Outline of Lecture 13

- The `Functor` type class (cont.)
- Folding with monoids – the type class `Foldable`
- The `Applicative` type class
- Examples of applicative functors
- The `Monad` type class (introduction, if time allows it)

Monoids and Functors (reminder)

- In the previous lecture, we have seen two common algebras:
  - `Monoid` gives us a means of combining (folding) two values of the same type together;
  - `Functor`, on the other hand, is for function application (mapping) over some structure we do not want to have to think about
- `Monoid`'s core operation, `mappend`, smashes two structures together (e.g., two lists become one)
- The core operation of `Functor`, `fmap`, applies a function to a value that is within some structure while leaving that structure unaltered
- The algebras are implemented in Haskell as type classes, fixing the required "interface"

Haskell type class `Functor` (reminder)

- The definition:

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

- The `Functor` type class is parameterised not by a polymorphic type variable like `a` or `b`, but with a type constructor `f`, which takes one polymorphic type parameter and producing a concrete type
- Suitable instances of `f`: `List([])`, `Maybe`, `Either`, `Tree`

Functor examples (reminder)

- Trees are functors
- A version of the `map` function for trees was defined as:

```
mapTree ::  (a-> b) -> Tree a -> Tree b
mapTree Nil = Nil
mapTree f (Node x t1 t2) =
  Node (f x) (mapTree f t1) (mapTree f t2)
```

- `Tree` is a functor:

```
instance Functor Tree where
  fmap = mapTree
```

Functor examples (cont.)

- What about `Either` – a type constructor with two type parameters?

```
data Either a b = Left a | Right b
```

- `Either` is not a functor, but `Either a` (a partially applied type constructor, still "waiting" for the second type parameter) is:

```
instance Functor (Either a) where
  fmap f (Right x) = Right (f x)
  fmap f (Left x) = Left x
```

Applying the given function `f` only on the right argument value!

Functor laws

- Identity: `fmap id = id`
  Passing the identity function should not have any effect at all
- Composition: `fmap (f . g) = fmap f . fmap g`
  If we compose two functions, f and g, and fmap that over some structure, we should get the same result as if we fmapped them separately and then composed them
- Both laws enforce the essential rule that functors must be structure preserving. If an implementation of fmap does not satisfy these laws, it is a broken functor

Stacked functors over nested layers of structure

- We can combine datatypes, usually by nesting them
- What if the data structure has more than one Functor type. Are we obligated to fmap only to the outermost datatype?
- No, we can actually compose several fmaps to reach the necessary layer. To demonstrate that, let's consider an example:

```
Prelude> lms = [Just "Ave", Nothing, Just "woohoo"]
Prelude> :t lms
lms ::  [Maybe [Char]]
Prelude> replaceWithP = const 'p'
Prelude> :t replaceWithP
replaceWithP ::  b -> Char
```

Stacked functors over nested layers of structure (cont.)

- Three layers of structure: a list, `Maybe` data type, and a list again
- By combining `fmap` functions, we can reach the layer we need:

```
Prelude> fmap replaceWithP lms
"ppp"
Prelude> (fmap . fmap) replaceWithP lms
[Just 'p',Nothing,Just 'p']
Prelude> (fmap . fmap . fmap) replaceWithP lms
[Just "ppp",Nothing,Just "pppppp"]
```

Stacked functors (cont.)

- How this composition even typechecks?

```
Prelude> :t (fmap . fmap)
(fmap . fmap) ::  (Functor f1, Functor f) => (a -> b)
-> f (f1 a) -> f (f1 b)
```

- The second half of one functor (e.g., `f m -> f n`) gets matched with the first part of the other functor (e.g., `x-> y`):

```
(.)   ::  (b->c) -> (a->b) -> a -> c
fmap  ::  Functor f => (m -> n) -> f m -> f n
fmap  ::  Functor g => (x -> y) -> g x -> g y
```

  thus ensuring that we go one more structural layer inside before applying the transformation function

Type constructors with more than single argument

- Can type constructors with more than single argument be made into functors? No because of the incompatible types
- We can solve this problem by "adjusting" a type constructor with partial application on type arguments
- Examples: `Either` is not accepted, while `Either a` can be defined as a functor, working on the Right elements of `Either a b`
- The same with pairs – `instance Functor ((,) a)`
- All instances of `Functor` available in Prelude:

```
Prelude> :info Functor
```

Folding with monoids – the type class `Foldable`

- As the class `Functor` is for type constructors that support mapping over, there is the class `Foldable` contains those type constructors that allow folding, e.g.,

```
ghci> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

- The interface of `Foldable` includes all the standard folding operations: `foldr`, `foldr1`, `foldl`, `foldl1` as well as generic functions `fold` and `foldMap`

Folding with monoids (cont.)

- To make a type constructor a member of `Foldable`, it is sufficient to only provide the generic `foldMap` function that relies on a monoid type:

```
ghci> :t foldMap
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

- The first parameter is a function takes a value that our foldable structure contains and returns a monoid value
- The second parameter is the structure to be folded
- `foldMap` maps the provided function over the foldable structure to produce monoid values. Then, by doing `mappend` between these monoid values, it joins them into a single monoid value

Folding with monoids (cont.)

