Recursion and Folds in the Haskell Programming Language

by Raimondo Butera

Abstract

Haskell's *folds*, `foldl` and `foldr` are used to recursively reduce data structures to a single value, a common design pattern in functional programming.

Recursion in Haskell

Algorithms can be either *recursive* or *iterative*. An *iterative* algorithm is composed of a series of steps, whereas a *recursive* algorithm is composed of a function that calls a (smaller) instance of itself. Recursively defined functions exist in both the functional and imperative programming styles.
Whilst recursive approaches are used in imperative languages, a non-recursively defined alternative is usually possible in imperative programming. Recursion is often used over iteration for the sake of elegance. For example, the following are two functions that calculates the factorial of an input, programmed in the iterative style (as `iFact`) and the recursive style (as `rFact`). The examples are written in JavaScript (a C-based language with support for both imperative and functional styles:

```
function iFact (n) {
  if(n <= 1){
    return 1;
  } else {
    for (n; n > 1; n--) {
      n *= n - 1;
    }
    return n;
  }
}

function rFact (n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * rFact(n- 1);
  }
}
```

Haskell, unlike JavaScript, is a purely functional programming language and uses a declarative programming paradigm where all data is immutable. It follows that there are no looping constructs in Haskell, and a problem requiring multiple steps to solve can only, therefore, be solved recursively in Haskell:

```
factorial :: a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Lists in Haskell

Haskell uses *list* structures to manage several values. Problems involving lists often demand the traversal of a list, repeatedly performing an action on a single member and the rest of a list.

A function `sum` that calculates the sum of a list's members would be written in Haskell as `sum (x:xs) = x + sum xs`. This pattern (repeatedly performing an action on a single member of a list and the rest of the list) is sufficiently common that Haskell has functions called *folds* (ie. `foldl` + `foldr`) to encapsulate it.

Haskell's Folds: `foldl` and `foldr`

- `foldl :: (a -> b -> a) -> a -> [b] -> a`[1]
- `foldr :: (a -> b -> b) -> b -> [a] -> b`[2]

The type definition of Haskell's folds show that they both take three arguments: a function to apply to the list, an accumulator to hold the result, and a *Foldable* (usually a list).

The definition of `foldr` is[3]:

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

which effectively translates to “if given an empty list, return the accumulator, otherwise apply `f` to the first element of the list and the result of folding the rest of the list”

Folds can be used whenever it is necessary to perform an action on individual members of a list to reduce the list to a single value. In most cases both `foldl` and `foldr` (a folding-left or folding-right approach) could be used almost interchangeably to solve the same problem. For example, a function `sum` which sums a list, written using `foldl` as `sum x = foldl (+) 0` and using `foldr` as `sum x = foldr (+) 0`.

Using `foldr` sometimes has advantages over using `foldl`. If `foldr`’s accumulator can return a result without looking at the accumulated value then `foldr` can return the result immediately. In contrast,`foldl` must traverse the entire list every time before it can return a result. This 1) often makes `foldr` faster/less computationally expensive and 2) enables `foldr` to be used on infinite lists in contexts that do not require the entire infinite result.

For example[4]:

```
echoes = foldr (\ x xs -> (replicate x x) ++ xs) []
take 10 (echoes [1..])     -- [1,2,2,3,3,3,4,4,4,4]
```

above Haskell code executes successfully because Haskell can lazily convert an infinite list into another lazy infinite data structure.

However, using `foldr` on very large finite lists may cause a stack overflow exception. The Haskell wiki[5] illustrates this problem: using `foldr` to find the sum of the list `[1..10000000]`, for example, creates a chain of addition operations which grows large enough to overflow the stack, due to the strict requirement of `(+)` that both arguments must be fully evaluated in order to return a result. The problem is solved by using `foldl'`, a version of `foldl` that forces the reduction of inner reducible expressions before outer ones.

Summary

Haskell’s folds, `foldl` and `foldr`, encapsulate a common programming pattern: reducing a list to a single value. They are implemented recursively due to the purely functional nature of the Haskell. One of the folds, `foldr`, has a broader range of uses than its counterpart `foldl`; it is often faster and works on infinite lists.

References

- [1] https://www.haskell.org/hoogle/?hoogle=foldl (https://www.haskell.org/hoogle/?hoogle=foldl)
- [2] https://www.haskell.org/hoogle/?hoogle=foldr (https://www.haskell.org/hoogle/?hoogle=foldr)
- [3] https://wiki.haskell.org/Fold (https://wiki.haskell.org/Fold)
- [4] https://en.wikibooks.org/wiki/Haskell/Lists_III (https://en.wikibooks.org/wiki/Haskell/Lists_III)
- [5] https://wiki.haskell.org/Foldr_Foldl_Foldl%27 (https://wiki.haskell.org/Foldr_Foldl_Foldl%27)