JFP, 14 pages, 2021.
doi:10.1017/xxxxx

FUNCTIONAL PEARL

Turner, Bird, Eratosthenes: An Eternal Burning Thread

JEREMY GIBBONS
Department of Computer Science, University of Oxford
e-mail: jeremy.gibbons@cs.ox.ac.uk

Abstract

Functional programmers have many things for which to thank the late David Turner: design decisions he made in his languages SASL, KRC, and Miranda over the last 50 years are still influential and inspiring now.

One example program that he popularized as an illustration of lazy evaluation and list comprehensions in SASL is a one-line recursive "sieve" to generate the infinite list of prime numbers. Turner called this algorithm The Sieve of Eratosthenes. In a lovely paper called "The Genuine Sieve of Eratosthenes" (JFP, 2009), Melissa O'Neill argued that Turner's algorithm is not in fact a faithful implementation of the algorithm, and gave a detailed presentation using priority queues of the real thing. She included a variation by Richard Bird, which uses only lists but makes clever use of circular programming. Bird describes his circular program again in his textbook "Thinking Functionally with Haskell", and sets its proof of correctness as an exercise. Unfortunately, his hint for a solution is incorrect. So what should a proof look like?

One of the last projects Turner worked on was the notion of "Total Functional Programming". He observed that most programs are already structurally recursive or corecursive, therefore guaranteed respectively terminating or productive, and conjectured that "with more practice we will find this is always true". Compelling as this vision is, it seems that we are still some way off achieving it. We explore Bird's circular Sieve of Eratosthenes as a challenge problem for Turner's Total Functional Programming.

The late David Turner had great taste in language design and programming. One example program that he introduced (Turner, 1982) to illustrate lazy evaluation and list comprehensions in SASL is a one-line recursive "sieve" to generate the infinite list of prime numbers:

$$\begin{array}{l} \mathit{primes} :: [\mathit{Integer}] \\ \mathit{primes} \;=\; \mathit{sieve}\,[2\,..]\;\textbf{where}\;\mathit{sieve}\,(p:\mathit{xs}) \;=\; p:\mathit{sieve}\,[x \mid x \leftarrow \mathit{xs}, x \bmod p \neq 0] \end{array}$$

That is, *sieve* takes a stream of candidate primes; the head p of this stream is a prime, and the remaining primes are obtained by removing all multiples of p from the candidates and sieving what's left. It's also a nice unfold (Gibbons and Jones, 1998; Meertens, 2004).

Turner called this algorithm "The Sieve of Eratosthenes". Unfortunately, as O'Neill (2009) observes, this nifty program is not in fact faithful to Eratosthenes. The problem is that for each prime p, every remaining candidate x is tested for divisibility by p. O'Neill calls this algorithm "trial division", and argues that the Genuine Sieve of Eratosthenes should eliminate every multiple of p without reconsidering all the candidates in between. That is, only at most every other natural number should be touched when eliminating multiples of 2, at most one in every three for multiples of 3, and so on. As an additional optimization, it suffices to eliminate multiples of p starting with p^2, since by that point all composite numbers with a smaller nontrivial factor will already have been eliminated.

O'Neill's paper presents a purely functional implementation of the Genuine Sieve of Eratosthenes. The tricky bit is keeping track of all the eliminations when generating an unbounded stream of primes, since obviously one can't eliminate all the multiples of one prime before producing the next prime. Her solution is to maintain a priority queue of iterators; indeed, the main argument of her paper is that functional programmers are often too quick to use lists, when other data structures such as priority queues might be more appropriate.

O'Neill's paper was published in the Journal of Functional Programming, when Richard Bird was the handling editor for Functional Pearls. The paper includes an epilogue that presents a purely list-based but circular implementation of the Genuine Sieve, contributed by Bird during the editing process. Bird describes his circular program again in his textbook "Thinking Functionally with Haskell" (Bird, 2014)*, and sets its proof of correctness as an exercise. Unfortunately, his hint for a solution is incorrect.

One of the last projects Turner worked on was the notion of "Total Functional Programming" (Turner, 2004), "designed to exclude the possibility of non-termination". He observed that most programs are already structurally recursive or corecursive, therefore guaranteed respectively terminating or productive, and conjectured that "with more practice we will find this is always true". But it seems that it is not always so easy. In this paper, we explore Bird's circular Sieve of Eratosthenes as a challenge problem for Turner's Total Functional Programming. What should Bird's proof hint have said?

1 The Genuine Sieve, using lists

Bird's program appears in §9.2 of his book (Bird, 2014), henceforth "TFWH". It deals with lists, but these will be infinite, sorted, duplicate-free streams, and these should be thought of as representing *infinite sets*, in this case sets of natural numbers. In particular, the program involves no empty or partial lists, only properly infinite ones (but our proofs later will have to deal with partial lists).

The prime numbers are what you get by eliminating the composite numbers from the "plural" naturals (those greater than one), and the composite numbers are the proper multiples of the primes—so the program is cleverly circular:

$$
\begin{aligned}
&\mathit{primes}, \mathit{composites} :: [\mathit{Integer}] \\
&\mathit{primes} \quad\;\;\, = \mathit{makeP}\ \mathit{composites} \\
&\mathit{composites} = \mathit{makeC}\ \mathit{primes}
\end{aligned}
$$

* JFP doesn't list O'Neill's paper as a Pearl, but Bird's book describes it that way. Either way, presumably Bird was the handling editor for the paper.

where

$$
\begin{aligned}
&\mathit{makeP}, \mathit{makeC} :: [\,\mathit{Integer}\,] \to [\,\mathit{Integer}\,]\\
&\mathit{makeP}\ \mathit{cs}\ =\ 2 : ([\,3\,..\,] \setminus\!\!\setminus \mathit{cs})\\
&\mathit{makeC}\ \mathit{ps}\ =\ \mathit{mergeAll}\ (\mathit{map}\ \mathit{multiples}\ \mathit{ps})
\end{aligned}
$$

(for later convenience, we have refactored the program as presented by Bird, here naming the components *makeP* and *makeC*).

