Efficiency of Functional Programs

Christian Sternagel

January 29, 2011

Until now, we have almost been recklessly careless about the computational expensiveness of a function. In real world applications however, it is often very important that functions are implemented efficiently (that is, do not make more steps than absolutely necessary). In the following it is shown that some intuitive function definitions are very inefficient (that is, there are much faster implementations) and two techniques are provided that often yield more efficient implementations.

1 The Fibonacci Numbers

In many textbooks one of the first examples of recursive functions is to compute the n-th Fibonacci number.

Definition 1.1. The *Fibonacci numbers* are given by the equations

$$\begin{aligned} \text{fib}(0) &= 1 \\ \text{fib}(1) &= 1 \\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) \qquad \text{for } n > 1 \end{aligned}$$

A straightforward implementation in Haskell can directly be inferred:

```
fib n | n <= 1     =  1
      | otherwise  =  fib (n-1) + fib (n-2)
```

Using the above definition of `fib`, the computation of the n-th Fibonacci number does need an exponential (with respect to n) number of recursive calls to itself. The claim is that $2 \cdot \texttt{fib}\ n - 1$ calls to `fib` are needed in order to compute `fib` n. This can be proved by induction on n. Notice that there are two base cases, since the recursion stops either at 1 or at 0.

Lemma 1.1. $2 \cdot \texttt{fib}\ n - 1$ *calls to the function* `fib` *are needed to compute the* n*-th Fibonacci number.*

Proof.

Base Case ($n = 0$). To evaluate `fib` 0, one call to `fib` is needed. Since the 0^{th} Fibonacci number is 1 this concludes the first base case.

Base Case ($n = 1$). Also to evaluate `fib` 1, one call to `fib` is needed. Since the 1^{st} Fibonacci number is 1 this concludes the second base case.

Step Case ($n = m + 2$). The IHs are that the number of calls to `fib` when computing $\texttt{fib}(m+1)$ equals $2 \cdot \texttt{fib}(m+1) - 1$ and the number of calls to `fib` when computing `fib` m equals $2 \cdot \texttt{fib}\ m - 1$. It is easily observed that the number of calls to `fib` when computing $\texttt{fib}(m+2)$ is equal to the number of calls needed for $\texttt{fib}(m+1)$ plus the number of calls needed for `fib` m plus 1. Hence

$$\begin{aligned}&(2 \cdot \texttt{fib}(m+1) - 1) + (2 \cdot \texttt{fib}\ m - 1) + 1\\ &\quad = 2 \cdot (\texttt{fib}(m+1) + \texttt{fib}\ m) - 1\\ &\quad = 2 \cdot (\texttt{fib}(m+2)) - 1\end{aligned}$$

□

It remains to be shown, that $2 \cdot \texttt{fib}\ n - 1$ is an exponential number with respect to n. Since $2 \cdot \texttt{fib}\ n - 1 \geq \texttt{fib}\ n$ it suffices if $\texttt{fib}\ n \geq 2^{C \cdot n}$ for some constant C. This can be shown by the following derivation:

$$\begin{aligned}\texttt{fib}\ n &= \texttt{fib}(n-1) + \texttt{fib}(n-2) && \text{(definition of } \texttt{fib}\text{)}\\ &= \texttt{fib}(n-2) + \texttt{fib}(n-3) + \texttt{fib}(n-2) && \text{(definition of } \texttt{fib}\text{)}\\ &= 2 \cdot \texttt{fib}(n-2) + \texttt{fib}(n-3)\\ &\geq 2 \cdot \texttt{fib}(n-2)\\ &\geq 2 \cdot (2 \cdot \texttt{fib}(n-2-2))\\ &= 4 \cdot (\texttt{fib}(n-4))\\ &\geq 4 \cdot (2 \cdot \texttt{fib}(n-4-2))\\ &\quad\ \vdots\\ &\geq 2^k \cdot \texttt{fib}(n - 2 \cdot k)\end{aligned}$$

for $n > 1$ and $n - 2 \cdot k \geq 0$. If n is even, a base case is reached at $n - 2 \cdot k = 0$, otherwise at $n - 2 \cdot k = 1$. In both cases $k = n/2$ (using integer division). Since $\texttt{fib}\ 0 = \texttt{fib}\ 1 = 1$, the result $\texttt{fib}\ n \geq 2^{n/2}$ is obtained. This inefficiency stems from the fact, that work is repeated unnecessarily. For example to compute `fib` 3, `fib` 2 and `fib` 1 are computed. Then to compute `fib` 2, `fib` 1 (again) and `fib` 0 are computed. Hence, `fib` 1 is computed twice, repeating work that has already been done. In order to make the implementation of `fib` more efficient, results that are needed later on in the computation have to be stored somehow.

2 Tupling

The technique that can be used to achieve this goal is called *tupling*. Consider for example the function

```
fibpair n | n <= 0     =  (0,1)
          | otherwise  =  (f2,f1+f2)
          where (f1,f2) = fibpair (n-1)
```

Since there is just a single recursive call to `fibpair` in the function body and the argument (n) is reduced by 1, it is clearly the case that only a linear number of recursive function calls is needed. Furthermore it is claimed, that `fibpair` can be used to compute the n-th Fibonacci number.

