JFP **21** (6): 563–572, 2011. © Cambridge University Press 2011
doi:10.1017/S0956796811000128 First published online 16 May 2011

FUNCTIONAL PEARL
A note on the genuine Sieve of Eratosthenes

MATTI NYKÄNEN
School of Computing, University of Eastern Finland, FI-70211 Kuopio, Finland
(*e-mail:* matti.nykanen@uef.fi)

Abstract

O'Neill (The genuine Sieve of Eratosthenes. *J. Funct. Program.* **19**(1), 2009, 95–106) has previously considered a functional implementation for the genuine Sieve of Eratosthenes, based on the well-known heap data structure. Here, we develop it further by adapting this data structure to this particular application.

1 Introduction

O'Neill (2009) discussed implementing the venerable Sieve of Eratosthenes for finding prime numbers in the lazy functional programming language Haskell (Peyton Jones, 2003). She began by pointing out that the usual 'one-liner'

```
primes = sieve [2..] where sieve (p:xs) = p:sieve [x|x←xs,x `mod` p > 0]
```

(as given, for instance in Chapter 12.6 of Hutton's Haskell textbook, 2007) is *not* the actual sieve[1] because it eliminates the multiples of already found primes differently than Eratosthenes had intended, namely by *trial division*; that is, by trying to divide each candidate still remaining in the list `xs` by the primes `p` found so far to see whether this candidate is prime or not. Moreover, it is inefficient even as a form of trial division, since it tries to divide in vain with primes larger than the square root of the current candidate. She then argued that he had intended instead to construct the multiples $2 \cdot \mathtt{p}, 3 \cdot \mathtt{p}, 4 \cdot \mathtt{p}, \ldots$ of each prime `p` *explicitly* and then cross them out from the remaining candidates. This explicit approach is more efficient than trial division in two ways: First, the sieve requires

$$\Theta(n \cdot \ln \ln n) \tag{1}$$

operations to find all the primes up to n, whereas trial division requires asymptotically more. Second, trial division requires laborious division operations, whereas these explicit multiples can be constructed less tediously via repeated additions. Given that Eratosthenes developed his sieve in times of calculations by hand, he must have intended this faster interpretation than trial division.

[1] I for one had disseminated this erroneous version in my own lectures. This note can be construed as an apology with an offering to redeem myself.

```
class Multiplex mx where
    starting :: Ord k ⇒ mx k
    withList :: Ord k ⇒ mx k → [k] → mx k
    get1st   :: Ord k ⇒ mx k → k
    but1st   :: Ord k ⇒ mx k → mx k
```

Listing 1: The type class for storing list of multiples.

O'Neill (2009) then went on to develop a Haskell implementation of this explicit approach. Her approach can be summarized as follows: Maintain a *heap*, where each entry is the list of still unconsumed multiples of a particular prime already found with the first element as the key. The laziness of Haskell constructs the next multiple in such a conceptually infinite list only when needed. When the algorithm needs to consume the next multiple, it can be found by taking the list with the smallest key from the heap, taking its first element (that is its key) and returning the rest of this list back to the heap. While this approach does attain the bound given by Equation (1) if we disregard maintaining the data structure, maintaining this heap incurs a logarithmic overhead with respect to the number of primes found.

In this note, we develop this approach further by fine tuning the generic heap data structure with some observations about this particular application (in Section 2) and then measuring how this improves performance (in Section 3). Since we wish to compare specifically just the effects of the data structure details while keeping all the other implementation aspects the same, let us first define a type class for different implementations of this *store for the multiples* of already found primes, given as Listing 1. That is, the type `mx` can act as such a store, if it offers the following four operations:

i. creating an empty initial store;
ii. adding a conceptually infinite ascending list of multiples into it (this is where the laziness of Haskell is employed);
iii. reading the smallest element currently stored in it and
iv. consuming this smallest element from it.

