Foldable and Traversable (/2014/07/30/foldable-and-traversable.html)

JUL 30, 2014

Before we can get into the more advanced topics on Lenses, it is important to really understand both `Foldable` and `Traversable`, which is the motivation behind this article.

Let's begin with `Foldable`. `Foldable` represents structures which can be folded. What does that mean? Here are a few examples:

- Calculating the sum of a list.
- Calculating the product of a list.
- Folding a tree to get a maximum value.

We can describe a *fold* as ***taking a structure and reducing it to a single result***. That's also why some languages have a `reduce` function instead of a `fold`, even though they mean the same thing.

It is important to really understand the concept behind a fold in general, not in terms of specific functions like `foldl` or `foldr`. Whenever you see the word `fold` in a function name, think *reducing a larger structure to a single result*.

Now comes the time to take a look at the `Foldable` type class.

```
class Foldable t where
    fold    :: Monoid m => t m -> m
    foldMap :: Monoid m => (a -> m) -> t a -> m

    foldr   :: (a -> b -> b) -> b -> t a -> b
    foldr'  :: (a -> b -> b) -> b -> t a -> b

    foldl   :: (b -> a -> b) -> b -> t a -> b
    foldl'  :: (b -> a -> b) -> b -> t a -> b

    foldr1  :: (a -> a -> a) -> t a -> a
    foldl1  :: (a -> a -> a) -> t a -> a
```

We won't go into detail on all of these, since `foldl`, `foldr`, `foldl'`, `foldr'`, `foldl1` and `foldr1` work the same as their counterparts from `Data.List`.

What is interesting here is that `fold` and `foldMap` require the elements of the `Foldable` to be `Monoid`s. Let's just quickly take a look at what a `Monoid` is.

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

Nothing really special here, `Monoid` just simply defines a zero element via `mempty` and an associative operation `mappend` for combining two

Recent articles

- Fibonacci Numbers (/2015/08/08/fibonacci-numbers.html)
- Parsing CSS with Parsec (/2014/08/10/parsing-css-with-parsec.html)
- Lens Tutorial - Stab & Traversal (Part 2) (/2014/08/06/lens-tutorial-stab-traversal-part-2.html)
- Foldable and Traversable (/2014/07/30/foldable-and-traversable.html)
- Building Monad Transformers - Part 1 (/2014/07/22/building-monad-transformers-part-1.html)
- Mutable State in Haskell (/2014/07/20/mutable-state-in-haskell.html)
- Lens Tutorial - Introduction (part 1) (/2014/07/14/lens-tutorial-introduction-part-1.html)
- Using Phantom Types in Haskell for Extra Safety - Part 2 (/2014/07/10/using-phantom-types-in-haskell-for-extra-safety-part-2.html)
- Using Phantom Types for Extra Safety (/2014/07/08/using-phantom-types-for-extra-safety.html)
- Evil Mode: How I Switched From VIM to Emacs (/2014/06/23/evil-mode-how-to-switch-from-vim-to-emacs.html)

Tags

- clojure (1) (/tags/clojure.html)
- testing (1) (/tags/testing.html)
- rspec (1) (/tags/rspec.html)
- ruby (1) (/tags/ruby.html)
- refactoring (1) (/tags/refactoring.html)
- haskell (9) (/tags/haskell.html)
- emacs (1) (/tags/emacs.html)
- lens (2) (/tags/lens.html)
- algorithms (1) (/tags/algorithms.html)

Archives

- 2015 (1) (/2015.html)
- 2014 (13) (/2014.html)
- 2013 (3) (/2013.html)

Google+ (https://plus.google.com/+JakubArnold?rel=author)

`Monoid`s into one. `mconcat` is just a convenience method which has a default implementation using `mappend`.

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

fold and foldMap

The interesting thing about `fold` and `foldMap` is that they use a `Monoid` instead of a function to give us the final result. This might not be obvious at first, but by picking the right `Monoid` it is essentialy the same as passing in a function, since it will just use the `mappend` defined for that `Monoid` instance.

One very very very important aspect to understand here is that it is the `fold` function that requires the elements of `Foldable` to have a `Monoid` instance, while `Foldable` itself does not have that restriction.

