Intro to Haskell Notes: Part 7

Adrian Brasoveanu*

October 7, 2013

Contents

1 Folds

Back when we were dealing with recursion, we noticed a theme throughout many of the recursive functions that operated on lists:

- we had an edge case for the empty list
- we introduced the $x : xs$ pattern and then we did some action that involved a single element and the rest of the list

*Based primarily on *Learn You a Haskell for Great Good!* by Miran Lipovača, `http://learnyouahaskell.com/`.

It turns out this is a very common pattern, so a couple of very useful functions were introduced to encapsulate it. These functions are called *folds*. They're like the *map* function, only they reduce the list that *map* outputs to a single value.

That is, a fold takes:

- a binary function
- a starting value, a.k.a. the accumulator
- a list to fold up

And the fold works as follows:

- the binary function is called with the accumulator and the first element of the list (or the last element, depending on whether we fold from the left or from the right), and produces a new accumulator
- then, the binary function is called again with the new accumulator and the now new first (or last) element, and so on
- once we've walked over the whole list, only the accumulator remains, which is what we've reduced the list to

The accumulator value (and hence the result) of a fold can be of any type. It can be a number, a boolean or even a new list.

Folds can be used to implement any function where you traverse a list once, element by element, and then return something based on that.

Whenever you want to traverse a list to return something, chances are you want a fold. That's why folds are, along with maps and filters, one of the most useful types of functions in functional programming.

2 Left folds

First let's take a look at the *foldl* function, also called the left fold:

- it folds the list up from the left side
- the binary function is applied to the starting accumulator and the head of the list
- that produces a new accumulator value and the binary function is called with that value and the next element of the list etc.

Here's its type:

```
ghci 1> :t foldl
    foldl :: (a → b → a) → a → [b] → a
```

And here's the documentation for *foldl*:

```
ghci 2> :! hoogle  -- info foldl
  Prelude foldl :: (a -> b -> a) -> a -> [b] -> a
```

foldl, applied to a binary operator, a starting value (typically the left-identity of the operator), and a list, reduces the list using the binary operator, from left to right:

$$foldl\ f\ z\ [x1, x2, ..., xn] \equiv (...((z\ `f`\ x1)\ `f`\ x2)\ `f`\ ...)\ `f`\ xn$$

The list must be finite.

From package base foldl :: (a -> b -> a) -> a -> [b] -> a

2.1 Implementing *sum* again

Let's implement *sum* again, only this time we'll use a fold instead of explicit recursion.

```
ghci 3> let {sum' :: (Num a) ⇒ [a] → a;
             sum' xs = foldl (λacc x → acc + x) 0 xs}
```

Note how the starting value / accumulator 0 is indeed the left (and right) identity of the binary operator $+$.

```
ghci 4> sum' [3, 5, 2, 1]
  11
```

Let's examine how this fold happens:

- $\lambda acc\ x \rightarrow acc + x$ is the binary function
- 0 is the starting accumulator and *xs* is the list to be folded up
- first, 0 is used as the *acc* argument of the binary function and 3 is used as the *x* (i.e., the current element) argument; $0 + 3$ produces a 3 and this becomes the new accumulator value
- next, 3 is used as the accumulator value and 5 as the current element and 8 becomes the new accumulator value
- now 8 is the accumulator value, 2 the current element, and the resulting new accumulator value is 10
- finally, 10 is used as the accumulator value and 1 as the current element, producing an 11

This example makes it very clear that folds are really the functional-language counterparts of list-based loops in imperative languages.

If we take into account that functions are curried, we can write the implementation of *sum′* even more succinctly in point free style:

```
ghci 5> let {sum' :: (Num a) ⇒ [a] → a;
             sum' = foldl (+) 0}
```

The lambda function $(\lambda acc\ x \rightarrow acc + x)$ is the same as $(+)$. And we can omit *xs* as the argument because calling *foldl* $(+)$ 0 will return a function that takes a list as its argument.

```
ghci 6> sum′ [3,5,2,1]
   11
```

2.2 Implementing *elem* again

Let's implement *elem* again, only this time with a left fold.

```
ghci 7> let {elem′ :: (Eq a) ⇒ a → [a] → Bool;
        elem′ y ys = foldl (λacc x → if x ≡ y then True else acc) False ys}
```

The starting accumulator here has to be a boolean value since the accumulator and the end result are always of the same type when dealing with folds. If you don't know what to use as a starting accumulator, this will give you some idea.

