Efficiency of Functional Programs

Christian Sternagel and Harald Zankl

December 13, 2018

Software gets slower faster
than hardware gets faster.
Niklaus Wirth

Software efficiency halves every
18 months, compensating
Moore's law.
David May

What Intel giveth,
Microsoft taketh.
Variant of *Wirth's Law*

Until now we have almost been recklessly careless about the computational cost 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 code.

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. The *Fibonacci numbers* are given by the equations

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

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

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

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

□

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 and sufficiently large n. This can be shown by the following derivation:

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

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 `fib` $0 =$ `fib` $1 = 1$, the result `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

One technique that can be used to maintain intermediate results 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, `fibpair` can be used to compute the n-th Fibonacci number.

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

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

for $n \geqslant 0$.

Proof.

Base Case $(n = 0)$. `fibpair` $1 = (1, 1) = ($`fib` $0,$ `fib` $1)$.

Step Case $(n = m + 1)$. The IH is that `fibpair` $(m+1) = ($`fib` $m,$ `fib` $(m+1))$. We have to show `fibpair` $(m+2) = ($`fib` $(m+1),$ `fib` $(m+2))$. The proof concludes by the derivation:

$$\begin{aligned}
\texttt{fibpair}\ (m+2) &= (f_2, f_1 + f_2) \quad (\text{with } (f_1, f_2) = \texttt{fibpair}\ (m+1)) \\
&\overset{\text{IH}}{=} (\texttt{fib}\ (m+1), \texttt{fib}\ m + \texttt{fib}\ (m+1)) \\
&= (\texttt{fib}\ (m+1), \texttt{fib}\ (m+2)).
\end{aligned}$$

□

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 `fibpair` $n = (\texttt{fib}(n-1), \texttt{fib}\ n)$, that is, for $n > 0$ the second component is the n-th Fibonacci number. Since `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) = (s + x, l + 1)
    where
      (s, l) = sumlen xs
```

Then `average` can be implemented by

```
average xs = s `div` l
    where
      (s, l) = sumlen xs
```

3 Tail Recursion

A special kind of recursion is *tail recursion*. A function is said to be tail recursive, if any recursive call is the *last* thing that happens.[1] This kind of recursion is special, since it can automatically be transformed (by the compiler) into a loop that only needs 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

[1] Note that as a consequence there can be at most one recursive call.

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 program is aborted.[2]

Tail recursion does circumvent this problem, since a tail recursive function can be automatically transformed into machine code using only constant stack space. The concept from the next section will give us the means to write tail recursive functions.

4 Parameter Accumulation

For tupling, the idea was to introduce additional result values to a function. In *parameter accumulation* the idea 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 (after the recursive call additions are performed and a pair is constructed). Consider the following implementation

```
sumlenAcc s l []     =  (s, l)
sumlenAcc s l (x:xs) =  sumlenAcc (s + x) (l + 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 in this lecture since most of the time the auxiliary functions are not needed anywhere else. Note however that even though `sumlen` is tail-recursive, calling it on a large list, will result in a tremendous amount of memory being used (e.g., for a list with 100,000,000 elements 20GB of RAM will not suffice). This is called a *memory leak*. The reason is the lazy evaluation strategy of Haskell. In every strict language the above definition will not result in a memory leak. Under lazy evaluation however, the expressions `s + x` and `l + 1` are not evaluated immediately, but rather stored unevaluated as *thunks*. As soon as we use the result of `sumlen`, these thunks are evaluated. This means that huge thunks of unevaluated expressions are built, for the list `[1..100000000]`:

$$0 + 1 + 2 + 3 + \cdots + 100000000$$

[2] That said, Haskell is special in that it does not use a conventional call stack. As a consequence, stack overflows are much less frequent than in other languages. However, we will see another form of memory problem that are specific to Haskell later.

for the sum, and

$$0 + \underbrace{1 + 1 + 1 + \cdots + 1}_{100000000 \text{ times}}$$

for the length. The difference is between storing two single integers for strict evaluation and two thunks of 100000001 integers each (at least 800MB for 32 bit `Int`s, but typically much much more) for lazy evaluation. For the moment we will just disregard the actual evaluation strategy of Haskell, but note that there are possibilities to enforce strict evaluation in Haskell in order to avoid memory leaks.

