Folds and monoids

CIS 194 Week 7
25 February 2013

Suggested reading:

- Learn You a Haskell, Only folds and horses
- Learn You a Haskell, Monoids
- Fold from the Haskell wiki
- Heinrich Apfelmus, Monoids and Finger Trees
- Dan Piponi, Haskell Monoids and their Uses
- Data.Monoid documentation
- Data.Foldable documentation

Folds, again

We've already seen how to define a folding function for lists… but we can generalize the idea to other data types as well!

Consider the following data type of binary trees with data stored at internal nodes:

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

leaf :: a -> Tree a
leaf x = Node Empty x Empty
```

Let's write a function to compute the size of a tree (*i.e.* the number of `Node`s):

```
treeSize :: Tree a -> Integer
treeSize Empty        = 0
treeSize (Node l _ r) = 1 + treeSize l + treeSize r
```

How about the sum of the data in a tree of `Integer`s?

```
treeSum :: Tree Integer -> Integer
treeSum Empty     = 0
treeSum (Node l x r)  = x + treeSum l + treeSum r
```

Or the depth of a tree?

```
treeDepth :: Tree a -> Integer
treeDepth Empty        = 0
treeDepth (Node l _ r) = 1 + max (treeDepth l) (treeDepth r)
```

Or flattening the elements of the tree into a list?

```
flatten :: Tree a -> [a]
flatten Empty        = []
flatten (Node l x r) = flatten l ++ [x] ++ flatten r
```

Are you starting to see any patterns? Each of the above functions:

1. takes a `Tree` as input
2. pattern-matches on the input `Tree`
3. in the `Empty` case, gives a simple answer
4. in the `Node` case:
    1. calls itself recursively on both subtrees
    2. somehow combines the results from the recursive calls with the data `x` to produce the final result

As good programmers, we always strive to abstract out repeating patterns, right? So let's generalize. We'll need to pass as parameters the parts of the above examples which change from example to example:

1. The return type
2. The answer in the `Empty` case
3. How to combine the recursive calls

We'll call the type of data contained in the tree `a`, and the type of the result `b`.

```
treeFold :: b -> (b -> a -> b -> b) -> Tree a -> b
treeFold e _ Empty        = e
treeFold e f (Node l x r) = f (treeFold e f l) x (treeFold e f r)
```

Now we should be able to define `treeSize`, `treeSum` and the other examples much more simply. Let's try:

```
treeSize' :: Tree a -> Integer
treeSize' = treeFold 0 (\l _ r -> 1 + l + r)

treeSum' :: Tree Integer -> Integer
treeSum' = treeFold 0 (\l x r -> l + x + r)

treeDepth' :: Tree a -> Integer
treeDepth' = treeFold 0 (\l _ r -> 1 + max l r)

flatten' :: Tree a -> [a]
flatten' = treeFold [] (\l x r -> l ++ [x] ++ r)
```

We can write new tree-folding functions easily as well:

```
treeMax :: (Ord a, Bounded a) => Tree a -> a
treeMax = treeFold minBound (\l x r -> l `max` x `max` r)
```

Much better!

Folding expressions

Where else have we seen folds?

Recall the `ExprT` type and corresponding `eval` function from Homework 5:

```
data ExprT = Lit Integer
           | Add ExprT ExprT
           | Mul ExprT ExprT

eval :: ExprT -> Integer
eval (Lit i)     = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

Hmm… this looks familiar! What would a fold for `ExprT` look like?

```
exprTFold :: (Integer -> b) -> (b -> b -> b) -> (b -> b -> b) -> ExprT -> b
exprTFold f _ _ (Lit i)     = f i
exprTFold f g h (Add e1 e2) = g (exprTFold f g h e1) (exprTFold f g h e2)
exprTFold f g h (Mul e1 e2) = h (exprTFold f g h e1) (exprTFold f g h e2)

eval2 :: ExprT -> Integer
eval2 = exprTFold id (+) (*)
```

Now we can easily do other things like count the number of literals in an expression:

```
numLiterals :: ExprT -> Int
numLiterals = exprTFold (const 1) (+) (+)
```

Folds in general

The take-away message is that we can implement a fold for many (though not all) data types. The fold for `T` will take one (higher-order) argument for each of `T`'s constructors, encoding how to turn the values stored by that constructor into a value of the result type—assuming that any recursive occurrences of `T` have already been folded into a result. Many functions we might want to write on `T` will end up being expressible as simple folds.

Monoids

Here's another standard type class you should know about, found in the `Data.Monoid` module:

```
class Monoid m where
    mempty  :: m
    mappend :: m -> m -> m

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

```
(<>) :: Monoid m => m -> m -> m
(<>) = mappend
```

`(<>)` is defined as a synonym for `mappend` (as of GHC 7.4.1) simply because writing `mappend` is tedious.

Types which are instances of `Monoid` have a special element called `mempty`, and a binary operation `mappend` (abbreviated `(<>)`) which takes two values of the type and produces another one. The intention is that `mempty` is an identity for `<>`, and `<>` is associative; that is, for all `x`, `y`, and `z`,

1. `mempty <> x == x`
2. `x <> mempty == x`
3. `(x <> y) <> z == x <> (y <> z)`

The associativity law means that we can unambiguously write things like

```
a <> b <> c <> d <> e
```

because we will get the same result no matter how we parenthesize.

There is also `mconcat`, for combining a whole list of values. By default it is implemented using `foldr`, but it is included in the `Monoid` class since particular instances of `Monoid` may have more efficient ways of implementing it.

`Monoid`s show up *everywhere*, once you know to look for them. Let's write some instances (just for practice; these are all in the standard libraries).

Lists form a monoid under concatenation:

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

As hinted above, addition defines a perfectly good monoid on integers (or rational numbers, or real numbers…). However, so does multiplication! What to do? We can't give two different instances of the same type class to the same type. Instead, we create two `newtype`s, one for each instance:

```
newtype Sum a = Sum a
  deriving (Eq, Ord, Num, Show)

getSum :: Sum a -> a
getSum (Sum a) = a

instance Num a => Monoid (Sum a) where
  mempty  = Sum 0
  mappend = (+)

newtype Product a = Product a
  deriving (Eq, Ord, Num, Show)
```

```haskell
getProduct :: Product a -> a
getProduct (Product a) = a

instance Num a => Monoid (Product a) where
  mempty  = Product 1
  mappend = (*)
```

Note that to find, say, the product of a list of `Integer`s using `mconcat`, we have to first turn them into values of type `Product Integer`:

```haskell
lst :: [Integer]
lst = [1,5,8,23,423,99]

prod :: Integer
prod = getProduct . mconcat . map Product $ lst
```

(Of course, this particular example is silly, since we could use the standard `product` function instead, but this pattern does come in handy somtimes.)

Pairs form a monoid as long as the individual components do:

```haskell
instance (Monoid a, Monoid b) => Monoid (a,b) where
  mempty = (mempty, mempty)
  (a,b) `mappend` (c,d) = (a `mappend` c, b `mappend` d)
```

Challenge: can you make an instance of `Monoid` for `Bool`? How many different instances are there?

Challenge: how would you make function types an instance of `Monoid`?