The result of this is that we can have something like `[Int]`, where the `[]` is a `Foldable`, but `Int` is not a `Monoid`, though as long as we don't use any of the functions from `Foldable` that require a `Monoid` we'll be OK. Here's an example

```
λ> foldr1 (+) [1,2,3,4]
10
λ> fold ["hello", "world"]
"helloworld" -- Strings are Monoids using concatenation
λ> fold [1,2,3,4]
<interactive>:1:1:
    No instance for (Monoid a0) arising from a use of 'it'
```

See how the problem only arises when we used `fold` with `Int`. We could however wrap those `Int`s in a `Monoid` such as `Sum` or `Product` and `fold` them then.

```
λ> fold [Sum 1, Sum 2, Sum 3, Sum 4]
Sum {getSum = 10}
```

This might seem tedious at first, but remember our `Foldable` type class, as it also defines a function that is perfect for this particular use case: `foldMap :: Monoid m => (a -> m) -> t a -> m`. We can read this as *Given a foldable containing things that aren't Monoids, and a function that can convert a single thing to a Monoid, I'll give you back a Monoid by traversing the foldable, converting everything to Monoids and folding them together.*

Here's our previous example, but now using `foldMap`.

```
λ> foldMap Sum [1,2,3,4]
Sum {getSum = 10}
```

If you think about this for a little while we might even implement `fold` in terms of `foldMap`. Why? When using `foldMap` we need to provide a way to convert each item to a `Monoid`, but if those items already are `Monoid`s, we don't need to do any conversion!

```
fold :: Monoid m => t m -> m
fold xs = foldMap id xs
```

Here’s the same in more steps.

```
λ> :t id
:: a -> a
λ> :t fold
:: (Monoid m, Foldable t) => t m -> m
λ> :t foldMap
:: (Monoid m, Foldable t) => (a -> m) -> t a -> m
λ> :t foldMap id
:: (Monoid m, Foldable t) => t m -> m
```

The actual `Foldable` type class requires either `foldMap` or `foldr`, but for the sake of this article we won’t be looking into `foldr`.

Traversable

Now that we have an understanding of `Foldable` we can move on to something more fun, `Traversable`. `Traversable` represents data structures which can be traversed while perserving the shape. This is why there is no `filter` or `concatMap`, since `Traversable` only defines a way to move through the data structure, but not a way to change it.

```
class (Functor t, Foldable t) => Traversable t where
    traverse  :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
```

If you look in the documentation for `Traversable` (https://hackage.haskell.org/package/base-4.7.0.0/docs/Data-Traversable.html) you might note that there is also `mapM` and `sequence`, but we won’t be covering those in this article, since their implementation isn’t interesting and can be done mechanically.

This might look a little intimidating at first, but don’t worry, we’ll do this step by step by implementing a `Traversable` instance for a list.

```
instance Traversable [] where
     traverse f xs = _
```

Since the implementation will be recursive we first need to define the base case for our recursion, which will be the empty list. The type that we’re looking for is `f [b]`, but because the list we’re traversing is empty, we just need to wrap it in the `Applicative` context.

```
instance Traversable [] where
    traverse _ [] = pure []
    traverse f (x:xs) = _
```

Next goes the actual recursive implementaion. We have a function `f :: a -> f b` and a head of the list which has the type `a`. The only thing we can do at this point is apply the function.

```
instance Traversable [] where
    traverse _ [] = pure []
    traverse f (x:xs) = f x
```

This won't typecheck of course, because we're returning `f b` instead of `f [b]`. We could cheat here a little bit and just try to apply a some function `f b -> f [b]` to get the result. We can use `(:[])` which has a type of `a -> [a]` and `fmap` it on what we have.

```
instance Traversable [] where
    traverse _ [] = pure []
    traverse f (x:xs) = fmap (:[]) (f x)
```

Now we have an implementation that type checks, but it is still wrong, since it doesn't satisfy the rule that *a traversal must not change the shape of the structure it is traversing*, and here we are just dropping the rest of the list. We need to find a way to use recursion and somehow combine the results.

By looking at the type of `traverse :: (a -> f b) -> t a -> f (t b)`, or in our case specifically `traverse :: (a -> f b) -> [a] -> f [b]` we can see that using `traverse` recursively on the tail of the list would give us the type we need.

```
instance Traversable [] where
    traverse _ [] = pure []
    traverse f (x:xs) = (f x) _ traverse f xs
```

Now we have two values, one of type `f b` and one of type `f [b]`, which are basically the head and the tail of the list, both wrapped in an `Applicative` context. We also have a function `(:) :: a -> [a] -> [a]`, which concatenates a head and a tail together into a single list.