In this particular case, it makes sense to use *False* as a starting accumulator:

- we assume the element is not in the set until proven otherwise;
- nice consequence: if we call the fold with an empty list, the result will be just the starting value.

The *elem′* function checks whether the current element in the list is the element we're looking for:

- if it is, we set the accumulator to *True*;
- if it isn't, we just leave the accumulator unchanged:
  - if it was *False* before, it stays that way;
  - and if it was *True*, we already have our final result so we don't touch it.

```
ghci 8> 3 ‘elem′‘ [2,3,5,7]
   True
```

```
ghci 9> 9 ‘elem′‘ [2,3,5,7]
   False
```

We can rewrite this function in (almost) point free style too:

```
ghci 10> let {elem″ :: (Eq a) ⇒ a → [a] → Bool;
         elem″ y = foldl (λacc x → if x ≡ y then True else acc) False}
```

```
ghci 11> 3 ‘elem″‘ [2,3,5,7]
   True
```

```
ghci 12> 9 ‘elem″‘ [2,3,5,7]
   False
```

3 Right folds

The right fold *foldr* works in a similar way to the left fold, except:

- we fold the list from the right;
- while the left fold's binary function has the accumulator as the first argument and the current value as the second one

  $$\lambda acc\ x \rightarrow \ldots \qquad \text{(binary operator for } \textit{foldl}\text{)}$$

  the right fold's binary function has the current list element as the first argument and the accumulator as the second one:

  $$\lambda x\ acc \rightarrow \ldots \qquad \text{(binary operator for } \textit{foldr}\text{)}$$

These two differences go together: the binary function in a right fold takes the accumulator on the right because we are folding the list from the right side.

Here's the type of *foldr*:

```
ghci 13> :t foldr
    foldr :: (a → b → b) → b → [a] → b
```

- as you can see, the type of the accumulator and also the type of the resulting value is *b*, while the type of entities in the list to be folded over is *a*;
- and as we already mentioned, the binary operator is of type $a \rightarrow b \rightarrow b$, i.e., it takes a list element as its first argument and the accumulator as its second argument;
- but unfortunately, the *foldr* function overall does not take the whole list (of type $[a]$) as its first argument and the accumulator as its second argument; instead, it preserves the same argument order as *foldl*: the first argument is the accumulator and the list is the second argument;
- while this makes things a bit confusing (the binary operator has its arguments reversed, but *foldr* as a whole doesn't), it helps if we want to use *foldr* point-free style: we can simply specify the accumulator and omit the list to be folded over, just like we do for *foldl*.

Finally, here's the documentation for *foldr*:

```
ghci 14> :! hoogle -- info foldr
    Prelude foldr :: (a -> b -> b) -> b -> [a] -> b
    foldr, applied to a binary operator, a starting value (typically the right-identity of the oper-
    ator), and a list, reduces the list using the binary operator, from right to left:

        foldr f z [x1, x2, ..., xn] ≡ x1 ‘f‘ (x2 ‘f‘ ...(xn ‘f‘ z)...)

    From package base foldr :: (a -> b -> b) -> b -> [a] -> b
```

3.1 Implementing *map*

We'll be implementing the *map* function with a right fold:

- the accumulator will be a list and we'll accumulate the mapped list element by element;
- so the starting accumulator has to be an empty list.

```
ghci 15> let {map' :: (a -> b) -> [a] -> [b];
              map' f xs = foldr (\x acc -> f x : acc) [] xs}
```

Implementing *map* with a right fold is the efficient thing to do: we start folding from the right, i.e., from the end of the list to be mapped over, so we can incrementally build the output list by prepending elements. And recall that the prepend operator : is much more efficient than concatenation ++, which is what we would have to use with a left fold:

- the list on the right-hand side of the prepend : or concat ++ operators is always left untouched,
- while the list on the left-hand side of the concat ++ operator is always 'decomposed' element by element from the end to the beginning, and these elements are prepended to the right-hand side list one at a time.