Lemma 2.1. *The two components of the result of* `fibpair` n *for* $n > 0$*, are the* $(n-1)$*-th and* n*-th Fibonacci numbers, that is,*

$$\texttt{fibpair}\ n = (\texttt{fib}(n-1), \texttt{fib}\ n)$$

for $n > 0$*.*

Proof. **Base Case** $(n = 1)$. $\texttt{fibpair}\ 1 = (1,1) = (\texttt{fib}\ 0, \texttt{fib}\ 1)$.

Step Case The IH is that $\texttt{fibpair}\ m = (\texttt{fib}(m-1), \texttt{fib}\ m)$. The proof concludes by the derivation:

$$\begin{aligned}\texttt{fibpair}(m+1) &= (f_2, f_1 + f_2) \quad (\text{with } (f_1, f_2) = \texttt{fibpair}\ m)\\ &\overset{\text{IH}}{=} (\texttt{fib}\ m, \texttt{fib}(m-1) + \texttt{fib}\ m)\\ &= (\texttt{fib}\ m, \texttt{fib}\ m + 1).\end{aligned}$$

□

Lemma 2.2. *The function* `fibpair` *can be used to implement* `fib` *as follows:*

```
fib = snd . fibpair
```

Proof. From Lemma 2.1 it is known that $\texttt{fibpair}\ n = (\texttt{fib}(n-1), \texttt{fib}\ n)$, that is, for $n > 0$ the second component is the n-th Fibonacci number. Since $\texttt{fibpair}\ 0 = (0,1)$, also for $n = 0$, the second component is the n-th Fibonacci number. □

In general, tupling is used to modify existing functions in a way that they return more than one result, aiming at a more efficient implementation. Consider for example a function `average`, computing the average of the elements of an integer list. Therefore, the sum of all elements and the number of elements is needed. This could be implemented as

```
average xs = sum xs `div` length xs
```

however, in this case the list `xs` is traversed twice, once to compute the sum, and a second time to compute the length of the list. This could be combined into the function

```
sumlen []      =  (0,0)
sumlen (x:xs)  =  (sum+x,len+1)
  where (sum,len) = sumlen xs
```

Then `average` can be implemented by

```
average xs = sum `div` len
  where (sum,len) = sumlen xs
```

3 Tail Recursion

A special kind of recursion is *tail recursion*. A function is said to be tail recursive, if the *last* thing to do is the recursive call. This kind of recursion is special, since it can automatically be transformed (by the compiler) into a loop, that does need only constant stack space. Therefore, tail recursive functions are sometimes also called *iterative*.

On a standard computer the function stack (or call stack, or execution stack) stores information about a function call until it is finished. Hence at the call, information is pushed on top of the stack and as the function finishes, popped off the stack. However, if a recursive call occurs within a function call, then, the information for this call is pushed on top of the stack before popping the former call. If the recursive call again causes a recursive call additional call information is pushed on top of the stack. This continues until the last recursive call, after which, the call information can be popped one after the other computing the result of the function. Hence the used stack space depends on the size of the input, for example, a recursive function on a list containing 100,000 elements, will push 100,000 entries on top of the stack before removing anything. If the stack grows too big, a stack overflow is generated and the function is aborted.

Tail recursion does circumvent this problem, since a tail recursive function can be transformed automatically to only use constant stack space.

4 Accumulating Parameters

For tupling, the idea was to introduce additional result values to a function. The idea of *accumulating parameters* is to introduce new parameters that are used to transfer intermediate results between function calls. In this way, often tail recursive variants of existing functions can be achieved. E.g., the above `sumlen` is not tail recursive. Consider the following implementation

```
sumlenAcc sum len []      =  (sum,len)
sumlenAcc sum len (x:xs)  =  sumlenAcc (sum+x) (len+1) xs

sumlen xs = sumlenAcc 0 0 xs
```

which can also be written as

```
sumlen = sl 0 0
  where sl s l []     = (s,l)
        sl s l (x:xs) = sl (s+x) (l+1) xs
```

The second variant is used more often, since most of the time the auxiliary functions are not needed somewhere else. Note however that even if `sumlen` seems tail-recursive, calling it on a large list, will result in a stack overflow. The reason is the lazy evaluation strategy of Haskell. In every strict language the above definition will do. Under lazy evaluation however, the `s+x` and `l+1` are not evaluated immediately, but only as soon as we use the result of `sumlen`. This means that huge thunks of unevaluated functions are built, for the list `[1..1000000]`:

$$0 + 1 + 2 + 3 + \cdots + 1000000$$

for the sum, and

$$0 + \underbrace{1 + 1 + 1 + \cdots + 1}_{1000000 \text{ times}}$$

for the length. The difference is between storing two single integers for strict evaluation and two thunks of 1000001 integers (at least 8MB for 32 bit `Int`s) for lazy evaluation, on the stack. For the moment we will just disregard the actual evaluation strategy of Haskell, but note that there are possibilities for strict evaluation (which would turn the above function into a proper tail-recursive one).

5 Chapter Notes

See also [1] for a discussion on how recursive algorithms should be treated in computer science courses. Additionally to Fibonacci numbers also binomial coefficients are used there as example.

References

[1] Ivan Stojmenovic. Recursive algorithms in computer science courses: Fibonacci numbers and binomial coefficients. 2000.