The intended meaning of this type class can be stated more formally by postulating a fifth operation `contents :: (Ord k) => mx k -> [k]` for 'the contents of the given store as a list' with the four axioms

i. `contents starting = []`
ii. `contents (m ‘withList‘ xs) = contents m ‘merge‘ xs`
where the `merge` function defined in Listing 3 merges the two ordered lists given as inputs together into one also ordered list
iii. `get1st = head . contents`
iv. `but1st = tail . contents`

characterizing the four operations. Note that this specifies `mx` to be a *bag* that contains as many copies of the same element as have been added. For instance, $225 = 3^2 \cdot 5^2$ would appear twice, since it is a multiple of both 3 and 5. However, the algorithm itself would require only a *set* with just one copy of each multiple.

https://doi.org/10.1017/S0956796811000128 Published online by Cambridge University Press

```
primes :: ∀ mx k . (Multiplex mx,Integral k) ⇒ mx k → [k]
primes _ =
    2 : found [3,5..] (starting :: mx k)
    where found ps@(p:ps') ms =
              p : sift ps' (ms `withList` multiplesOf ps)
          sift ps@(p:ps') ms =
              case compare p (get1st ms)
              of LT → found ps ms
                 GT → sift ps  $ but1st ms
                 EQ → sift ps' $ but1st ms
          multiplesOf qs@(q:_) =
              map (*q) qs
```

Listing 2: The list of primes using the type class from Listing 1.

```
instance Multiplex [] where
    starting = []
    withList = merge
    get1st   = head
    but1st   = tail

merge :: Ord t ⇒ [t] → [t] → [t]
merge [] ys = ys
merge xs [] = xs
merge xs@(x:xs') ys@(y:ys')
    | x < y =
        x : merge xs' ys
    | otherwise =
        y : merge xs  ys'
```

Listing 3: Using Haskell's built-in lists to store the multiples.

The type class in Listing 1 enables us to rewrite the Sieve of Eratosthenes as suggested by O'Neill (2009) as in Listing 2, where the dummy argument merely carries the type of store to use; the `undefined` value of that type suffices. This dummy argument also turns `primes` from an infinite list constant into a constant function so that it is not necessary to retain the already computed primes in memory.

The simplest store uses Haskell's built-in list type directly, as in Listing 3. It can be obtained directly from the axioms above by choosing `contents` to be the identity function `id`. This is obviously not a very efficient implementation, because the store `ms` will develop as

$$\texttt{merge (merge (merge [] } \underbrace{\texttt{[9,15,21,...]}}_{\mathsf{mop}(3),}\texttt{) } \underbrace{\texttt{[25,35,45,...]}}_{\mathsf{mop}(5),}\texttt{) } \underbrace{\texttt{[49,63,77,...]}}_{\mathsf{mop}(7),\ldots} \quad (2)$$

where

$$\mathsf{mop}(p) = \left[p^2, p^2 + 2 \cdot p, p^2 + 4 \cdot p, \ldots\right]$$

denotes the infinite ascending list of odd multiples of the prime p starting from its square p^2 and inserted by the algorithm. This is a badly skewed tree of these `merge` operations, and this skewedness is what slows down finding and consuming