We'll come back to this in a moment. Now let's look at an example involving the *map'* function to see how *foldr* works step by step:

```
ghci 16> map' (+3) [1,2,3]
    [4,5,6]
```

When we map the unary function $(+3)$ over the list $[1,2,3]$ with *map'*, we approach the list from the right:

- we take the last element, which is 3, and apply the function $(+3)$ to it, which results in 6
- we prepend the result to the accumulator, which is $[\,]$, so the new accumulator is $6:[\,]$, i.e., $[6]$
- now we apply $(+3)$ to 2, and we prepend the result to the current accumulator; so the accumulator is now $5:[6]$, i.e., $[5,6]$
- finally, we apply $(+3)$ to 1 and prepend that to the current accumulator; so the final result is $4:[5,6]$, i.e., $[4,5,6]$

Just as with functions involving *foldl*, we could write the *map'* function in point free style because of the (somewhat confusing, but convenient) order in which the accumulator and list arguments are fed to the *foldr* function:

```
ghci 17> let {map'' :: (a -> b) -> [a] -> [b];
              map'' f = foldr (\x acc -> f x : acc) []}
```

```
ghci 18> map'' (+3) [1,2,3]
    [4,5,6]
```

4 Right folds vs. left folds

Note that we can implement the *map* function with a left fold too.

```
ghci 19> let {map''' :: (a → b) → [a] → [b];
              map''' f = foldl (λacc x → acc ++ [f x]) [ ]}
```

```
ghci 20> map''' (+3) [1,2,3]
    [4,5,6]
```

But as we already discussed, the ++ function is much more expensive than :, so we usually use right folds when we're building up new lists from a list.

In addition, right folds work on infinite lists while left folds don't:

- if we take an infinite list, choose an element and fold the list up from the right starting with that element, we'll eventually reach the beginning of the list;
- however, if we take an infinite list and we try to fold it up from the left, we'll never reach the end.

5 Folding without explicit accumulators: *foldl1* and *foldr1*

The *foldl1* and *foldr1* functions work like *foldl* and *foldr*, except we don't need to provide an explicit starting accumulator:

- they take the first (or last) element of the list to be the starting accumulator and then start the fold with the element next to it.

Here are their types and documentation:

```
ghci 21> :t foldl1
    foldl1 :: (a → a → a) → [a] → a
```

```
ghci 22> :! hoogle  -- info foldl1
    Prelude foldl1 :: (a -> a -> a) -> [a] -> a
    foldl1 is a variant of foldl that has no starting value argument, and thus must be applied to
    non-empty lists.
    From package base foldl1 :: (a -> a -> a) -> [a] -> a
```

```
ghci 23> :t foldr1
    foldr1 :: (a → a → a) → [a] → a
```

```
ghci 24>  :! hoogle  -- info foldr1
    Prelude foldr1 :: (a -> a -> a) -> [a] -> a
    foldr1 is a variant of foldr that has no starting value argument, and thus must be applied to
    non-empty lists.
    From package base foldr1 :: (a -> a -> a) -> [a] -> a
```

Because *foldl1* and *foldr1* depend on the lists they fold up having at least one element, they cause runtime errors if called with empty lists. *foldl* and *foldr* on the other hand work fine with empty lists.

When making a fold, think about how it acts on an empty list: if the function doesn't make sense when given an empty list, you can probably use a *foldl1* or *foldr1* to implement it.

5.1 Implementing *sum* again

We can now implement the *sum* function in a very clear and concise (point free) way:

```
ghci 25>  let sum' = foldl1 (+)
```

```
ghci 26>  sum' [1 . . 10]
    55
```

6 Strict folds

foldl′ and *foldl1′* are stricter versions of their respective lazy incarnations. They are made available in the *Data.List* module, which will be discussed in the following set of lecture notes.

We need strict folds because when we use lazy folds on really big lists, we might get stack overflow errors:

- the accumulator value isn't actually updated as the folding happens because of the lazy nature of the folds;
- what actually happens is that the accumulator makes a promise that it will compute its value when asked to actually produce the result; this promise is called a thunk;
- and this happens for every intermediate accumulator, so all those thunks overflow the stack.