- Example – datatype `Tree`:

```
data Tree a = NilT | Node a (Tree a) (Tree a)
```

- Making `Tree` an instance of `Foldable`:

```
instance Foldable Tree where
  foldMap f NilT = mempty
  foldMap f (Node x left right) =
    (foldMap f left) `mappend` (f x)
    `mappend` (foldMap f right)
```

Haskell type classes: Applicative

- Applicatives are monoidal functors. What does it mean?
- The `Applicative` type class allows for function application lifted over structure (like `Functor`)
- However, with `Applicative`, the function we are applying and the value being applied are both embedded in some structure
- As a result, we have to apply the function and merge these structures together
- So, `Applicative` involves both monoids and functors

Haskell type classes: Applicative (cont.)

- The definition:

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

- Every type that can have an `Applicative` instance must also have a `Functor` instance
- The `pure` function does a very simple thing: it lifts (wraps) something into a functorial structure (minimal context)
- The infix operator `(<*>)` (called apply) combines two structural values (one of them containing an embedded function) into one

Examples of `pure`

- Embedding a value of any type in the structure we are working with:

```
Prelude> pure 1 ::  [Int]
[1]
Prelude> pure 1 ::  Maybe Int
Just 1
Prelude> pure 1 ::  Either a Int
Right 1
Prelude> pure 1 ::  ([a], Int)
([],1)
```

- The left type is handled differently from the right in the final two examples since the left type is part of the structure, and the structure is not transformed by the function application.
  For the same reason as `fmap (+1) (4,5) == (4,6)`

Applicative functors are monoidal functors

- Applicative? Connected to function application? Let us check the types:

```
($) ::  (a -> b) -> a -> b
(<$>) ::  (a -> b) -> f a -> f b
(<*>) ::  f (a -> b) -> f a -> f b
```

  where `(<$>) = fmap` (i.e., infix operator alias for `fmap`)

- While `(<$>)` looks like a generalisation of `($)` over a given structure, `(<*>)` additionally "wraps" a given function into the structure `f`

- Moreover, one of Applicative laws requires:

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

- "Wrapping" of a given function (`pure f`) looks like additional (redundant?) step. What are advantages of this?

Applicative functors are monoidal functors (cont.)

- When we were dealing with fmap, we had only one bit of structure, so it was left unchanged
- With `(<*>)`, we have two bits of structure of type `f` that we need to deal with before returning a value of type `f b`. We cannot simply leave them unchanged; we must unite them somehow
- A typical solution – rely on `Monoid` for our structure and function application for our values!
- What are we gaining from that?

Examples of (<*>)

- Applying (<*>) between lists: the values of [a -> b] and [a]:

```
Prelude> [(*3)] <*> [4, 5]
[12,15]
Prelude> [(*2), (*3)] <*> [4, 5]
[8,10,12,15]
Prelude> [] <*> [Just 2]
[]
Prelude> [(*2), (*3)] <*> []
[]
```

- What are we gaining from embedding (a-> b) into [a -> b]? List-ness (a property of being a list, with all its advantages)
- The first case could have been encoded as simply fmap (*3) [4,5] (no need for Applicative). In the other cases, we have now a list of functions that give us more expressive power

Examples of (<*>) (cont.)

- Applying (<*>) between the values of Maybe (a -> b) and Maybe a:

```
Prelude> Just (*2) <*> Just 2
Just 4
Prelude> Just (*2) <*> Nothing
Nothing
Prelude> Nothing <*> Just 2
Nothing
```

- What are we gaining from embedding (a-> b) into Maybe (a -> b)? Maybe-ness
- We can now express how uncertainty of different Maybe values can be combined

Applicative instances

- The Applicative instance for lists:

```
instance Applicative [] where
  pure a = [a]
  [] <*> _ = []
  (f:fs) <*> xs =
    (fmap f xs) `mappend` (fs <*> xs)
```

- For lists, mappend = (++)
- One of several ways to combine such list values (one alternative – ZipList)

`Applicative` instances (cont.)

