CSE 382: Week 04

The Functor Pattern Part 2: reduce/fold and unfold

Recursive programs have a systematic translation to unfold and fold.

- Conal Elliott

Beyond Map and Filter

The functors map and filter are great. Use them whenever you can and they are the best choice for that situation. They do not cover all situations. That's why there are other basic functors that are part of every functional programmer's tool set. These are reduce (formally known as fold left and fold right) and unfold. It's these functors that really flesh out your tool set.

Fold Left and Fold Right

The basic concept of folding is that you start with something large and reduce its exposed surface area. This is true not only for clothes, paper, and and other such things, but for data as well. If your code snippet, function, or application accepts a bunch of data and then produces, as output, fewer computed results than the amount of data it started with, your snippet, function, or application was a data fold. Maybe your app took in a bunch of historical data about the weather and produced a prediction for the next few days. That app is a fold.

Definition 1 - *Fold*:

Maybe your app accessed a database of customers and produced a filtered, ordered list of customers a function type where the input count is greater than the output count.

Definition 1

that your sales people should contact due to the customer's specific needs. This is a partial rather than a complete list. That behavior is a fold.

Or maybe you use an application to help you track and fix bugs in your code. It accesses the entire database of defects and selects just the ones that match details you provide it. That's another fold app.

When you stop and think about it, the number of applications, functions, and code snippets that perform a fold-type operation is huge...and thankfully so. Imagine if the data produced by any of the apps mentioned was as large as the input data. They would be nearly useless.😬

Fold behavior comes in two flavors, fold left, moving in the list from left to right, and fold right, moving in the list from right to left. There will be times when selecting between right and left types of folds is important. There will be other times when it is not.

foldl

Fig. 1 - The declaration and anti-pattern definition of the `foldl` functor.

Let's start with the `foldl` function. Take a look at Figure 2. There you will see `foldl`'s declaration states

```
$foldl :: [a]\ accum\ (\lambda ::\ a\ b \to c) \to c$
$foldl :: [a]\ accum\ \ (\lambda ::\ a\ b \to c) \to accum$
        $when\ [a]$ is empty ;
$foldl :: [h \mid t]\ accum\ \ (\lambda ::\ a\ b \to c) \to$
      $temp = \lambda\ ::\ accum\ h$
      $foldl\ ::\ t\ \ temp\ (\lambda ::\ a\ b \to c).$
```

that the functor's set of parameters is a list, an accumulator, and a lambda function. It also states the value of foldl is a single value. It is important to understand that the accumulator can be of any type but that the type of the lambda's parameter 'a' must match the type of the accumulator for the functor. Also, if the accumulator is numeric, its initial value could be 0 if you are wanting to do something similar to addition, or its initial value could be 1 if you are wanting to do something like multiplication.

Definition 2 - *Anti-pattern*:

> a common response to a common problem that has a high probability of being ineffective or counterproductive.

There are other things your lambda function could be doing with the list. If it is concatenating the list, the accum parameter could be an empty string. If, however, you are thinking of passing in an empty list as the accumulator, you should think again. That would be an anti-pattern. You would be better served by using a combination of map and filter functors in this situation rather than a fold functor.

Notice in the second clause of foldl a local variable called temp is created and is assigned the value of the lambda when the lambda is passed the accumulator and the head of the list. In the last phrase of the second clause this temporary variable is passed as the accumulator value to recursive call in the second phrase of this same clause of foldl.

Fig. 2 - The declaration and good definition of the foldl functor.

```
$foldl :: [a]\ accum\ (\lambda ::\ a\ b \to c) \to c$
$foldl :: [a]\ accum\ \ (\lambda ::\ a\ b \to c) \to accum$
        $when\ [a]$ is empty ;
$foldl :: [h \mid t]\ accum\ \ (\lambda ::\ a\ b \to c) \to$
        $foldl\ \ ::\ \ t\ \ (\lambda\ \ ::\ \ accum\ h)\ (\lambda ::\ \ a\ b \to c).$
```

While organizing your code like it is in Figure 1 may seem to make sense, it should be avoided.