We'll come back in a minute to *mergeAll*, which unions a set of sets to a set; but (\\) is the obvious implementation of list difference of strictly increasing streams (hence, representing set difference):

$$
\begin{aligned}
&(\setminus\!\!\setminus) :: \mathit{Ord}\ a \Rightarrow [\,a\,] \to [\,a\,] \to [\,a\,]\\
&(x : \mathit{xs}) \setminus\!\!\setminus (y : \mathit{ys})\\
&\quad \mid x < y\ =\ x : (\mathit{xs} \setminus\!\!\setminus (y : \mathit{ys}))\\
&\quad \mid x == y\ =\ \mathit{xs} \setminus\!\!\setminus \mathit{ys}\\
&\quad \mid x > y\ =\ (x : \mathit{xs}) \setminus\!\!\setminus \mathit{ys}
\end{aligned}
$$

and *multiples p* generates the multiples of p starting with p^2:

$$
\mathit{multiples}\ p\ =\ \mathit{iterate}\ (p+)\ (p \times p)
$$

Thus, the composites are obtained by merging together the infinite stream of infinite streams $[\,[4, 6\,..\,], [9, 12\,..\,], [25, 30\,..\,], \ldots\,]$. You might think that you could have defined instead $\mathit{primes}\ =\ [\,2\,..\,] \setminus\!\!\setminus \mathit{composites}$, but this doesn't work: this won't compute the first prime without first computing some composites, and you can't compute any composites without at least the first prime, so this definition would be unproductive. Somewhat surprisingly, it suffices to "prime the pump" (so to speak) just with 2, and everything else flows freely from there.

Returning to *mergeAll*, here is the obvious implementation of *merge*, which merges two strictly increasing streams into one (hence, representing set union):

$$
\begin{aligned}
&\mathit{merge} :: \mathit{Ord}\ a \Rightarrow [\,a\,] \to [\,a\,] \to [\,a\,]\\
&\mathit{merge}\ (x : \mathit{xs})\ (y : \mathit{ys})\\
&\quad \mid x < y\ =\ x : (\mathit{merge}\ \mathit{xs}\ (y : \mathit{ys}))\\
&\quad \mid x == y\ =\ x : \mathit{merge}\ \mathit{xs}\ \mathit{ys}\\
&\quad \mid x > y\ =\ y : \mathit{merge}\ (x : \mathit{xs})\ \mathit{ys}
\end{aligned}
$$

Then *mergeAll* is basically a stream fold with *merge*. You might think you could define this simply by $\mathit{mergeAll}\ (\mathit{xs} : \mathit{xss})\ =\ \mathit{merge}\ \mathit{xs}\ (\mathit{mergeAll}\ \mathit{xss})$, but again this would be unproductive. After all, you can't merge the infinite stream of sorted streams $[\,[5, 6\,..\,], [4, 5\,..\,], [3, 4\,..\,], \ldots\,]$ into a single sorted stream, because there is no least element with which to start. Instead, we have to make the assumption that we have a *sorted* stream of sorted streams; then the binary merge can exploit the fact that the head of the left stream is the head of the result, without even examining the right stream. So, we define:

$$
\begin{aligned}
&\mathit{mergeAll} :: \mathit{Ord}\ a \Rightarrow [\,[\,a\,]\,] \to [\,a\,]\\
&\mathit{mergeAll}\ (\mathit{xs} : \mathit{xss})\ =\ \mathit{xmerge}\ \mathit{xs}\ (\mathit{mergeAll}\ \mathit{xss})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{xmerge} :: \mathit{Ord}\ a \Rightarrow [a] \rightarrow [a] \rightarrow [a] \\
&\mathit{xmerge}\ (x : \mathit{xs})\ \mathit{ys}\ =\ x : \mathit{merge}\ \mathit{xs}\ \mathit{ys}
\end{aligned}
$$

This program is now productive, and *primes* yields the infinite sequence of prime numbers, using the genuine algorithm of Eratosthenes.

2 The Approx Lemma

Bird uses this circular program as an illustration of the *Approx Lemma*. Define

$$
\begin{aligned}
&\mathit{approx} :: \mathit{Int} \rightarrow [a] \rightarrow [a] \\
&\mathit{approx}\ (n+1)\ [\,] \qquad\ \ = [\,] \\
&\mathit{approx}\ (n+1)\ (x : \mathit{xs})\ =\ x : \mathit{approx}\ n\ \mathit{xs}
\end{aligned}
$$

Then we have:

Lemma 1 (Approx Lemma)**.** For finite, partial, or infinite lists *xs*, *ys*,

$$(\mathit{xs}\ =\ \mathit{ys}) \Longleftrightarrow (\forall n \in \mathbb{N}\ .\ \mathit{approx}\ n\ \mathit{xs}\ =\ \mathit{approx}\ n\ \mathit{ys})$$

Note that $\mathit{approx}\ 0\ \mathit{xs}$ is undefined; the function $\mathit{approx}\ n$ preserves the outermost n constructors of a list, but then chops off anything deeper and replaces it with $\perp$ (undefined), returning a partial list if the input was longer. So, the lemma states that to prove two lists equal, it suffices to prove equal all their partial approximations.

