JFP **35**, e5, 13 pages, 2025.
doi:10.1017/S0956796824000194

FUNCTIONAL PEARL

Turner, Bird, Eratosthenes: An eternal burning thread

JEREMY GIBBONS
Department of Computer Science, University of Oxford, Oxford, UK
(*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 inspirational now. In particular, Turner was a strong advocate of lazy evaluation and of list comprehensions. As an illustration of these techniques, he popularized 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", Melissa O'Neill argued that Turner's program 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. In particular, why is this circular program productive? Unfortunately, Bird's 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; he conjectured that "with more practice we will find this is always true". We explore Bird's circular Sieve of Eratosthenes as a challenge problem for Turner's Total Functional Programming.

1 Introduction

The late David Turner had great taste in language design and programming. In particular, he was a strong advocate for lazy evaluation and list comprehensions. One example program that he introduced in order to illustrate these techniques (Turner, 1976, 1982) is a one-line recursive "sieve" to generate the infinite list of prime numbers:

$$\begin{aligned}&\mathit{primes} :: [\mathit{Integer}]\\&\mathit{primes} = \mathit{sieve}\,[2\,..]\ \textbf{where}\ \mathit{sieve}\,(p:\mathit{xs}) = p:\mathit{sieve}\,[x \mid x \leftarrow \mathit{xs}, x \bmod p \not= 0]\end{aligned}$$

That is, *sieve* takes a stream of candidate primes, initially the “plural” naturals (those greater than one); the head p of this stream is a prime, and the subsequent primes are obtained by removing all multiples of p from the candidates and sieving what remains. It’s also a nice unfold (Gibbons & 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 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, at most every other natural number should be tested 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 moving on to 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 functional pearl 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 as an exercise its proof of correctness, specifically productivity. Unfortunately, Bird’s 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; he conjectured that “with more practice we will find this is always true”. But it seems that it is not always so easy. In particular, Bird’s circular Sieve of Eratosthenes is apparently productive; but it is not clear how it might fit within Turner’s vision for Total Functional Programming. In this paper, we take on this challenge. What should Bird’s proof hint have said?

2 The Genuine Sieve, using lists

Bird’s program appears in Section 9.2 of his book (Bird, 2014), henceforth “TFWH”. It deals with lists, but in this paper these will be infinite, sorted, duplicate-free streams, 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 as well).

[*] JFP doesn’t list O’Neill’s paper as a Pearl, but this was a production error (Tranah, 2024).

The prime numbers are what you get by eliminating the composite numbers from the plural naturals, and the composite numbers are the proper multiples of the primes. So the program is cleverly circular:

```
primes, composites :: [Integer]
primes     = makeP composites
composites = makeC primes
```

where

```
makeP, makeC :: [Integer] → [Integer]
makeP cs = 2 : ([3 . .] \\ cs)
makeC ps = mergeAll (map multiples ps)
```

For later convenience, we have slightly refactored the program as presented by Bird, naming the components *makeP* and *makeC*. In particular, *primes* is a fixpoint of *makeP* · *makeC*.

Here, (\\) is the obvious implementation of list difference of strictly increasing streams, hence representing set difference:

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

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

```
multiples p = [p × p, p × p + p . .]
```

Thus, the composites are obtained by merging together the infinite stream of infinite streams [[4, 6 . .], [9, 12 . .], [25, 30 . .], . . .]. You might think that you could have defined instead *makeP cs* = [2 . .] \\ *cs*, 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 using this in the definition of *primes* would be unproductive. Somewhat surprisingly, it suffices to "prime the pump" just with 2; everything else flows freely from there.

Now for *mergeAll*, which represents the union of a set of sets. Here is the obvious implementation of *merge*, which merges two strictly increasing streams into one, hence representing set union:

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

Then *mergeAll* is basically a stream fold with *merge*. You might think you could define this simply by *mergeAll* (*xs* : *xss*) = *merge xs* (*mergeAll xss*), but again this would be unproductive. After all, you can't merge the infinite stream of sorted streams

$[[5,6\,..], [4,5\,..], [3,4\,..], \dots]$ into a single sorted stream: 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 needing to examine the right stream. So, we define:

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

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

Incidentally, although it is faithful, O'Neill (2009) points out that this program is not asymptotically optimal. After all, it is basically simulating her priority queues using nothing but ordered lists, incurring a logarithmic penalty. But that has no bearing on this paper.

3 The Approx Lemma

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

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

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 truncates anything deeper and replaces it with $\perp$ (the undefined value), returning a partial list if the input was longer. That is, the lemma states that two lists are equal if (and of course only if) all their partial approximations agree.

