JFP **14** (6): 759–763, 2004. © 2004 Cambridge University Press
DOI: 10.1017/S0956796804005210 Printed in the United Kingdom

FUNCTIONAL PEARL
Calculating the Sieve of Eratosthenes

LAMBERT MEERTENS
Kestrel Institute, Palo Alto, CA, USA
and
Utrecht University, Utrecht, The Netherlands
(*e-mail:* lambert@kestrel.edu)

1 The Sieve of Eratosthenes

The *Sieve of Eratosthenes* is an efficient algorithm for computing the successive primes. Rendered informally, it is as follows:

1. Write down the successive "plurals": 2, 3, 4, …
2. Repeat:
   (a) Take the first number that is not circled or crossed out.
   (b) Circle it.
   (c) Cross out its proper multiples.
3. What is left (i.e. the circled numbers) are the successive prime numbers.

Here are the first few rounds shown in action:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	…
②	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~	11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~	…
②	③	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~	…
②	③	~~4~~	⑤	~~6~~	7	~~8~~	~~9~~	~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~	…

Although formulated in an imperative style, this algorithm must be executed somewhat lazily, for otherwise the process will hang already in the first step. For manual execution by calculating non-prodigies the main efficiency gain is in the avoidance of division, since 'manual' division takes considerably more effort than addition. As this is not a big issue for modern electronic computers, we shall ignore this aspect. What remains then as the essence of this sieve algorithm is this: it produces a *stream* of primes, and that stream is used *while it is being produced* to filter itself. Thus, each number being tested for primality is only tested for divisibility by smaller primes, instead of all smaller plurals.

There is a well-known two-line functional program embodying this sieve. Although not the most efficient functional program for Eratosthenes' Sieve, this elegant little program may be considered part of the folklore of functional programming. When first confronted with this program, many a student of functional programming finds it hard to understand "why" it works. In fact, most may not know how to approach this other than by the error-prone technique of building a mental model

of the behaviour of the program under the standard operational semantics. Yet it is possible to give simple and precise arguments not employing operational notions, and among those, derivation by program calculation is perhaps the most satisfying and convincing. So, what we set out to do in the remainder is to construct that folklore functional program by calculation.

In order to do so we recapitulate some standard theory about streams and about primes in the following two sections.

2 Streams

For our present purposes, a *stream* is always an *infinite* list.

The data type of streams can be viewed as the so-called carrier of a final coalgebra. The corresponding general *anamorphism* (Malcolm, 1990; Meijer *et al.*, 1991) – in particular for streams also known as an *unfold* – with suitably typed ingredient functions f and g is the function h satisfying

$$h\,x \;=\; f\,x \;:\; h(g\,x) \tag{1}$$

Informally, the value computed by function h applied to argument x is the stream $[f\,x,\ f(g\,x),\ f(g(g\,x)),\ \ldots\,]$. Note that equation (1), given functions f and g, fully characterizes anamorphism h. In other words, viewed as a functional equation in unknown h, (1) has a *unique* solution. A notation for that function is $(\!|f \vartriangle g|\!)$.

A particular case is the anamorphism $(\!|f \vartriangle (+1)|\!)$, where the second ingredient function $(+1)$ is the successor function on the naturals. We prove that

$$(\!|f \vartriangle (+1)|\!)\ n \;=\; map\ f\ [n..] \tag{2}$$

by verifying that the right-hand side, viewed as a function of n, satisfies (1) with ingredients $f := f$ and $g := (+1)$. So we calculate:

$$\begin{array}{ll} & map\ f\ [n..] \\ = & \quad \{\ \text{definition of `..'}\ \} \\ & map\ f\ (n\ :\ [n{+}1..]) \\ = & \quad \{\ \text{definition of } map\ \} \\ & f\ n\ :\ map\ f\ [n{+}1..] \end{array}$$

Since the solution to (1) is unique, to establish that some function h equals $\lambda n \rightarrow map\ f\ [n..]$ it now suffices to show that it satisfies the equation $h\ n \;=\; f\ n : h(n{+}1)$.

All Haskell lists used explicitly or implicitly in what follows will actually be ascending streams of naturals.