So to prove that *primes* does indeed produce the prime numbers, it suffices to prove that

$$\mathit{approx}\ n\ \mathit{primes}\ =\ p_1 : \ldots : p_n : \perp$$

for all n, where p_j is the jth prime (we take $p_1\ =\ 2$—for consistency with TFWH, we count the primes starting from one). Bird therefore defines

$$\mathit{prs}\ n\ =\ \mathit{approx}\ n\ \mathit{primes}$$

and claims that

$$
\begin{aligned}
\mathit{prs}\ n\ &=\ \mathit{approx}\ n\ (\mathit{makeP}\ (\mathit{crs}\ n)) \\
\mathit{crs}\ n\ &=\ \mathit{makeC}\ (\mathit{prs}\ n)
\end{aligned}
$$

To prove the claim, he observes that it suffices for $\mathit{crs}\ n$ to be well defined at least up to the first composite number greater than p_{n-1}, because then $\mathit{crs}\ n$ delivers enough composite numbers to supply $\mathit{prs}\ (n+1)$, which will in turn supply $\mathit{crs}\ (n+1)$, and so on. It is a "non-trivial result in Number Theory" that $p_{n-1} < (p_n)^2$; therefore it suffices that

$$\mathit{crs}\ n\ =\ c_1 : \ldots : c_m : \perp$$

where c_j is the jth composite number (so $c_1\ =\ 4$) and $c_m\ =\ (p_n)^2$. Completing the proof is set as Exercise 9.I of TFWH, and Answer 9.I gives a hint about using induction to show that $\mathit{crs}\ (n+1)$ is the result of merging $\mathit{crs}\ n$ with $\mathit{multiples}\ p_{n+1}$.[†]

[†] Incidentally, there is a typo in TFWH: the body of the chapter, the exercise, and its solution all have "$m\ =\ (p_n)^2$" instead of "$c_m\ =\ (p_n)^2$".

Unfortunately, the hint in Answer 9.I is at best unhelpful. For example, it implies that $crs\ 2$ (which equals $4 : 6 : 8 : 9 : \bot$) could be constructed from $crs\ 1$ (which equals $4 : \bot$) and $multiples\ 3$ (which equals $[9, 12\,..]$); but where do the 6 and 8 come from? Nevertheless, the claim in Exercise 9.I is valid. What should the hint for the proof have been?

3 The Membership Lemma

Bird's program is a dance involving two partners, with the definitions of the lists *primes* and *composites* (and likewise, the functions *prs* and *crs*) depending on each other. However, the two dancers move at different speeds. The first few primes indeed correspond to the first few composites, but each with different numbers of defined elements: $approx\ n\ primes$ corresponds to $approx\ m\ composites$ for some m, but it is hard to work out which m. This means that the Approx Lemma alone is not really sufficient when trying to prove the program correct.

We introduce a new result that is better suited to this problem; in particular, better suited to proving equality between two infinite lists representing infinite sets of naturals, being duplicate-free and strictly increasing.

Define membership of partial or infinite strictly increasing lists as follows:

$$
\begin{array}{l}
elem :: Ord\ a \Rightarrow a \rightarrow [a] \rightarrow Bool \\
elem\ z\ (x : xs)\ \mid z < x\ =\ False \\
\qquad\qquad\qquad\ \mid z == x\ =\ True \\
\qquad\qquad\qquad\ \mid z > x\ =\ elem\ z\ xs
\end{array}
$$

For properly infinite strictly increasing lists with fully defined elements, this is always defined. But for a partial list with defined elements, it is defined only for z at most the last defined element. (For such a list xs, there is a least n such that $xs = approx\ n\ xs$. Then $xs = x_0 : x_1 : \ldots : x_{n-1} : \bot$, and $elem\ z\ xs$ is defined iff $z \leqslant x_{n-1}$.) We will only use *elem* on partial or infinite lists, so we do not need a case for $[\,]$.

Then we have:

Lemma 2 (Membership Lemma)**.** For partial or infinite strictly increasing lists xs, ys over a flat element type,

$$(xs = ys) \Longleftrightarrow (\forall z\,.\ elem\ z\ xs = elem\ z\ ys)$$

Note that the lemma does not hold for unordered or even for weakly increasing lists: it corresponds to set equality, not bag or list equality. Nor does it hold for finite lists; for example, $[\,]$ and $\bot$ agree everywhere on membership (because we have left *elem* undefined on the empty list), but are different. Similarly, it does not hold for partial or infinite lists over non-flat element types; for example, consider $\bot$ and $\bot : \bot$.

Proof. Clearly the implication holds from left to right. For the other direction, suppose $\forall z\,.\ elem\ z\ xs = elem\ z\ ys$. We conduct a case analysis on whether xs is partial or infinite.

Case xs is partial. Let n be the least such that $xs = approx\ n\ xs$, so $xs = x_0 : x_1 : \ldots : x_{n-1} : \bot$.