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

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

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

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

and claims that

$$\mathit{prs}\ n = \mathit{approx}\ n\,(\mathit{makeP}\,(\mathit{crs}\ n))$$

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

$$crs\,n = c_1 : \ldots : c_m : \bot$$

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 $crs\,(n+1)$ is the result of merging $crs\,n$ with $multiples\,p_{n+1}$.[†]

Unfortunately, the hint in Answer 9.I is at best unhelpful. For instance, 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: the program is productive. What should the hint for the proof have been?

4 Proving the Sieve of Eratosthenes correct

Here is a direct and non-recursive specification of the primes and composites:

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

By convention, 1 is considered neither prime nor composite (Sloane, 1999).

The interesting question is not really the elements of the lists, but productivity—that is, not so much showing that $primes_{spec}$ is *a* fixed point of $makeP \cdot makeC$, but that it is the *least* fixed point. So we state the following lemma without proof:

Lemma 2 (relating specification and implementation)**.**

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

We will prove that $primes = primes_{spec}$.

4.1 Approximations

We will need two variations on *approx*, using a predicate instead of a count for termination:

```
$approxWhile, approxUntil :: (a \to Bool) \to [a] \to [a]$
$approxWhile\ p\ (x:xs)$ | $p\ x$       $= x : approxWhile\ p\ xs$
                       | $otherwise$ $= \bot$
$approxUntil\ \ p\ (x:xs)$ | $p\ x$       $= x : \bot$
                       | $otherwise$ $= x : approxUntil\ p\ xs$
```

[†] 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 "$c_m = (p_n)^2$".

In words, $\mathit{approxWhile}\ p\ \mathit{xs}$ gives the longest approximation to xs all of whose elements satisfy p, and $\mathit{approxUntil}\ p\ \mathit{xs}$ gives the shortest approximation to xs containing an element satisfying p (or xs itself, if no element satisfies p). That is, $\mathit{approxUntil}\ p$ stops with and includes the first element that satisfies p, whereas $\mathit{approxWhile}\ p$ stops with and excludes the first element that fails to satisfy p. Our lists will be strictly increasing, and we will use an upper bound for *approxWhile* and a lower bound for *approxUntil*; for example,

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

For integer x, we write "$x \in \mathit{xs}$" when $x = \mathit{xs}\,!!\,n$ for some n, and say then that "xs is defined at least as far as x".

4.2 Properties of approximation

The two functions *approxWhile* and *approx* are related by:

Lemma 3 (introducing *approxWhile*)**.** For strictly increasing, partial or infinite xs,

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

provided that xs is defined at least as far as $\mathit{xs}\,!!\,n$.

Moreover, *approxWhile* and *approxUntil* are related by:

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

$$\mathit{approxWhile}\ (\leqslant x)\ \mathit{xs} = \mathit{approxUntil}\ (\geqslant x)\ \mathit{xs}$$

and *approxWhile* and set difference by:

Lemma 5 (*approxWhile* of difference)**.** For strictly increasing, partial or infinite xs, ys with $y \in \mathit{ys}$, $x \in (\mathit{xs} \setminus\setminus \mathit{ys})$, and $x < y$,

$$\mathit{approxWhile}\ (\leqslant x)\ (\mathit{xs} \setminus\setminus \mathit{ys}) = \mathit{approxWhile}\ (\leqslant x)\ (\mathit{xs} \setminus\setminus \mathit{approxWhile}\ (\leqslant y)\ \mathit{ys})$$

and *mergeAll* and *approx* by:

Lemma 6 (*mergeAll* and *approx*)**.** For $n \geqslant 0$ and partial or infinite strictly increasing list xss of properly infinite, strictly increasing lists, defined at least as far as $\mathit{xss}\,!!\,n$,

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

Proof. By induction on n.

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

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

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

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

Note the following property of *merge* and *approxUntil*:

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

for infinite xs, ys with $b \in \mathit{ys}$, since *merge* becomes undefined as soon as either argument does. Then we have

$$\begin{array}{l}
\quad \mathit{mergeAll}\,(\mathit{approx}\,(n+2)\,((x:\mathit{xs}):\mathit{xss})) \\
= \quad [\![\ \text{definition of } \mathit{approx} \ ]\!] \\
\quad \mathit{mergeAll}\,((x:\mathit{xs}):\mathit{approx}\,(n+1)\,\mathit{xss}) \\
= \quad [\![\ \text{definition of } \mathit{mergeAll}, \mathit{xmerge} \ ]\!] \\
\quad x:\mathit{merge}\,\mathit{xs}\,(\mathit{mergeAll}\,(\mathit{approx}\,(n+1)\,\mathit{xss})) \\
= \quad [\![\ \text{inductive hypothesis} \ ]\!] \\
\quad x:\mathit{merge}\,\mathit{xs}\,(\mathit{approxUntil}\,(\geqslant b)\,(\mathit{mergeAll}\,\mathit{xss})) \\
= \quad [\![\ \text{observation above about } \mathit{merge} \text{ and } \mathit{approxUntil} \ ]\!] \\
\quad x:\mathit{approxUntil}\,(\geqslant b)\,(\mathit{merge}\,\mathit{xs}\,(\mathit{mergeAll}\,\mathit{xss})) \\
= \quad [\![\ \text{definition of } \mathit{approxUntil}, \text{ since } x < \mathit{head}\,(\mathit{xss}\,!!\,n) = b \ ]\!] \\
\quad \mathit{approxUntil}\,(\geqslant b)\,(x:\mathit{merge}\,\mathit{xs}\,(\mathit{mergeAll}\,\mathit{xss})) \\
= \quad [\![\ \text{definition of } \mathit{mergeAll}, \mathit{xmerge} \ ]\!] \\
\quad \mathit{approxUntil}\,(\geqslant b)\,(\mathit{mergeAll}\,((x:\mathit{xs}):\mathit{xss}))
\end{array}$$

□

4.3 Bertrand's Postulate

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

Proposition 7 (number theory)**.** For $n \geqslant 0$,

$$\begin{aligned}&\mathit{approx}\,(n+1)\,\mathit{primes}_{\mathit{spec}}\\&\quad = \mathit{approxWhile}\,(\leqslant p_{n+1})\,(\mathit{makeP}\,(\mathit{approxWhile}\,(\leqslant (p_n)^2)\,\mathit{composites}_{\mathit{spec}}))\end{aligned}$$

Informally, truncating the composites at $(p_n)^2$ provides enough input for *makeP* to generate the primes at least as far as p_{n+1}.

Proof of Proposition 7. For $n \geqslant 1$,

$$\begin{aligned}&\mathit{approx}\,(n+1)\,\mathit{primes}_{\mathit{spec}}\\=\;&[\![\ \text{Lemma 3, and } \mathit{primes}_{\mathit{spec}}\,!!\,n = p_{n+1}\ ]\!]\\&\mathit{approxWhile}\,(\leqslant p_{n+1})\,\mathit{primes}_{\mathit{spec}}\\=\;&[\![\ \text{Lemma 2}\ ]\!]\\&\mathit{approxWhile}\,(\leqslant p_{n+1})\,([2\,..]\setminus\!\setminus \mathit{composites}_{\mathit{spec}})\\=\;&[\![\ \text{Lemma 5, with } y = (p_n)^2 > p_{n+1} \text{ by Bertrand's Postulate}\ ]\!]\\&\mathit{approxWhile}\,(\leqslant p_{n+1})\,([2\,..]\setminus\!\setminus \mathit{approxWhile}\,(\leqslant (p_n)^2)\,\mathit{composites}_{\mathit{spec}})\\=\;&[\![\ \text{2 is not composite}\ ]\!]\\&\mathit{approxWhile}\,(\leqslant p_{n+1})\,(2 : ([3\,..]\setminus\!\setminus \mathit{approxWhile}\,(\leqslant (p_n)^2)\,\mathit{composites}_{\mathit{spec}}))\\=\;&[\![\ \text{definition of } \mathit{makeP}\ ]\!]\\&\mathit{approxWhile}\,(\leqslant p_{n+1})\,(\mathit{makeP}\,(\mathit{approxWhile}\,(\leqslant (p_n)^2)\,\mathit{composites}_{\mathit{spec}}))\end{aligned}$$

The step invoking Lemma 5 is not valid when $n = 0$, because p_0 is undefined, and hence so too is the set difference. Nevertheless, the overall proposition

$$\begin{aligned}&\mathit{approx}\,1\,\mathit{primes}_{\mathit{spec}}\\&\quad = \mathit{approxWhile}\,(\leqslant 2)\,(\mathit{makeP}\,(\mathit{approxWhile}\,(\leqslant \bot^2)\,\mathit{composites}_{\mathit{spec}}))\end{aligned}$$

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

4.4 Completing the proof

We prove the following result:

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

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

Proof. 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

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

Note that the second equation implies that *composites* is defined at least as far as $(p_n)^2$. Therefore, by Proposition 7, also *makeP* (*approxWhile* $(\leqslant (p_n)^2)$ *composites*) is defined at least as far as p_{n+1}. We refer to these facts as "$(p_n)^2$ is present in *composites*" and "p_{n+1} is present in *primes*" below. Then we have:

$$
\begin{array}{ll}
& \mathit{approx}\,(n+1)\,\mathit{primes}_{\mathit{spec}} \\
= & [\![\ \text{Proposition 7 (Bertrand's Postulate)}\ ]\!] \\
& \mathit{approxWhile}\,(\leqslant p_{n+1})\,(\mathit{makeP}\,(\mathit{approxWhile}\,(\leqslant (p_n)^2)\,\mathit{composites}_{\mathit{spec}})) \\
= & [\![\ \text{inductive hypothesis}\ ]\!] \\
& \mathit{approxWhile}\,(\leqslant p_{n+1})\,(\mathit{makeP}\,(\mathit{approxWhile}\,(\leqslant (p_n)^2)\,\mathit{composites})) \\
= & [\![\ \text{definition of } \mathit{makeP}\text{; see } (*) \text{ below}\ ]\!] \\
& \mathit{approxWhile}\,(\leqslant p_{n+1})\,([2\,..]\ \backslash\backslash\ \mathit{approxWhile}\,(\leqslant (p_n)^2)\,\mathit{composites}) \\
= & [\![\ \text{Lemma 5, since } (p_n)^2 \text{ is present in } \mathit{composites}\ ]\!] \\
& \mathit{approxWhile}\,(\leqslant p_{n+1})\,([2\,..]\ \backslash\backslash\ \mathit{composites}) \\
= & [\![\ \text{definition of } \mathit{makeP}\text{; see } (*) \text{ below}\ ]\!] \\
& \mathit{approxWhile}\,(\leqslant p_{n+1})\,(\mathit{makeP}\ \mathit{composites}) \\
= & [\![\ \text{definition of } \mathit{primes}\ ]\!] \\
& \mathit{approxWhile}\,(\leqslant p_{n+1})\,\mathit{primes} \\
= & [\![\ \text{Lemma 3, since } p_{n+1} \text{ is present in } \mathit{primes}\ ]\!] \\
& \mathit{approx}\,(n+1)\,\mathit{primes}
\end{array}
$$

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

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

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

Then

$$
\begin{array}{ll}
& \mathit{approxUntil}\,(\geqslant b)\,\mathit{composites} \\
= & [\![\ \text{definition of } \mathit{composites}\ ]\!] \\
& \mathit{approxUntil}\,(\geqslant b)\,(\mathit{makeC}\ \mathit{primes}) \\
= & [\![\ \text{definition of } \mathit{makeC}\ ]\!] \\
& \mathit{approxUntil}\,(\geqslant b)\,(\mathit{mergeAll}\,(\mathit{map}\ \mathit{multiples}\ \mathit{primes})) \\
= & [\![\ \text{Lemma 6, given that } \mathit{primes} \text{ is defined at least as far as } p_{n+1}\ ]\!] \\
& \mathit{mergeAll}\,(\mathit{approx}\,(n+1)\,(\mathit{map}\ \mathit{multiples}\ \mathit{primes})) \\
= & [\![\ \text{naturality of } \mathit{approx}\ ]\!] \\
& \mathit{mergeAll}\,(\mathit{map}\ \mathit{multiples}\,(\mathit{approx}\,(n+1)\,\mathit{primes})) \\
= & [\![\ \text{first equation}\ ]\!] \\
& \mathit{mergeAll}\,(\mathit{map}\ \mathit{multiples}\,(\mathit{approx}\,(n+1)\,\mathit{primes}_{\mathit{spec}})) \\
= & [\![\ \text{naturality of } \mathit{approx}\ ]\!] \\
& \mathit{mergeAll}\,(\mathit{approx}\,(n+1)\,(\mathit{map}\ \mathit{multiples}\ \mathit{primes}_{\mathit{spec}})) \\
= & [\![\ \text{Lemma 6}\ ]\!] \\
& \mathit{approxUntil}\,(\geqslant b)\,(\mathit{mergeAll}\,(\mathit{map}\ \mathit{multiples}\ \mathit{primes}_{\mathit{spec}}))
\end{array}
$$

$$\begin{array}{l} = \quad [\![\ \text{definition of } \mathit{makeC} \ ]\!] \\ \quad \mathit{approxUntil}\ (\geqslant b)\ (\mathit{makeC}\ \mathit{primes}_{\mathit{spec}}) \\ = \quad [\![\ \text{Lemma 2} \ ]\!] \\ \quad \mathit{approxUntil}\ (\geqslant b)\ \mathit{composites}_{\mathit{spec}} \end{array}$$

In particular, b is in *composites*; therefore also

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

by Lemma 4, dealing with the second equation too. □

Finally, we have:

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

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

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

This completes the proof of correctness of Bird's program.

