Lazy Evaluation, Fibonacci, & the Golden Ratio

Akash Garg

Version 1.0 - June 15, 2012

I remember an interesting computational puzzle: find the sum of all even digits (less than 1000000) in a Fibonacci sequence.

A simple brute force solution brute force to this is simply creating a Fibonacci sequence, filtering out the odd terms, and then taking the sum.

To that end, one way to compute the the Fibonacci sequence recursively is with a procedure like the following:

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-2) `myadd` fib (n-1)
    where myadd x y = trace (" Adding: " ++ show x ++
                             " & " ++ show y ++ "\n") (x+y)
```

This should be fairly easy to understand. ‘fib 0‘ and ‘fib 1‘ are the base conditions that simply return the first two numbers in the Fibonacci sequence. For anything else, the Fibonacci number is the addition of the previous two numbers in the sequence, namely ‘fib (n-2)‘ and ‘fib (n-1)‘. I use a custom addition operator here that prints whenever two numbers are added together. This function can be used for example to compute the fifth term in the Fibonacci sequence:

```
*Main> fib 5
 Adding: 1 & 1
 Adding: 1 & 2
 Adding: 1 & 1
 Adding: 1 & 1
 Adding: 1 & 2
 Adding: 2 & 3
 Adding: 3 & 5
8
```

Note the repeated number of additions we had to do. Here the function computes the addition of 1 & 1, 3 times! Obviously a lot of extra computation takes place, which only gets worse if as the function evaluates

later terms in the sequence. This is because the evaluation of every digit in the sequence must evaluate every number before it . . . this compounds the later you get into the sequence.

Consider an alternate solution in Haskell:

```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
sum $ filter (even) $ takeWhile (<1000000) fibs
```

For those unfamiliar with Haskell, just note that ':' prepends to a list. The 'zipWith' function takes an operation that is then applied to each pair of the given two lists. In this case, it applies the '+' operator to each pair of elements in the 'fibs' list and the 'tail fibs' list. The latter is simply the 'fibs' list without the first element. Convince yourself that this recursive definition of 'fibs' actually computes the Fibonacci sequence by working it out on paper.

The second statement is simple. From the list of Fibonacci numbers stored in the list 'fibs', take all the numbers that are less than 1000000, filter out all the even numbers, and then sum them together. The result is a summation of all even Fibonacci numbers less than 1000000.

This may seem just an inefficient as the naive recurisve implementation of the Fibonacci sequence at the beginning of the article, however, Haskell's prodigious use of lazy evaluation (sometimes called non-strict evaluation) actually makes this solution quite elegant. Let's take a look at what Haskell does under the hood while evaluating the above.

The very first thing to note is that 'fibs' is an infinite list. A language that makes use of strict-evaluation, i.e., a language that needs to evaluate a function's arguments before applying the function to it, cannot handle an infinite list; since evaluating an argument that is an infinite list will take infinite time...

In Haskell, infinite lists are possible because instead of evaluating the arguments, it simply makes a record that when needed, the argument will be evaluated so that the calling function can be properly applied.

To understand this, consider a step by step evaluation of the above with one slight modification, instead of considering `takeWhile (<1000000)`, let's consider `takeWhile (<8)`. This will effectively only take the sum of the first 3 even terms in the Fibonacci sequence.

The first function that actually gets called is a call made to `takeWhile (<8) fibs`. The function `takeWhile` will evaluate the list 'fibs' until the criteria < 8 is met. Now note that 'fibs' is actually a self-referential data-structure. In theory, it is an infinite list that contains the list of all Fibonacci numbers. This data-structure is actually a lazy data-structure where later parts of the structure refer to earlier parts by name. Below is a graphic representation of what the 'fibs' list looks like at initialization.

Initially, the structure is just a computation with unevaluated pointers back to itself. As it is unfolded, values are created in the structure. Later references to already-computed parts of the structure are able to find the value already there waiting for them. No need to re-evaluate the pieces, and no extra work to do! Let's take a look at how the data-structure unfolds after each iteration of `takeWhile (<8) fibs`. Below is a slight modification of the above code that simply prints to the terminal when addition of two values are taking place.

```
fibs = 1 : 1 : zipWith myadd fibs (tail fibs)
```

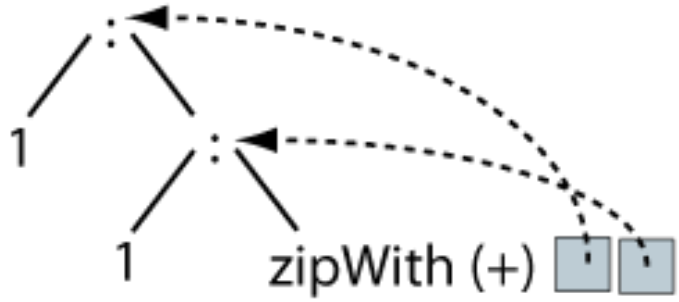

Figure 1: Here we see the first two elements of the list and the rest of the list is simply a record that indicates that when needed a ‘zipWith‘ function will be applied to earlier parts of the list.