3 Characterizing the primes

The stream of primes can be described as:

$$primes \;=\; map\ nthPrime\ [0..] \tag{3}$$

using the fact that there is an inexhaustible supply of primes: $nthPrime\ 0 = 2$, $nthPrime\ 1 = 3$, $nthPrime\ 2 = 5$, etc. Of course, this needs to be supplemented by a characterization of function *nthPrime*.

https://doi.org/10.1017/S0956796804005210 Published online by Cambridge University Press

What is a prime number? A *plural* is a natural number that is at least 2. The stream of all plurals is given by $[2..]$. A *prime* is then a plural that is not divisible by any other plural. Using the fact that a natural divisor of a plural p is at most p, this can equivalently be restated as: a prime is a plural that is not divisible by any smaller plural. It follows immediately from this characterization that 2 is prime. Since a non-prime plural (a.k.a. a *composite*) is divisible by a smaller plural, it is divisible by some smaller prime. This is the first step towards the proof of the prime factorization theorem, but here we use it for yet another characterization of the primes, one that is geared towards the Sieve:

a prime is a plural that is not divisible by any smaller prime.

In an ascending stream *smaller* is the same as *prior*. Therefore, $nthPrime\ n$ is the head of the stream of plurals that remain after removing from $[2..]$ the multiples of the primes $nthPrime\ 0$, $nthPrime\ 1$, and so on, up to but not including $nthPrime\ n$.

The preceding is deemed part of elementary number theory. We can express it in Haskell, as follows. The operation of removing the multiples of primes prior to $nthPrime\ n$ from a stream is given by $removeTo\ n$, using this definition of *removeTo*:

$$removeTo\ 0 \quad = \quad id \tag{4}$$

$$removeTo\ (n{+}1) \quad = \quad filter\ (notdiv\ (nthPrime\ n)) \cdot removeTo\ n \tag{5}$$

where

$$notdiv\ d\ n \ = \ n\ `mod`\ d \neq 0 \tag{6}$$

So, applying this filter to the plurals and taking the head of the resulting stream, we have:

$$nthPrime\ n \ = \ head\ (removeTo\ n\ [2..]) \tag{7}$$

The intermediate stream $removeTo\ n\ [2..]$ corresponds, for successive values of n, to the 'untouched' plurals (i.e. neither circled nor crossed out) in successive rounds of the Sieve as shown in action in Section 1. Actually, (4)–(7) constitute a ludicrously inefficient, but nevertheless effective, Haskell program for computing the primes. It is so inefficient because the smaller primes are recomputed again and again, and so the time to compute the next prime roughly doubles each time. If function *nthPrime* is 'tabulated' or 'memoized' (Bird, 1980), this program is tolerably inefficient. In a sense, memoizing is what the Sieve does: it 'reuses' the previously computed primes.

Note that, for each natural n, $removeTo\ n$ is a filter, i.e., a function on streams that can be expressed in the form $filter\ r$ for some predicate r. In general, if f is a filter, and $h \ = \ head\ (f\ [2..])$, it holds that $f\ [2..] \ = \ h \ : \ f\ [h{+}1..]$. This property of filters, which we state without proof, can be used to strengthen (7) to a property of the stream $removeTo\ n\ [2..]$:

$$removeTo\ n\ [2..] \ = \ nthPrime\ n \ : \ removeTo\ n\ [(nthPrime\ n) + 1\ ..] \tag{8}$$

https://doi.org/10.1017/S0956796804005210 Published online by Cambridge University Press

4 Calculating the solution

A straightforward generalization of (3) is:

$$primesFrom\ n\ =\ map\ nthPrime\ [n..] \tag{9}$$

So, by (2), $primesFrom\ =\ [\![nthPrime_{\triangle}(+1)]\!]$. We want to find an algorithm in sieve form for this, meaning that it uses the stream of primes it produces to filter the very same stream. To find such a solution we let ourselves be guided by the fact that it has to satisfy (1) for $f := nthPrime$ and $g := (+1)$, or

$$primesFrom\ n\ =\ nthPrime\ n\ :\ primesFrom\ (n{+}1) \tag{10}$$

An 'inspired guess' is now that we can express function *primesFrom* in sieve form by:

$$primesFrom\ n\ =\ sieve\ (removeTo\ n\ [2..]) \tag{11}$$

for some function *sieve*. This guess is, of course, informed by the form of (7).