Subcase $n = 0$. Then $xs = \bot$, so $elem\ z\ xs = \bot$ for any z, so therefore also $elem\ z\ ys = \bot$ for any z, so $ys = \bot = xs$ too.
Subcase $n > 0$. Then

$$\begin{aligned} elem\ z\ xs &= True, \text{ if } z = x_i \text{ for some } 0 \leq i < n \\ &= False, \text{ if } z < x_0, \text{ or } x_{i-1} < z < x_i \text{ for some } 0 < i < n \\ &= \bot, \quad\ \text{ if } z > x_{n-1} \end{aligned}$$

By the premise, $elem\ z\ ys$ satisfies the same properties; that is, $elem\ z\ ys$ is false for $z < x_0$, true for $z = x_0$, false for $x_0 < z < x_1$, and so on up to $z = x_{n-1}$; therefore, $approx\ n\ ys = x_0 : x_1 : \ldots : x_{n-1} : \bot = approx\ n\ xs$. Moreover, we must have $ys\ !!\ n = \bot$ (because otherwise $ys\ !!\ n > x_{n-1}$, and then

$$elem\ (ys\ !!\ n)\ ys = True \neq \bot = elem\ (ys\ !!\ n)\ xs$$

contradicting the premise); therefore $ys = approx\ n\ ys$, and hence $ys = xs$.
Case xs is infinite. Then $xs = x_0 : x_1 : \ldots$. Similarly to the non-empty partial case,

$$\begin{aligned} elem\ z\ xs &= True, \text{ if } z = x_i \text{ for some } 0 \leq i < n \\ &= False, \text{ if } z < x_0, \text{ or } x_{i-1} < z < x_i \text{ for some } 0 < i \end{aligned}$$

But $elem\ z\ ys$ must satisfy the same properties, and therefore $ys = x_0 : x_1 : \ldots = xs$ too.

□

We use Lemma 2 in particular for the proof of Proposition 8, our key result.

4 Proving the Sieve of Eratosthenes correct

Now we can turn to the proof of correctness of Bird's program; in particular, the proof of productivity. Here is the direct specification of the primes and composites:

$$\begin{aligned} primes_{spec} &= filter\ isPrime\ [2\,..] \\ composites_{spec} &= [2\,..] \setminus\setminus primes_{spec} \end{aligned}$$

$$\begin{aligned} divisors\ n &= [d \mid d \leftarrow [2\,..\,n], n \bmod d == 0] \\ isPrime\ n &= (divisors\ n == [n]) \end{aligned}$$

By convention, 1 is considered neither prime nor composite (Sloane, 1999).
We state the following lemma without proof:

Lemma 3 (relating specification and implementation)**.**

$$\begin{aligned} primes_{spec} &= makeP\ composites_{spec} \\ composites_{spec} &= makeC\ primes_{spec} \end{aligned}$$

4.1 Approximations

We will use some lemmas about membership of partial approximations to various components of the primes program. Some are statements about partial lists, and hence equalities between partial expressions. For these, we introduce the form

$$lhs \;=\; g \triangleleft rhs$$

where $\triangleleft$ "guards" a value by a condition:

$$\begin{aligned}&(\triangleleft) :: Bool \to a \to a\\ &g \triangleleft x \mid g \;=\; x\end{aligned}$$

That is, *rhs* may be more defined than *lhs*, but guarding *rhs* by g to yield $g \triangleleft rhs$ makes something precisely equal to *lhs*: either both sides are defined and evaluate to the same result, or both are undefined. We make $\triangleleft$ loose binding for notational convenience—it will mostly be the outermost operator, and then we do not need parentheses around the guard.

Here are two variations on *approx*, using a predicate for termination instead of a count:

$$\begin{aligned}&approxWhile, approxUntil :: (a \to Bool) \to [a] \to [a]\\ &approxWhile\; p\; (x : xs) \;=\; p\, x \triangleleft x : approxWhile\; p\; xs\\ &approxUntil\;\; p\; (x : xs) \;=\; x : (not\; (p\, x) \triangleleft approxUntil\; p\; xs)\end{aligned}$$

That is, $approxWhile\; p\; xs$ gives the longest approximation to xs all of whose elements satisfy p, and $approxUntil\; p\; xs$ gives the shortest approximation to xs containing an element satisfying p. Our lists will be strictly increasing, and we will use an upper bound for *approxWhile* and a lower bound for *approxUntil*; for example,

$$\begin{aligned}&approxWhile\; (\leqslant 5)\; [1, 3\,..] \;=\; 1 : 3 : 5 : \bot\\ &approxWhile\; (\leqslant 6)\; [1, 3\,..] \;=\; 1 : 3 : 5 : \bot\\ &approxUntil\;\; (\geqslant 5)\; [1, 3\,..] \;=\; 1 : 3 : 5 : \bot\\ &approxUntil\;\; (\geqslant 4)\; [1, 3\,..] \;=\; 1 : 3 : 5 : \bot\end{aligned}$$

The two functions are related by the following result:

Lemma 4 (*approxWhile* and *approxUntil*)**.** For partial or infinite xs with $x \in xs$,

$$approxWhile\; (\leqslant x)\; xs \;=\; approxUntil\; (\geqslant x)\; xs$$

(we write "$x \in xs$" when $x \;=\; xs\; !!\; n$ for some n).

4.2 Bertrand's Postulate

Bird's "non-trivial result in Number Theory" is Bertrand's Postulate (Bertrand, 1845), which states that $p_{n+1} < 2\, p_n$ for $n > 0$. As a corollary, $p_{n+1} < (p_n)^2$; this is the key fact that makes Bird's program productive. We encapsulate this in the following proposition:

Proposition 5 (number theory)**.** For $n \geqslant 0$,

$$\begin{aligned}&approx\; (n+1)\; primes_{spec}\\ &\quad = approxWhile\; (\leqslant p_{n+1})\; (makeP\; (approxWhile\; (\leqslant (p_n)^2)\; composites_{spec}))\end{aligned}$$

Proposition 5 rests on the following two lemmas, stated without proof:

Lemma 6 (introducing *approxWhile*)**.** For strictly increasing xs, whether partial or infinite,

$$approx\ (n+1)\ xs \ = \ approxWhile\ (\leqslant (xs\ !!\ n))\ xs$$

provided that xs is defined at least as far as $xs\ !!\ n$ (that is, $xs\ !!\ n \in xs$).

