Under consideration for publication in J. Functional Programming

A list-based Sieve of Eratosthenes

David Spies

Abstract

We introduce a Sieve of Eratosthenes in Haskell which produces an infinite list of prime numbers, and achieves $O(n \log n \log\log n)$ performance without making use of *any* data structure besides infinite lists.

1 Introduction

In "The Genuine Sieve of Eratosthenes" (O'NEILL, 2009), they refute the claim that a well-known Haskell one-liner for generating a list of prime numbers is an implementation of the Sieve of Eratosthenes and show that its computaitonal behavior is fundamentally different (not to mention significantly slower). They then go on to produce an actual implementation making use first of binary-tree-based maps and later of queues claiming:

> *An implementation of the actual sieve has its own elegance, showing the utility of well-known data structures over the simplicity of lists [...] choosing the right data structure for the problem at hand [can make] an order of magnitude performance difference.*

Here, we produce an implementation of the Sieve of Eratosthenes which *is* computationally similar to O'Neill's in that it has the same big-O performance and ultimately works by merging together streams of composites and then excluding them from a base-list. But it does so without ever making explicit use of any data-structure besides infinite lists. The heap which merges together lists of composites is implicitly built into the way the function calls which generate these lists are structured rather than being encoded explicitly as data.

1.1 Literate Haskell

This document (or rather `primes.lhs`, the LaTeX file used to generate this document) is literate Haskell. It has been tested on GHC 8.2.2. To compile it we will need a main method. We can take a positive number n as the sole command line argument print out the n^{th} prime by indexing into the infinite list of primes, `primes` on page 4:

```
import System.Environment (getArgs)

main :: IO ()
main = do
  [ind] <- map read <$> getArgs
  print $ primes !! (ind - 1)
```

To run it, try:

```
$ ghc -O2 primes.lhs
$ ./primes 1000000
15485863
```

Finding the millionth prime takes about 7 seconds on an Intel Core™ i5-4200M CPU.

Additionally, "`ghci -i primes.lhs`" can be used to otherwise play with the various functions and structures defined in this paper.

2 Merging Lists

All lists used in this document are infinite and sorted. All lists of lists are infinite and sorted on their heads. When dealing with sorted lists, it is useful to have a merge function.

```
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. Instead of this merge function, we will need one which prepends the head of the left list before merging the remainder with the right list. If we can prove via some *other* means that the head of the left input is less than the head of the right, then this will return the same result as just calling `merge`. But obtaining the head of the result won't require any evaluation of the right argument.

```
fmerge :: Ord a => [a] -> [a] -> [a]
fmerge (x : xs) ys = x : merge xs ys
```

To see how this might be useful, here is an ordered list of all numbers whose only prime factors are 2 and 3:

```
twosThreesOnly :: [Integer]
twosThreesOnly = fmerge (iterate (2 *) 1) [3 * x | x <- twosThreesOnly]
```

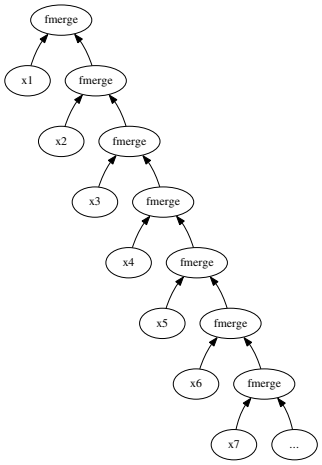
Were we to define it using `merge` rather than `fmerge`, attempting to evaluate `twosThreesOnly` would simply cause our program to loop forever unable to determine the recursively-defined first element.

`fmerge` can *also* be used to merge an infinite list of sorted lists, provided 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 theoretically is not very performant. In the worst case, evaluating the kth element can require $\Omega(k)$ running time. To see why this is, take a look at what structure results from evaluating `fmergeAllNaive` on a list. Suppose we have the list:

Figure 1. Structure of `fmergeAllNaive`




```
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, we will first create a helper function for efficiently merging *just the length-k prefix* of a list of lists (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
```

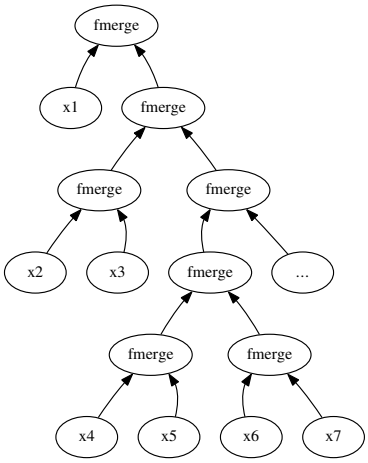
This should look famillliarly like a standard merge-sort, except that we're using `fmerge` rather than `merge` (and all our lists are infinite).

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 is 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

Figure 2. Structure of `fmergeAll`



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).

3 Excluding From a List

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
```

4 Primes and Composites

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. The number 2 is explicitly given as the head of the list to avoid infinite looping to find the head of the list at runtime.

```
primes :: [Integer]
primes = 2 : ([3..] `excluding` composites)

composites :: [Integer]
composites = fmergeAll [[p * p, p * (p + 1)..] | p <- primes]
```

Since the list of primes is in ascending order and the operation of squaring is monotonic on positive integers, we can be sure that the heads of the composite lists to be merged are also in ascending order. Thus it is safe to use `fmergeAll` here for merging together all the lists of composites.

5 Rolling The Wheel

In (O'NEILL, 2009) they additionally show how to build a "wheel" sieve by skipping multiples of the smallest (and hence most common) primes. We can do the same with the list-based version. First, observe how we can ignore all the even numbers in our computation:

```
oddPrimes :: [Integer]
oddPrimes = 3 : ([5,7..] `excluding` oddComposites)

oddComposites :: [Integer]
oddComposites = fmergeAll [[p * p, p * (p + 2)..] | p <- primes]

primes2 :: [Integer]
primes2 = 2 : oddPrimes
```

Re-running with `primes` changed to `primes2` in the main method on page 1 gives us the millionth prime in only 3.5 seconds. Nearly exactly half of what it costs using the straightforward sieve.

To ignore multiples of 3 as well, we start with the list of prime `candidates` and then filter out the composites based on those.

```
wheelCandidates :: [Integer]
wheelCandidates = 5 : concat [[n + 1, n + 5] | n <- [6, 12..]]

wheelPrimes :: [Integer]
wheelPrimes = 5 : (tail wheelCandidates) `excluding` wheelComposites

wheelComposites :: [Integer]
wheelComposites =
  fmergeAll [[p * k | k <- dropWhile (< p) wheelCandidates]
    | p <- wheelPrimes]

primes3 :: [Integer]
primes3 = 2 : 3 : wheelPrimes
```

And by changing `primes` to `primes3` we observe an almost-negligible speedup from 3.5 to 3.3 seconds to find the millionth prime. Perhaps a greater speedup could be obtained by replacing the `dropWhile` above with a direct computation of the first term for each list or by finding a way to generate successive composites by addition rather than multiplication as is done in the `primes` and `primes2` case.

References

O'NEILL, MELISSA E. (2009). The genuine sieve of eratosthenes. *Journal of functional programming*, **19**(1), 95–106.