https://doi.org/10.1017/S0956796811000128 Published online by Cambridge University Press

```
data PriorityQ k v = Lf
                   | Br !k v !(PriorityQ k v) !(PriorityQ k v)
                   deriving (Eq, Ord, Read, Show)

emptyPQ :: PriorityQ k v
emptyPQ = Lf

minKeyPQ :: PriorityQ k v → k
minKeyPQ (Br k v _ _)         = k
minKeyPQ _                    = error "Empty heap!"

insertPQ :: Ord k ⇒ k → v → PriorityQ k v → PriorityQ k v
insertPQ wk wv (Br vk vv t1 t2)
               | wk ⩽ vk   = Br wk wv (insertPQ vk vv t2) t1
               | otherwise = Br vk vv (insertPQ wk wv t2) t1
insertPQ wk wv Lf             = Br wk wv Lf Lf

siftdown :: Ord k ⇒
            k → v → PriorityQ k v → PriorityQ k v → PriorityQ k v
siftdown wk wv Lf _             = Br wk wv Lf Lf
siftdown wk wv (t @ (Br vk vv _ _)) Lf
    | wk ⩽ vk                   = Br wk wv t Lf
    | otherwise                 = Br vk vv (Br wk wv Lf Lf) Lf
siftdown wk wv (t1 @ (Br vk1 vv1 p1 q1)) (t2 @ (Br vk2 vv2 p2 q2))
    | wk ⩽ vk1 && wk ⩽ vk2    = Br wk wv t1 t2
    | vk1 ⩽ vk2               = Br vk1 vv1 (siftdown wk wv p1 q1) t2
    | otherwise               = Br vk2 vv2 t1 (siftdown wk wv p2 q2)
```

Listing 4: A functional binary heap.

the smallest element. It is found by considering the first argument of each `merge` operation, and this makes it linear in the number of primes found so far.

This is why O'Neill (2009) chose instead a heap as the store: It keeps the data structure balanced, and thereby reduces this search for the smallest element from linear to logarithmic. Subsequently, Smith collected and made the code in her article available (O'Neill & Smith, 2009). This code uses the functional generic *binary* heap shown in Listing 4, which is a Haskell translation of the Standard ML code given by Paulson (1996) in Chapter 4.16.

With this generic heap as the store, (one copy of) the smallest element is consumed as follows:

> The key for the root node r can be consumed by moving the first element from the datum for r to become its new, larger key and restoring the heap property by *shifting* down this changed node into its proper place in the heap. (3)

Listing 5 implements this key consumption rule (3), where the function `siftdown` implements the actual shifting. Figure 1 illustrates its operation: The tree on the left shows the two highest levels of the heap when key 21 is being consumed, marked by crossing it out. The circled numbers are the keys, with the associated data shown to their right. The tree on the right shows the result: 27 becomes the new key, and the

https://doi.org/10.1017/S0956796811000128 Published online by Cambridge University Press

```
newtype Heap0 k
    = Heap0 (PriorityQ k [k])

instance Multiplex Heap0 where
    starting =
        Heap0 emptyPQ
    (Heap0 h) `withList` (x:xs) =
        Heap0 $ insertPQ x xs h
    get1st (Heap0 h) =
        minKeyPQ h
    but1st (Heap0 (Br _ (x:xs) t1 t2)) =
        Heap0 $ siftdown x xs t1 t2
```

Listing 5: Using the traditional heap in Listing 4.

Fig. 1. Deleting the key at the root and shifting its list down in the heap.

corresponding root node has swapped places with the old left child, as indicated by the dotted arrow.

2 Stealing your children's keys

The heap is useful whenever we must maintain a collection of data, where each datum has its own associated key, and this data is processed in the order prescribed by these keys. However, in this particular application, we can make the following three observations, where the first two are immediate while the third merits a small proof:

I. Here, each datum is, in fact, a source of future keys, rather than some generic value.
   That is, in this sense, we have here a *stronger* connection between a key and its datum than in the generic heap data structure.

II. Here, we are only interested in consuming these current and future keys in ascending order. Hence, we are allowed to associate a key with any datum we wish, and not just the one originally associated with it, as long as we maintain the heap property.
   That is, in this sense, we have here a *weaker* connection between a key and its datum than in the generic heap data structure.

III. Suppose that the algorithm has just determined that the current candidate p is indeed a prime and is therefore adding its list $\mathsf{mop}(p)$ into the current heap. Then, the key p^2 of this new list is larger than any key in the current heap.

https://doi.org/10.1017/S0956796811000128 Published online by Cambridge University Press

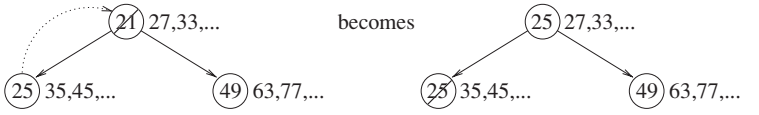

Fig. 2. Deleting the key at the root and shifting another into its place.