Lemma 7 (*approxWhile* of difference)**.** For partial or infinite, strictly increasing xs, ys with $y \in ys$, $x \in (xs \setminus\!\setminus ys)$, and $x < y$,

$$approxWhile\ (\leqslant x)\ (xs \setminus\!\setminus ys) \ = \ approxWhile\ (\leqslant x)\ (xs \setminus\!\setminus approxWhile\ (\leqslant y)\ ys)$$

Proof of Proposition 5. For $n \geqslant 1$,

$$\begin{array}{ll}
& approx\ (n+1)\ primes_{spec} \\
= & [\![\ \text{Lemma 6, and } primes_{spec}\ !!\ n \ = \ p_{n+1}\ ]\!] \\
& approxWhile\ (\leqslant p_{n+1})\ primes_{spec} \\
= & [\![\ \text{Lemma 3}\ ]\!] \\
& approxWhile\ (\leqslant p_{n+1})\ ([2\,..] \setminus\!\setminus composites_{spec}) \\
= & [\![\ \text{Lemma 7, with } y \ = \ (p_n)^2 > p_{n+1}\ ]\!] \\
& approxWhile\ (\leqslant p_{n+1})\ ([2\,..] \setminus\!\setminus approxWhile\ (\leqslant (p_n)^2)\ composites_{spec}) \\
= & [\![\ \text{2 is not composite}\ ]\!] \\
& approxWhile\ (\leqslant p_{n+1})\ (2 : ([3\,..] \setminus\!\setminus approxWhile\ (\leqslant (p_n)^2)\ composites_{spec})) \\
= & [\![\ \text{definition of } makeP\ ]\!] \\
& approxWhile\ (\leqslant p_{n+1})\ (makeP\ (approxWhile\ (\leqslant (p_n)^2)\ composites_{spec}))
\end{array}$$

The above application of Lemma 7 is not valid when $n \ = \ 0$, because p_0 is undefined, and hence so too is the set difference; nevertheless, the overall proposition

$$\begin{array}{l}
approx\ 1\ primes_{spec} \\
\quad = approxWhile\ (\leqslant 2)\ (makeP\ (approxWhile\ (\leqslant \bot^2)\ composites_{spec}))
\end{array}$$

still holds, both sides being equal to $2 : \bot$. □

4.3 Approximating primes and composites

We prove the following result:

Proposition 8 (approximations)**.** For all n,

$$\begin{array}{lcl}
approx\ n\ primes & = & approx\ n\ primes_{spec} \\
approxWhile\ (\leqslant (p_n)^2)\ composites & = & approxWhile\ (\leqslant (p_n)^2)\ composites_{spec}
\end{array}$$

The proof is in Section 4.5. Then:

Theorem 9 (the *primes* program is correct)**.**

$$primes \ = \ primes_{spec}$$

Proof. A direct corollary of Proposition 8, by Lemma 1. □

4.4 Subsidiary lemmas

We collect here two lemmas needed for the proof of Proposition 8, which are themselves not specifically about primes.

Lemma 10 (*mergeAll* and *approx*)**.** For $n \geqslant 0$ and partial or infinite list xss of properly infinite lists, such that xss is defined at least as far as $xss\ !!\ n$,

$$mergeAll\ (approx\ (n+1)\ xss)\ =\ approxUntil\ ({\geqslant}\ head\ (xss\ !!\ n))\ (mergeAll\ xss)$$

Proof. By induction on n.

Base case. For $n\ =\ 0$, we have

$$\begin{array}{ll} & mergeAll\ (approx\ (n+1)\ ((x : xs) : xss)) \\ = & [[\ \text{definition of } approx\ ]] \\ & mergeAll\ ((x : xs) : \bot) \\ = & [[\ \text{definition of } mergeAll, xmerge\ ]] \\ & x : merge\ xs\ (mergeAll\ \bot) \\ = & [[\ \text{definition of } mergeAll, merge\ ]] \\ & x : \bot \\ = & [[\ \text{definition of } approxUntil\ ]] \\ & approxUntil\ ({\geqslant}\, x)\ (x : merge\ xs\ (mergeAll\ xss)) \\ = & [[\ \text{definition of } mergeAll, xmerge\ ]] \\ & approxUntil\ ({\geqslant}\, x)\ (mergeAll\ ((x : xs) : xss)) \end{array}$$

Inductive step. Let $n \geqslant 0$ and $b\ =\ head\ (xss\ !!\ n)$, and assume as inductive hypothesis that

$$mergeAll\ (approx\ (n+1)\ xss)\ =\ approxUntil\ ({\geqslant}\, b)\ (mergeAll\ xss)$$