```
    where myadd x y = trace (" Adding: " ++ show x ++
                             " & " ++ show y ++ "\n") (x+y)
```

Applying this modification, we get the following result. Note that we are only adding each pair of numbers *once*.

```
*Main> takeWhile (<8) fibs
[1,1 Adding: 1 & 1
,2 Adding: 1 & 2
,3 Adding: 2 & 3
,5 Adding: 3 & 5
]
```

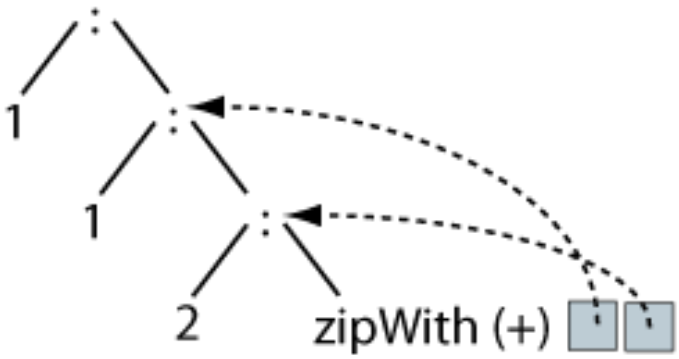

Figure 2: As values need to be evaluated, the list is unfolded with the newly computed values. Later values are still records that refer to earlier parts of the list. Since earlier values already exist in the list, the next time we need access to them, they do not need to be recomputed

Furthermore, this data-structure remains in memory, so repeated calls do not do additional computation. Below we access the fifth element in the Fibonacci sequence[1]:

[1] ‘!i is the index operator. ‘fibs !! 5‘ returns the fifth element in the ‘fibs‘ list

```
*Main> fibs !! 5
8
```

Note that the answer, 8, is simply returned without doing any additional work. A call made to the ‘fib‘ function however to get the fifth element will need to redo all the work again:

```
*Main> fib 5
 Adding: 1 & 1
 Adding: 1 & 2
 Adding: 1 & 1
 Adding: 1 & 1
 Adding: 1 & 2
 Adding: 2 & 3
 Adding: 3 & 5
8
```

This should be fairly obvious to most people, after all, a function has no chance of caching intermediate results without using auxiliary data-structures. As nice as they are, self-referential data-structures may not be feasible in all applications. A more general approach is to augment functions that have repeated sub-problems with a technique called memoization [Cor+01]. I won't go into details on memoization here and will leave that for another note [2], but the basic idea is that a function is augmented with a dictionary lookup of cached values. If a value is not available yet, it is computed and stored in the cache. If it is available, the cached version of the value is used. Note that this process is exactly what our self-referential data-structure is doing! It's not using a "dictionary" per se, but it is using a mechanism to cache computed values so that it does not need to recompute them again.

An alternative solution to the above problem is to use the elegant relation of the Golden Ratio and the Fibonacci sequence. By definition, the Golden Ratio is:

$$\phi = \lim_{n \to \text{inf}} \frac{F(n+1)}{F(n)}.$$

where $F(n)$ is the n^{th} Fibonacci term.

Note that every 3rd item in the Fibonacci sequence is an even number. Convince yourself of this before continuing. We need a way to sum up those even numbers. Using ratios of the Fibonacci terms and the Golden Ratio, we can define the following properties:

$$\frac{F(n+2)}{F(n)} = \frac{\left(\frac{F(n+2)}{F(n+1)}\right)}{\left(\frac{F(n)}{F(n+1)}\right)} = \frac{\phi}{\left(\frac{F(n)}{F(n+1)}\right)} = \frac{\phi}{\frac{1}{\phi}} \equiv \phi^2$$

Similarly,

$$\frac{F(n+3)}{F(n)} = \frac{\left(\frac{F(n+3)}{F(n+2)}\right)}{\left(\frac{F(n)}{F(n+2)}\right)} = \frac{\phi}{\frac{1}{\phi^2}} = \phi^3$$

[2] A good description of memoization in Haskell can be found on the Haskell Wiki and references there in. The page also contains several examples to memoize the Fibonacci sequence.

Since ϕ is the approximate ratio between two consecutive numbers in the Fibonacci sequence, this tells us that in the limit, the ratio between two *even* numbers in the Fibonacci sequence approaches ϕ^3. Once we have this relation, we can simply start with an even Fibonacci number and get the next even Fibonacci number by multiplying by ϕ^3 and rounding the result to the nearest integer value. Starting with the first even Fibonacci number 2, we get the following sequence: 2, 8, 34, 144, 610, ... multiplying by ϕ^3 each time to get the next even Fibonacci number. We can then simply take the sum of these values to get the answer we were originally looking for.

Yet another way to solve this problem is to consider the Fibonacci sequence as a second order linear difference equation. It turns out that solving it will involve a closed form solution that makes use of the powers of, you guessed it, the golden ratio, ϕ! I recommend trying this out. This article on Wikipedia provides a nice description on deriving closed form solutions for recurrence relations.

References

[Cor+01] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. The MIT Press, 2001. ISBN: 0262032937.

Revision	Date	Description
1.0	June 15, 2012	Initial draft