Haskell: Tail Recursion

Volker Sorge

March 20, 2012

While recursively implemented functions are generally more concise, easier to understand and regarded as more elegant, they can be more memory intensive if not programmed carefully. In particular, care has to be taken, that for large computations, not too many operations on the stack have to be performed. For recursive functions this means we have to minimise the number of function calls that have to be stored on the stack. This can be achieved by *tail recursion*.

The basic idea of tail recursion is to effectively simulate an efficient iteration using the simplicity and elegance of a recursion. As a consequence tail recursive functions execute slightly faster than non-tail recursive ones, as they do not have to perform stack operations, and, more importantly, tail recursive functions can recurse indefinitely since no stack resources are used and no stack overflow should occur.

The Simple Case

How do we achieve this in practice? In traditional recursion one normally uses the result of a recursive call in a computation. The principle of tail recursion is to perform all computation first before the recursive call, often giving the results of the computation as additional argument to the recursively called function. Thus in tail recursion the recursive call is the *last logic instruction* in the recursive function. This has the effect that not all previous recursive calls have to be recorded on the call stack and, as a consequence, makes tail recursion essentially equivalent to straight forward iteration.

Let's have a look at an example; a naïve implementation of the factorial function.

```
fac 0 = 1
fac n = n * fac (n - 1)
```

Why is this factorial function not tail recursive? Although it appears that in the step case the last call is `fac (n - 1)`, it actually is not. The last function call is `(*)`. Let's observe what happens on the call stack, using *source reduction*. We take the example of a call of `fac 5`.

```
fac 5 = 5 * (fac 4)
      = 5 * (4 * (fac 3))
      = 5 * (4 * (3 * (fac 2)))
      = 5 * (4 * (3 * (2 * (fac 1))))
      = 5 * (4 * (3 * (2 * (1 * (fac 0)))))
      = 5 * (4 * (3 * (2 * (1 * 1))))
      = 5 * (4 * (3 * (2 * 1)))
      = 5 * (4 * (3 * 2))
      = 5 * (4 * 6)
      = 5 * 24
      = 120
```

As we can see first all the function calls of `fac` are expanded before the first multiplication is carried out. Our goal is now to first carry out the multiplication before calling `fac`. This is

Code given between solid horizontal bars can be loaded into Haskell from a `.hs` file. Code between dotted horizontal bars can be typed in the interpreter at the prompt. It will always compute correctly but the result might occassionally cause a display error (e.g. if a particular type does not inherit from the `Show` type class).

best achieved by using an *accumulator* to pass an intermediate result to the recursive call. The following function does exactly this:

```
facAux 0 r = r
facAux n r = facAux (n − 1) (r ∗ n)
```

The parameter r of facAux holds the result that is passed through the recursive calls. Observe that the parameter n is already multiplied with r *before* the recursive call. Obviously we now have to call facAux with two parameters, n and r. Since r will always be the same for every first call of facAux, namely the default result 1, we can wrap the function into a more traditional single parameter factorial function.

```
facTail n = facAux n 1
```

Finally, we observe the effect of the tail call looking at the source reduction. Note, that at no point in the function execution is it necessary to store a function call on the stack.

```
facAux 5 1 = facAux 4 5
           = facAux 3 20
           = facAux 2 60
           = facAux 1 120
           = facAux 0 120
           = 120
```

The sequence of intermediate results might appear slightly odd as we are now "counting backwards". If we want to multiply in increasing order of parameters, we can achieve this with the following implementation that employs an internal auxiliary function via the where construct. Observe that facAux2 is still tail recursive.

```
facTail2 n = facAux2 1 1
    where facAux2 c r
            | c == n = (r ∗ c)
            | otherwise = facAux2 (c + 1) (r ∗ c)
```

The Complex Case

The factorial function is very simple, in that it has only one recursive call to itself in its body. Functions of this nature can always be transformed in a straight forward manner into tail recursive functions. The same holds for functions that contain more than one recursive call, but each in a distinct case. However, often functions contain more than one recursive call within the computation of a single case. These functions are sometimes called *tree recursive* as their computations resemble a tree structure and are consequently very memory intensive.

While some problems are naturally tree recursive (e.g., printing a binary tree) many problems that appear tree recursive at first, can be turned into tail recursion when examined more closely. We will look at the example of Fibonacci numbers. A naïve recursive function is the following:

```
fib 0 = 1
fib 1 = 1
fib n = fib (n−1) + fib (n−2)
```

This computation can be drawn as a tree, where the root node is fib(n), that has a left sub-tree with root fib(n−1) and a right sub-tree with root fib(n−2). Clearly, when we compute fib(n−1) we have to compute this right sub-tree, i.e., fib(n−2), as well, therefore computing it twice. If we examine the tree further we can see that fib(n−3) needs to be computed 3 times, and so forth. In general, in the above function, fib(n−i) has to be computed i times.

However, the above consideration reveals that the left most branch of the recursion tree already contains all necessary computations. Indeed all information needed to compute any fib(i)

is given in the two previous levels of this branch. Thus we can simply reorder the computation to

$$\mathsf{fib}\,(0),\ \mathsf{fib}\,(1), \mathsf{fib}\,(2), \ldots,\ \mathsf{fib}\,(\mathsf{n}-1),\ \mathsf{fib}\,(\mathsf{n})$$

and instead of remembering one intermediate result as in the case of the factorial function, we remember two — the result of the current computation and of the previous one. The following function achieves this:

```
fibAux n result previous
  | n == 0 = result
  | otherwise = fibAux (n - 1) (result + previous) result

fibTail n
  | n == 0 = 0
  | otherwise = fibAux n 1 0
```

While for the factorial function the advantage of tail recursion was only really measurable for values of $n > 50000$, for the Fibonacci numbers, it is quite obvious already for small n.