CIS 1904: Haskell

Functor, Foldable

Logistics

- HW7 will be released tonight
- Today’s class is recorded

Functors

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

Functors

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

```
showElems :: [Int] -> [String]
showElems xs = map show xs
```

Functors

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : xs) = Just x

safeShowHead :: (Show a) => [a] -> Maybe String
safeShowHead xs = case safeHead xs of
    Nothing -> Nothing
    Just x -> Just (show x)
```

Functors

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : xs) = Just x

safeShowHead :: (Show a) => [a] -> Maybe String
safeShowHead xs = case safeHead xs of
    Nothing -> Nothing
    Just x -> Just (show x)
```

Functors

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

Functors

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

safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : xs) = Just x

safeShowHead :: (Show a) => [a] -> Maybe String
safeShowHead xs = case safeHead xs of
    Nothing -> Nothing
    Just x -> Just (show x)
```

Functors

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

safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : xs) = Just x

safeShowHead :: (Show a) => [a] -> Maybe String
safeShowHead xs = map show (safeHead xs)
```

Functors

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

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

Functors

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

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

Map generally seems like it should work on “containers” for other values:

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

where `f` is something like `[]` that can “contain” values of type `a`.

Functors

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

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

Map generally seems like it should work on “containers” for other values:

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

where `f` is something like `[]` that can “contain” values of type `a`.

What tool have we seen for restricting a type variable to a certain group of types?

Functors

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

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

Map generally seems like it should work on “containers” for other values:

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

where `f` is something like `[]` that can “contain” values of type `a`.

What tool have we seen for restricting a type variable to a certain group of types?
Typeclasses!

Functors

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

Conceptually, `fmap` should always satisfy the following “functor laws”:

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

(Haskell has no mechanism for enforcing this in general.)

Functors

```
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
```

Functors

```haskell
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 x) = Just (f x)
```

Functors

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

data Stream a = Cons a (Stream a)

instance Functor Stream where
    fmap :: (a -> b) -> Stream a -> Stream b
    fmap f (Cons x xs) =
```

Functors

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

data Stream a = Cons a (Stream a)

instance Functor Stream where
    fmap :: (a -> b) -> Stream a -> Stream b
    fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

Functors

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

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

instance Functor Stream where
    fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

Functors

```
class Show a where
    show :: a -> String

instance Show Bool where
    show True = “True”
    show False = “False”

instance Show a => Show (Maybe a) where
    show Nothing = “Nothing”
    show (Just x) = “Just ” ++ show x
```

Functors

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

not `instance Functor [a] where`

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

not `instance Functor (Maybe a) where`

```haskell
instance Functor Stream where
    fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

not `instance Functor (Stream a) where`

Functors

```
class Show a where
    show :: a -> String
```

`a` is a type

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

`f` is a type *constructor*

Functors

What does it mean for something to be a “container” here?

1. It contains 0 or more *values*, potentially with some structure on them
2. For `fold` and `map` to make sense, all those *values* must have the same type

How can we formalize this idea?

Functors

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


data Maybe a
    = Nothing
    | Just a

data Stream a = Cons a (Stream a)
```

Functors

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


data Maybe a
    = Nothing
    | Just a

data Stream a = Cons a (Stream a)
```

each takes in a single type as an argument and returns another type

Functors

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

data Maybe a
    = Nothing
    | Just a

data Stream a = Cons a (Stream a)
```

```
List :: Type -> Type

Maybe :: Type -> Type

Stream :: Type -> Type
```

Kinds

In Haskell, our types have types!

Regular types get the type `Type`, also written `*`:

```
Int :: *
Char :: *
Bool :: *
```

We call the types of types *kinds*.

Functors

What does it mean for something to be a “container” here?

1. It contains 0 or more *values*, potentially with some structure on them
2. For `fold` and `map` to make sense, all those *values* must have the same type

How can we formalize this idea?

A container in this context is something with kind `* -> *`.

Functors

What does it mean for something to be a “container” here?

1. It contains 0 or more *values*, potentially with some structure on them
2. For `fold` and `map` to make sense, all those *values* must have the same type

How can we formalize this idea?

A container in this context is something with kind `* -> *`.

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

Kinds

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

```
Maybe :: * -> *
```

```
Maybe Int ::
```

Kinds

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

```
Maybe :: * -> *

Maybe Int :: *
```

Kinds

```
List ::

List Int ::