Proof

Assume to the contrary that the list for some previously found prime $q < p$ starts with some element $q^2 + 2 \cdot r \cdot q > p^2$, where $r > 0$ is the number of elements already consumed from this list. This inequality can be rearranged into $r > \frac{p-q}{2} \cdot \frac{p+q}{q}$ where the first fraction equals the number of candidates tested by the sieve for primality between finding q and reaching p, and the second fraction is more than 1. This, in turn, contradicts the fact that testing a candidate for primality consumes at most one element from any list. □

Listing 5 did not take any of these three observations into account while updating its heap, so let us now fine tune it by doing so.

The first two observations suggest the following rule for consuming a key instead of the earlier key consumption rule (3):

> If the first element in the datum for n, the node whose current key is being consumed, can become its new key without violating the heap property, then so be it, by observation I. Otherwise, *steal* the smallest key from the (one or two) children of n, and the child who lost its key is then responsible for finding itself a new key, using again this rule (4) recursively, by observation II. (4)

On the one hand, note that the proof given above for observation III continues to hold for this new key consumption rule (4) as well. On the other hand, note that the 'pleasing but minor optimization', as characterized by O'Neill (2009), of starting $\mathsf{mop}(p)$ with p^2 instead of $2 \cdot p$ now becomes central: While the analogue for observation III would hold even without this optimization when the old rule (3) is used, this might no longer be guaranteed when this new rule (4) is used instead.

Figure 2 illustrates the operation of this new rule (4): 27 cannot replace the consumed key 21, because this would violate the heap property, so 25 gets stolen from the left child, as indicated by the dotted arrow, which must then find another key to replace it. If 35 suffices as the replacement for the left child, so be it; otherwise, the left child will, in turn, steal itself a new replacement key from its own children.

The usefulness of this new rule (4) versus the original rule (3) can be seen by comparing how the computation proceeds after Figure 1 versus Figure 2: First the next prime 23 is found and its list $\mathsf{mop}(23)$ is added, but this does not alter these two top levels in either figure. Then, the key 25 gets consumed. In Figure 1, this results in swapping the root and its left child back to their original positions, whereas in Figure 2, it suffices to take 27 as the new key from the list, which has stayed in its original place. This demonstrates that rule (4) can involve less work and fewer node allocations than rule (3), warranting its further study.

https://doi.org/10.1017/S0956796811000128 Published online by Cambridge University Press

In general, such savings stem from the fact that *key consumption retains the lists in the same positions* within the heap when rule (4) is used – only the keys move to restore their heap order. This opens another optimization possibility: *Ensure that each list will be in a good position* within the heap. By this, we mean that the most frequently accessed lists should be positioned closest to the root. The list $\mathsf{mop}(p)$ is accessed more frequently than the list $\mathsf{mop}(p)$ whenever $p < q$. Hence, we see that a good position for the next list to insert is (a) on the one hand as close to the root as possible, (b) but on the other hand no closer than any of the previously inserted lists. For instance, the `merge` expression tree in Equation (2) fails miserably both of these desirable properties (a) and (b).

Fortunately, observation III and a binary heap together guarantee such a good positioning: Observation III ensures that when the list $\mathsf{mop}(p)$ for the most recently found prime p is being inserted, its initial key p^2 is larger than any key already in the heap. A binary heap ensures, in turn, that when such a key is inserted, its new node will be positioned as close to the root as possible (so property (a) is now satisfied) without having to alter the distances of any existing nodes from the root (so property (b) is now satisfied as well).