Knowing all of this it just comes down to a basic use of `Applicative` where we have a function of two arguments and need to apply it to two values in the `Applicative` context. We can do this in two different ways.

```
instance Traversable [] where
    traverse _ [] = pure []
    traverse f (x:xs) = (:) <$> f x <*> traverse f xs
```

And an alternative definition using `liftA2`.

```
instance Traversable [] where
    traverse _ [] = pure []
    traverse f (x:xs) = liftA2 (:) (f x) (traverse f xs)
```

It should be pretty clear now that we need the `Applicative` to be able to actually implement `traverse`. If all we had was a `Functor` we wouldn't be able to combine the `f b` and `f [b]` together.

sequenceA

Now that we have `traverse` we can move on to define `sequenceA`. Here's a specific type for our list instance.

```
sequenceA :: Applicative f => [f a] -> f [a]
```

If you're familiar with `sequence :: Monad m => [m a] -> m [a]` from `Control.Monad` then you can see how these two functions are doing the same thing. It simply takes the `Applicative` effects, runs them and pulls them out of the list.

The implementation is really simple. Starting out with an empty list, we just need to wrap it in the `Applicative` context.

```
sequenceA [] = pure []
```

Next comes the actual recursive implementaiton. If we pattern match on the head and the tail of the list, we'll yet again get `f a` and `[f a]`.

```
sequenceA (x:xs) = _
```

We can call `sequenceA` recursively on the tail to get `f [a]`.

```
sequenceA [] = pure []
sequenceA (x:xs) = sequenceA xs
```

But of course this isn't good enough. We need a way to combine the head and the tail while they're both wrapped in an `Applicative` context. This can be done in the same way as we did previously with `traverse`, using `(:)` and the `Applicative` functions `<$>` and `<*>`.

```
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```

Or alternatively using `liftA2` again.

```
sequenceA [] = pure []
sequenceA (x:xs) = liftA2 (:) x (sequenceA xs)
```

That's it, we have a working implementation for `sequenceA`.

Implementing sequenceA with traverse and vice versa

If we now look at our implementations for `traverse` and `sequenceA` we can definitely see some similarity there.

```
traverse _ [] = pure []
traverse f (x:xs) = (:) <$> f x <*> traverse f xs

sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```

The only difference is that `traverse` takes a function and applies it to the head of the list, while `sequenceA` simply uses the head as it is. Knowing this we can actually define `sequenceA` using `traverse` and the `id` function.

```
sequenceA :: (Traversable t, Applicative f) => t (f a) -> f (t a)
sequenceA xs = traverse id xs
```

Could we do the same thing the other way around though? Yes! We most certainly can define `traverse` by using `sequenceA` and the fact that every `Traversable` is also a `Functor`. Let's take this step by step.

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

We only have one way of applying our function `a -> f b` to the `t a` and that is using `fmap`, which would give us `t (f b)`.

```
traverse :: (Traversable t, Applicative f) => (a -> f b) -> t a -> f (t b)
traverse f xs = _ $ fmap f xs
```

Now we'll get an error saying that we need a function `t (f b) -> f (t b)`, which is exactly what `sequenceA` does!

```
traverse :: (Traversable t, Applicative f) => (a -> f b) -> t a -> f (t b)
traverse f xs = sequenceA $ fmap f xs
```

Traversable with default implementations

Given the two implementations we just got we can rewrite our initial `Traversable` type class to use those as a default implementation for both functions.

```
class (Functor t, Foldable t) => Traversable t where
    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
    traverse f xs = sequenceA $ fmap f xs

    sequenceA :: Applicative f => t (f a) -> f (t a)
    sequenceA xs = traverse id xs
```

This is actually how it's done in the `Data.Traversable` module, except that if you look at the source code you'll see the functions defined in point free style.

```
class (Functor t, Foldable t) => Traversable t where
    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
    traverse f = sequenceA . fmap f

    sequenceA :: Applicative f => t (f a) -> f (t a)
    sequenceA = traverse id
```

Default implementation for Functor and Foldable using Traversable

It might not be so obvious at first, but a `Traversable` is a very powerful concept. So powerful that it actually allows us to define both `Functor` and `Foldable` if we have just a single function from `Traversable`. The `Data.Traversable` module defines two functions, `fmapDefault` and `foldMapDefault`, which can be used as an implementation for `fmap` and `foldMap` if we so desire.

```
fmapDefault :: Traversable t => (a -> b) -> t a -> t b
foldMapDefault :: (Traversable t, Monoid m) => (a -> m) -> t a -> m
```

The way we're going to implement these is very similar to what we did

in the Lens introduction article (http://blog.jakubarnold.cz/2014/07/14/lens-tutorial-introduction-part-1.html). If this section is too hard for you to understand I recommend reading the Lens article first and then come back here. Everything will make a lot more sense.

