Functors and Type Constructor Classes

A pure functional language with Class!
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 * 

\[
\text{String} \quad \text{Int} \quad \text{[Bool]} \quad \text{Maybe Int} \\
\text{Tree a} \quad \text{Map k v} \quad \text{Int -> Int} \quad \text{a -> a -> Bool}
\]

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

\[
\text{Maybe :: * -> *} \quad \text{Tree :: * -> *} \\
\text{Map :: * -> * -> *} \quad \text{[] :: * -> *}
\]

\[
\text{Map :: * -> * -> *} \quad \text{(->) :: * -> * -> *}
\]

*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

```
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:

\[
\begin{align*}
    \text{fmap id} & \iff \text{id} \\
    \text{fmap (f \cdot g)} & \iff \text{fmap f \cdot fmap g}
\end{align*}
\]

Means that \text{fmap} preserves the structure of values

\text{... code written against the Functor interface can assume this}
Foldable \textit{class of data structures that can be accumulated over}

\[
\text{class Foldable } t \text{ where }
\begin{align*}
\text{foldr} & : (a \to b \to b) \to b \to [a] \to b \\
\text{foldr} & \ _\ b \ [\ ] \ = \ b \\
\text{foldr} \ f \ b \ (x:xs) \ = \ f \ x \ (\text{foldr} \ f \ b \ xs)
\end{align*}
\]

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

Accumulator functions have different types

… can we refactor \texttt{tfold} to make them the same?
Foldable class of data structures that can be accumulated over

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

typewriter!

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

```haskell
tfold :: (a -> b -> b -> b) -> b -> Tree a -> b
```

typewriter!

```haskell
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)
```

```haskell
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)

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

```
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
• 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

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

Equations that every Applicative instance should satisfy:

**identity**

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

**homomorphism**

\[
\text{pure f} <*> \text{pure x} \iff \text{pure (f x)}
\]

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

\[
\text{pure (.) <*> u <*> v <*> w} \iff \text{u <*> (v <*> w)}
\]

**interchange**

\[
\text{u <*> pure y} \iff \text{pure ($ y) <*> u}
\]
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