Listing 6 implements this approach. Note how `myInsertPQ` simplifies and optimizes the `insertPQ` in Listing 4 through observation III. Indeed, this optimization would apply also to Listing 5, but there its effects would be minor, since its key consumption rule (3) would later shuffle the positions of the lists within the heap anyway.

3 Measurement results

Since the original implementation in Listing 5 already attains the asymptotically optimal number of arithmetic operations given as Equation (1), and both the original and our implementation incur the asymptotically same overhead for the heap operations, we can only hope that our own implementation in Listing 6 with its fine-tuned heap manages to reduce the constant factors. Hence, we turn to measuring and comparing their performance with respect to each other.

Our measurements were carried out with compiled and optimized code, to avoid measuring the perhaps substantial overheads caused by using the typeclass `Multiplex`. More precisely, we used the Glasgow Haskell Compiler (GHC) version 6.10.3 with the options `-O2 -funbox-strict-fields` on the Linux Fedora Core 11 operating system. We used one processor core of a Lenovo ThinkPad model T60p with a 2.33 GHz Intel Core 2 Duo T7600 processor. Arithmetic was performed using the built-in type `Int` so that Equation (1) remains valid.

We measured the user time spent on computing the mth odd prime. The results are shown as Figure 3, which does indeed show the improvement that we hoped for. The intermittent staircase-like jumps in the measured timings are probably due to the effects of garbage collections. They too show an improvement as delays in these jumps due to fewer allocations required by our method.

One way to summarize the improvement shown in Figure 3 is as follows. Take Equation (1) as an asymptotic estimate for the work performed by the algorithm,

https://doi.org/10.1017/S0956796811000128 Published online by Cambridge University Press

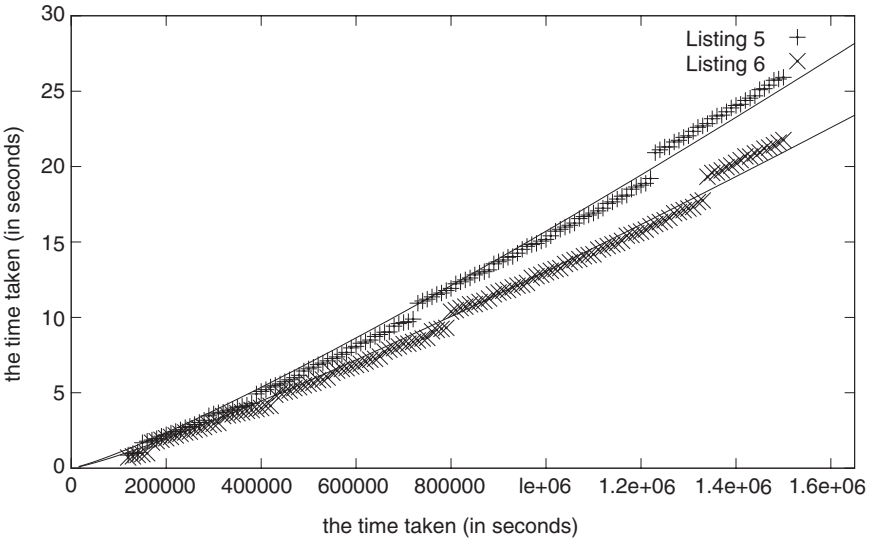

Fig. 3. The measurement results.

except for the heap overhead. The mth prime is about $m \cdot \ln m$, so we can replace the quantity n in it with this estimate to get a function of m, the quantity used in our tests. The heap overhead can, in turn, be incorporated by multiplying it further with $\ln m$. This gives us an estimate of the form $a \cdot m \cdot (\ln m)^2 \cdot \ln \ln(m \cdot \ln m) + b$ for the running time. Fitting this estimate to our measurements yields

coefficient a	coefficient b	
$2.94105 \cdot 10^{-8}$	$2.0341 \cdot 10^{-7}$	for Listing 5
$2.44391 \cdot 10^{-8}$	$2.03409 \cdot 10^{-7}$	for Listing 6.