Then we have

$$\begin{array}{ll} & mergeAll\ (approx\ (n+2)\ ((x : xs) : xss)) \\ = & [[\ \text{definition of } approx\ ]] \\ & mergeAll\ ((x : xs) : approx\ (n+1)\ xss) \\ = & [[\ \text{definition of } mergeAll, xmerge\ ]] \\ & x : merge\ xs\ (mergeAll\ (approx\ (n+1)\ xss)) \\ = & [[\ \text{inductive hypothesis}\ ]] \\ & x : merge\ xs\ (approxUntil\ ({\geqslant}\, b)\ (mergeAll\ xss)) \\ = & [[\ merge \text{ and } approxUntil \text{ (see below)}\ ]] \\ & x : approxUntil\ ({\geqslant}\, b)\ (merge\ xs\ (mergeAll\ xss)) \\ = & [[\ x < head\ (xss\ !!\ n)\ =\ b\ ]] \\ & approxUntil\ ({\geqslant}\, b)\ (x : merge\ xs\ (mergeAll\ xss)) \\ = & [[\ \text{definition of } mergeAll, xmerge\ ]] \\ & approxUntil\ ({\geqslant}\, b)\ (mergeAll\ ((x : xs) : xss)) \end{array}$$

The hint about *merge* and *approxUntil* is that

$$approxUntil\ ({\geqslant}\, b)\ (merge\ xs\ ys)\ =\ merge\ xs\ (approxUntil\ ({\geqslant}\, b)\ ys)$$

for infinite xs, ys with b an element of ys, which follows from the fact that *merge* becomes undefined as soon as either argument does.

$\square$

Lemma 11 (membership of *approxWhile*)**.** For partial or infinite list xs with $y \in xs$,

$$\mathit{elem}\; z\; (\mathit{approxWhile}\; ({\leqslant}\, y)\; xs) \;=\; z \leqslant y \triangleleft \mathit{elem}\; z\; xs$$

Proof. Let n be such that $y = xs\; !!\; n$; then

$$\mathit{approxWhile}\; ({\leqslant}\, (xs\; !!\; n))\; xs \;=\; \mathit{approx}\; (n+1)\; xs$$

from which the result follows. $\square$

4.5 Completing the proof

Proof of Proposition 8. By induction on n.

Base case. When $n = 0$, both equations trivially hold, because $\mathit{approx}\; 0$ and p_0 are undefined. When $n = 1$, both equations hold by inspection.

Inductive step. We now consider the case $n + 1$ with $n > 0$. Assume the inductive hypothesis

$$\begin{aligned}
\mathit{approx}\; n\; \mathit{primes} &= \mathit{approx}\; n\; \mathit{primes}_{spec} \\
\mathit{approxWhile}\; ({\leqslant}\, (p_n)^2)\; \mathit{composites} &= \mathit{approxWhile}\; ({\leqslant}\, (p_n)^2)\; \mathit{composites}_{spec}
\end{aligned}$$

Note that the second equation implies that *composites* is defined at least as far as $(p_n)^2$. Therefore, by Proposition 5, also $\mathit{makeP}\; (\mathit{approxWhile}\; ({\leqslant}\, (p_n)^2)\; \mathit{composites})$ is defined at least as far as p_{n+1}; we refer to this fact as “p_{n+1} is present” in hints below. Then we have:

$$\begin{aligned}
& \mathit{elem}\; z\; (\mathit{approx}\; (n+1)\; \mathit{primes}_{spec}) \\
=\;& [\![\; \text{Proposition 5} \;]\!] \\
& \mathit{elem}\; z\; (\mathit{approxWhile}\; ({\leqslant}\, p_{n+1})\; (\mathit{makeP}\; (\mathit{approxWhile}\; ({\leqslant}\, (p_n)^2)\; \mathit{composites}_{spec}))) \\
=\;& [\![\; \text{Lemma 11, since } p_{n+1} \text{ is present} \;]\!] \\
& z \leqslant p_{n+1} \triangleleft \mathit{elem}\; z\; (\mathit{makeP}\; (\mathit{approxWhile}\; ({\leqslant}\, (p_n)^2)\; \mathit{composites}_{spec})) \\
=\;& [\![\; \text{inductive hypothesis} \;]\!] \\
& z \leqslant p_{n+1} \triangleleft \mathit{elem}\; z\; (\mathit{makeP}\; (\mathit{approxWhile}\; ({\leqslant}\, (p_n)^2)\; \mathit{composites})) \\
=\;& [\![\; \text{Lemma 11, since } p_{n+1} \text{ is present} \;]\!] \\
& \mathit{elem}\; z\; (\mathit{approxWhile}\; ({\leqslant}\, p_{n+1})\; (\mathit{makeP}\; (\mathit{approxWhile}\; ({\leqslant}\, (p_n)^2)\; \mathit{composites}))) \\
=\;& [\![\; \text{definition of } \mathit{makeP}\text{; see } (*) \text{ below} \;]\!] \\
& \mathit{elem}\; z\; (\mathit{approxWhile}\; ({\leqslant}\, p_{n+1})\; ([2\,..]\; \backslash\backslash\; \mathit{approxWhile}\; ({\leqslant}\, (p_n)^2)\; \mathit{composites})) \\
=\;& [\![\; \text{Lemma 7} \;]\!] \\
& \mathit{elem}\; z\; (\mathit{approxWhile}\; ({\leqslant}\, p_{n+1})\; ([2\,..]\; \backslash\backslash\; \mathit{composites})) \\
=\;& [\![\; \text{definition of } \mathit{makeP}\text{; see } (*) \text{ below} \;]\!] \\
& \mathit{elem}\; z\; (\mathit{approxWhile}\; ({\leqslant}\, p_{n+1})\; (\mathit{makeP}\; \mathit{composites})) \\
=\;& [\![\; \text{definition of } \mathit{primes} \;]\!] \\
& \mathit{elem}\; z\; (\mathit{approxWhile}\; ({\leqslant}\, p_{n+1})\; \mathit{primes})
\end{aligned}$$

$$\begin{aligned}
&= \quad [\![\ \text{Lemma 6, since } p_{n+1} \text{ is present}\ ]\!] \\
&\quad \mathit{elem}\ z\ (\mathit{approx}\ (n+1)\ \mathit{primes})
\end{aligned}$$

