Fibolucci

EXERCISE 'FIBOLUCCI' in *Programming, The Derivation of Algorithms*[1].

[1] A. Kaldewaij. *Programming, The Derivation of Algorithms*. Prentice Hall, 1990

Problem

Write a program that calculates the function

$$f(n) = \sum_{i=0}^{n} fib(i)fib(n-i), \text{ for } n \geq 0$$

where *fib* is the Fibonacci sequence defined by:

$$fib(0) = 0$$
$$fib(1) = 1$$
$$fib(n+2) = fib(n+1) + fib(n), \text{ for } n \geq 0$$

To solve the Fibolucci sum we adopt the same notation used in *Programming in the 1990s*[2]: The notation of function application is the "dot" notation with name of function, followed by arguments, each separated by a dot. The notation of quantified expressions has the operator followed by the bounded variables, then a colon followed by the range for the bounded variables and ended with a colon and the actual expression. So

[2] Edward Cohen. *Programming in the 1990s, An Introduction to the Calculation of Programs*. Springer-Verlag, 1990

$$(\sum k : i \leq k < j : x_k)$$

corresponds to the more classical mathematical notation $\sum_{k=i}^{j-1} x_k$.

For our derivation steps in predicate calculus we will use the following notation:

$$\begin{array}{ll} & A \\ = & < \text{reason why A equals B } > \\ & B \\ \leq & < \text{reason why B is less than C} > \\ & C \end{array}$$

We start by finding a recursive expression for f. We will use properties of quantified expressions as covered in Chapter 3 of *Programming in the 1990s*[3]. Since $\mathit{fib}.(0) = 0$ we can use an equivalent definition expression for f:

[3] Edward Cohen. *Programming in the 1990s, An Introduction to the Calculation of Programs*. Springer-Verlag, 1990

$$f(n) = (\sum i : 1 \leq i < n : \mathit{fib}.i\ \mathit{fib}.(n-i))$$

We derive:

$$\begin{aligned}
& f.(n+2) \\
= \quad & < \text{definition of } f \ > \\
& (\textstyle\sum i : 1 \leq i < n+2 : \mathit{fib}.i\ \mathit{fib}.(n+2-i)) \\
= \quad & < \text{range split, 1-point rule} > \\
& (\textstyle\sum i : 1 \leq i < n+1 : \mathit{fib}.i\ \mathit{fib}.(n+2-i)) + \mathit{fib}.(n+1)\ \mathit{fib}.(1) \\
= \quad & < \ \mathit{fib}.(1) = 1 \ > \\
& (\textstyle\sum i : 1 \leq i < n+1 : \mathit{fib}.i\ \mathit{fib}.(n+2-i)) + \mathit{fib}.(n+1) \\
= \quad & < \text{definition of } \mathit{fib} \ > \\
& (\textstyle\sum i : 1 \leq i < n+1 : \mathit{fib}.i\ (\mathit{fib}.(n+1-i) + \mathit{fib}.(n-i))) + \mathit{fib}.(n+1) \\
= \quad & < \text{splitting the term} > \\
& (\textstyle\sum i : 1 \leq i < n+1 : \mathit{fib}.i\ \mathit{fib}.(n+1-i)) + \\
& (\textstyle\sum i : 1 \leq i < n+1 : \mathit{fib}.i\ \mathit{fib}.(n-i)) + \mathit{fib}.(n+1) \\
= \quad & < \text{definition of } f \ > \\
& f.(n+1) + (\textstyle\sum i : 1 \leq i < n+1 : \mathit{fib}.i\ \mathit{fib}.(n-i)) + \mathit{fib}.(n+1) \\
= \quad & < \text{range split, 1-point rule}, \mathit{fib}.(0) = 0 \ > \\
& f.(n+1) + (\textstyle\sum i : 1 \leq i < n : \mathit{fib}.i\ \mathit{fib}.(n-i)) + \mathit{fib}.(n+1) \\
= \quad & < \text{definition of } f \ > \\
& f.(n+1) + f.n + \mathit{fib}.(n+1)
\end{aligned}$$

We get the recursive definition of f:

$$\begin{aligned}
& f.0 = 0 \\
& f.1 = 0 \\
& f.(n+2) = \mathit{fib}.(n+1) + f.(n+1) + f.n, \text{ for } n \geq 0
\end{aligned}$$

It is straightforward to write a program that computes f from this recursive definition, either iteratively with a loop that step by step computes next values of f starting with $f(2)$ and remembering the last two computed values of f and of *fib* for the next computations, or in Haskell by simply declaring the above recursions for f and *fib*. This will lead to a runtime of $O(n)$. But can we do better than linear ?