These two curves are also drawn in Figure 3. Comparing their leading coefficients a summarizes our improvement as $1 - \frac{2.44391}{2.94105} \approx 17\%$.

Note finally that O'Neill (2009) also showed how the sieve can be optimized further by adding a wheel for the first few primes. For instance, she obtained a threefold speedup to her heap-based sieve by adding a wheel for the first four primes 2, 3, 5 and 7. The gist of this optimization is to avoid having to maintain their multiples in the store, and not in altering the properties of the store itself. In contrast, our optimization addresses the properties of the store. Hence, these two optimizations could also coexist in the same code.

4 Conclusion

We have reconsidered O'Neill's (2009) functional implementation for the genuine Sieve of Eratosthenes, which was based on using the well-known heap data structure. Here, we developed her approach further by fine tuning this generic data structure for this particular application. This led to a performance increase of about 17% according to our measurements.

https://doi.org/10.1017/S0956796811000128 Published online by Cambridge University Press

```
mySiftdown :: Ord k ⇒
            [k] → PriorityQ k [k] → PriorityQ k [k] → PriorityQ k [k]
mySiftdown (wk:wv) Lf _              = Br wk wv Lf Lf
mySiftdown (wk:wv) (t @ (Br vk vv _ _)) Lf
    | wk ⩽ vk                        = Br wk wv t Lf
    | otherwise                      = Br vk vv (Br wk wv Lf Lf) Lf
mySiftdown w@(wk:wv) (t1 @ (Br vk1 vv1 p1 q1)) (t2 @ (Br vk2 vv2 p2 q2))
    | wk ⩽ vk1 && wk ⩽ vk2           = Br wk wv t1 t2
    | vk1 ⩽ vk2                      = Br vk1 w (mySiftdown vv1 p1 q1) t2
    | otherwise                      = Br vk2 w t1 (mySiftdown vv2 p2 q2)

myInsertPQ :: Ord k ⇒ [k] → PriorityQ k [k] → PriorityQ k [k]
myInsertPQ w (Br vk vv t1 t2) =
    Br vk vv (myInsertPQ w t2) t1
myInsertPQ (wk:wv) Lf =
    Br wk wv Lf Lf

newtype Heap1 k
    = Heap1 (PriorityQ k [k])

instance Multiplex Heap1 where
    starting =
        Heap1 emptyPQ
    (Heap1 h) `withList` xs =
        Heap1 $ myInsertPQ xs h
    get1st (Heap1 h) =
        minKeyPQ h
    but1st (Heap1 (Br _ xs t1 t2)) =
        Heap1 $ mySiftdown xs t1 t2
```

Listing 6: Our alternative for Listing 5.

In closing, we would like to point out two open questions on how this approach could be improved further:

- First, is there an efficient functional data structure for the set of multiples instead of the bag used here? Or failing that, is there some data structure between them which would be able to eliminate at least some of the repeated elements internally?
- Second, we have developed here a good positioning for the lists of multiples of already found primes, based on balanced binary trees and the order in which these lists arrive. Can even better placements be developed instead, based on some other kinds of trees or the contents of these lists?

Acknowledgments

The author thanks the editor and the anonymous referees for their valuable comments on improving the presentation.

References

Hutton, G. (2007) *Programming in Haskell*. Cambridge University Press.

O'Neill, M. E. (2009) The genuine Sieve of Eratosthenes. *J. Funct. Program.* **19**(1), 95–106.

O'Neill, M. E. & Smith, L. P. (2009) The NumberSieves Package [online]. Accessed July 13, 2009. Available at: `http://hackage.haskell.org/package/NumberSieves`

Paulson, L. C. (1996) *ML for The Working Programmer*. 2nd ed. Cambridge University Press.

Peyton Jones, S. (ed) (2003) *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press. Also available as *J. Funct. Program.*, Special Issue, **13**(1), 2003, and [online]. Available at: `http://haskell.org/onlinereport/`