(For the two steps marked $(*)$, we switch freely between $\mathit{makeP}\ cs = 2 : ([3\,..] \setminus\!\setminus cs)$ and $[2\,..] \setminus\!\setminus cs$ for different values of cs; this is sound, because in both cases cs is defined at least as far as its head, namely 4.) Then by the Membership Lemma (Lemma 2),

$$\mathit{approx}\ (n+1)\ \mathit{primes} = \mathit{approx}\ (n+1)\ \mathit{primes}_{spec}$$

which deals with the first equation. Note that therefore *primes* is defined at least as far as p_{n+1}. For the second equation, let $b = (p_{n+1})^2$, so that

$$b = \mathit{head}\ (\mathit{map\ multiples\ primes}_{spec}\ !!\ n) = \mathit{head}\ (\mathit{map\ multiples\ primes}\ !!\ n)$$

Then

$$\begin{aligned}
&\quad \mathit{approxUntil}\ (\geqslant b)\ \mathit{composites} \\
&= \quad [\![\ \text{definition of } \mathit{composites}\ ]\!] \\
&\quad \mathit{approxUntil}\ (\geqslant b)\ (\mathit{makeC\ primes}) \\
&= \quad [\![\ \text{definition of } \mathit{makeC}\ ]\!] \\
&\quad \mathit{approxUntil}\ (\geqslant b)\ (\mathit{mergeAll}\ (\mathit{map\ multiples\ primes})) \\
&= \quad [\![\ \text{Lemma 10, given that } \mathit{primes} \text{ is defined at least as far as } p_{n+1}\ ]\!] \\
&\quad \mathit{mergeAll}\ (\mathit{approx}\ (n+1)\ (\mathit{map\ multiples\ primes})) \\
&= \quad [\![\ \text{naturality of } \mathit{approx}\ ]\!] \\
&\quad \mathit{mergeAll}\ (\mathit{map\ multiples}\ (\mathit{approx}\ (n+1)\ \mathit{primes})) \\
&= \quad [\![\ \text{above}\ ]\!] \\
&\quad \mathit{mergeAll}\ (\mathit{map\ multiples}\ (\mathit{approx}\ (n+1)\ \mathit{primes}_{spec})) \\
&= \quad [\![\ \text{naturality of } \mathit{approx}\ ]\!] \\
&\quad \mathit{mergeAll}\ (\mathit{approx}\ (n+1)\ (\mathit{map\ multiples\ primes}_{spec})) \\
&= \quad [\![\ \text{Lemma 10}\ ]\!] \\
&\quad \mathit{approxUntil}\ (\geqslant b)\ (\mathit{mergeAll}\ (\mathit{map\ multiples\ primes}_{spec})) \\
&= \quad [\![\ \text{definition of } \mathit{makeC}\ ]\!] \\
&\quad \mathit{approxUntil}\ (\geqslant b)\ (\mathit{makeC\ primes}_{spec}) \\
&= \quad [\![\ \text{Lemma 3}\ ]\!] \\
&\quad \mathit{approxUntil}\ (\geqslant b)\ \mathit{composites}_{spec}
\end{aligned}$$

Moreover, b is in $\mathit{composites}_{spec}$, so also in *composites*; therefore also

$$\mathit{approxWhile}\ (\leqslant b)\ \mathit{composites} = \mathit{approxWhile}\ (\leqslant b)\ \mathit{composites}_{spec}$$

by Lemma 4, as required.

□

This completes the proof of Proposition 8, and hence of Theorem 9:

$$\mathit{primes} = \mathit{primes}_{spec}$$

5 Conclusion

Total Functional Programming: David Turner's ambition (Turner, 2004) was for languages "designed to exclude the possibility of non-termination". He observed that most programs are already structurally recursive or corecursive, therefore guaranteed respectively terminating or productive, and conjectured that "with more practice we will find this is always true". He explicitly admits that "rewriting the well known sieve of Eratosthenes [by which he means trial division] program in this discipline involves coding in some bound on the distance from one prime to the next". We have coded that bound by appeal to Bertrand's Postulate (Proposition 5)—but Turner's vision would require that appeal at least to be acknowledged by the totality checker. One could go as far as full dependent types, in which case the relevant assumption can be formally expressed as a theorem—but still, one would either have to prove the theorem (a decidedly non-trivial matter) or accept it as an unverified axiom; Turner said that he was "interested in finding something simpler". Much as I find the idea of total functional programming appealing, I fear that we are still some way off, even after 20 years of "more practice". But I would love to be shown to be unnecessarily pessimistic.

Trial division: Turner popularized the trial division algorithm in various publications; I believe his first publications of it is in the SASL Manual. Interestingly, SASL changed from eager semantics (Turner, 1975) to lazy semantics (Turner, 1976); the primes program appears only in the later of those two documents, despite them both having the same technical report number. Turner (2020) notes that the program appeared in Kahn and MacQueen (1977):

> *Did I see a preprint of that in 1976? I don't recall but it's possible, in which case my contribution was to express the idea using recursion and lazy lists.*

Kahn and MacQueen (1977) in turn credit it to McIlroy (1968). McIlroy (2014) records:

> *For examples in a talk at the Cambridge Computing Laboratory (1968) I cooked up some interesting coroutine-based programs. One, a prime-number sieve, became a classic, spread by word of mouth.*

Turner (1976) and Kahn and MacQueen (1977) call the trial division algorithm "The Sieve of Eratosthenes", but McIlroy (1968, 2014) does not.