Figure 1 is included in this reading as an example of a common anti-pattern. Anti-patterns are patterns that seem good but have hidden badness. Don't use them. Instead, the temp variable should be discarded and the call to lambda should be placed in the last phrase of the clause as you see in Figure 2.

foldr

`Foldr` is much like `foldl`. That's why the declarations of the two functors is nearly identical. The difference between the two shows up in the definitions. In the pseudocode found in Figure 3 you'll find that the major difference is how `h` and `t` are dealt with. For this bit of pseudocode t represents the last element of the

Fig. 3 - The declaration and definition of the foldr functor.

```
$foldr :: [a]\ accum\ (\lambda ::\ a\ b \to c) \to c$
$foldr :: [a]\ accum\ \ (\lambda ::\ a\ b \to c) \to accum$
        $when\ [a]$ is empty ;
$foldr :: [h : t]\ accum\ \ (\lambda ::\ a\ b \to c) \to$
        $foldr\ \ ::\ \ h\ \ (\lambda\ \ ::\ \ accum\ t)\ (\lambda ::\ \ a\ b \to c).$
```

list and h represents the rest of the remaining elements of the list. You'll find as you examine Figure 3 that while `foldl` moves from the head element to the tail element, `foldr` moves from the tail element to the head element. That's why you will see the h and t variables switching places if you compare the last clause of foldl with the last clause of foldr.

Unfold

While the fold functors accumulate a list of values into a single value, unfold does the opposite. Unfold produces a potentially infinite set of results from a set of initial values called the seed. This seed set may have only one element.

Definition 3 - *Unfold*:

> a function type where the input count is less than the output count.

You've used `unfold` functors before. Think of when you've needed a pseudo-random number. When you used a function to get that number, you used an unfold functor. Random number generators need a seed to get started. You may not have needed to provide a seed value since there is usually a façade function provided by the library writers that, by default, uses the current time in milliseconds as the seed.

In many languages, `unfold` functors that produce random numbers are called Random Number Streams since after the initial seed is provided, repeated calls to the functor produces a predictable, calculable result. In other words, if the same seed is used for two different calls of the functor, the same results are generated in the same order.

Fig.4 - The declaration and definition of a naive unfold functor.

$$\begin{aligned}
&unfold\ ::\ c\ s\ (\lambda\ ::\ c\ s \to \{u,r\}) \to [r]\\
&unfold\ ::\ c\ s\ (\lambda\ ::\ c\ s \to \{u,r\}) \to\\
&\qquad [\,]\ when\ c = 0;\\
&unfold\ ::\ c\ s\ (\lambda\ ::\ c\ s \to \{u,r\}) \to\\
&\qquad \lambda\ ::\ c\ s \to \{a,b\}\\
&\qquad [b] +\!\!+\ unfold\ ::\ a\ b\ (\lambda\ ::\ c\ s \to \{u,r\})\ otherwise;
\end{aligned}$$

The version of unfold in Figure 4 is naive. It is not an infinite stream. Later in the course `unfold` is revisited when the lazy execution and stream patterns are covered in depth. Those patterns will show you how to create a non-naive, infinite `unfold` functors. For now, this version of `unfold` will suffice with `c` being the current count and `s` being the current state.

Notice that in this case, the second phrase of the second clause of the unfold functor is not an anti-pattern. It isn't an anti-pattern because the result of the call to the lambda function can not be directly included in the next call to the functor. Why?? Because, `b`, one of the elements of the 2-tuple that is the value of the lambda function, needs to be prepended to the list that is the value of the `unfold` functor.

As an oversimplified example of how the naive unfold could be used, think of generating the first 50 even numbers. In this case the functor would be executed like this $unfold\ ::\ 50\ 0\ (\lambda\ ::\ c\ s \to \{c-1, 2+s\})$ and a list of the first 50 even numbers would be the value.

The Laws

For functors there are laws that must be followed if a something is to be a functor. While these laws are often expressed in an arcane way, explanations of the laws are included here to make them easier to apply.

Please remember that functors were originally defined in the Category Theory branch of mathematics. A category can be thought of as a grouping of things. Any category may have 0 or more elements. That's why functors in functional programming often act on lists. Those lists may have 0, 1, or more elements, but the functor applies a list of functions to a list of values of some type. The items in the list may also be of type $function$ since functions are first-class citizens. 😎

Laws that Determine if Something is a Functor

1. $functor\ ::\ a\ id = a$

   This rule states that the potential functor must be able to apply an identity function id to an item a and the value of the functor be the same as the item. The item may be a single item or any grouping of items. This also implies that the functor has no side effects since a is the complete value of the functor.

2. $functor\ ::\ a\ (f \circ g) = (functor\ ::\ a\ f) \circ (functor\ ::\ a\ g)$

   This rule states that the potential functor must be able to apply a composed function and achieve the same results as applying the individual functions separately and then composing their values..

3. The relations between elements does not change when examining the items acted on by the functor and those that are the value of the functor.