- The `Applicative` instance for `Maybe`:

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

- `Nothing` values propagate through the structure!

`Applicative` instances (cont.)

- The `Applicative` instance for `(,) a`:

```
instance Monoid a => Applicative ((,) a) where
  pure x = (mempty, x)
  Nothing <*> _ = Nothing
  (u, f) <*> (v, x) = (u `mappend` v, f x)
```

- Explicit dependency (constraint) `Monoid a`
- Monoidal merging and function application on different pair elements!

One more example: lookups

- The `lookup` function searches inside a list of tuples for a value that matches the input and returns the paired value wrapped inside a Maybe context:

```
:t lookup
lookup ::  Eq a => a -> [(a, b)] -> Maybe b
```

- Example:

```
import Control.Applicative

f x = lookup x [ (3, "hello"), (4, "julie"), (5, "kbai")]
g y = lookup y [ (7, "sup?"), (8, "chris"), (9, "aloha")]
h z = lookup z [(2, 3), (5, 6), (7, 8)]
m x = lookup x [(4, 10), (8, 13), (1, 9001)]
```

One more example: lookups (cont.)

- Combining `Maybe` values:

```
Prelude> f 3
Just "hello"
Prelude> g 8
Just "chris"
Prelude> (++) <$> f 3 <*> g 7
Just "hellosup?"
Prelude> (+) <$> h 5 <*> m 1
Just 9007
Prelude> (+) <$> h 5 <*> m 6
Nothing
```

- `f <$> (x :: Maybe a)` embedds a function into the `Maybe` structure by partial application (e.g., `Prelude> (++) <$> f 3` becomes `Just (++ "hello")`)

Haskell type classes: Monads

- A functor maps a function over some structure; an applicative maps a function that is contained in some structure over some other structure and then combines the two layers of structure like `mappend`
- Monads are applicative functors, but they have something special about them that makes them different from and more powerful than either `<*>` or `fmap` alone
- Definition:

```
class Applicative m => Monad m where
  (>>=) ::  m a -> (a -> m b) -> m b
  (>>) ::  m a -> m b -> m b
  return ::  a -> m a
  x >> y = x >>= (\_ -> y)
```

Haskell type classes: Monads (cont.)

- Dependencies between type classes:

  `Functor => Applicative => Monad`

  Whenever you have implemented an instance of Monad for a type you necessarily have an Applicative and a Functor as well

- `return` is just the same as `pure` of *Applicative*. Essentially, a simple constructor for a monadic value

- The operator `(>>)` is called the sequencing operator. It is a restricted version of the main `Monad` function `(>>=)`, called *bind*

Haskell type classes: Monads (cont.)

- Let us check the types:

```
fmap ::  Functor f => (a -> b) -> f a -> f b
(<*>) ::  Applicative f => f (a -> b) -> f a -> f b
(>>=) ::  Monad f => f a -> (a -> f b) -> f b
```

- Bind `(>>=)` looks quite similar to both `fmap` and `(<*>)`, but with the first two arguments flipped. Yet another version of mapping a function over a value while bypassing its surrounding structure
- Can we express `(>>=)` via `fmap` by simply instantiating `b` to `f b`?

Haskell type classes: Monads (cont.)

- Example:

```
Prelude> andOne x = [x, 1]
Prelude> andOne 10
[10,1]
Prelude> :t fmap andOne [4, 5, 6]
fmap andOne [4, 5, 6] :: Num t => [[t]]
Prelude> fmap andOne [4, 5, 6]
[[4,1],[5,1],[6,1]]
```

- We ended up with an extra layer of structure, and now we have a result of nested lists
- Our mapped function has itself generated more structure! How to discard one unnecesary layer of that structure?

Haskell type classes: Monads (cont.)

- We know how to do it with lists, using the function `concat`:

```
concat ::  [[a]] -> [a]
```

- `Monad`, in a sense, is a generalisation of concat! The unique part of `Monad` is the following function:

```
import Control.Monad (join)

join ::  Monad m => m (m a) -> m a
```

- The ability to flatten those two layers of structure into one is what makes `Monad` special
- Using `join`, `bind` can be simply defined as
  `(>>=) f x = join (fmap f x)`

Haskell type classes: Monads (cont.)

- General intuition: a monad allows to sequence operations on structure, while feeding the result of one action as the input value to the next
- Each operation may return a structured value, which is internally flattened and combined with others
- A structured value may indicate non-determinism (e.g., several possible results as a list of values), uncertainty (a Maybe value), ...
- Monad allows to sequence such operations and their structured results in a convenient way