Foldable and Traversable

FIT2102 Programming Paradigms

S2 2017 Week 9

Preamble

In this week's tutorial, we will explore the power of foldable and traversable structures. Both are typeclasses such as Functor or Applicative. Actually, we only need three things for this week:

1. **Functor**. A functor is a container, we can apply a function over (or inside) of it.
2. **Foldable**. A foldable is a container that we can reduce to a single value.
3. **Traversable**. A traversable is both a foldable and a functor.

You will have to copy your `Functor.hs` and `Applicative.hs` file from last week into this week's `src/` folder to have access to Functor and Applicative instances.[1]

[1] To keep the project working as a single entity, please keep the header (`import` declarations) in the files while copying your code over.

Functions

*You have to write the body **and** the type signature for all exercises.*

First, we will explore the power of `fold`. Folding is a very basic operation in functional programming. A lot of functions are actually implemented as folds.

In our case, we will focus on a right-fold. A right-fold takes a binary function which merges elements, a base case, and a list of elements to return the *fold* of these elements through the function.

```
foldr ::         -- right-fold
  (a -> b -> b) -- binary function
  -> b           -- base case
  -> [a]         -- list of elements
  -> b           -- result
```

Another way to look at a right-fold is: replace all the *cons* `(:)` in a list with a function and the *nil* `[]` with the base case.

```
-- Sum the elements of a list with fold
[1, 2, 3, 4, 5]        -- Rewrite using cons
1 : 2 : 3 : 4 : 5 : [] -- Fold (+) on the list
1 + 2 + 3 + 4 + 5 + 0  -- 0 is the base case
15                     -- Apply
```

Interlude: Monoid

A Monoid is a type with an associative binary operation that follows the law of identity. Monoids allow folding operations using the binary function `(<>)`.

1. Law of identity (left and right).

$$\forall x : \text{id} <> x = x$$
$$\forall x : x <> \text{id} = x$$

2. Law of associativity.

$$\forall x, y, z : x <> (y <> z) = (x <> y) <> z$$

Monoids, conveniently, provide a default value for their operation.[2]

[2] In mathematics we call that an "identity element."

```
mempty :: (Monoid m) => m
```

Monoids also provide a generalised `fold`, also called `mconcat`.[3]

[3] Note how `mconcat` does not require a base case as it is defined within the monoid.

```
mconcat :: (Monoid m) => [m] -> m
```

Foldable

A foldable is a structure which can be reduced to a single value. Think of it as a structure on which we can use `foldr`. To define an instance of foldable, we need to define the following function:

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

That is, given a monoid and a function to fold an item into it, and a sequence of non-monoidal items, we can reduce the sequence to a single element.

Defining an instance of foldable allows us to derive a number of very useful functions *for free*. For example, a generalised `fold`, both left- and right-fold, a list converter, a length function, element existence, etc.[4]:

[4] See the list on hackage.

```
class Foldable t where
  fold   :: (Monoid m) => t m -> m
  length :: t a -> Int
  toList :: t a -> [a]
  elem   :: (Eq a) => a -> t a -> Bool
  -- ...
```

Traversable

Traversable are structures which can be traversed while performing an action. A traversable has to be a foldable and a functor. They are defined using `traverse`.

```
traverse :: (Traversable t, Applicative f) => (a -> f b) -> t a -> f (t b)
```

All instances of traversable have to respect the following laws:

1. The law of identity.[5]

$$\text{traverse Id} = \text{Id}$$

2. The law of naturality.[6]

$$\forall f, \forall t \in \mathcal{A} : t \circ \text{ traverse } f = \text{traverse } (t \circ f)$$

3. The law of composition.[7]

$$\text{traverse}(\odot \circ \text{ <\$> } g \circ f) = \odot \circ \text{ <\$> } (\text{traverse } g) \circ (\text{traverse } f)$$

A traversable is for a monoid what an applicative is for a functor.

[5] Where Id is the identity functor.

[6] Where $t \in \mathcal{A}$ is a function that respects the applicative operations.

[7] Where $\odot$ is the composition of functors.