Let's look again at the recursive expressions of the two functions involved, leaving out the base cases and computing one additional next value:

$$
\begin{aligned}
&f.(n+2) = fib.(n+1) + f.(n+1) + f.n\\
&f.(n+3) = fib.(n+2) + f.(n+2) + f.(n+1)\\
&fib.(n+2) = fib.(n+1) + fib.n\\
&fib.(n+3) = fib.(n+2) + fib.(n+1)
\end{aligned}
$$

The key observation we can make here is that new values of the two functions are linear combinations of previously computed values. Linear combinations implies linear applications with matrix representations from linear algebra. How many previously computed values, i.e. how far back do we need to go: we need the last computed value last and the value computed before that, so 2 previous values. Looks like we could try something in a linear space of dimension 2.

Let's try first with *fib* which is simpler and doesn't depend on f. We define the function $Fib : \mathbb{N} \to \mathbb{N}^2$ into the two-dimensional space $\mathbb{N}^2$:

$$Fib.n = \begin{pmatrix} fib.n \\ fib.(n+1) \end{pmatrix}, \text{ for } n \geq 0$$

For a recursive expression for *Fib* we have:

$$
\begin{aligned}
& Fib.(n+1)\\
= \quad & < \text{definition of } Fib \ >\\
& \begin{pmatrix} fib.(n+1) \\ fib.(n+2) \end{pmatrix}\\
= \quad & < \text{definition of } fib \ >\\
& \begin{pmatrix} fib.(n+1) \\ fib.(n+1) + fib.n \end{pmatrix}\\
= \quad & < \text{matrix multiplication} >\\
& \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} fib.n \\ fib.(n+1) \end{pmatrix}\\
= \quad & < \text{definition of } Fib \ >\\
& \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} Fib.n
\end{aligned}
$$

So

$$Fib.(n+1) = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} Fib.n = \ldots = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n+1} Fib.0$$

The same approach can be used for f. We define a function $F : \mathbb{N} \to \mathbb{N}^4$ into the four-dimensional space $\mathbb{N}^4$:

$$F.n = \begin{pmatrix} fib.n \\ fib.(n+1) \\ f.n \\ f.(n+1) \end{pmatrix}, \text{ for } n \geq 0$$

For a recursive expression for F we have:

$$\begin{aligned}
& F.(n+1) \\
= \; & < \text{definition of } F \; > \\
& \begin{pmatrix} fib.(n+1) \\ fib.(n+2) \\ f.(n+1) \\ f.(n+2) \end{pmatrix} \\
= \; & < \text{definitions of } fib \text{ and } f \; > \\
& \begin{pmatrix} fib.(n+1) \\ fib.(n+1) + fib.n \\ f.(n+1) \\ f.(n+1) + f.n + fib.(n+1) \end{pmatrix} \\
= \; & < \text{matrix multiplication} > \\
& \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} fib.n \\ fib.(n+1) \\ f.n \\ f.(n+1) \end{pmatrix} \\
= \; & < \text{definition of } F \; > \\
& \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} F.n
\end{aligned}$$

and

$$F.(n+1) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} F.n = \ldots = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}^{n+1} F.0$$

Calculating $F.n$ also calculates $f.n$ so if we can calculate $F.n$ faster than linear we also solve the original problem faster than linear. $F.n$ is basically an exponentiation so let's look at the exponentiation function $exp(x,n) = x^n$. The following recursive expression holds for exp:

$$exp.x.n = \begin{cases} exp.(x\ x).(n/2) & \text{if } n = 0 \text{ mod } 2 \\ x\ exp.x.(n-1) & \text{if } n = 1 \text{ mod } 2 \end{cases}$$

At least at every other step in the above recursion n is halved so computing $exp(x,n)$ has $O(log\ n)$ runtime which also implies $O(log\ n)$ runtime for F.