5 Linear vs. Quadratic Complexity

In this section we demonstrate that for large values of n it can already be problematic if a function runs in time $O(n^2)$.

Consider the append function for lists

```
[]     ++ ys  =  ys
(x:xs) ++ ys  =  x : (xs ++ ys)
```

Obviously this function takes linear time in the size of its first parameter, which is harmless. More precisely if *xs* has n elements, then *xs* `++` *ys* takes $n + 1$ computation steps (do not forget the base case, that is, when *xs* = `[]`).

Let us in addition consider the function

```
reverse []       =  []
reverse (x:xs)   =  reverse xs ++ [x]
```

which calls itself and '`++`' linearly often. While this might also look harmless (at first sight), it definitely is not. What happens is the following. The function '`++`' is called linearly often but has linear complexity itself, resulting in a quadratic complexity in total. This can be seen if we evaluate `reverse [1 .. n]` as follows:

$$\begin{aligned}
\texttt{reverse}\ [1,\ldots,n] &\to^{n+1} ((((\texttt{[]}\ \texttt{++}\ [n])\ \texttt{++}\ [n-1])\ \texttt{++}\ [n-2])\ \texttt{++}\cdots)\ \texttt{++}\ [1] \\
&\to^{1} (((\texttt{[}n\texttt{]}\ \texttt{++}\ [n-1])\ \texttt{++}\ [n-2])\ \texttt{++}\cdots)\ \texttt{++}\ [1] \\
&\to^{2} ((\texttt{[}n,n-1\texttt{]}\ \texttt{++}\ [n-2])\ \texttt{++}\cdots)\ \texttt{++}\ [1] \\
&\to^{3} (\texttt{[}n,n-1,n-2\texttt{]}\ \texttt{++}\cdots)\ \texttt{++}\ [1] \\
&\ \ \vdots \\
&\to^{n} \texttt{[}n,n-1,n-2,\ldots,1\texttt{]}
\end{aligned}$$

Hence in total we have

$$\begin{aligned}
(n+1) + 1 + 2 + 3 + \ldots + n &= (n+1) + \frac{n \cdot (n+1)}{2} \\
&= \frac{(n+1) \cdot (n+2)}{2} \in O(n^2)
\end{aligned}$$

computation steps.

Consider in contrast the alternative implementation:

```
reverseAcc xs = revAcc [] xs
    where
      revAcc acc [] = acc
      revAcc acc (x:xs) = revAcc (x:acc) xs
```

With the help of the accumulator this function needs linear time only (to be more precise for a list of length n this function requires $n + 1$ computation steps). We leave it as an exercise to evaluate these two functions for lists of arbitrary length to see the (dramatic) differences in execution time for larger n, e.g., $n = 100000$.

6 Further Remarks

See Ivan Stojmenovic's article [1] for a discussion on how recursive algorithms should be treated in computer science courses. He uses the example of binomial coefficients in addition to Fibonacci numbers. Note that for the Fibonacci numbers a closed expression is known, namely

$$\mathsf{fib}(n) = \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n.$$

However, this expression is not ideal for implementations because of precision problems when using floating point arithmetic. An efficient implementation of Fibonacci numbers can be derived from the identity

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n = \begin{pmatrix} \mathsf{fib}(n) & \mathsf{fib}(n+1) \\ \mathsf{fib}(n+1) & \mathsf{fib}(n+2) \end{pmatrix}$$

References

[1] Ivan Stojmenovic. Recursive algorithms in computer science courses: Fibonacci numbers and binomial coefficients. *IEEE Transactions on Education*, 43(3):273–276, 2000. doi:10.1109/13.865200.