Outline of Lecture 13

- Records in Haskell
- Algebras as Haskell type classes
- The `Monoid` type class
- Type constructors and kinds
- The `Functor` type class
- The `Foldable` type class

Records in Haskell

- A very simple simple implementation (essentially syntactic sugaring over the existing datatype definition mechanism)
- Records in Haskell are product types with additional syntax to provide convenient accessors to fields (functions) within the record
- A simple product type (a person with a name and an age):

```
data Person = MkPerson String Int
deriving (Eq,Show)
```

- We can extract necessary values by using pattern matching and/or writing our own functions:

```
name ::  Person -> String
name (MkPerson s _) = s
```

Records in Haskell (cont.)

- Let's see how we could define a similar product type but with record syntax:

```
data Person = Person {name :: String, age :: Int}
deriving (Eq,Show)
```

- Defining it as a record means there are now named record field accessors. They are just generated functions that go from the product type to a member of product:

```
Prelude> :t age
age :: Person -> Int
Prelude> pp = Person "Ann" 5
Person {name = "Ann", age = 5}
Prelude> age pp
5
```

Abstract patterns and algebras

- Haskell allows to recognise abstract patterns in code, which have well-defined and analysed representations in mathematics
- A word frequently used to describe these abstractions is algebra, by which we mean one or more operations and the set they operate over
- Examples of such algebras: monoids, semigroups, functors, monads, ..
- In Haskell, these algebras can be implemented with type classes
- Type classes define the set of operations, while their instances define how each operation will perform for a given type or set

Type class `Monoid`

- In mathematics, a monoid is an algebraic structure with a single associative binary operation and an identity element
- In other words, it is a data type for which we can define a binary function such as:
  - the function takes two parameters of the same type;
  - there exists such a value that does not change other values when used with the function (identity element);
  - If we have three or more values and use the function to reduce them to a single result, the application order does not matter (associativity).
- Examples: `Integer` with `(*)` and 1, `List a ([a])` with `(++)` and `[]`

Type class `Monoid` (cont.)

- The class definition:

```
class Monoid a where
  mempty ::  a
  mappend ::  a -> a -> a
  mconcat ::  [a] -> a
  mconcat = foldr mappend mempty
```

`mempty` – the identity element,
`mappend` – the binary monoid operation,
`mconcat` – generalisation of `mappend` over a list of values

Monoids are ideal for folding

Monoid examples

- Lists are monoids:

```
instance Monoid [a] where
  mempty = []
  mappend = (++)
```

- Maybe a is a monoid:

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  Nothing `mappend` m = m
  m `mappend` Nothing = m
  Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

Reusing algebras

- The last example of monoid instance demonstrates that algebras can be reused:

  ```
  instance Monoid a => Monoid (Maybe a) ...
  ```

- More such examples:
  - `instance Monoid b => Monoid (a -> b) ...`
  - `instance (Monoid a, Monoid b) => Monoid (a, b) ...`
  - ```
    instance (Monoid a, Monoid b, Monoid c) =>
      Monoid (a, b, c) ...
    ```
    ```

Monoid laws

- Three mathematical properties (laws) that are expected from any monoid instance
- Left identity:
  `mappend mempty x = x`
- Right identity:
  `mappend x mempty = x`
- Associativity:
  `mappend x (mappend y z) = mappend (mappend x y) z`
- Validating/checking the laws for an instance candidate: with QuickCheck, ...

Two possible monoid structures for the same type

- The type Integer does not have a Monoid instance. None of the numeric types do. Why?
- Both summation and multiplication can be used as monoid operations!
- Restriction: each type should only have one unique instance for a given typeclass
- To resolve the conflict, we have the Sum and Product newtypes (in Data.Monoid) to wrap numeric values and signal which Monoid instance we want
- Reminder: using `newtype` "wraps" the existing type, forcing Haskell to treat it as new

Two possible monoid structures for the same type

- The `Sum` record data types declared by `newtype`, e.g. :

```
newtype Sum a = Sum {getSum :: a}
newtype Product a = Product {getProduct :: a}
```

- Both `Sum` and `Product` (for any `Num a =>`) are declared as instances of the `Monoid` type class
- Checking:

```
Lecture13> mappend (Sum 1) (Sum 99)
Sum {getSum 100}
Lecture13> mappend (Product 33.3) (Product 2.5)
Product {getProduct = 83.25}
Lecture13> mappend (Sum 2, Sum 3) (Sum 3, Sum 4)
Sum {getSum = 5},Sum {getSum = 7}
```

Functors

- Functor – pattern of mapping over or around some structure that we do not want to alter
- That is, we want to apply the function to the value that is "inside" of some structure and leave the structure intact
- Example: a function gets applied for each element of a list and the list structure remains. No elements are removed or added, only transformed
- The type class `Functor` generalises this pattern for many types of structure

Intuition behind functors

- Applying data transformations within the given context / structure / "box" / "wrapper"
- Functors encode
  - going inside the structure (list, tree, any data constructor),
  - applying the given transformation on the extracted inside values,
  - reconstructing the original structure
- Often a sequence of actions when the values are extracted from the context, transformed, and then the context is restored are needed

Haskell type class `Functor`

- The `Functor` type class: the types that can be mapped over
- The definition:

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

- The type class contains a single operation `fmap` for working within a given structure
- Looks very similar to the familiar `map`:

```
map ::  (a -> b) -> [a] -> [b]
```

- What's a structure here? What is `f` stands for? The answers soon

`fmap` examples

- Looks like a whole lot of `fmap` is going around:

```
Prelude> fmap (*10) [2,7]
[20,70]
Prelude> fmap (+1) (Just 1)
(Just 2)
Prelude> fmap (+1) Nothing
Nothing
Prelude> fmap (+10/) (4,5)
(4,2.0)
Prelude> fmap (++ "Esq.") (Right "Chris Allen")
(Right "Chris Allen, Esq.")
```

- The same principle: transformations that happen within some external structure (a list, a tuple or a data type)

What's `f` stands for in the `fmap` type?

- There are two kinds of constructors in Haskell: type constructors and data constructors. Type constructors are used only at the type level, in type signatures and typeclass declarations and instances
- Type constructors: functions that take types and produce types. Examples: `[]`, `(,)`, `Maybe`, `Either`, `Tree` `...` User-defined data type names are also type constructors, if the type definition contains at least one type variable
- The `Functor` type class is parameterised over such a type constructor (`f`)
- Essentially, `f` introduces the structure that `fmap` works inside on!

Kinds

- To distinguish between the basic types and type constructors, the notion of *kind* is used
- Kinds are the types of types, or types one level up. We represent kinds in Haskell with `*`, `* -> *`, `* -> * -> *`, ...
- We know something is a fully applied, concrete type when it is represented as `*`. When it is `* -> *`, it, like a function, is still waiting to be applied.
- Checking kinds within GHCI:

```
Prelude> :k []
[] ::  * -> *
```

Kinds (cont.)

- More examples of kinds:

```
Prelude> :k Int
Int ::  *
Prelude> :k Maybe
Maybe ::  * -> *
Prelude> :k Either
Either ::  * -> * -> *
Prelude> :k Person
Person ::  *
Prelude> :k Sum
Person ::  * -> *
```

Lists as Functors

- It is not coincidence that the definition of `fmap` function looks like the `map` function on lists

```
map ::  (a -> b) -> [a] -> [b]
```

- Lists are an instance of the `Functor` type class:

```
instance Functor [] where
  fmap = map
```

- Having `[a]` instead of `[]` here would generate an error: a function on types (a type constructor) is expected, not a concrete type like `[a]`

Functor examples

- Mapping through elements of some type is often required and useful feature
- Example: transmitting the error through `mapMaybe`

```
mapMaybe ::  (a->b) -> Maybe a -> Maybe b

mapMaybe g Nothing  = Nothing
mapMaybe g (Just x) = Just (g x)
```

- Maybe is a functor:

```
instance Functor Maybe where
  fmap = mapMaybe
```

Again, writing `Maybe`, not `Maybe a`

Functor examples (cont.)

- Trees are functors too
- 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 partial 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 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> :i 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)
```