List (Int -> Int) ::

List (Maybe Int) ::
```

Kinds

```
List :: * -> *

List Int ::

List (Int -> Int) ::

List (Maybe Int) ::
```

Kinds

```
List :: * -> *

List Int :: *

List (Int -> Int) ::

List (Maybe Int) ::
```

Kinds

```
List :: * -> *

List Int :: *

List (Int -> Int) :: *

List (Maybe Int) ::
```

Kinds

```
List :: * -> *

List Int :: *

List (Int -> Int) :: *

List (Maybe Int) :: *
```

Kinds

Kinds have two “constructors”, or two forms:

`*`

k_1 `->` k_2 where k_1 and k_2 are both kinds

Kinds

Kinds have two “constructors”, or two forms:

`*`

k_1 `->` k_2 where k_1 and k_2 are both kinds

Examples:

```
*
* -> *
* -> * -> *
(* -> *) -> *
```

Kinds: Digression

```
data Container tc a = App (tc a)


App [True] :: Container [] Bool

App (Just “Hello”) :: Container Maybe String
```

Kinds: Digression

```
data Container tc a = App (tc a)


App [True] :: Container [] Bool

App (Just “Hello”) :: Container Maybe String



Container ::
```

Kinds: Digression

```
data Container tc a = App (tc a)


App [True] :: Container [] Bool

App (Just “Hello”) :: Container Maybe String


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

Kinds: FAQ

- You can check the kind of something in GHCi with `:k`, just like `:t`
  - e.g., `:k Int`
- “container” is a very overloaded word
  - If you search “Haskell containers” you will get an unrelated library
- `* :: *` in Haskell

Foldable

What else might we want to do with a “container”?

If it has multiple elements, we might want to combine them.

```
class Foldable t where
    foldr :: (a -> b -> b) -> b -> t a -> b
    elem :: (Eq a) => a -> t a -> Bool
    maximum :: (Ord a) => t a -> a
    …
```

Aside: Functor vs. Foldable

Why not make `fmap` part of `Foldable`?

`fmap` preserves structure, which sometimes we do not want:

```
data SortedList a
    = SNil | SCons a (SortedList a)

negateAll :: SortedList Int -> SortedList Int
negateAll xs = fmap negate xs

ghci> negateAll (SCons 1 (SCons 2 SNil))
```

Aside: Functor vs. Foldable

Why not make `fmap` part of `Foldable`?

`fmap` preserves structure, which sometimes we do not want:

```
data SortedList a
    = SNil | SCons a (SortedList a)

negateAll :: SortedList Int -> SortedList Int
negateAll xs = fmap negate xs

ghci> negateAll (SCons 1 (SCons 2 SNil))
SCons (-1) (Scons (-2) SNil)
```

Aside: Functor vs. Foldable

Why not make `fmap` part of `Foldable`?

`fmap` preserves structure, which sometimes we do not want:

In Haskell, a `Set` is a size-balanced binary tree.

Mapping would require potentially very expensive rebalancing.

Foldable

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

instance Foldable Stream where
    foldr :: (a -> b -> b) -> b -> Stream a -> b
    foldr f z (Cons x xs) =
```

Foldable

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

instance Foldable Stream where
    foldr :: (a -> b -> b) -> b -> Stream a -> b
    foldr f z (Cons x xs) = f x (foldr f z xs)
```

Foldable

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

instance Foldable Maybe where
    foldr :: (a -> b -> b) -> b -> Maybe a -> b
    foldr f z =
```

Foldable

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

instance Foldable Maybe where
    foldr :: (a -> b -> b) -> b -> Maybe a -> b
    foldr f z Nothing = z
    foldr f z (Just x) = f x z
```

Foldable

```
instance Foldable Maybe where
    foldr :: (a -> b -> b) -> b -> Maybe a -> b
    foldr f z Nothing = z
    foldr f z (Just x) = f x z

headOrDefault :: a -> [a] -> a
headOrDefault d = foldr (fun x _ -> x) d safeHead
```

Foldable

```
instance Foldable Maybe where
    foldr :: (a -> b -> b) -> b -> Maybe a -> b
    foldr f z Nothing = z
    foldr f z (Just x) = f x z

headOrDefault :: a -> [a] -> a
headOrDefault d = foldr (fun x _ -> x) d safeHead

ghci> headOrDefault 0 [1,2,3]
1
ghci headOrDefault 0 []
0
```