Before we write the actual code for computing F let's first see if we can find a more compact representation for the powers of matrix A involved in the computation:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

We are searching for patterns in the powers of A:

$$A^2 = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 2 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 2 & 1 & 2 \end{pmatrix},\ A^3 = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 1 & 2 & 1 & 2 \\ 2 & 5 & 2 & 3 \end{pmatrix},\ A^4 = \begin{pmatrix} 2 & 3 & 0 & 0 \\ 3 & 5 & 0 & 0 \\ 2 & 5 & 2 & 3 \\ 5 & 10 & 3 & 5 \end{pmatrix}$$

We make the conjecture that A^k for any natural k is of the form:

$$A^k = \begin{pmatrix} a & b & 0 & 0 \\ b & a+b & 0 & 0 \\ c & d & a & b \\ e & f & b & a+b \end{pmatrix},\ \text{for some } a,b,c,d,e,f \in \mathbb{N} \tag{1.1}$$

and prove this by induction. The base case for $k = 1$ is established with values $(0,1,0,0,0,1)$ for (a,b,c,d,e,f). Assuming that the conjecture holds for A^k we look at A^{k+1} and get:

$$A^{k+1} = A^k\,A = \begin{pmatrix} b & a+b & 0 & 0 \\ a+b & a+2b & 0 & 0 \\ d & b+c+d & b & a+b \\ f & a+b+e+f & a+b & a+2b \end{pmatrix}$$

so A^{k+1} has the same form as stated in the conjecture if we substitute $(b, a+b, d, b+c+d, f, a+b+e+f)$ for (a,b,c,d,e,f). This proves conjecture (1.1).

It means that in our program we can use a tuple representation (a,b,c,d,e,f) of 6 values instead of the whole 16 values to represent the powers of A. We need to define multiplication in this tuple space consistent with the matrix multiplication:

$$\begin{aligned}(a,b,c,d,e,f)(a',b',c',d',e',f') = \\ &(aa' + bb', \\ &\ ab' + b(a'+b'), \\ &\ ca' + db' + ac' + be', \\ &\ cb' + d(a'+b') + ad' + bf', \\ &\ ea' + fb' + bc' + (a+b)e', \\ &\ eb' + f(a'+b') + bd' + (a+b)f')\end{aligned}$$

We read this definition off the matrix multiplication:

$$\begin{pmatrix} a & b & 0 & 0 \\ b & a+b & 0 & 0 \\ c & d & a & b \\ e & f & b & a+b \end{pmatrix} \begin{pmatrix} a' & b' & 0 & 0 \\ b' & a'+b' & 0 & 0 \\ c' & d' & a' & b' \\ e' & f' & b' & a'+b' \end{pmatrix}$$

The last expression we need is:

$$A^n F.0 = \begin{pmatrix} a & b & 0 & 0 \\ b & a+b & 0 & 0 \\ c & d & a & b \\ e & f & b & a+b \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} b \\ a+b \\ d \\ f \end{pmatrix}$$

so we are interested in d which corresponds to the $F.n$ coordinate of the vector.

Putting all the pieces together we get the final Haskell program:

Listing 1.1: Haskell code

```
type Tuple6Ints = (Int, Int, Int, Int, Int, Int)

tmul :: Tuple6Ints -> Tuple6Ints -> Tuple6Ints

tmul (a, b, c, d, e, f) (a', b', c', d', e', f') =
    (a * a' + b * b',
     a * b' + b * (a' + b'),
     c * a' + d * b' + a * c' + b * e',
     c * b' + d * (a' + b') + a * d' + b * f',
     e * a' + f* b' + b * c' + (a + b) * e',
     e * b' + f* (a' + b') + b * d' + (a + b) * f')

fibexp :: Tuple6Ints -> Int -> Tuple6Ints

fibexp tuple n | n == 0 = error "undefined"
               | n == 1 = tuple
               | n `mod` 2 == 0 =
                   fibexp (tuple `tmul` tuple)
                              (n `div` 2)
               | n `mod` 2 == 1 =
                   tuple `tmul` (fibexp tuple (n - 1))
               | otherwise = error "wrong␣input"

fourth :: Tuple6Ints -> Int

fourth (a, b, c, d, e, f) = d

fibolucci :: Int -> Int

fibolucci n | n == 0 = 0
            | otherwise =
                fourth (fibexp (0, 1, 0, 0, 0, 1) n)
```

Bibliography

Edward Cohen. *Programming in the 1990s, An Introduction to the Calculation of Programs*. Springer-Verlag, 1990.

A. Kaldewaij. *Programming, The Derivation of Algorithms*. Prentice Hall, 1990.