Let's first compare the types of `traverse` and `fmap`.

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

The difference is that the function passed to `traverse` returns a value wrapped in `Applicative` context, and the result is also wrapped. If we could find a way to wrap the value after we apply the function, and then unwrap it at the end, we would get exactly the same type as `fmap`.

We can use the `Identity` functor to do this, which defines a way to unwrap it using `runIdentity :: Identity a -> a`.

```
fmapDefault :: Traversable t => (a -> b) -> t a -> t b
fmapDefault f x = _
```

We don't have that many options here. To be able to give the function `f` to a `traverse` we need to change it's type from `a -> f a`. That's where `Identity` comes in.

```
fmapDefault :: Traversable t => (a -> b) -> t a -> t b
fmapDefault f x = traverse (Identity . f) x
```

Now our types don't align, since we are supposed to return `t b` but we are returning `Identity (t b)`. The solution here is the above mentioned `runIdentity` which simply unwraps the value.

```
fmapDefault :: Traversable t => (a -> b) -> t a -> t b
fmapDefault f x = runIdentity $ traverse (Identity . f) x
```

And once more in point free style.

```
fmapDefault :: Traversable t => (a -> b) -> t a -> t b
fmapDefault f = runIdentity . traverse (Identity . f)
```

Compare this to the definition of `over` and you can see how it *looks and feels* almost exactly the same.

```
over :: Lens s a -> (a -> a) -> s -> s
over ln f = runIdentity . ln (Identity . f)
```

We'll explain how this relates to Lenses in more detail in a followup article, but for now let's move on to `foldMapDefault`.

Implementing foldMapDefault

This part is very hard to understand, so be careful.

If we compare the type of `foldMapDefault` with `traverse` we can yet again see some similarity.

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

The difference from `fmapDefault` is that now we need a way to convert each element of the `Traversable` to a `Monoid`.

We will use the `Const` applicative here, which as it so happens also defines a `Monoid` instance.

```
foldMapDefault :: (Traversable t, Monoid m) => (a -> m) -> t a -> m
foldMapDefault f x = _
```

As previously we can only use `traverse` together with a function `a -> f b`, but we have `a -> m`, where by using `Const` we can do the `m -> f b`.

```
foldMapDefault :: (Traversable t, Monoid m) => (a -> m) -> t a -> m
foldMapDefault f x = traverse (Const . f) x
```

Again we're faced with the problem of having `Const m (t b)` instead of `m`, which can be solved using `getConst`.

```
foldMapDefault :: (Traversable t, Monoid m) => (a -> m) -> t a -> m
foldMapDefault f x = getConst $ traverse (Const . f) x
```

And a point free version.

```
foldMapDefault :: (Traversable t, Monoid m) => (a -> m) -> t a -> m
foldMapDefault f = getConst . traverse (Const . f)
```

This is also very similar to one of the functions Lens provides, in particular `view`.

```
view :: Lens s a -> s -> a
view ln = getConst . ln Const
```

Implementing Functor and Foldable with Traversable

Now that we understand how both `fmapDefault` and `foldMapDefault` work, we can use them to define a `Functor` and a `Foldable` instance for any `Traversable` we might have.

We can test this out by defining a simple list type.

```haskell
data List a = Nil
            | Cons a (List a)
            deriving Show

instance Functor List where
    fmap = fmapDefault

instance Foldable List where
    foldMap = foldMapDefault

instance Traversable List where
    traverse _ Nil = pure Nil
    traverse f (Cons x xs) = fmap Cons (f x) <*> traverse f xs
```

We used `fmap = fmapDefault` and `foldMap = foldMapDefault` to define our `Functor` and `Foldable` instances, which is all made possible by also having a `Traversable` instance. Let's test this out to make sure it works!

```
λ> traverse (\x -> Just (x + 1)) (Cons 1 (Cons 2 (Cons 3 Nil)))
Just (Cons 2 (Cons 3 (Cons 4 Nil)))
λ> fold (Cons "hello" (Cons "world" Nil))
"helloworld"
λ> fmap (+1) (Cons 1 (Cons 2 (Cons 3 Nil)))
Cons 2 (Cons 3 (Cons 4 Nil))
```

It might be a surprising, but everything works as it is supposed to.