5 Conclusion

Total Functional Programming: As discussed in the introduction, David Turner's ambition (Turner, 2004) was for future programming languages that were "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 in that paper that "rewriting the well known sieve of Eratosthenes program [by which he means trial division] 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 a weakening of Bertrand's Postulate (Proposition 7)—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 (Théry, 2003)—or accept it as an unverified axiom; Turner said that he was "interested in finding something simpler" than full dependent types. Much as I find Turner's vision for total functional programming appealing, I fear that we are still some way off, even after 20 years of "more practice". However, I would be delighted to be shown to be unnecessarily pessimistic.

Trial division: Turner popularized the trial division algorithm in various publications; I believe that the earliest of these is 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. (List comprehensions appeared with KRC (Turner, 1982), originally called "ZF expressions", apparently being retrofitted to SASL the following year (Turner, 1983); the 1976 primes program was written instead with a filter.) Turner (2020) acknowledged that the program appeared in a famous paper by Kahn & 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.*

The program also appeared in papers about the dataflow language Lucid; for example, Ashcroft & Wadge (1977) again attribute it to Kahn. Kahn & 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), Kahn & MacQueen (1977), and Wadge & Ashcroft (1985) call the trial division algorithm "The Sieve of Eratosthenes", but McIlroy (1968, 2014) does not.

Proofs about infinite lists: In developing this proof, we also considered an ApproxWhile Lemma, analogous to the Approx Lemma (Lemma 1):

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

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

But this is much less general: the elements must now be ordered; and moreover, the bounds must grow without limit, so it doesn't hold universally for rationals, or pairs, or strings. Note that we used naturality of *approx* in the proof of Proposition 8; the corresponding property of *approxWhile* is not so straightforward. Perhaps it is possible to phrase a proof in terms solely of *approxWhile* without using *approx*? We have not pursued this further.

Bird's exercise: What of TFWH (Bird, 2014)? This paper was prompted by a series of ten emails from Francisco Lieberich (Lieberich, 2018) pointing out this and other errors in the book. 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 $(p_3)^2\ =\ 25$. A tighter but still sufficient condition is

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

I have added that observation to the errata for the book (Bird, 2014). Nevertheless, the margin of TFWH is too narrow to contain the proof presented here, so I still have no appropriate correction to apply.

[‡] Turner was an undergraduate at Oxford before starting his DPhil there in 1969, so it seems unlikely that he was present at McIlroy's talk in Cambridge. I conjecture that Turner learnt of the program via someone who did attend—perhaps Christopher Strachey, Turner's original DPhil supervisor.

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), and from the anonymous JFP reviewers. Thanks are due to Francisco Lieberich for alerting me to the error in TFWH, among many other errors. And of course this paper couldn't have happened without the seminal contributions of David Turner and Richard Bird.

Conflicts of interest

None.

Supplementary material

For supplementary material for this article, please visit https://doi.org/10.1017/S0956796824000194.

References

Ashcroft, E. A. & Wadge, W. W. (1977) Lucid, a nonprocedural language with iteration. *Comm. ACM*. **20**(7), 519–526.

Bertrand, J. (1845) Mémoire sur le nombre de valeurs que peut prendre une fonction quand on y permute les lettres qu'elle renferme. *J. l'École Royale Polytech*. **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. https://www.cs.ox.ac.uk/publications/books/functional/.

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

Kahn, G. & MacQueen, D. B. (1977) Coroutines and networks of parallel processes. In 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. *J. Funct. Program*. **14**(6), 759–763.

O'Neill, M. E. (2009) The genuine Sieve of Eratosthenes. *J. Funct. Program*. **19**(1), 95–105.

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

Théry, L. (2003) Proving pearl: Knuth's algorithm for prime numbers. In *Theorem Proving in Higher Order Logic*. Springer, pp. 304–318.

Tranah, D. (2024) "JFP 622". Personal communication (email).

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, pp. 1–28.

Turner, D. A. (1983) SASL language manual, "revised November 1983 for inclusion of ZF expressions". Technical report.

Turner, D. A. (2004) Total functional programming. *J. Univers. Comput. Sci*. **10**(7), 751–768.

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

Wadge, W. W. & Ashcroft, E. A. (1985) *Lucid, the Dataflow Programming Language*. Academic.