The remaining task is to find a suitable definition for *sieve*. Let us substitute the right-hand side of (11) for *primesFrom* in both sides of (10) and see how far we can symbolically evaluate the results. For the sake of brevity '*nthPrime n*' is abbreviated throughout by '*p*'. For the left-hand side we calculate then:

$$\begin{array}{ll} & primesFrom\ n \\ = & \quad \{\ (11)\ \} \\ & sieve\ (removeTo\ n\ [2..]) \\ = & \quad \{\ (8)\ \} \\ & sieve\ (p\ :\ removeTo\ n\ [p{+}1..]) \\ = & \quad \{\ \text{abbreviating '}removeTo\ n\ [p{+}1..]\text{' to '}ns\text{'}\ \} \\ & sieve\ (p\ :\ ns) \end{array}$$

For the right-hand side:

$$\begin{array}{ll} & p\ :\ primesFrom\ (n{+}1) \\ = & \quad \{\ (11)\ \} \\ & p\ :\ sieve\ (removeTo\ (n{+}1)\ [2..]) \\ = & \quad \{\ (5),\ \text{definition of '}\cdot\text{'}\ \} \\ & p\ :\ sieve\ (filter\ (notdiv\ p)\ (removeTo\ n\ [2..])) \\ = & \quad \{\ (8)\ \} \\ & p\ :\ sieve\ (filter\ (notdiv\ p)\ (p\ :\ removeTo\ n\ [p{+}1..])) \\ = & \quad \{\ \text{abbreviating as above}\ \} \\ & p\ :\ sieve\ (filter\ (notdiv\ p)\ (p\ :\ ns)) \\ = & \quad \{\ notdiv\ p\ p\ =\ False,\ \text{definition of } filter\ \} \\ & p\ :\ sieve\ (filter\ (notdiv\ p)\ ns) \end{array}$$

So (10) holds if the final expressions in these two calculations are equal. If we treat *p* and *ns* in these expressions as ordinary variables rather than abbreviations for any specific expressions, and take the equation equating these two final expressions as the definition of *sieve*, the desired equality is certainly achieved.

https://doi.org/10.1017/S0956796804005210 Published online by Cambridge University Press

For *primes* we find then:

$$\begin{array}{ll} & \mathit{primes} \\ = & \quad \{\ (3)\ \} \\ & \mathit{map}\ \mathit{nthPrime}\ [0..] \\ = & \quad \{\ (9)\ \} \\ & \mathit{primesFrom}\ 0 \\ = & \quad \{\ (11)\ \} \\ & \mathit{sieve}\ (\mathit{removeTo}\ 0\ [2..]) \\ = & \quad \{\ (4), \text{definition of } \mathit{id}\ \} \\ & \mathit{sieve}\ [2..] \end{array}$$

The final program is now

$$\begin{array}{l} \mathit{primes} = \mathit{sieve}\ [2..] \\ \mathit{sieve}\ (p : ns) = p : \mathit{sieve}\ (\mathit{filter}\ (\mathit{notdiv}\ p)\ ns) \\ \mathit{notdiv}\ d\ n = n\ \grave{}\mathit{mod}\grave{}\ d \neq 0 \end{array}$$

a two-line program if *notdiv* is expanded in place. Note that *primesFrom* has departed from the stage, having played its role as a catalyst for the calculations.

5 Conclusion

We have arrived at a well-known functional program for the Sieve of Eratosthenes. Clearly, the initial definitions were set up to facilitate the steps leading to this particular solution, and there is no intended claim that invention *ab initio* might have proceeded equally smoothly. The derivation uses, apart from pure formal equational reasoning as originally proposed for program derivation in Burstall & Darlington (1977), and a few snippets of elementary number theory, only the fact that equation (1) is a characterization (also known as a *universal property*). There are other proof methods that will establish the same result (for a comparison of proof methods, see Gibbons & Hutton (1991)), but the present one is arguably the simplest, and without doubt the easiest to teach. It deserves to be better known.

References

Bird, R. S. (1980) Tabulation techniques for recursive programs. *Comput. Surv.* **12**(4), 403–417.

Burstall, R. M. and Darlington, J. (1977) A transformation system for developing recursive programs. *J. ACM*, **24**(1), 44–67.

Gibbons, J. and Hutton, G. (1999) Proof methods for structured corecursive programs. *Proceedings 1st Scottish Functional Programming Workshop*, Stirling, Scotland.

Malcolm, G. R. (1990) *Algebraic Data Types and Program Transformation*. PhD thesis, Department of Computing Science, Groningen University, The Netherlands.

Meijer, E., Fokkinga, M. and Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (editor), *Proceedings ACM Conference on Functional Programming Languages and Computer Architecture: LNCS 523*, pp. 124–144. Springer-Verlag.

https://doi.org/10.1017/S0956796804005210 Published online by Cambridge University Press