Proofs about infinite lists: Our Membership Lemma (Lemma 2) is applicable to partial or infinite strictly increasing lists over any totally ordered flat element type; but not for non-flat element types, unordered lists or lists with duplicates, or (as observed above) for finite lists. We also considered an ApproxWhile Lemma, more closely analogous to the Approx Lemma (Lemma 1):

Lemma (ApproxWhile Lemma)**.** For infinite sequence $b_0 < b_1 < \cdots$ of integer bounds, and two lists xs, ys of integers, whether finite, partial, or infinite,

$$(xs = ys) \Longleftrightarrow (\forall i \,.\, approxWhile\ (\leqslant b_i)\ xs = approxWhile\ (\leqslant b_i)\ ys)$$

But this is more restrictive than the Membership Lemma: the bounds must grow without bound, so it doesn't hold universally for rationals, or pairs, or strings. Moreover, it did not seem very helpful in proving the primes program correct.

Bird's exercise: What of Bird (2014)? This paper was prompted by a series of ten emails (Lieberich, 2018) pointing out errors in TFWH, including this particular error. Recall that Bird's hint towards the proof implies that $\mathit{crs}\ 2 = 4 : 6 : 8 : 9 : \bot$ can be obtained by merging $\mathit{crs}\ 1 = 4 : \bot$ and $\mathit{multiples}\ 3 = [9, 12 \,..]$. In fact, a more helpful hint that Bird could have given is that $\mathit{crs}\ 2$ can be constructed from $\mathit{crs}\ 1$ alone, without needing $\mathit{multiples}\ 3$ at all: $\mathit{crs}\ 2 = \mathit{makeC}\ (\mathit{makeP}\ (\mathit{crs}\ 1))$. This doesn't quite work for higher values, because the right-hand side is *too* productive: $\mathit{makeC}\ (\mathit{makeP}\ (\mathit{crs}\ 2))$ yields the composites up to 49, whereas $\mathit{crs}\ 3$ needs composites only up to $(p_3)^2 = 25$. But the general answer is

$$\mathit{crs}\ (n+1) = \mathit{makeC}\ (\mathit{approx}\ (n+1)\ (\mathit{makeP}\ (\mathit{crs}\ n)))$$

Nevertheless, the proof of that claim is neither short nor simple, so perhaps this is not an appropriate correction for TFWH.

Acknowledgements

I am grateful for helpful comments on this work throughout its gestation, from members of the Algebra of Programming research group at Oxford (especially Geraint Jones, Guillaume Boisseau, and Zhixuan Yang) and IFIP Working Group 2.1 (especially Tom Schrijvers). Thanks are due to Francisco Lieberich for alerting me to the error in TFWH, among many others. And of course this couldn't have happened without the seminal contributions of David Turner and Richard Bird.

Conflicts of interest

None.

References

Bertrand, J. (1845) Mémoire sur le nombre de valeurs que peut prendre une fonction quand on y permute les lettres qu'elle renferme. *Journal de l'École Royale Polytechnique*. **18** (Cahier 30), 123–140. In French; see also `https://en.wikipedia.org/wiki/Bertrand's_postulate`.

Bird, R. (2014) *Thinking Functionally with Haskell*. Cambridge University Press. `https://www.cs.ox.ac.uk/publications/books/functional/`.

Gibbons, J. & Jones, G. (1998) The under-appreciated unfold. International Conference on Functional Programming. Baltimore, Maryland. pp. 273–279.

Kahn, G. & MacQueen, D. B. (1977) Coroutines and networks of parallel processes. IFIP Congress. IFIP. pp. 993–998.

Lieberich, F. (2018) "Errata". Personal communication (email).

McIlroy, M. D. (1968) Coroutines. Internal report. Bell Telephone Laboratories. Murray Hill, New Jersey. `http://www.iq0.com/notes/coroutine.html`.

McIlroy, M. D. (2014) Coroutine prime number sieve. `https://www.cs.dartmouth.edu/~doug/sieve/sieve.pdf`.

Meertens, L. (2004) Calculating the Sieve of Eratosthenes. *Journal of Functional Programming*.

14(6).

O'Neill, M. E. (2009) The genuine Sieve of Eratosthenes. *Journal of Functional Programming*. **19**(1), 95–105.

Sloane, N. (1999) The composite numbers. In *The On-Line Encyclopedia of Integer Sequences*. https://oeis.org/A002808.

Turner, D. A. (1975) SASL language manual. Technical Report CS/75/1. University of St Andrews, Dept of Computational Science. Revised 16/9/75.

Turner, D. A. (1976) SASL language manual. Technical Report CS/75/1. University of St Andrews, Dept of Computational Science. Revised 1/12/76.

Turner, D. A. (1982) Recursion equations as a programming language. In *Functional Programming and its Applications*, Darlington, J., Henderson, P., & Turner, D. A. (eds). Cambridge University Press. pp. 1–28.

Turner, D. A. (2004) Total functional programming. *Journal of Universal Computer Science*. **10**(7), 751–768.

Turner, D. A. (2020) "SASL manual". Personal communication (email).