The strict folds aren't lazy and actually compute the intermediate values as they go along instead of filling up the stack with thunks.

So if you ever get stack overflow errors when doing lazy folds, try switching to their strict versions.

```
ghci 27>  :! hoogle  -- info "foldl'"
    Data.List foldl' :: (a -> b -> a) -> a -> [b] -> a
    A strict version of foldl.
    From package base foldl' :: (a -> b -> a) -> a -> [b] -> a
```

```
ghci 28> :! hoogle -- info "foldl1'"
    Data.List foldl1' :: (a -> a -> a) -> [a] -> a
    A strict version of foldl1
    From package base foldl1' :: (a -> a -> a) -> [a] -> a
```

Strict right folds, i.e., the *foldr′* and *foldr1′* functions, are made available in various other modules, for example:

```
ghci 29> :! hoogle -- info "foldr'"
    Data.Foldable foldr' :: Foldable t => (a -> b -> b) -> b -> t a -> b
    Fold over the elements of a structure, associating to the right, but strictly.
    From package base foldr' :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

```
ghci 30> :! hoogle -- info "foldr1'"
    Data.ByteString.Char8 foldr1' :: (Char -> Char -> Char) -> ByteString -> Char
    A strict variant of foldr1
    From package bytestring foldr1' :: (Char -> Char -> Char) -> ByteString -> Char
```

7 More examples with folds

To see how powerful folds are, we're going to implement a bunch of standard library functions by using folds. This is also excellent practice for writing 'looping' functions in a purely functional manner.

We'll use point free style extensively because it enables us to focus on the functions themselves rather than the data we operate on.

7.1 Implementing *maximum* again

```
ghci 31> let {maximum' :: (Ord a) ⇒ [a] → a;
              maximum' = foldr1 (λx acc → if x > acc then x else acc)}
```

7.2 Implementing *reverse* again

```
ghci 32> let {reverse' :: [a] → [a];
              reverse' = foldl (λacc x → x : acc) [ ]}
```

Our *reverse′* definition takes the empty list as the starting accumulator, approaches our list from the left and prepends to the accumulator. In the end, we build up a reversed list.

Note that $\lambda acc\ x \rightarrow x : acc$ looks like the prepend : function, only its arguments are flipped. That's why we could have also written our *reverse′* as:

$$foldl\ (flip\ (:))\ [\,]$$

7.3 Implementing *product*

```
ghci 33> let {product' :: (Num a) ⇒ [a] → a;
              product' = foldr1 (∗)}
```

7.4 Implementing *filter*

```
ghci 34> let {filter' :: (a → Bool) → [a] → [a];
              filter' p = foldr (λx acc → if p x then x : acc else acc) []}
```

7.5 Implementing *head* again

```
ghci 35> let {head' :: [a] → a;
              head' = foldr1 (λx _ → x)}
```

head is more clearly implemented by pattern matching, which is what we did before, but we can also do it using folds.

The previous pattern-matching implementation is provided below for ease of reference:

```
(1)  head' :: [a] → a
     head' [] = error "Can't call head on an empty list, dummy!"
     head' (x: _) = x
```

7.6 Implementing *last*

