= \text{foldr}~f~(f~x~(\text{foldr}~f~b~r))~l
\]
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 f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

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

instance Applicative Maybe where
    pure x = Just x
    Just f <*> mx = fmap f mx
    Nothing <*> _ = Nothing
Applicative functor laws

Equations that every Applicative instance should satisfy:

**identity**

\[ \text{pure id} <*> v \iff v \]

**homomorphism**

\[ \text{pure f} <*> \text{pure x} \iff \text{pure (f x)} \]
Applicative functor laws (cont.)

class Functor f => Applicative f where
    pure    :: a -> f a
    (<?,>)   :: f (a -> b) -> f a -> f b

Equations that every Applicative instance should satisfy:

- **composition**
  
  $\text{pure (.)} \ <\&\> \ u \ <\&\> \ v \ <\&\> \ w \iff u \ <\&\> \ (v \ <\&\> \ w)$

- **interchange**
  
  $u \ <\&\> \ \text{pure } y \iff \text{pure } ($ $y$ $) \ <\&\> \ u$
Relationship to Functor

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

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

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