

Foldable and Traversable

A quick tour of two common patterns.

Ben Kolera

bfpg.org

September 23, 2014

Foldable: The Typeclass

Something that can be reduced with any monoid. Instances must have foldMap ¹:

```
data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)
    deriving (Show, Functor)

instance Foldable Tree where
-- foldMap :: (Monoid m) => (a -> m) -> Tree a -> m
    foldMap f Empty          = mempty
    foldMap f (Leaf x)       = f x
    foldMap f (Node l k r)   = foldMap f l <> f k <> foldMap f r
```

¹Can implement foldr instead and there are a few functions overridable for performance reasons.

Foldable: Execution

```
exampleTree1 = (Node (Leaf 1) 2 (Node Empty 3 (Leaf 4)))
```

```
foldMap f Empty      = mempty
```

```
foldMap f (Leaf x)   = f x
```

```
foldMap f (Node l k r) = foldMap f l <> f k <> foldMap f r
```

```
example1 = foldMap Sum exampleTree1
```

```
-- foldMap Sum (Leaf 1) <> Sum 2 <> foldMap Sum (Node Empty 3 (Leaf 4))
```

```
-- Sum 1 <> Sum 2 <> mempty <> Sum 3 <> foldMap Sum (Leaf 4)
```

```
-- Sum 1 <> Sum 2 <> mempty <> Sum 4 <> Sum 4
```

```
-- Sum 10
```

Functions Derived From Foldable

```
example2 = mapM_ print exampleTree1
-- Prints 1 then 2 then 3 then 4 all on their own lines.

-- |
-- >>> example3
-- "abc"
example3 = fold (Node (Leaf "a") "b" (Leaf "c"))

-- |
-- >>> example4
-- Just 2
example4 = find even exampleTree1

-- |
-- >>> example5
-- [1,2,3,4]
example5 = toList exampleTree1
```

Traversable: The Typeclass

Walks structure like foldable but runs an applicative at each node rather than reducing.

```
instance Traversable Tree where
-- traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  traverse f Empty          = pure Empty
  traverse f (Leaf x)       = Leaf <$> f x
  traverse f (Node l k r) = Node <$> traverse f l <*> f k <*> traverse f r
```

Traversable Execution

```
-- |
-- >>> example6
-- Node (Leaf "I'm from A") "I'm from B" Empty
example6 = traverse readFile (Node (Leaf "fileA") "fileB" Empty)

-- Node
--   <$> traverse readFile (Leaf "fileA")
--   <*> readFile "fileB"
--   <*> traverse readFile Empty

-- Node
--   <$> (Leaf <$> readFile "fileA")
--   <*> readFile "fileB"
--   <*> pure Empty
```

Functions Derived from Traversable

(These are already defined in `Data.Traversable` as `foldMapDefault` and `fmapDefault`)

```
newtype Id a = Id { getId :: a } deriving (Functor)
instance Applicative Id where
    pure = Id
    Id f <*> Id x = Id (f x)
-- |
-- >>> fmap' (*2) exampleTree1
-- Node (Leaf 2) 4 (Node Empty 6 (Leaf 8))
fmap' :: Traversable t => (a -> b) -> t a -> t b
fmap' f = getId . traverse (Id . f)

-- |
-- >>> foldMap' Sum exampleTree1
-- Sum {getSum = 10}
foldMap' :: (Traversable t, Monoid m) => (a -> m) -> t a -> m
foldMap' f = getConst . traverse (Const . f)
```

The point to all of this

- We don't really gain lots of free code, so what's the point?
- But we do give a name to two very common patterns with data.
- At its crudest, you avoid some namespace collisions and save some imports.
- At its finest you can now right APIs that work on traversable giving the user the ability to use any traversable rather than the one you picked.
- E.g: The user can use `Data.List` or `Data.List.NonEmpty` based on their needs.

Package `base-prelude` is an alternative prelude that hides the `sequence`, `mapM`, `foldr`, etc that are implemented around `Data.List` and exports the `Foldable` and `Traversable` ones.

`Base prelude` is a nice step towards a more abstract and reusable prelude. Check it out! :)

The End

Thanks for listening!

(I'll put this online sometime tomorrow and send the deets to the list.)