```
ghci 36> let {last' :: [a] → a;
              last' = foldl1 (λ_ x → x)}
```

7.7 Summary: Unpacking the folds into successive function applications

Another way to picture right and left folds (already hinted at in the documentation) is as follows.

Suppose we have a right fold with the binary function f and the starting accumulator z. If we're right folding over the list $[3,4,5,6]$, we're essentially doing this:

$$f\ 3\ (f\ 4\ (f\ 5\ (f\ 6\ z)))$$

That is, f is called with the last element in the list and the accumulator, the result is given as the accumulator to the penult value, and so on.

For example, if we take f to be $+$ and the starting accumulator value to be 0, we have:

$$3 + (4 + (5 + (6 + 0)))$$

Similarly, doing a left fold over that list with g as the binary function and z as the starting accumulator is the equivalent of:

$$g\ (g\ (g\ (g\ z\ 3)\ 4)\ 5)\ 6$$

For example, if g is the *flip* (:) function and [ ] is the starting accumulator (so we're reversing the list), then calling *foldl* on the list $[3,4,5,6]$ is equivalent to:

$$\mathit{flip}\ (:)\ (\mathit{flip}\ (:)\ (\mathit{flip}\ (:)\ (\mathit{flip}\ (:)\ [\,]\ 3)\ 4)\ 5)\ 6$$

And if we evaluate this expression, we get $6 : 5 : 4 : 3 : [\,]$, i.e., $[6,5,4,3]$, as expected.

8 Scans

scanl and *scanr* are like *foldl* and *foldr*, but they report all the intermediate accumulator states in the form of a list. Similarly, *scanl1* and *scanr1* are analogous to *foldl1* and *foldr1*.

```
ghci 37> :! hoogle -- info scanl
    Prelude scanl :: (a -> b -> a) -> a -> [b] -> [a]
    scanl is similar to foldl, but returns a list of successive reduced values from the left:

        scanl f z [x1, x2, ...] ≡ [z, z `f` x1, (z `f` x1) `f` x2, ...]

    Note that

        last (scanl f z xs) ≡ foldl f z xs ∘

    From package base scanl :: (a -> b -> a) -> a -> [b] -> [a]
```

```
ghci 38> :! hoogle -- info scanr
    Prelude scanr :: (a -> b -> b) -> b -> [a] -> [b]
    scanr is the right-to-left dual of scanl. Note that

        head (scanr f z xs) ≡ foldr f z xs ∘

    From package base scanr :: (a -> b -> b) -> b -> [a] -> [b]
```

As the documentation makes clear, when using a *scanl*, the final result will be in the last element of the resulting list while a *scanr* will place the result in the head.

Scans are used to monitor the progression of a function that can be implemented as a fold. Here are some examples:

```
ghci 39> scanl (+) 0 [3, 5, 2, 1]
    [0, 3, 8, 10, 11]
```

```
ghci 40> scanl1 (+) [3, 5, 2, 1]
    [3, 8, 10, 11]
```

Note that *scanl* (+) 0, and *scanl1* (+) in particular, are just ways of obtaining cumulative sums over a list, which are useful for the examination of empirical probability distributions among other things.

```
ghci 41> scanl (flip (:)) [] [3,2,1]
    [[],[3],[2,3],[1,2,3]]
```

```
ghci 42> scanr (+) 0 [3,5,2,1]
    [11,8,3,1,0]
```

```
ghci 43> scanr1 (+) [3,5,2,1]
    [11,8,3,1]
```

```
ghci 44> scanl1 (\acc x -> if x > acc then x else acc) [3,4,5,3,7,9,2,1]
    [3,4,5,5,7,9,9,9]
```

8.1 Sum the square roots of natural numbers until we exceed 1000

Let's answer this question: how many numbers does it take for the sum of the square roots of all natural numbers to exceed 1000?

- to get the square roots of all natural numbers, we just do *map sqrt* [1 ..];
- now, to get the sum, we could do a fold, but we're going to do a scan because we're interested in how the sum progresses;
- once we've done the scan, we just see how many sums are under 1000.

```
ghci 45> let {sqrtSums :: Int;
              sqrtSums = 1 + length (takeWhile (<1000) $ scanl1 (+) $ map sqrt [1 ..])}
```

- the first sum in the scan list will be 1;
- the second will be 1 plus the square root of 2;
- the third will be that plus the square root of 3, and so on;
- if there are X sums under 1000, then it takes $X + 1$ elements for the sum to exceed 1000.

```
ghci 46> sqrtSums
    131
```

We use *takeWhile* here instead of *filter* because *filter* doesn't work on infinite lists:

- we know the list is ascending, but *filter* doesn't;
- so we use *takeWhile* to cut the scan list off at the first occurence of a sum greater than 1000.

```
ghci 47> sum $ map sqrt [1 .. 131]
    1005.0942035344083
```

```
ghci 48> sum $ map sqrt [1 .. 130]
    993.6486803921487
```