A list-based Sieve of Eratosthenes

David Spies

March 11, 2018

Abstract

In [1], they claim that in order to write an efficient (within $O(\log n)$ performance of the imperative version) Sieve of Eratosthenes, one must resort to more complex data-structures. In this paper, we introduce a Sieve of Eratosthenes which achieves the same big-O performance using *only* infinite lists.

This document is literate Haskell. It compiles and runs on GHC 8.2.2. To compile it we'll need a main method. Let's print out the first 30 primes:

```
main :: IO ()
main = print (take 30 primes)
```

All lists used in this document are infinite and sorted. All lists of lists are sorted on their heads. First, lets define a merge function (for infinite sorted lists).

```
merge :: Ord a ⇒ [a] → [a] → [a]
merge (x : xs) (y : ys)
  | y < x = y : merge (x : xs) ys
  | otherwise = x : merge xs (y : ys)
```

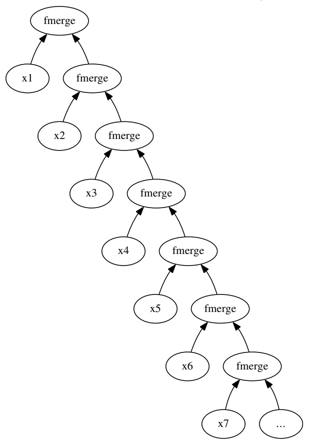
This takes two sorted lists and merges them into a single sorted list. Note that evaluating the head of the result triggers the evaluation of both arguments' heads. We'll need a version that takes the head of the left argument and puts that as the head of the result before ever evaluating the right argument. Then we'll be careful to call it only where we know the head of the right list is at least as large as the head of the left.

```
fmerge :: Ord a ⇒ [a] → [a] → [a]
fmerge (x : xs) ys = x : merge xs ys
```

With this in hand, we could define a function which lists all numbers whose only prime factors are 2 and 3 as follows:

```
twosThreesOnly :: [Integer]
twosThreesOnly = fmerge (iterate (2∗) 1) [3 ∗ x | x ← twosThreesOnly]
```

Figure 1: Structure of `fmergeAllNaive`



We can also use `fmerge` to merge an infinite list of sorted lists together if we know the heads of the lists are already sorted.

```
fmergeAllNaive :: Ord a ⇒ [[a]] → [a]
fmergeAllNaive (x : xs) = fmerge x (fmergeAllNaive xs)
```

This works and uses an impressively small amount of code, but isn't very performant. In the worst case, evaluating the kth element can require $O(k)$ running time. To see why this is, let's look at what structure results from evaluating `fmergeAllNaive` on a list. Suppose we have the list

```
xs = x1 : x2 : x3 : x4 : ...
```

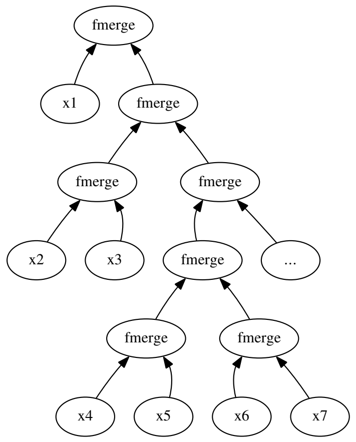
When we call `fmergeAllNaive xs`, the resulting structure looks something like Figure 1.

If an element belongs to list x_k, then to "bubble up" to the top of our evaluation structure, it must be compared against the first element of x_{k-1}, followed by the first element of x_{k-2} etc. until finally being compared against x_1.

To rectify this, let's first create a helper function for efficiently merging a list prefix of length-k (where $k > 0$). The `fmergePrefix` function returns both the merged prefix and the unmerged remainder. As before, we assume the heads of the lists are themselves already sorted.

```
fmergePrefix :: Ord a ⇒ Int → [[a]] → ([a], [[a]])
fmergePrefix 1 (x : xs) = (x, xs)
fmergePrefix k xs = (fmerge y z, zs)
  where
    (y, ys) = fmergePrefix (k `quot` 2) xs
    (z, zs) = fmergePrefix ((k + 1) `quot` 2) ys
```

Figure 2: Structure of `fmergeAll`



This should look familliarly like a standard merge-sort, except that we're using `fmerge` rather than `merge` and all our lists are infinite.

Notice that evaluating the first n elements of the resulting merged prefix requires at most $O(n \log k)$ steps as any element needs to be compared with at most $\log_2 k$ others to bubble to the top. Now here's a more efficient `fmergeAll` implementation which makes use of `fmergePrefix`.

```
fmergeAll :: Ord a ⇒ [[a]] → [a]
fmergeAll = go 1
  where
    go k xs = let (ys, zs) = fmergePrefix k xs in fmerge ys (go (k ∗ 2) zs)
```

`fmergeAll` is quite similar to `fmergeAllNaive` except that instead of merging lists together one at a time, we merge them in batches of exponentially growing size. Each batch is efficiently merged using fmergePrefix. A consequence is that any element from the k^{th} list needs to be compared with at most $O(\log k)$ elements to bubble to the top of the resulting structure of thunks (see Figure 2).

In addition to merging lists, it will also be useful to *exclude* elements from a list. The implementation is straightforward.

```
excluding :: Ord a ⇒ [a] → [a] → [a]
(x : xs) ‘excluding‘ (y : ys) = case compare x y of
  LT → x : (xs ‘excluding‘ (y : ys))
  EQ → xs ‘excluding‘ ys
  GT → (x : xs) ‘excluding‘ ys
```

With this in hand, we can mutually recursively define two lists: `primes` and `composites` which respectively are lists of all the prime and composite integers. The composites are the merged multiples of the prime numbers. Note that if we start from the square of each prime, then any composite number n will occur once in the composites list for each of its prime factors $p \leq \sqrt{n}$ (every composite number must have such a factor). The primes are then just the list of all numbers larger than 1 excluding the composites. To avoid unbounded recursion, the number 2 must be explicitly given as a prime.

$$\begin{aligned}
&\mathit{primes} :: [\mathit{Integer}] \\
&\mathit{primes} = 2 : ([3\,..]\ `\mathit{excluding}`\ \mathit{composites}) \\
&\mathit{composites} :: [\mathit{Integer}] \\
&\mathit{composites} = \mathit{fmergeAll}\ [[p * p, p * (p + 1)\,..] \mid p \leftarrow \mathit{primes}]
\end{aligned}$$

And that's it. `primes` is an efficient list of all the prime numbers.

References

[1] MELISSA E O'NEILL. The genuine sieve of eratosthenes. *Journal of Functional Programming*, 19(1):95–106, 2009.